# COMP 532
# Machine Learning and BioInspired Optimization

## Lecture 10&11: Deep Learning

**Dr. Shan Luo**

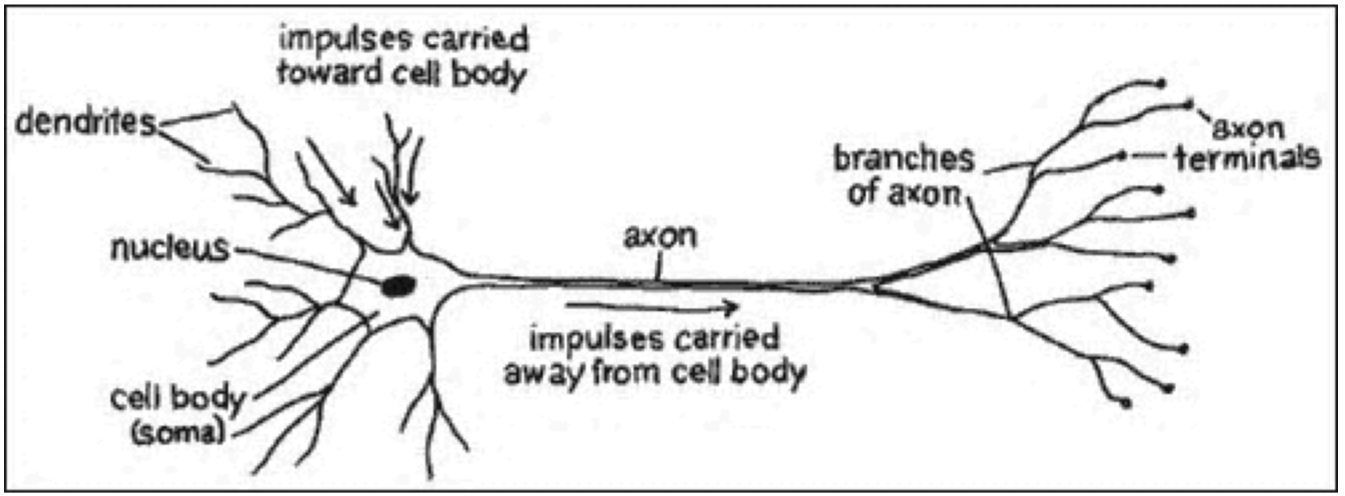
**Department of Computer Science**

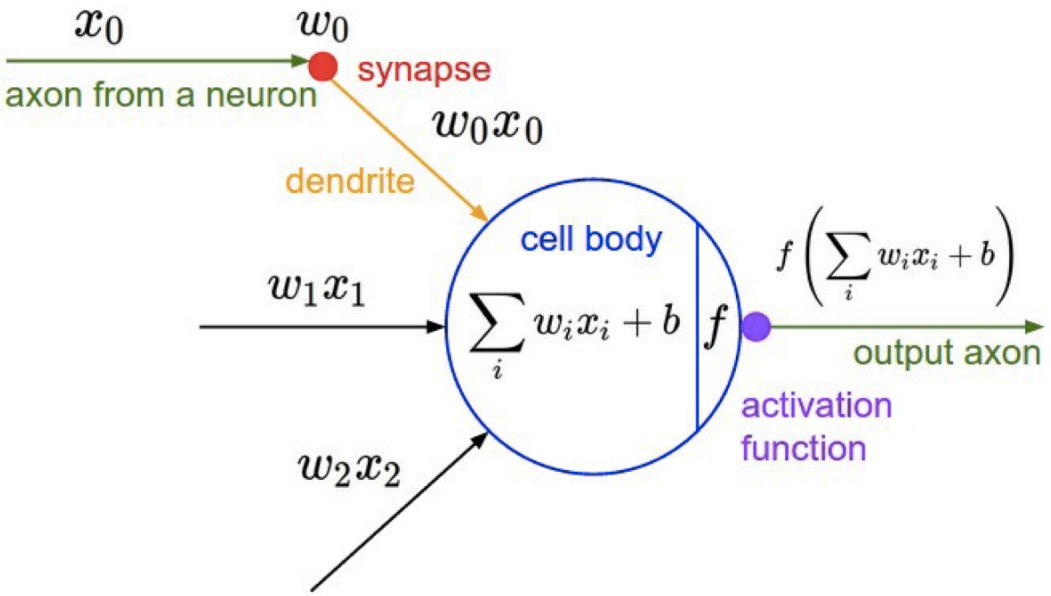
**shan.luo@liverpool.ac.uk**

# Overview (2 lectures)

- A first look at Artificial Neural Networks
  - Artificial Neurons
  - Activation Functions
- More on Perceptrons
  - Weights and bias
  - How to train them
- Training a neural network
  - Backpropagation basics

# From Real to Artificial Neurons



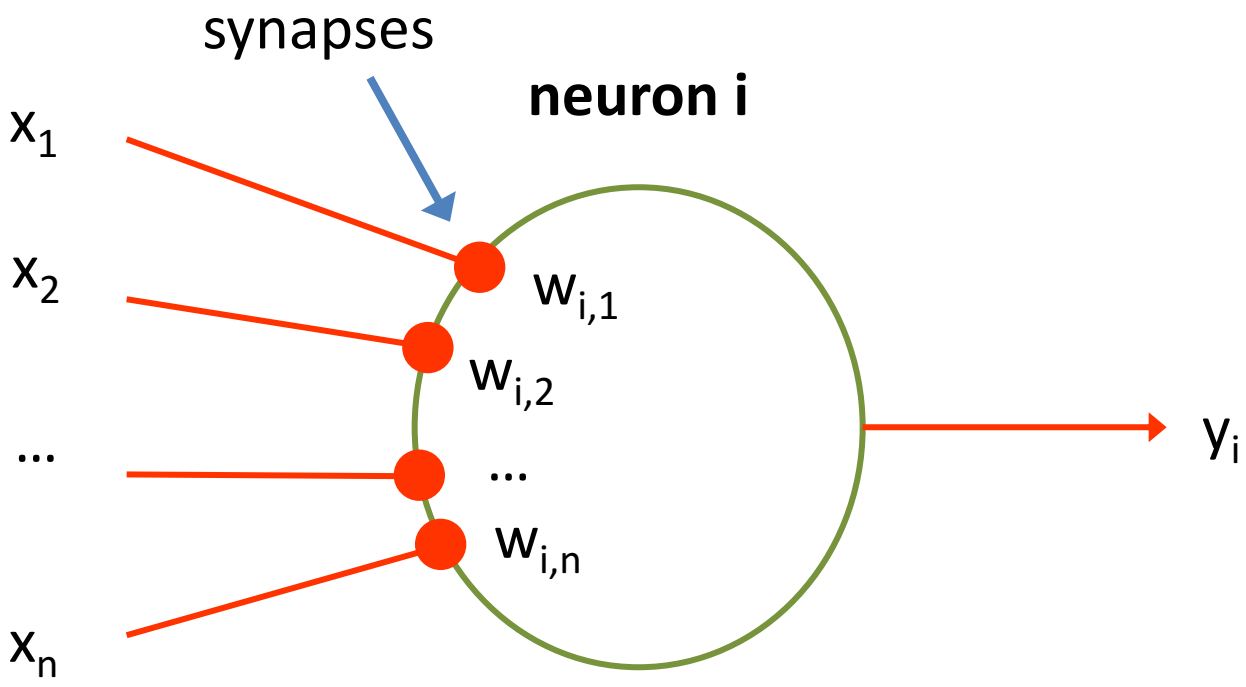
# How do NNs and ANNs work?

- NNs are able to **learn** by **adapting their connectivity patterns** so that the organism improves its behavior in terms of reaching certain (evolutionary) goals.

- The strength of a connection, or whether it is excitatory or inhibitory, depends on the state of a receiving neuron's **synapses**.

- The NN achieves **learning** by appropriately adapting the states of its synapses.

# An Artificial Neuron




net input signal: $\text{net}_i(t) = \sum_{j=1}^{n} w_{i,j}(t)x_j(t)$

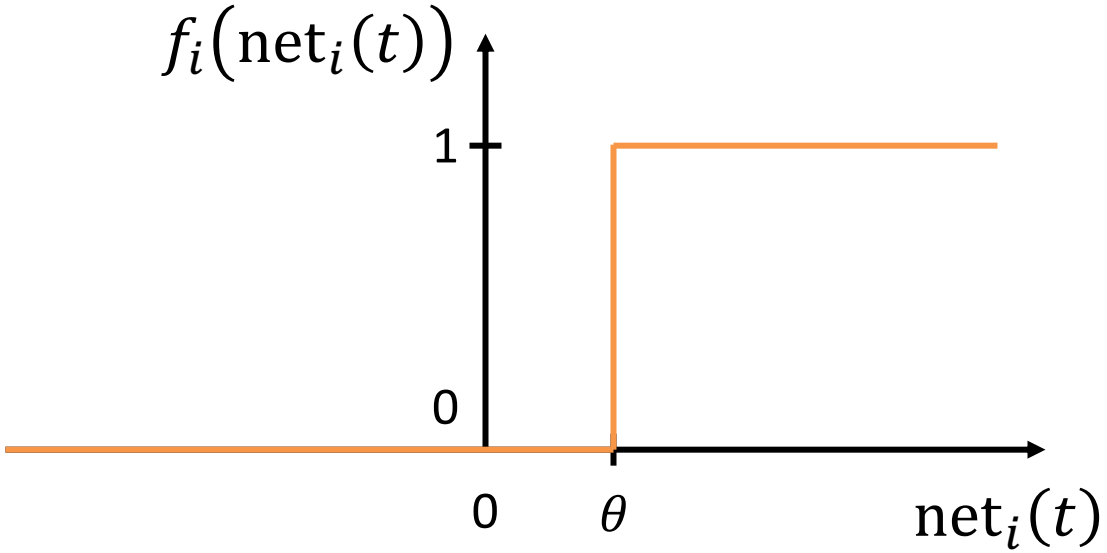
output signal: $y_i(t) = f_i\big(\text{net}_i(t)\big)$

# The Activation Function

One possible choice is a **threshold function**:

$$f_i\big(\text{net}_i(t)\big) = \begin{cases} 1 & \text{if } \text{net}_i(t) \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

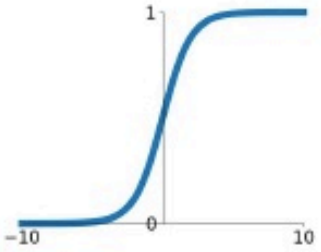
The graph of this function looks like this:



# The Activation Function

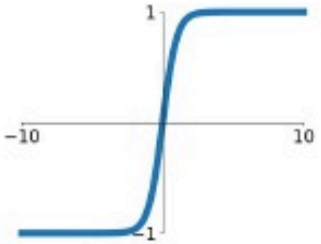
**Sigmoid**

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



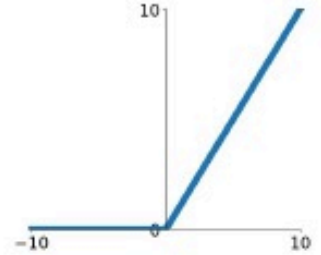

**tanh**

$$\tanh(x)$$



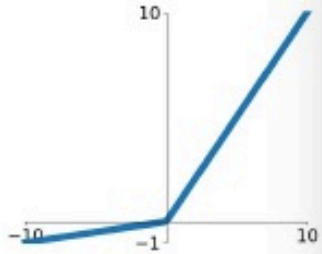

**ReLU**

$$\max(0, x)$$




**Leaky ReLU**

$$\max(0.1x, x)$$

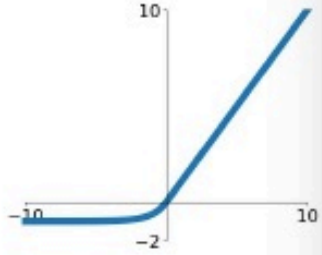



**Maxout**

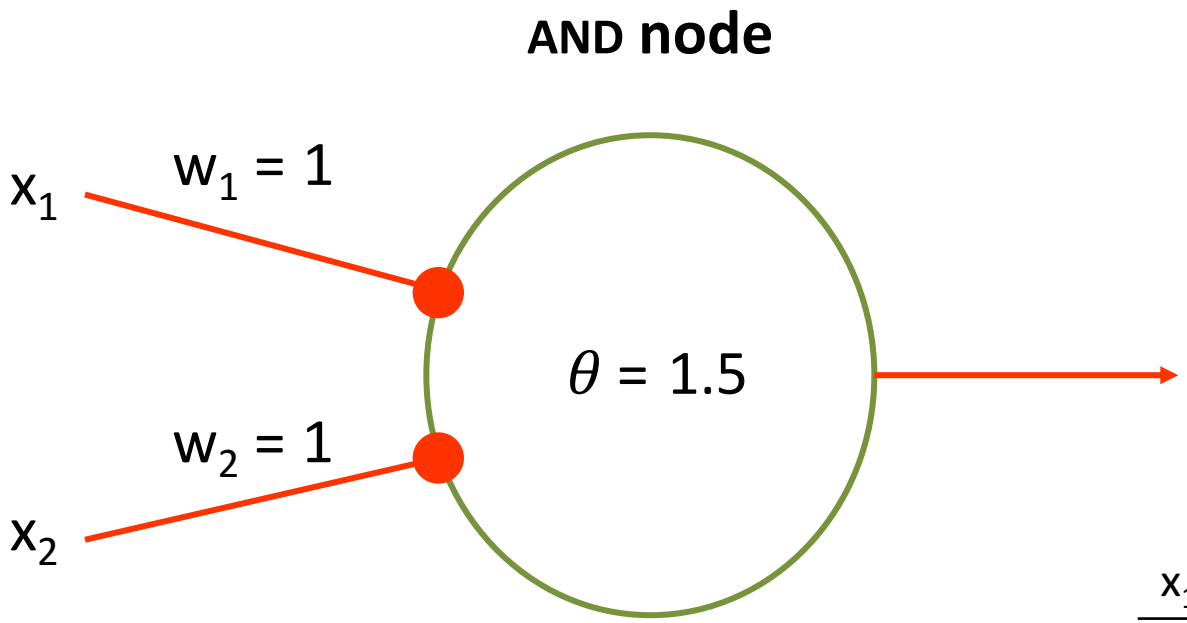
$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

**ELU**

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



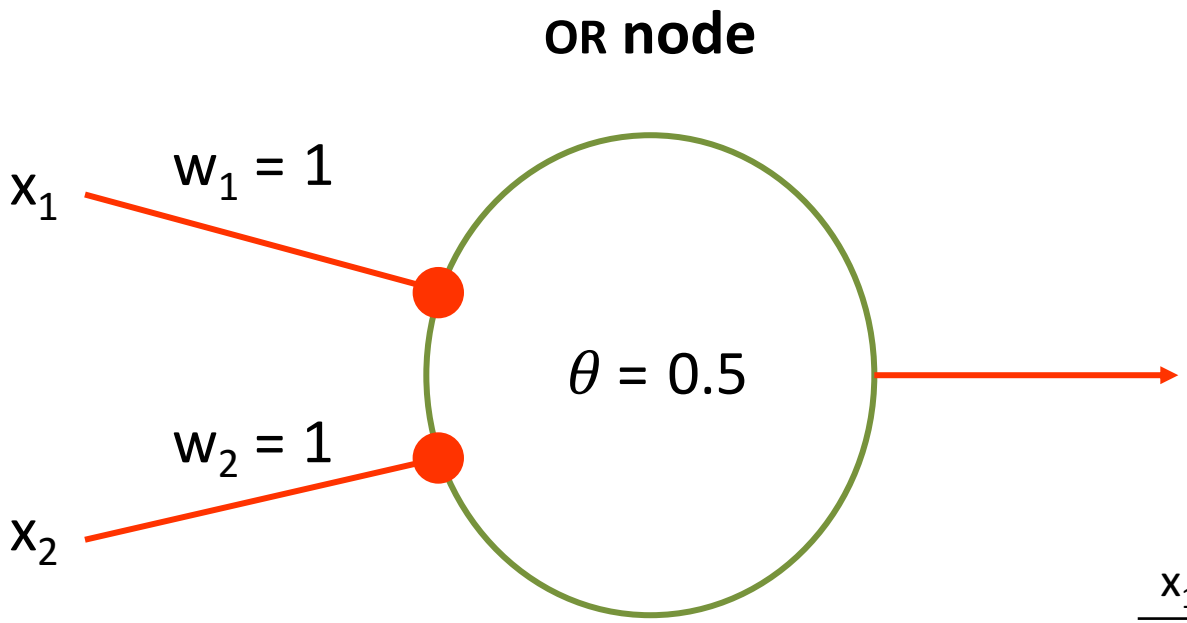
# Binary Analogy: Threshold Logic Units (TLUs)




| $x_1$ | $x_2$ | $x_1 \wedge x_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

TLUs in technical systems are similar to the threshold neuron model, except that TLUs only accept binary inputs (0 or 1).

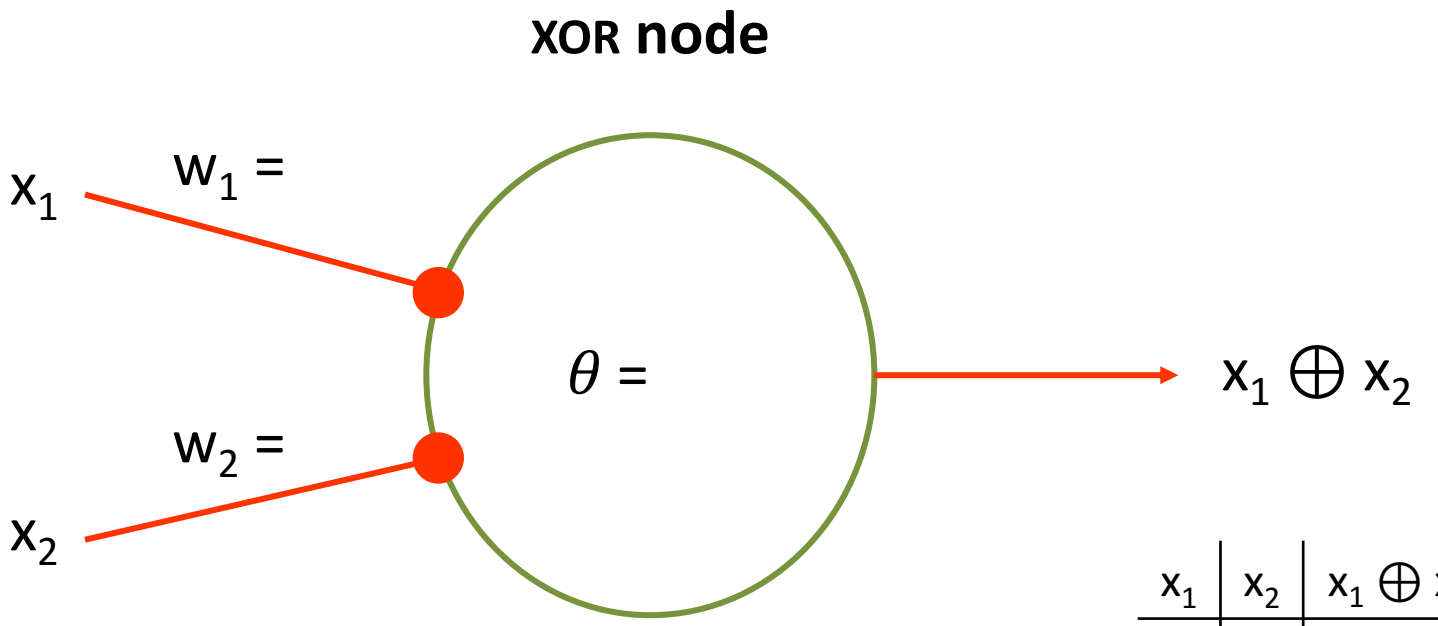
# Binary Analogy: Threshold Logic Units (TLUs)




| $x_1$ | $x_2$ | $x_1$ V $x_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

TLUs can implement various simple logical functions.

# Binary Analogy: Threshold Logic Units (TLUs)




| $x_1$ | $x_2$ | $x_1 \oplus x_2$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

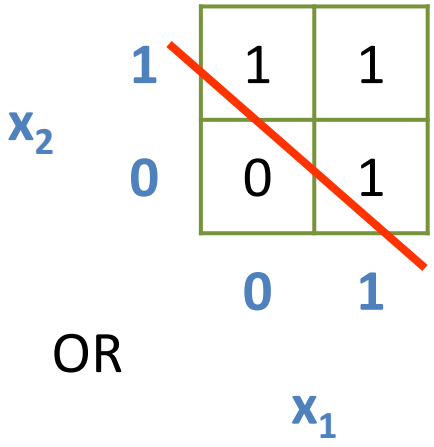
What about XOR (exclusive OR)?

**Impossible!** TLUs can only realize **linearly separable** functions.

# Linear Separability

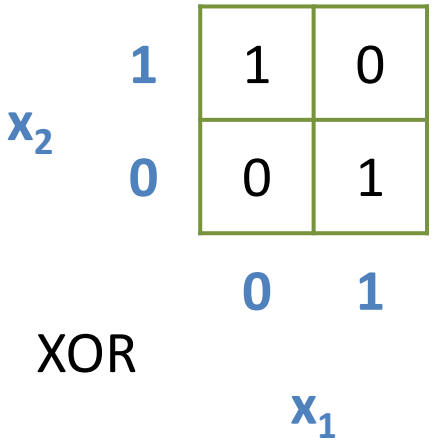
- A function $f:\{0, 1\}^n \rightarrow \{0, 1\}$ is **linearly separable** if the space of input vectors yielding 1 can be separated from those yielding 0 by a **linear surface** (hyperplane) in n dimensions.

- Examples (two dimensions):




OR

**linearly separable**




XOR

**not linearly separable**

# Linear Separability

- To explain linear separability, let us consider the function f:$\mathbb{R}^n \rightarrow \{0, 1\}$ with

$$f(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

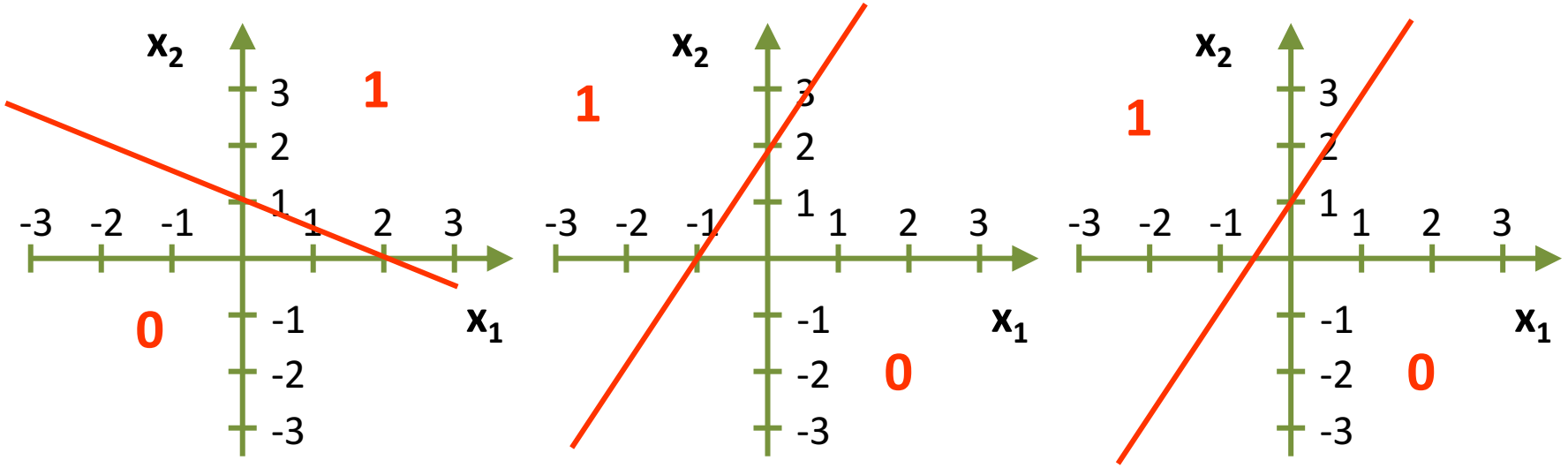
where $x_1, x_2, \ldots, x_n$ represent real numbers.

- This is exactly the function that our threshold neurons use to compute their output from their inputs.

# Linear Separability

$$f(x_1, x_2, \ldots, x_n) = \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0 & \text{otherwise} \end{cases}$$

Input space in the two-dimensional case (n = 2):




$w_1 = 1, w_2 = 2,$
$\theta = 2$

$w_1 = -2, w_2 = 1,$
$\theta = 2$

$w_1 = -2, w_2 = 1,$
$\theta = 1$

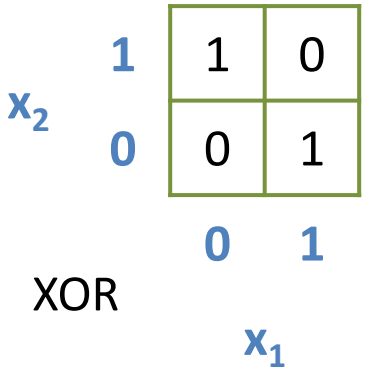
# Linear Separability

- By varying the weights and the threshold, we can realize **any linear separation** of the input space into a region that yields output 1, and another region that yields output 0.
- As we have seen, a **two-dimensional** input space can be divided by any straight line.
- A three-dimensional input space can be divided by any **two-dimensional plane**.
- In general, an **n-dimensional** input space can be divided by an **(n-1)-dimensional plane** (hyperplane).
- Of course, for n > 3 this is hard to visualize.

# Linear Separability

- Of course, the same applies to our original function f of the TLU using binary input values.
  - The only difference is the restriction in the input values.

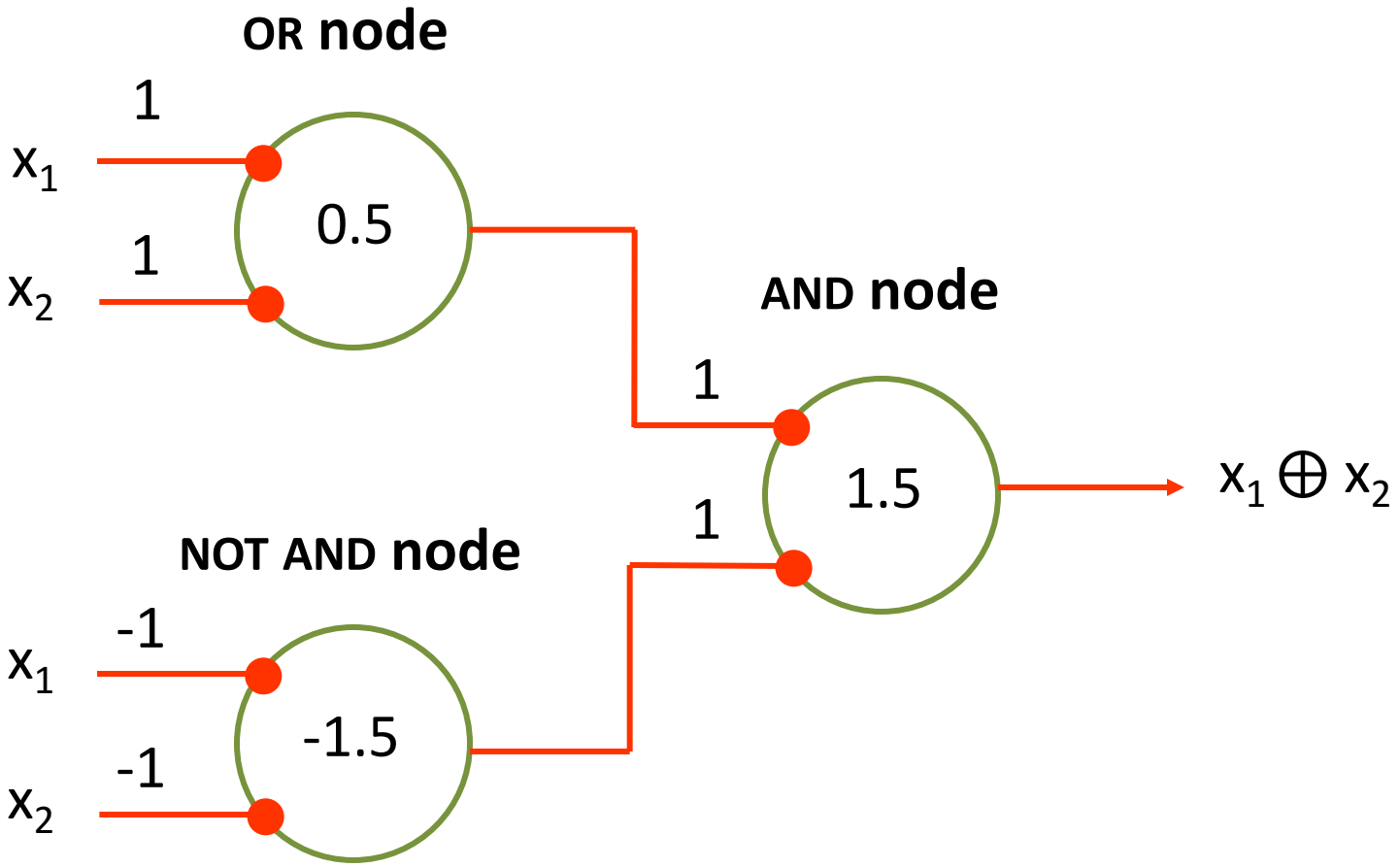
- Obviously, we cannot find a straight line to realize the XOR function:




- In order to realize XOR with TLUs, we need to combine multiple TLUs into a **network**.

# Multi-Layered XOR Network

Logic: XOR $\longleftrightarrow$ (OR) AND (NOT AND)

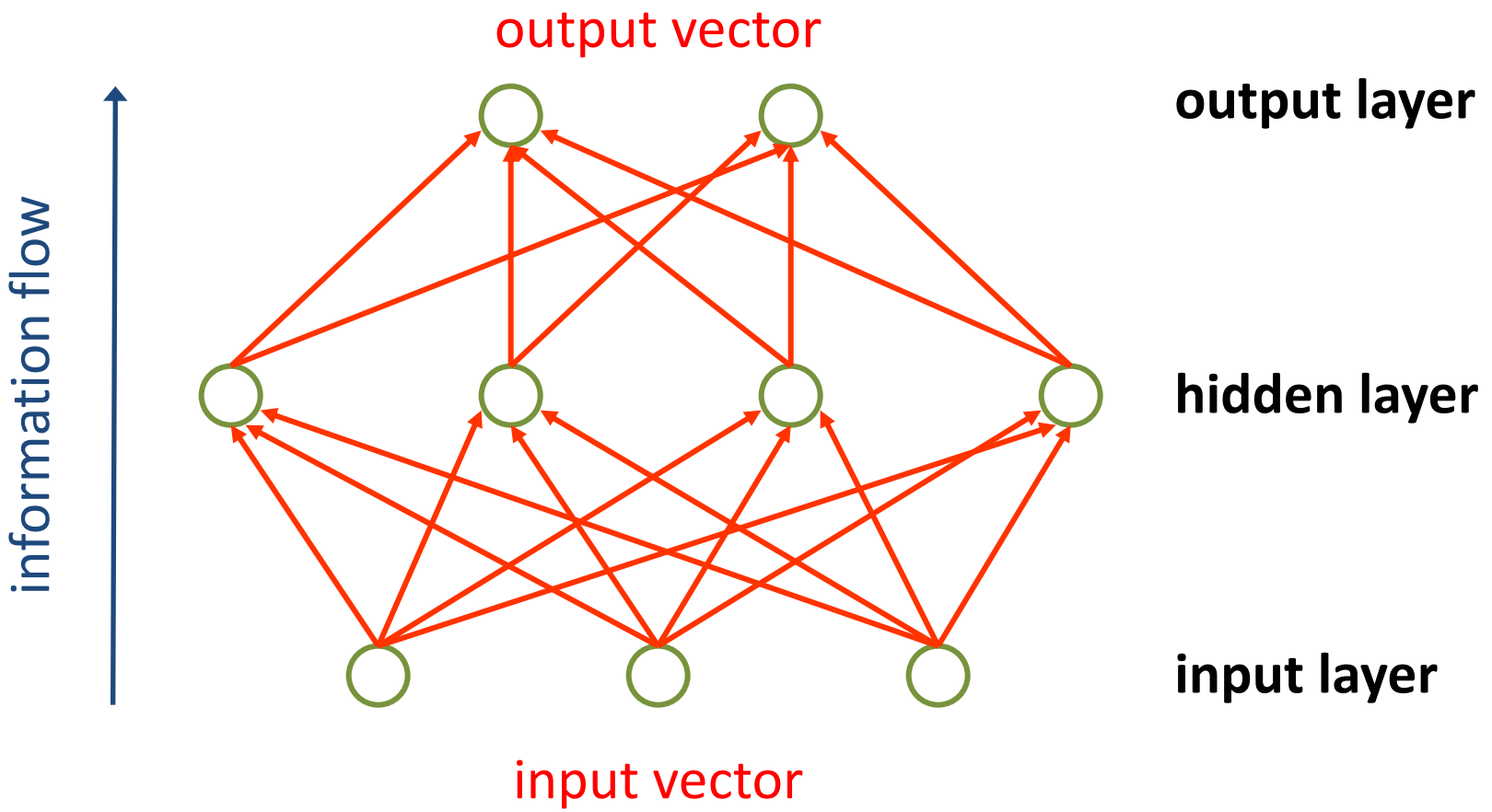


# Multi-Layer Networks

- The first layer, called **input layer**, just contains the input vector and does not perform any computations.
- The second layer, called **hidden layer**, receives input from the input layer and sends its output to the output layer.
- After applying their activation function, the neurons in the **output layer** contain the output vector.

# Feed-Forward Neural Network

**Example**: network function $f:\mathbb{R}^3 \rightarrow \{0, 1\}^2$



# Activation Functions

- We have already seen:
  - Threshold Neurons (**Perceptron**)
  - Threshold Logic Units (binary input)

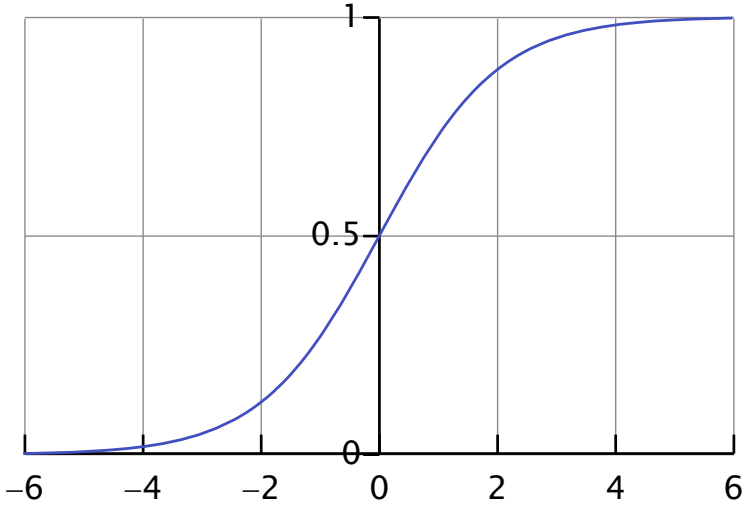
- Other options:
  - Linear Neuron (output is a linear combination of inputs)
  - Rectified Linear Unit (ReLU, we will see this later)
  - Sigmoidal Neurons

# Sigmoidal Neurons

- Very common type of artificial neuron, especially in learning networks

- Logistic activation function
  - "Squashes" the input to the interval [0, 1]

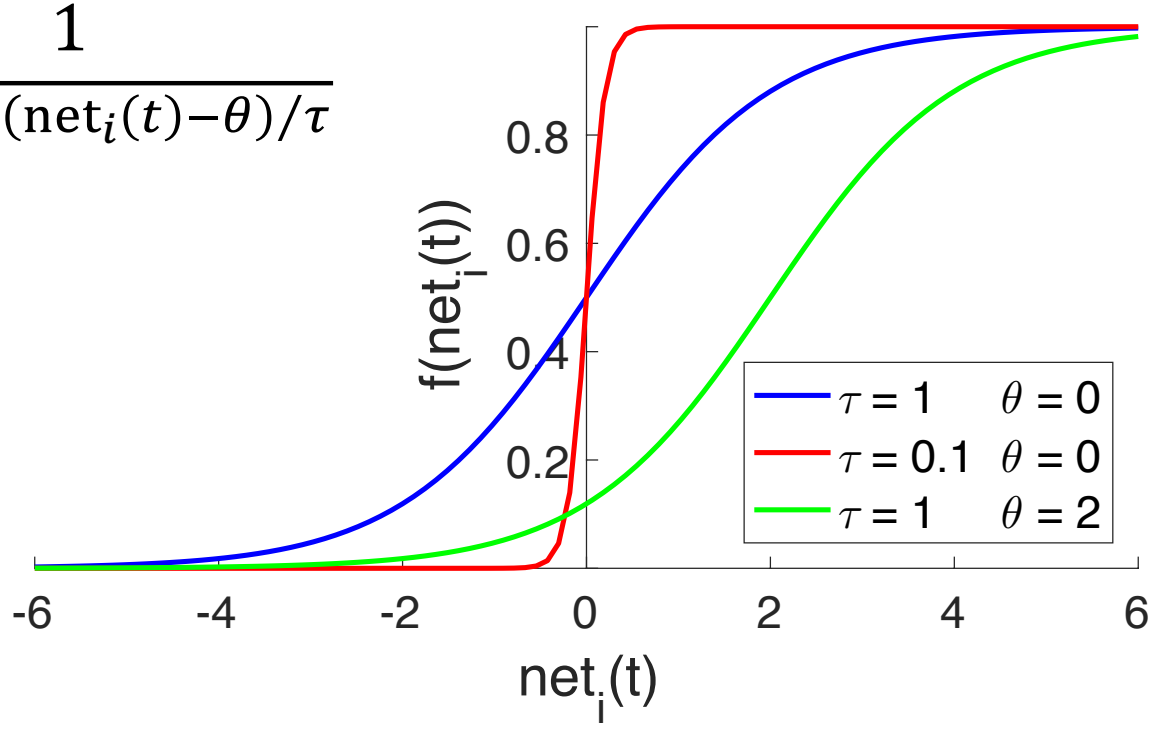
$$\sigma(x) = \frac{1}{1+e^{-x}}$$




- A network of sigmoidal neurons with $n$ inputs and $m$ outputs computes the function $f: \mathbb{R}^n \rightarrow (0, 1)^m$

# Sigmoidal Neurons

$$f_i\left(\text{net}_i(t)\right) = \frac{1}{1 + e^{-(\text{net}_i(t) - \theta)/\tau}}$$




The parameter $\tau$ controls the **slope** of the sigmoid, while the parameter $\theta$ controls the **horizontal offset** in a way similar to the threshold neurons.

# How to set Weights and Threshold?

- Typically: **supervised learning**
- In supervised learning, we train an ANN with a set of vector pairs, so-called **examples** or **training data**.
- By providing enough examples, the network may learn good **features** that describe the data “in general”

# Supervised Learning

- Each training pair $(\mathrm{x}, \mathrm{y})$ consists of an **input vector $\mathbf{x}$** and a corresponding **output vector $\mathbf{y}$**.
  - Whenever the network receives input $\mathrm{x}$, we would like it to provide output $\mathrm{y}$.
  - The examples thus describe the **function** that we want the network to **learn**.

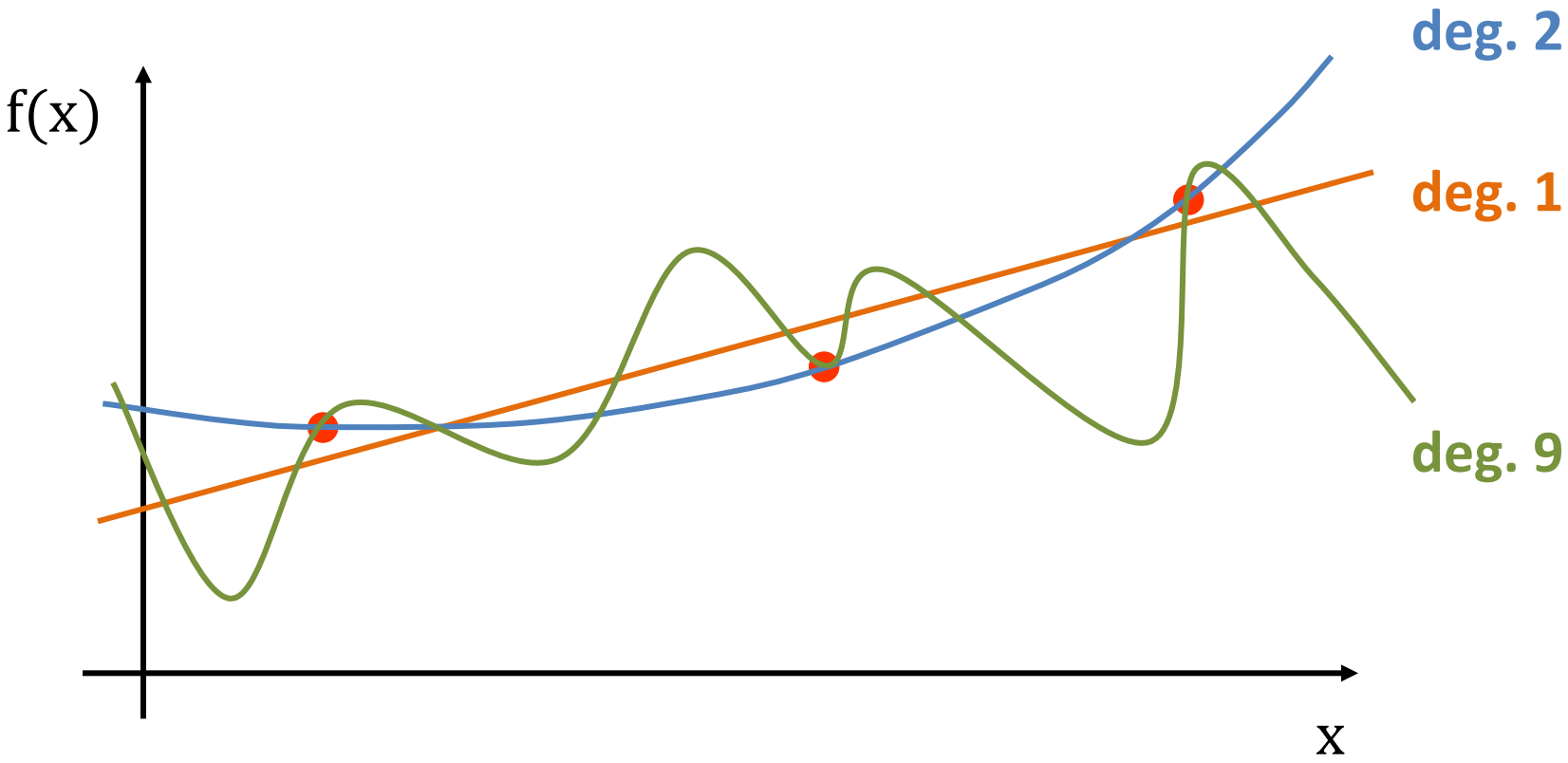
- Besides learning the exemplars, we would like our network to **generalize**, that is, give plausible output for inputs that the network had not been trained with.

# Supervised Learning

- There is a **tradeoff** between a network's ability to **precisely learn the given examples** and its ability to **generalize**.

- This problem is similar to **fitting a function** to a given set of data points.
  - Let's assume that we want to find a fitting function $f:\mathbb{R}\rightarrow\mathbb{R}$ for a set of three data points.
  - We can try to do this with polynomials of different degree (complexity).

# Function Approximation




Perhaps the polynomial of degree 2 provides the most **plausible** fit…

# Supervised Learning

- The same principle applies to ANNs:
  - **Too few neurons**: it may not have enough degrees of freedom to precisely approximate the desired function.
    **=> underfitting!**
  - **Too many neurons**: it will learn the examples perfectly, but its additional degrees of freedom may cause it to show implausible behavior for untrained inputs, i.e. poor generalization.
    **=> overfitting!**

- Unfortunately, no known equations will tell you the optimal size of your network for a given application; there are only heuristics.

# Evaluation of Networks

- **Basic idea**: define **error function** and measure error for untrained data (testing set)
  - Typical:

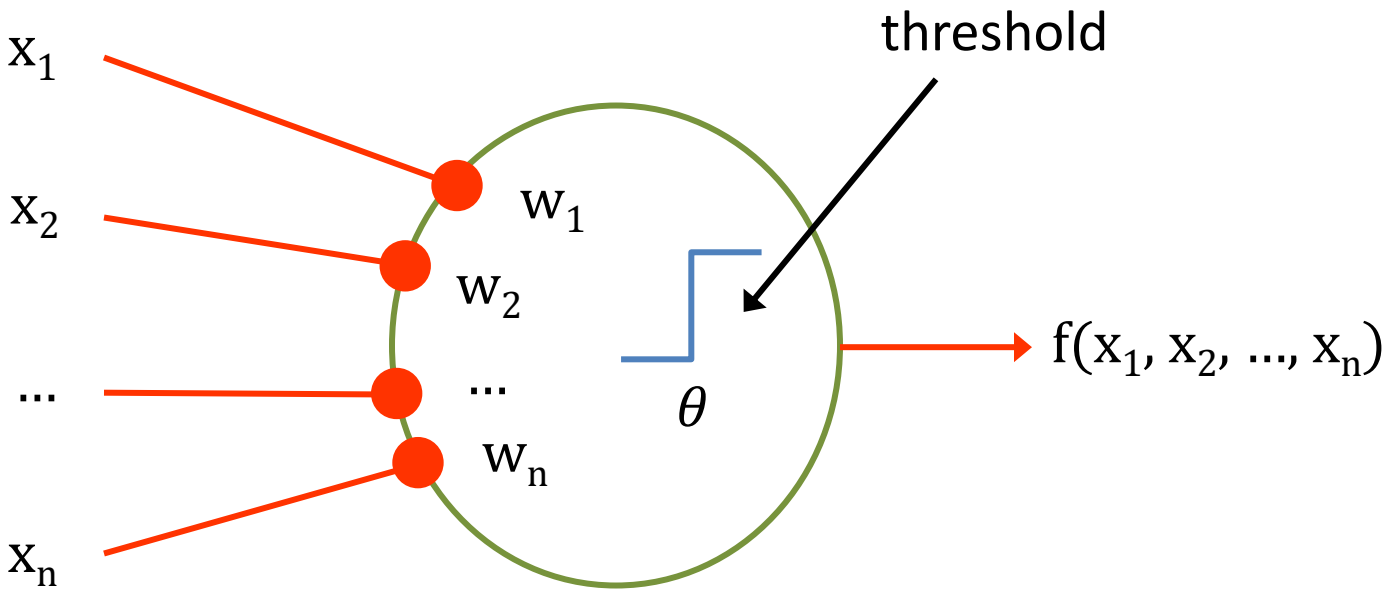
$$E = \sqrt{\sum_i (d_i - o_i)^2}$$

    where $d$ is the desired output, and $o$ is the actual output.

  - For classification:

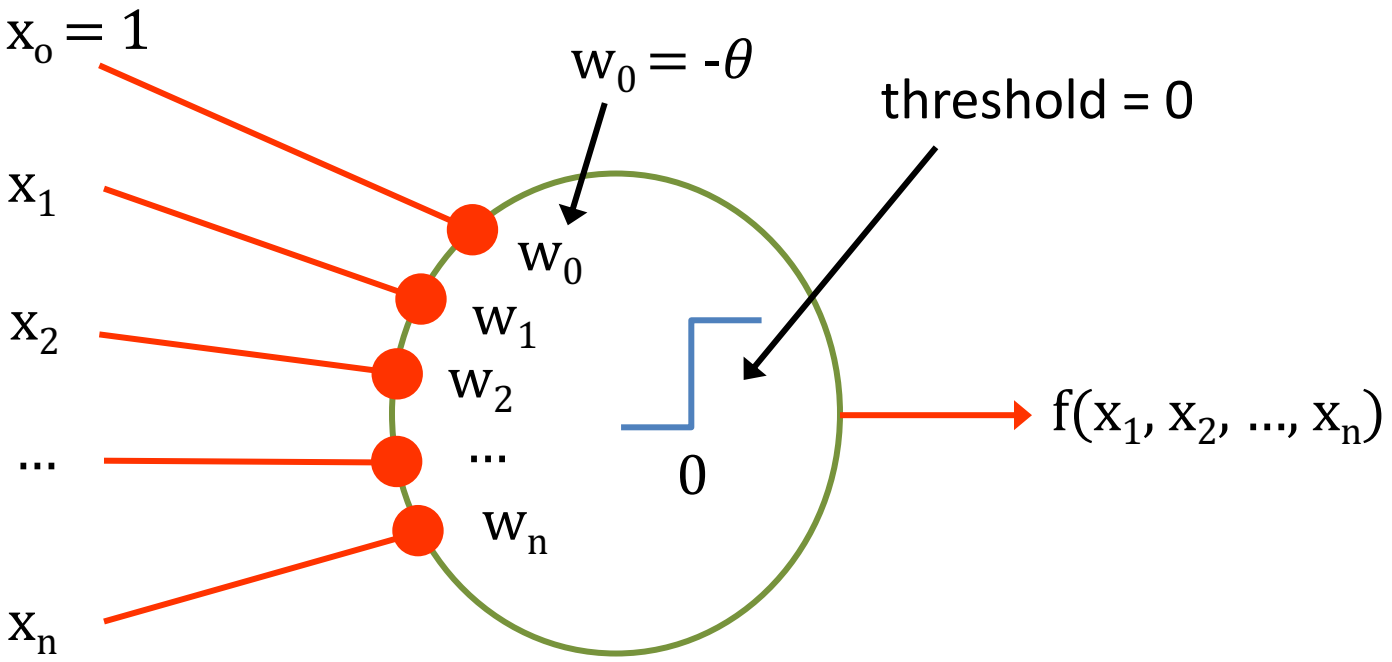
$$E = \frac{\#\text{correctly classified samples}}{\#\text{samples}}$$

# The Perceptron




- Net input signal: $\text{net} = \sum_{i=1}^{n} w_i x_i$
- Output signal: $f(\text{net}) = \begin{cases} +1 & \text{if net} > \theta \\ -1 & \text{otherwise} \end{cases}$

# The Perceptron – Bias!




- Net input signal: $\text{net} = \sum_{i=0}^{n} w_i x_i$
- Output signal: $f(\text{net}) = \begin{cases} +1 & \text{if net} > 0 \\ -1 & \text{otherwise} \end{cases}$

**Only the weight vector is adaptable, not the threshold!**

# Perceptron Computation

- Similar to a TLU, a perceptron divides its *n*-dimensional input space by an (*n - 1*)-dimensional hyperplane given by:

$$w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = 0$$

  - for $w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0$, its output is 1, and
  - for $w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n \leq 0$, its output is -1.

- With the right weight vector $(w_0, \ldots, w_n)^\intercal$, a single perceptron can compute any linearly separable function.
- So, how to find such a weight vector for a given function?

# Perceptron Training

- For example, suppose input $x$ with $\text{class}(x) = -1$
  - If $w \cdot x > 0$, then we have a misclassification.
  - Then the weight vector needs to be modified to $w + \Delta w$ with $(w + \Delta w) \cdot x < w \cdot x$ to possibly improve classification.

- We can choose $\Delta w = -\eta x$, because

$$(w + \Delta w) \cdot x = (w - \eta x) \cdot x = w \cdot x - \eta x \cdot x < w \cdot x$$

and $x \cdot x$ is the square of the length of vector $x$ and is thus positive.

# Perceptron Training

- Same for $\text{class}(x) = 1$, but with opposite signs
  - we introduce $\tilde{x}$ to unify these two cases.

- Suppose misclassified input $x$
  - Let $\tilde{x} = \text{class}(x) \cdot x$, then $w^{k-1} \cdot \tilde{x} < 0$
  - Then we can always set $\Delta w = \eta \tilde{x}$ such that

$$w^k = w^{k-1} + \eta \tilde{x}$$

  - **Intuition**:
    - if $\text{class}(x) = 1$, then $w^{k-1} \cdot \tilde{x} < 0$ and $\Delta w = \eta \tilde{x} > 0$
    - if $\text{class}(x) = -1$, then $w^{k-1} \cdot \tilde{x} < 0$ and $\Delta w = \eta \tilde{x} > 0$
    - so, we always move $w$ in the right direction

# Perceptron Training Algorithm

**ALGORITHM** Perceptron Training;

Start with a randomly chosen weight vector $w^0$;

Let $k = 1$;

**WHILE** there exist input vectors that are misclassified by $w^{k-1}$, **DO**

Let $x$ be a misclassified input vector;

Let $\tilde{x} = \text{class}(x) \cdot x$, implying that $w^{k-1} \cdot \tilde{x} < 0$;

Update the weight vector to $w^k = w^{k-1} + \eta\tilde{x}$;

Increment $k$;

**END-WHILE**

# Learning Rate and Termination

- **Ideally**: terminate when all samples are correctly classified

- If the number of misclassified samples has not changed in a large number of steps, the problem could be the choice of learning rate $\eta$:
  - If $\eta$ is too large, classification may just be swinging back and forth, taking a long time to stabilize
  - If $\eta$ is too small, changes in classification can be extremely slow

- If changing $\eta$ does not help, the samples **may not be linearly separable**, and training should terminate

# Visualizing Perceptrons in 2D

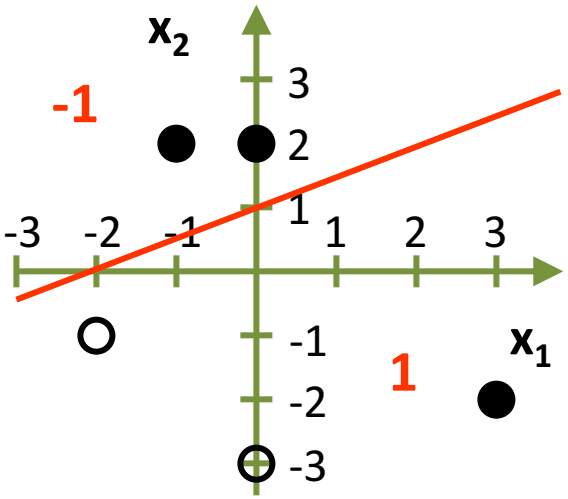
- How can we visualize a straight line defined by an equation such as $w_0 + w_1 x_1 + w_2 x_2 = 0$?

- One possibility is to determine the points where the line crosses the coordinate axes:
  - $x_1 = 0 \Rightarrow w_0 + w_2 x_2 = 0 \Rightarrow w_2 x_2 = -w_0 \Rightarrow x_2 = -w_0/w_2$
  - $x_2 = 0 \Rightarrow w_0 + w_1 x_1 = 0 \Rightarrow w_1 x_1 = -w_0 \Rightarrow x_1 = -w_0/w_1$

- Thus, the line crosses at $(0, -w_0/w_2)$ and $(-w_0/w_1, 0)$.
  - If $w_1$ or $w_2$ is 0, it just means that the line is horizontal or vertical, respectively.
  - If $w_0$ is 0, the line hits the origin, and its slope is $-w_1/w_2$.

# Perceptron Learning Example

- We would like our perceptron to correctly classify the five 2-dimensional data points below.
- Let the random initial weight vector $w^0 = (2,1,-2)^T$.
  - The dividing line crosses at $(0, 1)^T$ and $(-2, 0)^T$.




○ class -1
● class 1

Let us pick the misclassified point $(-2, -1)^T$ for learning:

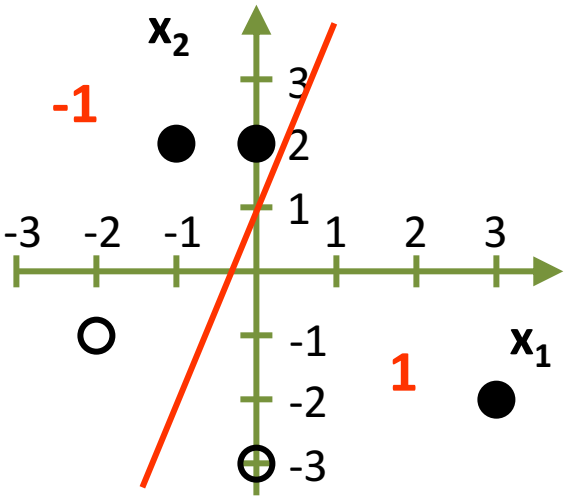
$\mathbf{x} = (1, -2, -1)^T$ (include bias $x_0 = 1$)

$\tilde{\mathbf{x}} = (-1)\cdot(1, -2, -1)^T$ ($\mathbf{x}$ is in class -1)

$\tilde{\mathbf{x}} = (-1, 2, 1)^T$

# Perceptron Learning Example

- $w^1 = w^0 + \tilde{x}$ (assume $\eta = 1$ for simplicity)
- $w^1 = (2,1,-2)^T + (-1,2,1)^T = (1,3,-1)^T$
  - The dividing line crosses at $(0, 1)^T$ and $(-1/3, 0)^T$.




Let us pick the next misclassified point $(0, 2)^T$ for learning:

$\mathbf{x} = (1, 0, 2)^T$ (include bias $x_0 = 1$)

$\tilde{\mathbf{x}} = (1)\cdot(1, 0, 2)^T$ ($\mathbf{x}$ is in class 1)

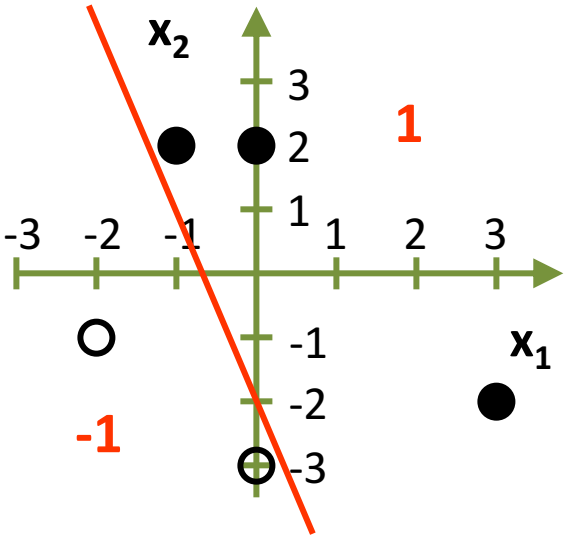
$\tilde{\mathbf{x}} = (1, 0, 2)^T$

○ class -1

● class 1

# Perceptron Learning Example

- $w^2 = w^1 + \tilde{x}$
- $w^2 = (1,3,-1)^T + (1,0,2)^T = (2,3,1)^T$
  - Now the line crosses at $(0, -2)^T$ and $(-2/3, 0)^T$.

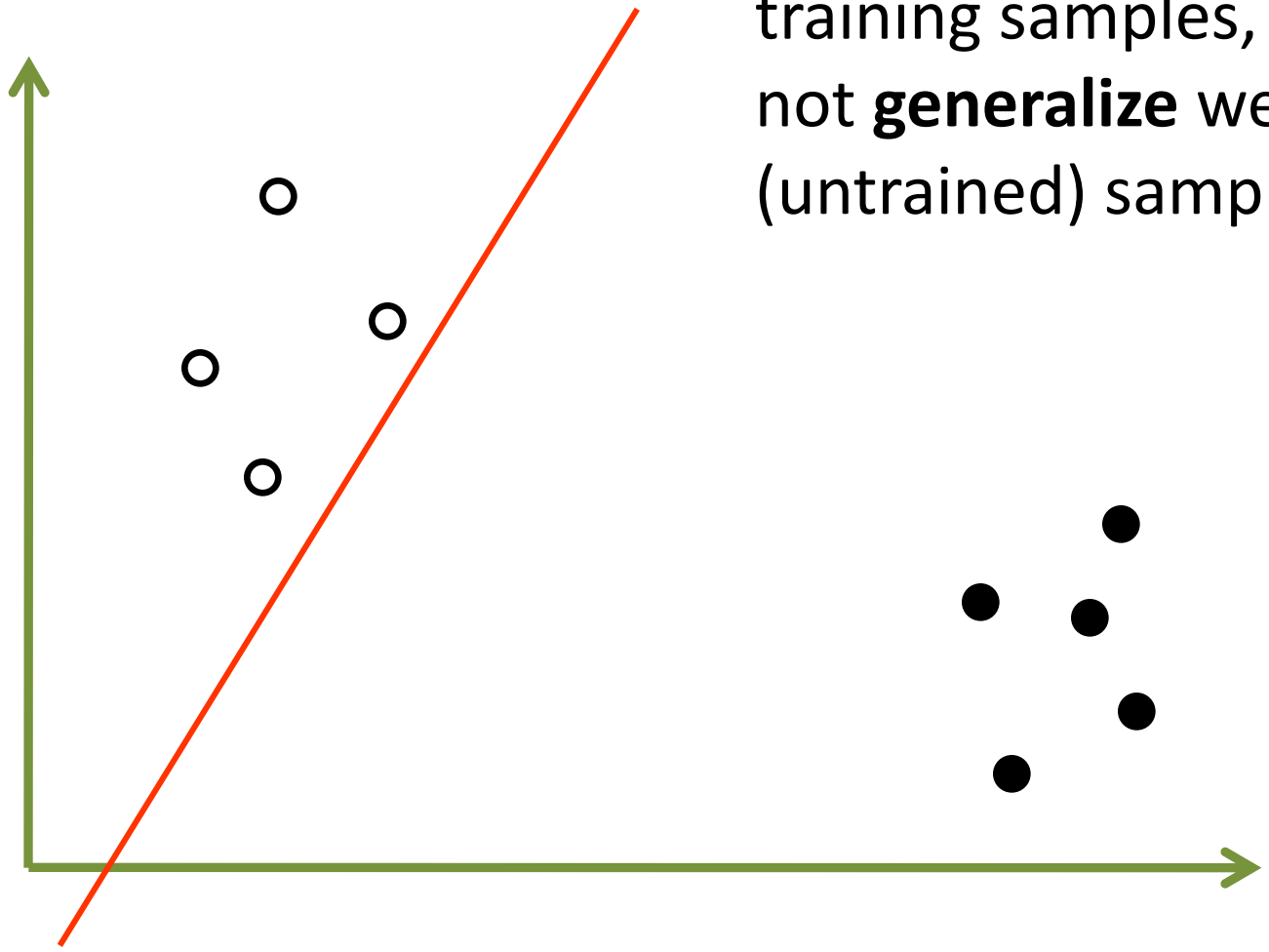



**All points are now correctly classified!**

The learning process terminates.

Typically, (many) more iterations are needed.

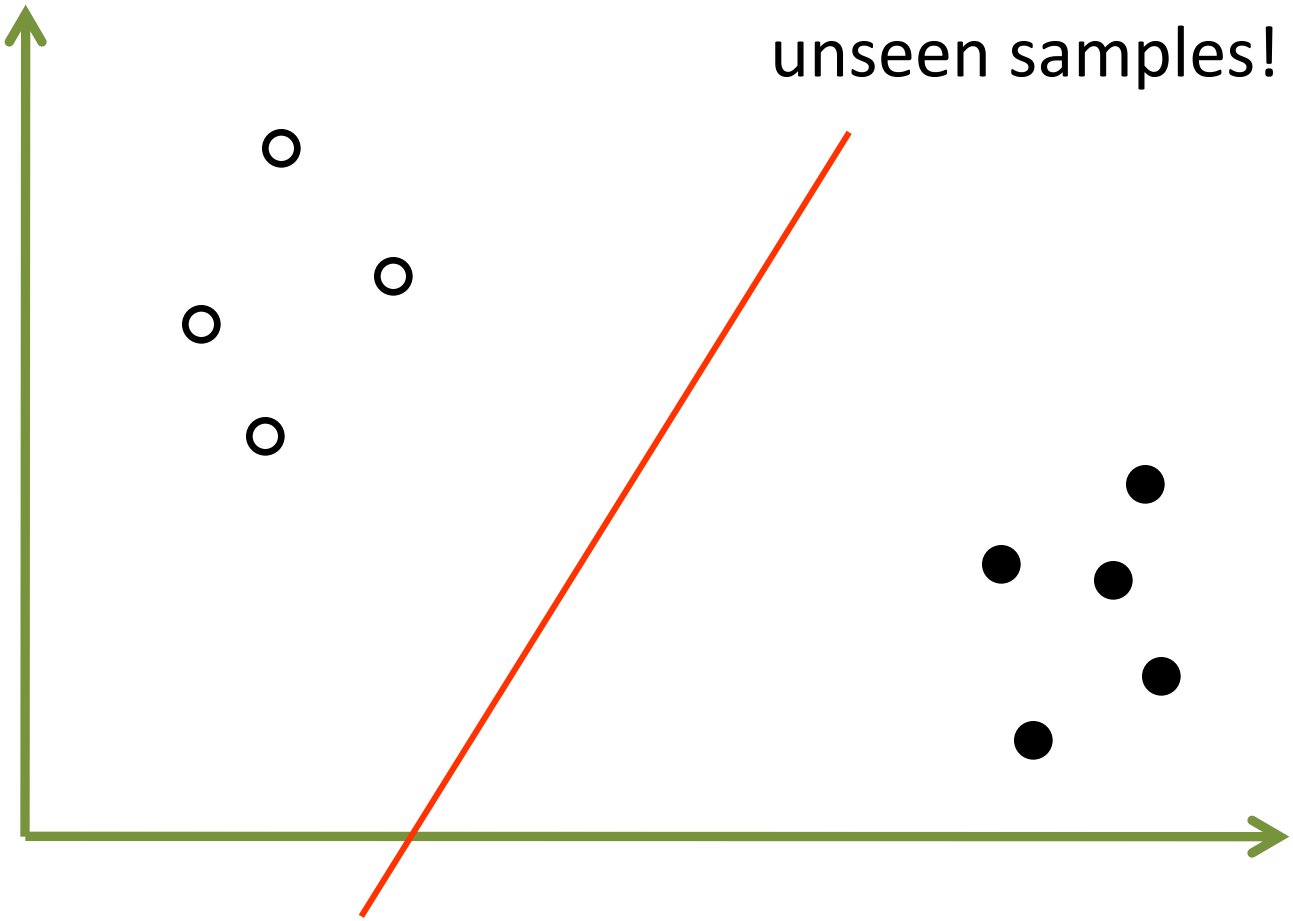
# Perceptron Learning Results



Perfect classification of training samples, but may not **generalize** well to new (untrained) samples.

# Perceptron Learning Results

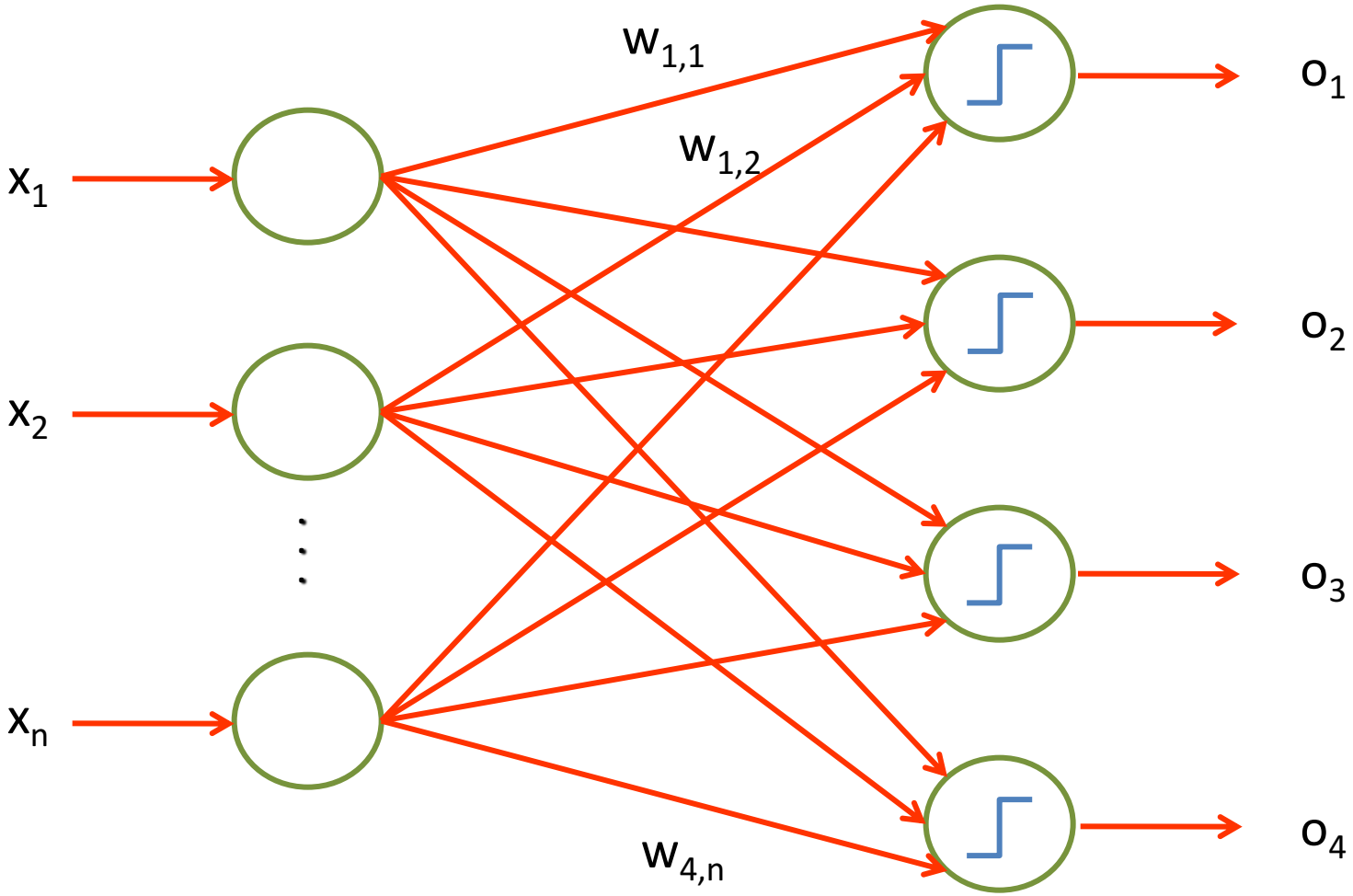
This function is **likely** to perform better on unseen samples!



# Multiclass Discrimination

- Often, our classification problems involve **more than two classes**.
- For example, character recognition requires at least 26 different classes.
- We can perform such tasks using **layers** of perceptrons.

# Multiclass Discrimination




A four-node perceptron for a four-class problem in n-dimensional input space

# Multiclass Discrimination

- Each perceptron learns to recognize one particular class, i.e., output 1 if the input is in that class, and 0 otherwise.
  - The units can be trained separately and in parallel.

- In production mode, the network decides that its current input is in the $k^{th}$ class if and only if $o_k = 1$, and for all $j \neq k$, $o_j = 0$, otherwise it is misclassified.

- For units with real-valued output, the neuron with maximal output can be picked to indicate the class of the input.
  - This maximum should be significantly greater than all other outputs, otherwise the input is misclassified.

# Multilayer Networks

- Although **single-layer perceptron networks** can distinguish between any number of classes, they still require **linear separability** of inputs.

- To overcome this serious limitation, we can use **multiple layers** of neurons.

- However, their non-differentiable output function led to an inefficient and weak learning algorithm.

- The idea that eventually led to a breakthrough was the use of **continuous output functions** and **gradient descent**.

# Multilayer Networks

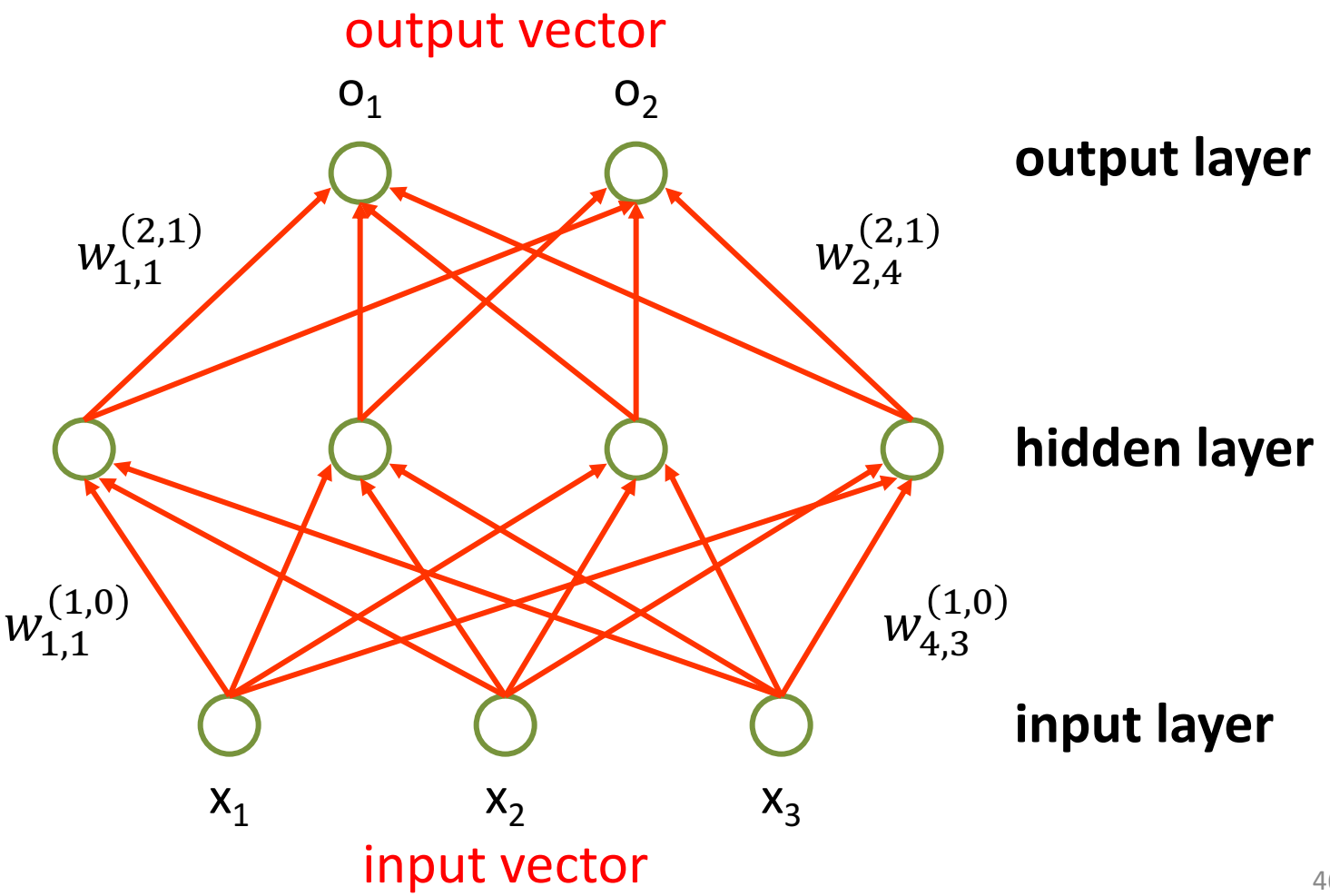
- The resulting **backpropagation** algorithm was popularized by Rumelhart, Hinton, and Williams in 1986.

- This algorithm solved the "credit assignment" problem, i.e., crediting or blaming individual neurons across layers for particular outputs.

- The **error at the output layer** is **propagated backwards** to units at lower layers, so that the weights of all neurons can be adapted appropriately.

# Terminology

- **Example**: network function $f:\mathbb{R}^3 \rightarrow \mathbb{R}^2$



# Backpropagation Learning

- The goal of the **backpropagation algorithm** is to modify the network's weights so that its output vector

$$o_p = (o_{p,1}, o_{p,2}, \dots, o_{p,K})$$

is as close as possible to the desired output vector

$$d_p = (d_{p,1}, d_{p,2}, \dots, d_{p,K})$$

for $K$ output neurons and input patterns $p = 1, \dots, P$.

- The set of input-output pairs (examples) $\{(x_p, d_p): p = 1, \dots, P\}$ constitutes the **training set**.

# Backpropagation Learning

- First, we need a cumulative error function that is to be minimized

$$\text{Error} = \sum_{p=1}^{P} \text{Err}(o_p, d_p)$$

- Typical choice: **mean square error** (MSE)

$$\text{MSE} = \frac{1}{P} \sum_{p=1}^{P} \sum_{j=1}^{K} (o_{p,j} - d_{p,j})^2$$

- How to minimize the MSE: **gradient descent**

# Gradient Descent

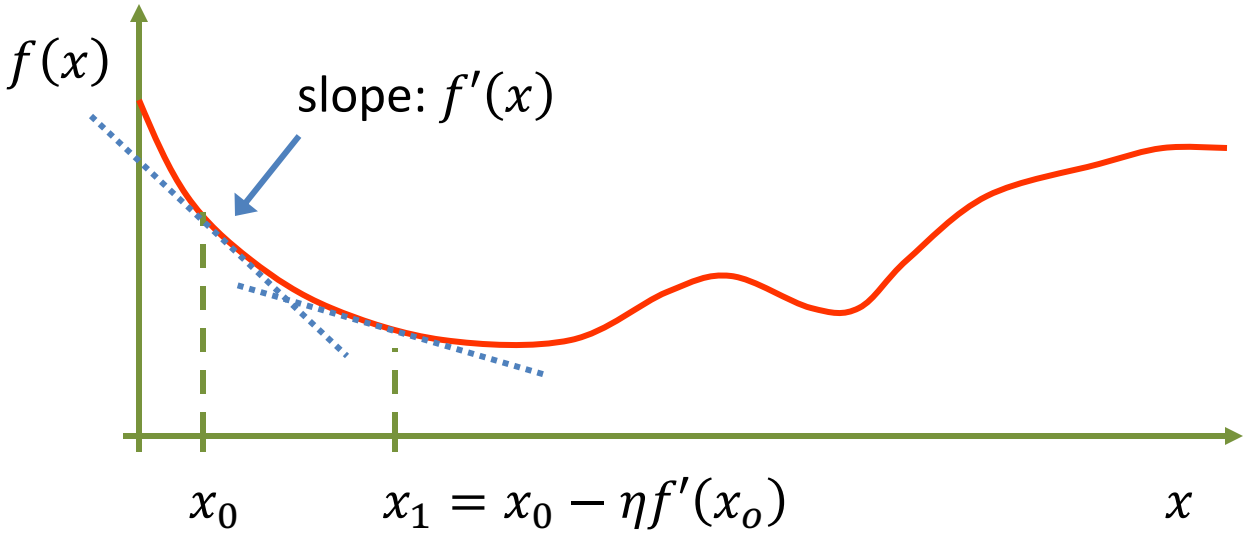
- **Gradient descent** is a very common technique to find the absolute minimum of a function.

- It is especially useful for high-dimensional functions.

- It is used to iteratively minimize the network's (or neuron's) error by **finding the gradient** of the error surface in weight-space and **adjusting the weights** in the opposite direction.

# Gradient Descent

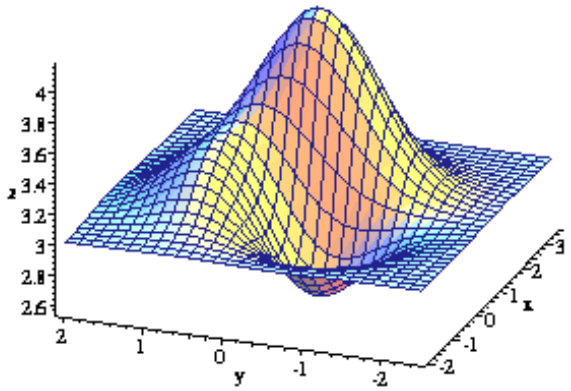
- **Gradient-descent example**: finding the absolute minimum of a one-dimensional error function $f(x)$:

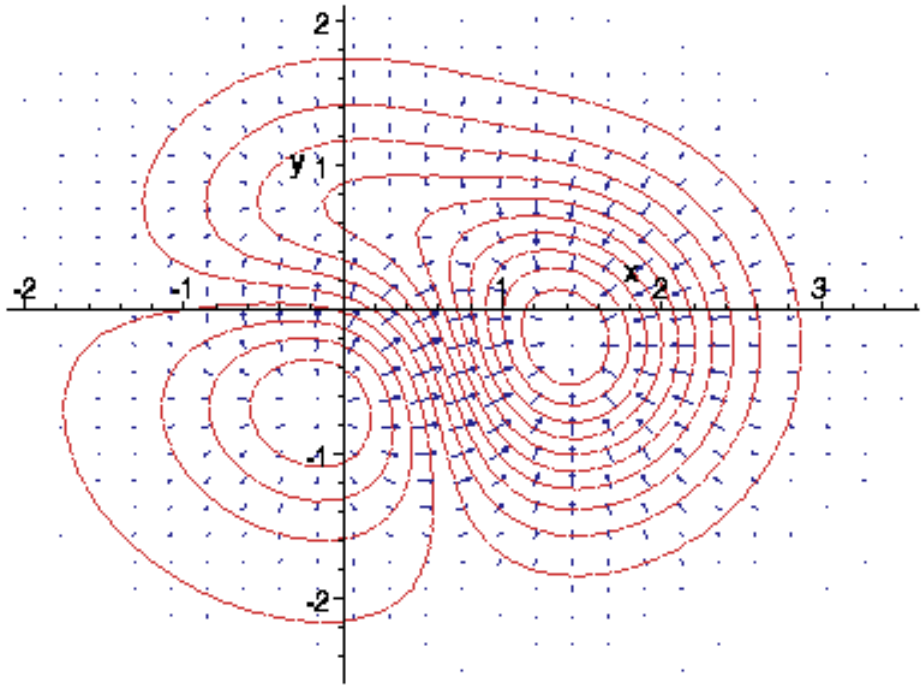



- Repeat this iteratively until for some $x_i$, $f'(x_i)$ is sufficiently close to 0.

# Gradient Descent

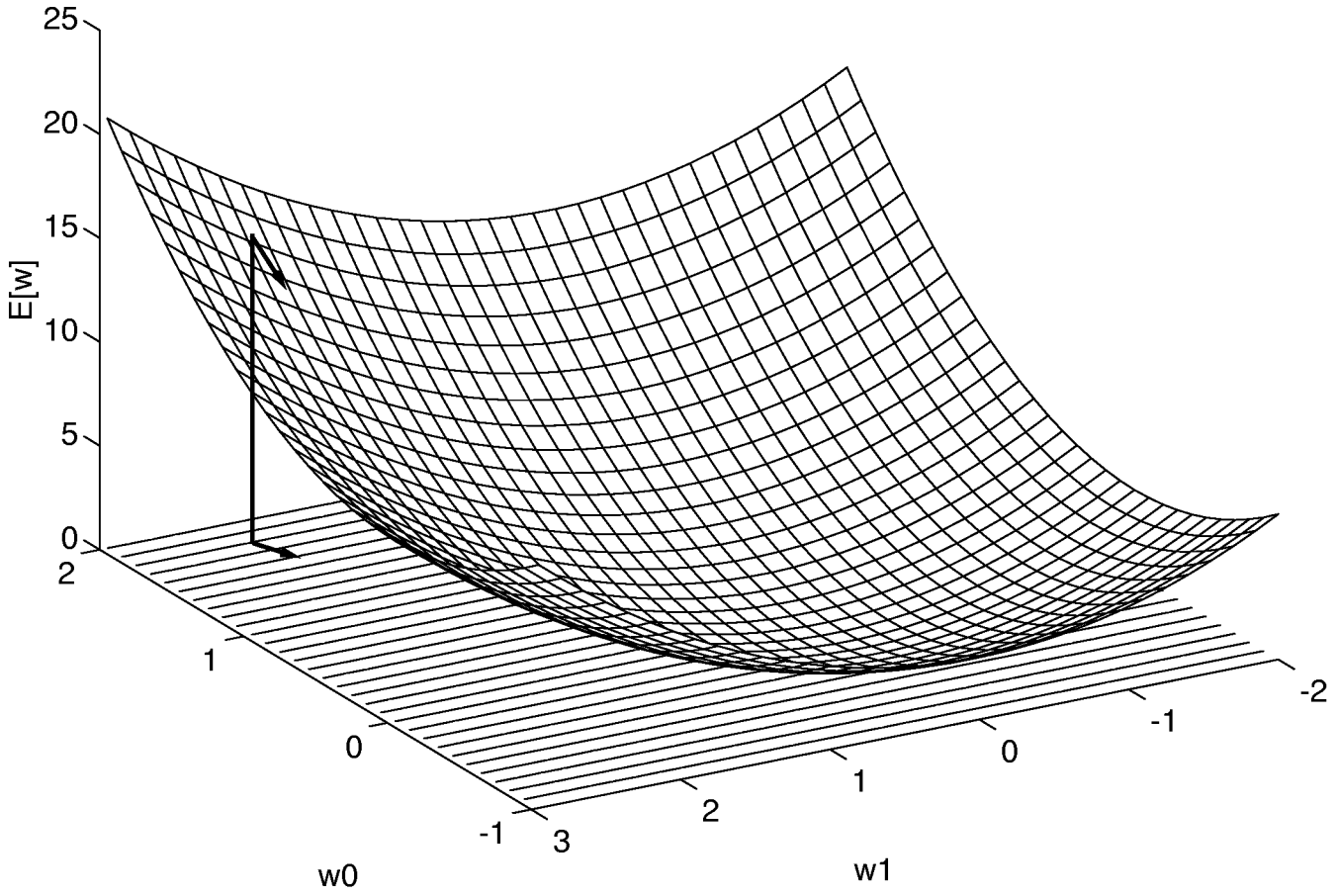
Gradients of
**two-dimensional functions:**




Arrows in the contour plot indicate the gradient of the function at different locations. The gradient is always pointing in the direction of the **steepest increase** of the function. In order to find the function's minimum, we should always **move against the gradient**.

# Gradient Descent



# Computing Gradients

- Some calculus: **derivatives**

- **Partial derivatives** for multi-variable functions

  – e.g. for $f(x, y) = xy$ we have $\frac{\partial f}{\partial x} = y$ and $\frac{\partial f}{\partial y} = x$

  – for $f(x, y) = x + y$ we have $\frac{\partial f}{\partial x} = 1$ and $\frac{\partial f}{\partial y} = 1$

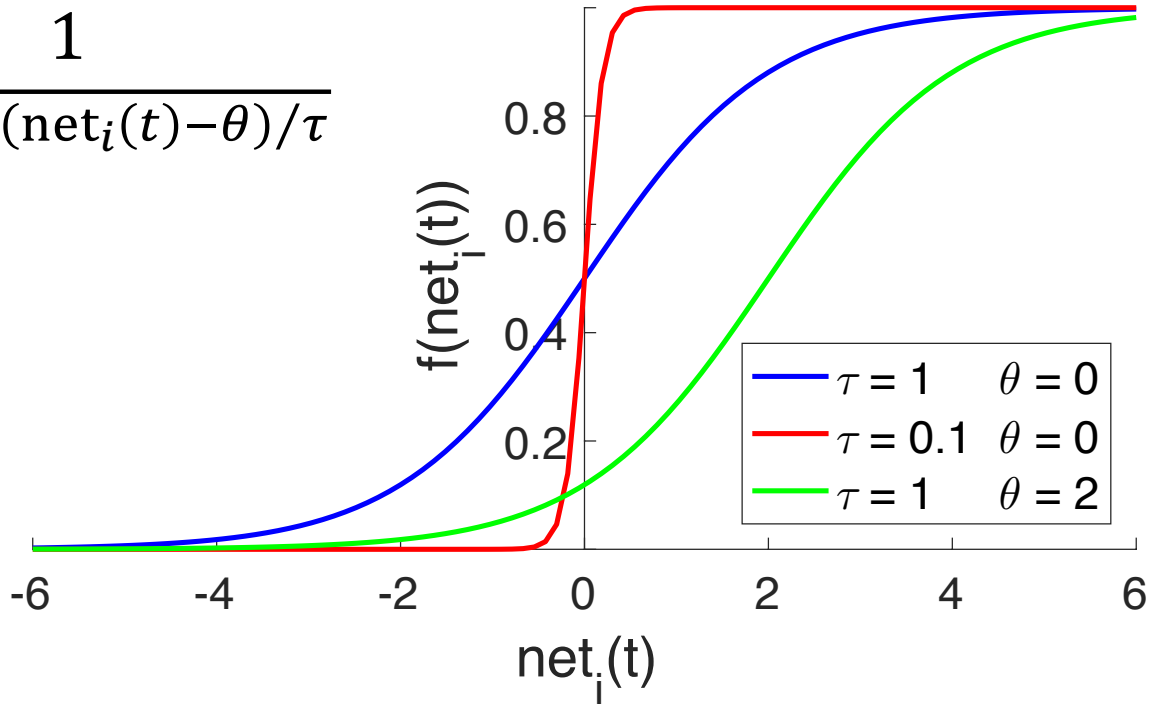
# Computing Gradients

- Derivatives for compound functions
  - e.g. $f(x, y, \mathrm{z}) = (x + y)z$
  - we break down the expression into $q = x + y$ and $f = qz$
  - we have $\frac{\partial f}{\partial q} = z$, $\frac{\partial f}{\partial z} = q$, $\frac{\partial q}{\partial x} = 1$, and $\frac{\partial q}{\partial y} = 1$
  - but we want $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ !
  - **chain rule**: $\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z$ and $\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z$

# Backpropagation for One Neuron

**Sigmoidal neuron**:

$$f_i\left(\text{net}_i(t)\right) = \frac{1}{1 + e^{-(\text{net}_i(t) - \theta)/\tau}}$$




In backpropagation networks, we typically choose $\tau = 1$ and $\theta = 0$.

# Backpropagation for One Neuron

- This leads to the simplified sigmoid function (dropping indices $i$ and $t$ for convenience)

$$f(\text{net}) = \frac{1}{1 + e^{-\text{net}}}$$

which gives

$$f'(\text{net}) = \frac{df(\text{net})}{d\text{net}} = \frac{e^{-\text{net}}}{(1 + e^{-\text{net}})^2}$$

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

$$f'(\text{net}) = f(\text{net})\big(1 - f(\text{net})\big)$$

# Backpropagation for One Neuron

- We now have the derivative of the sigmoid activation function, but what we want are the **gradients of the error function on the individual weights**.

- Assume the error function is $E = \frac{1}{2}(o - d)^2$

- Let's call the sigmoid function $f$, then we have

$$o = f\left(\sum_{j=0}^{n} w_j x_j\right)$$

- We can break up this expression for $E$ in parts:

$E = \frac{1}{2}(o - d)^2$ with $o = f(q)$ and $q = \sum_{j=0}^{n} w_j x_j$

# Backpropagation for One Neuron

- Then we find the (partial) derivatives of the parts

$$E = \frac{1}{2}(o-d)^2 \text{ with } o = f(q) \text{ and } q = \sum_{j=0}^{n} w_j x_j$$

- This gives us

$$\frac{\partial E}{\partial o} = o - d,\ \frac{\partial o}{\partial q} = f(q)\big(1 - f(q)\big),\ \text{and } \frac{\partial q}{\partial w_i} = x_i$$

- Then, by the chain rule:

$$\frac{\partial E}{\partial w_i} = \frac{\partial E}{\partial o} \cdot \frac{\partial o}{\partial q} \cdot \frac{\partial q}{\partial w_i} = (o-d)\big(o(1-o)\big)x_i$$

$$\text{where o} = \frac{1}{1+e^{-w \cdot x}} \text{ and } w \cdot x = \sum_{j=0}^{n} w_j x_j$$

# Backpropagation for One Neuron

- So we know the **gradient of the error on the weights**

$$\frac{\partial E}{\partial w_i} = (o - d)\big(o(1 - o)\big)x_i$$

where $o = \frac{1}{1+e^{-w \cdot x}}$ and $w \cdot x = \sum_{j=0}^{n} w_j x_j$

- Then, we want to update the weights **against the gradient** to find the minimum of the error function:

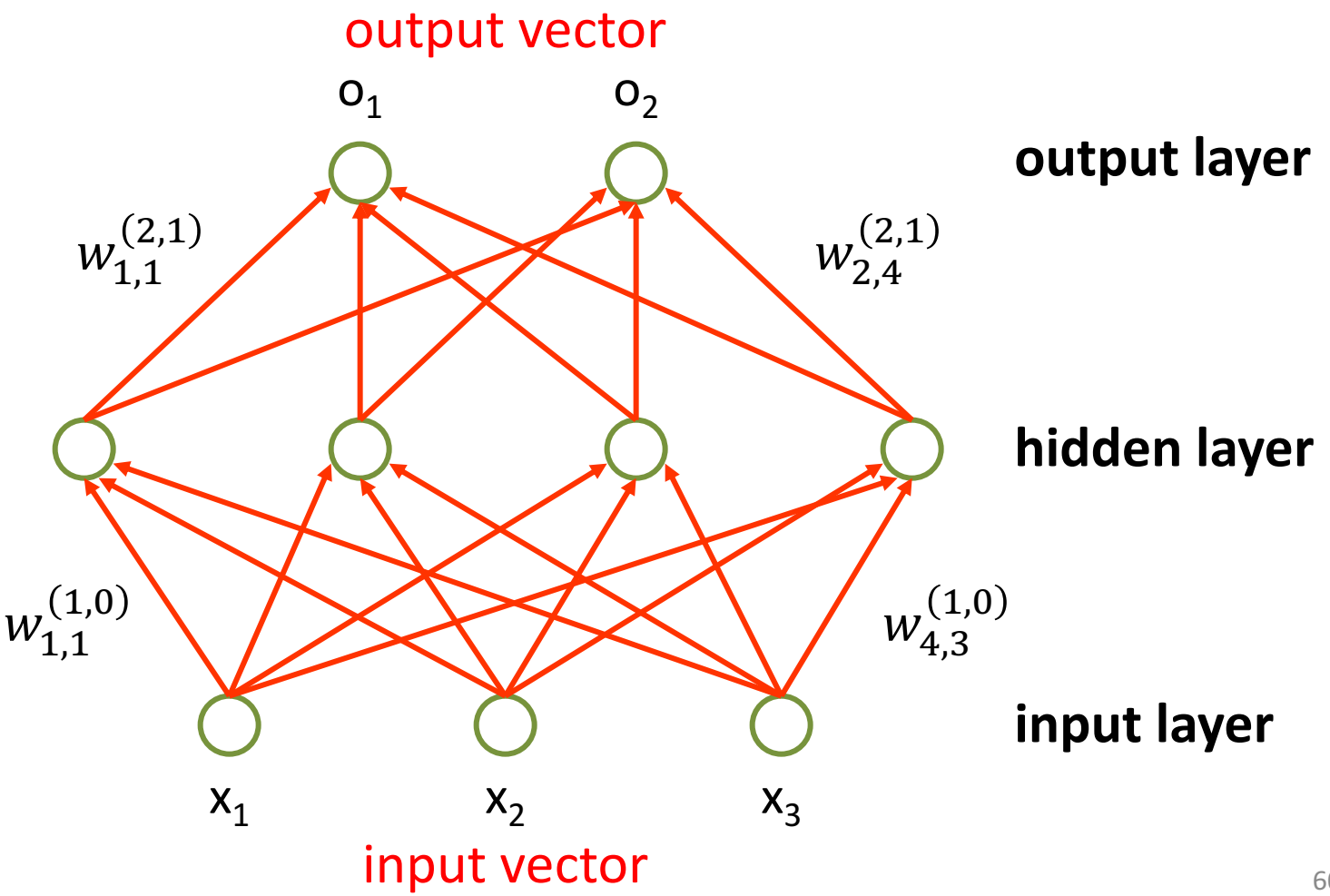
$$\Delta w_j = -\eta(o - d)\big(o(1 - o)\big)x_i$$

step size

- And finally $w_j \leftarrow w_j + \Delta w_j$ for all $j = 0, \ldots, n$

# Terminology

- **Remember**: network function $f:\mathbb{R}^3 \rightarrow \mathbb{R}^2$



# Backpropagation on Multiple Layers

- For nodes in the **output layer**, not much changes
  - inputs are now $(x_0, x_1, \dots x_n)^\intercal = \left(x_0, o_1^{(2,1)}, \dots, o_m^{(2,1)}\right)^\intercal$ which are the outputs of the $m$ nodes in the hidden layer
  - the gradients remain the same

- For nodes in the **hidden layer**, things are a bit more complicated
  - The gradient of the error on the output $\frac{\partial E}{\partial o}$ is now defined as a function over the gradients of all $k$ nodes in the output layer:

$$\frac{\partial E\left(o_i^{(2,1)}\right)}{\partial o_i^{(2,1)}} = \frac{\partial E\left(q_1^{(2)}, \dots, q_k^{(2)}\right)}{\partial o_i^{(2,1)}}$$

# Backpropagation on Multiple Layers

- The gradient of the error on the **output of the hidden layer** is defined as a function over the gradients of all $k$ nodes in the **output layer**:

$$\frac{\partial E\left(o_i^{(2,1)}\right)}{\partial o_i^{(2,1)}} = \frac{\partial E\left(q_1^{(2)}, \ldots, q_k^{(2)}\right)}{\partial o_i^{(2,1)}}$$

- We can write the total derivative recursively as

$$\frac{\partial E\left(o_i^{(2,1)}\right)}{\partial o_i^{(2,1)}} = \sum_{j=1}^{k}\left(\frac{\partial E}{\partial q_j^{(2)}} \cdot \frac{\partial q_j^{(2)}}{\partial o_i^{(2,1)}}\right) = \sum_{j=1}^{k}\left(\frac{\partial E}{\partial o_j^{(2)}} \cdot \frac{\partial o_j^{(2)}}{\partial q_j^{(2)}} \cdot w_{j,i}^{(2,1)}\right)$$

# Backpropagation on Multiple Layers

- Sticking it all together, we have the following general expression for the gradients on the weights of any node $i$:

$$\frac{\partial E}{\partial w_{i,j}} = \delta_i x_j$$

$$\delta_i = \begin{cases} (o_i - d_i)\big(o_i(1-o_i)\big) & \text{if } i \text{ is an output node} \\ \left(\sum_l \delta_l \, w_{l,i}^{(2,1)}\right)\big(o_i(1-o_i)\big) & \text{if } i \text{ is a hidden node} \end{cases}$$

- And finally $\Delta w_{i,j} = -\eta \delta_i x_j$ for all $i, j$.

# Backpropagation Algorithm

**ALGORITHM** Backpropagation

Start with randomly chosen weights

**WHILE** MSE is above desired threshold and computational bounds are not exceeded, **DO**

**FOR EACH** input pattern $x_p$, $1 \leq p \leq P$,

Compute hidden node inputs

Compute hidden node outputs

Compute inputs to the output nodes

Compute the network outputs

Compute the error between output and desired output

Modify the weights between hidden and output nodes

Modify the weights between input and hidden nodes

**END-FOR**

**END-WHILE**