

# Lecture: Machine Learning for Data Science

Winter semester 2021/22

Lectures 8: Classification (Neural networks - Perceptron)

Prof. Dr. Eirini Ntoutsi

# Outline

- Neural Networks: Basic intuition and basic notions
- Perceptron
- Beyond simple perceptron
- Things you should know from this lecture & reading material

# Biological inspiration

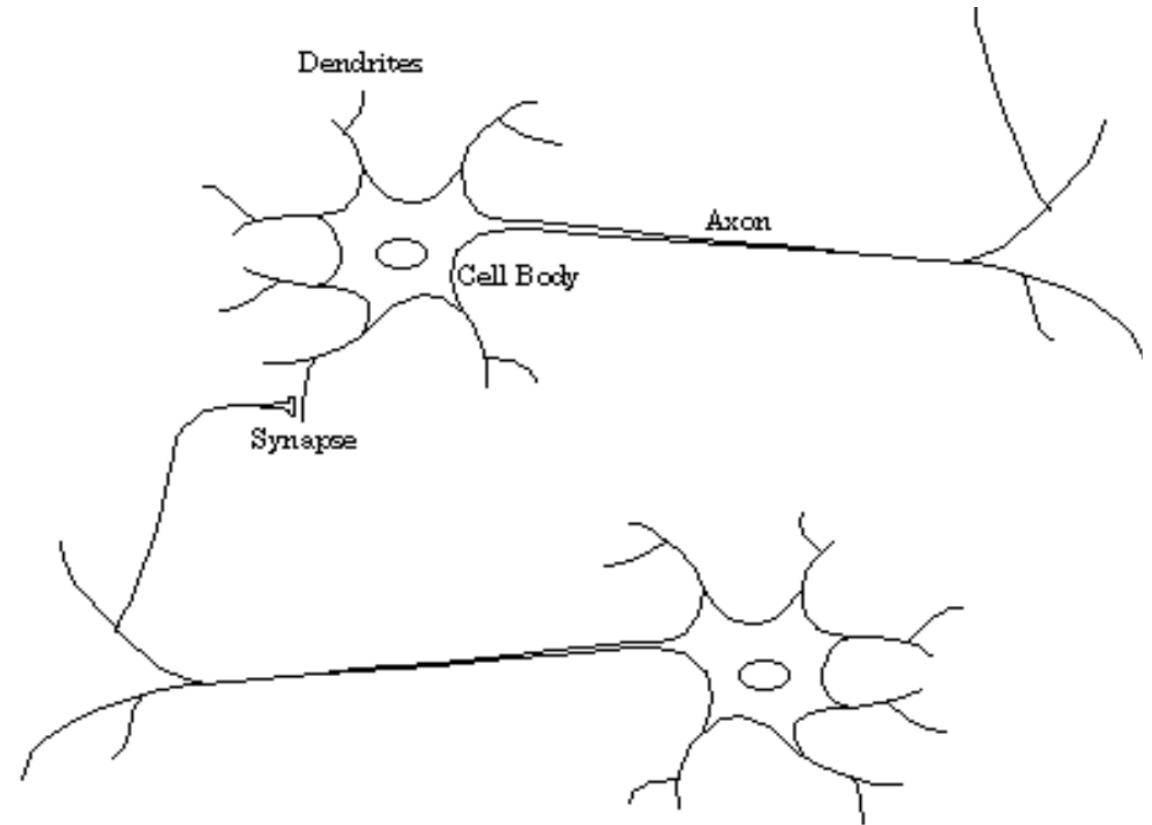
- Humans perform complex tasks like vision, motor control, or language understanding very well.
- One way to build intelligent machines is to try to **imitate the (organizational principles of) human brain**.

# Human brain

- The brain is a highly complex, non-linear, and parallel computer, composed of some  $10^{11}$  neurons that are densely connected ( $\sim 10^4$  connection per neuron). We have just begun to understand how the brain works...
- A neuron is much slower ( $10^{-3}$ sec) compared to a silicon logic gate ( $10^{-9}$ sec), however the massive interconnection between neurons make up for the comparably slow rate.
  - Complex perceptual decisions are arrived at quickly (within a few hundred milliseconds)
- **100-Steps rule**: Since individual neurons operate in a few milliseconds, calculations do not involve more than about 100 serial steps and the information sent from one neuron to another is very small (a few bits)
- **Plasticity**: Some of the neural structure of the brain is present at birth, while other parts are developed through learning, especially in early stages of life, to adapt to the environment (new inputs).

# Biological neuron

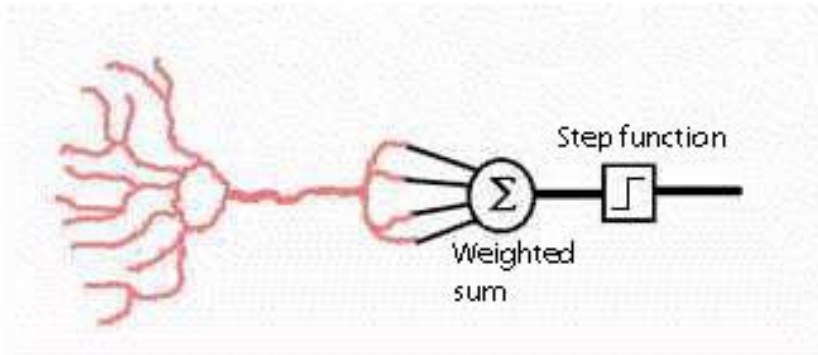
- **dendrites**: nerve fibres carrying electrical signals to the cell
- **cell body (soma)**: computes a non-linear function of its inputs
- **axon**: single long fiber that carries the electrical signal from the cell body to other neurons
- **synapse**: the point of contact between the axon of one cell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



A biological neuron

# Artificial neuron (perceptron)

- The perceptron is a mathematical model of a biological neuron.



Source: [link](#)

An artificial neuron (perceptron)

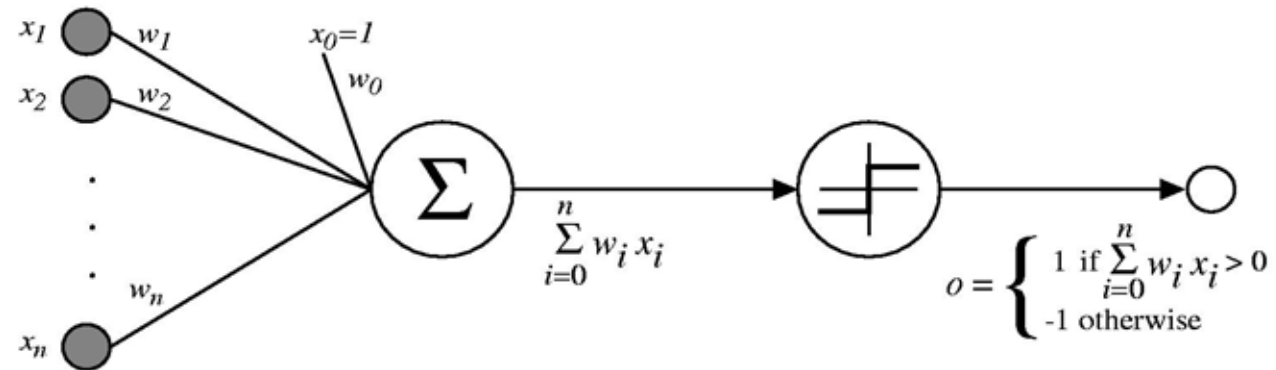
- While in actual neurons the dendrite receives electrical signals from the axons of other neurons, in the perceptron these **electrical signals are represented as numerical values**.
- At the synapses between the dendrite and axons, electrical signals are modulated in various amounts. This is also modeled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

# Outline

- Neural Networks: Basic intuition and basic notions
- Perceptron
- Beyond simple perceptron
- Things you should know from this lecture & reading material

# The perceptron classifier

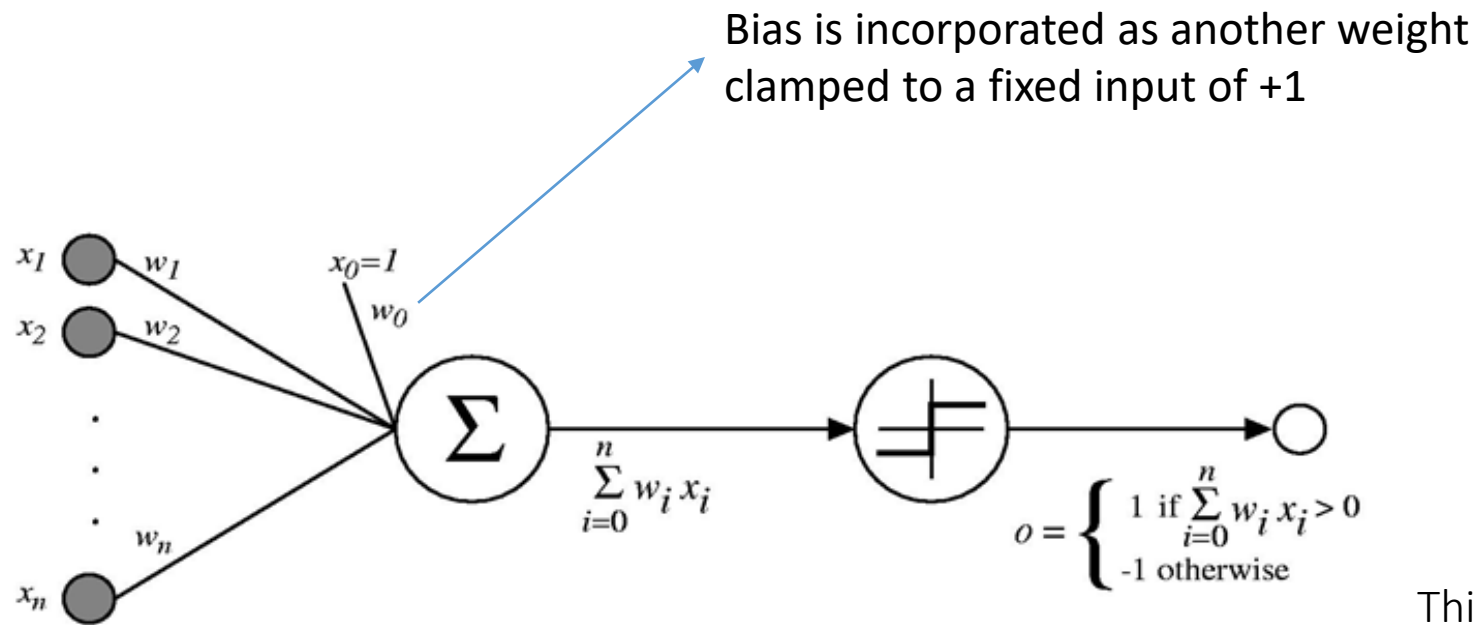
- Loosely motivated by a simple model of how neurons fire
  - First model proposed by McCulloch and Pitts Model in 1943
- A perceptron



- takes a vector of real-valued inputs (the so-called, **network input**)
- calculates a **linear combination** of these inputs
- Adds the **bias** term
- Passes this value through a **step function** (an example of an **activation function**)
- The neuron **fires** (i.e., becomes active) if its output is above 0



# The perceptron classifier



This is the equation of a **linear classifier**

- Line in 2D
- A hyperplane in general

$$o(x_1, \dots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \dots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$



$$o(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w} \cdot \vec{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

# The perceptron learning rule

- The perceptron algorithm is about learning the weights for the input signals in order to draw a linear decision boundary that allows us to discriminate between the two linearly separable classes
- **Perceptron learning rule** (Frank Rosenblatt, 1958)

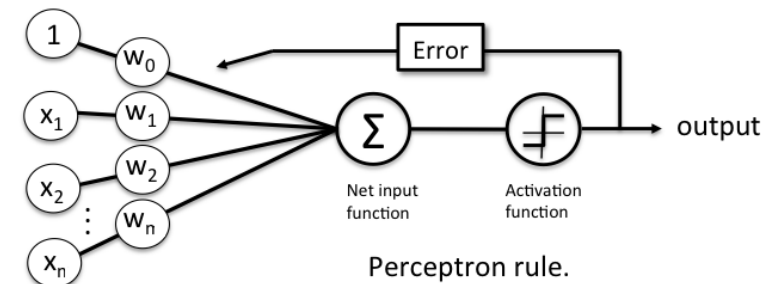
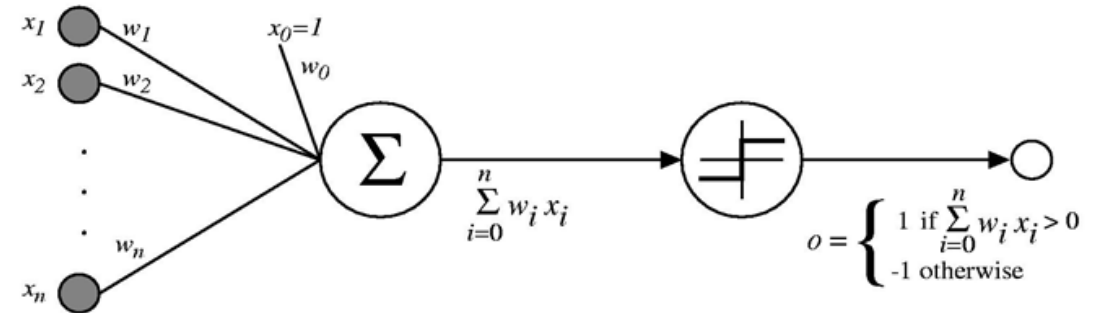
- Initialize the weights to 0 or, some small values
- For each training example  $x_i$ 
  - Calculate the output  $o_i$
  - Update the weights

- Weight update formula:  $w_j = w_j + \Delta w_j$

where  $\Delta w_j = \eta(y^i - o^i)x_j^i$

$\eta$  is the learning rate (a constant in [0-1])

- In case of correct predictions, weights do not change
- It can be proven that the perceptron converges if the classes are linearly separable



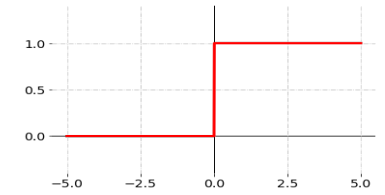
**0/1 loss:**  $L_{0/1}(h(x_i), y_i) = 0$ , if  $h(x_i) = y_i$ ; 1 otherwise

# Problems with perceptron

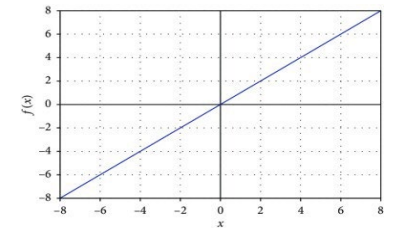
- Although the perceptron learning rule converges if the two classes can be separated by linear hyperplane, problems arise if the classes cannot be separated perfectly by a linear classifier.
- **Delta-rule update of the weights**
  - Replace the step function with a **linear activation function**
  - The linear function is continuous and therefore differentiable
- This allows us to define a cost function that we can minimize in order to update the weights - **sum of squared errors (SSE)** or **squared-loss** or **L2 loss**

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

- $D$ : the training set
- $t_d$ : the real class value of the instance  $d$  (target value)
- $o_d$ : perceptron's output
- $\frac{1}{2}$  is used for convenience



Step function  
(threshold=0)

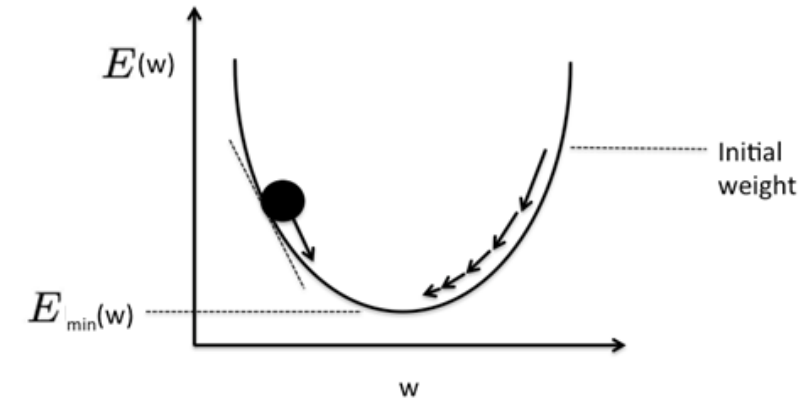


Linear function

# Gradient descent

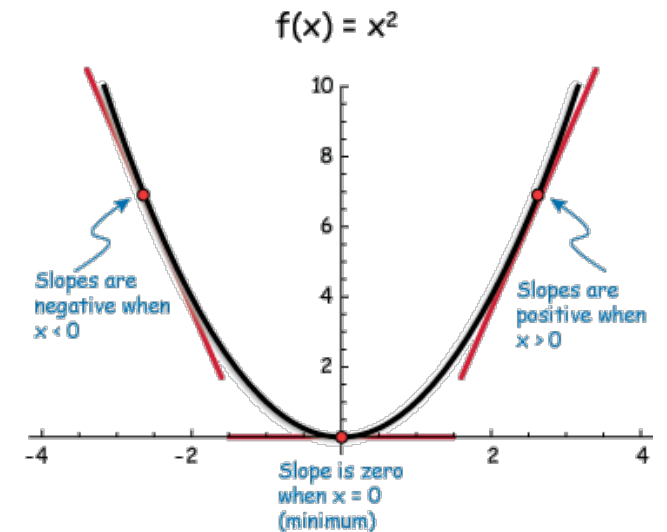
- In order to minimize the SSE cost function, we will use gradient descent
- Gradient descent idea:
  - Start with an arbitrary initial weight vector
  - Repeatedly modify it in small steps
  - At each step, the weight vector is altered in the direction that produces the steepest descent along the error surface.
- Schematic of gradient descent with a single weight  $w$ 
  - At each step, we take a step into the opposite direction of the gradient, and the step size is determined by the value of the **learning rate  $\eta$**  as well as the **slope of the gradient**.

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$



# Back to basics: First derivative

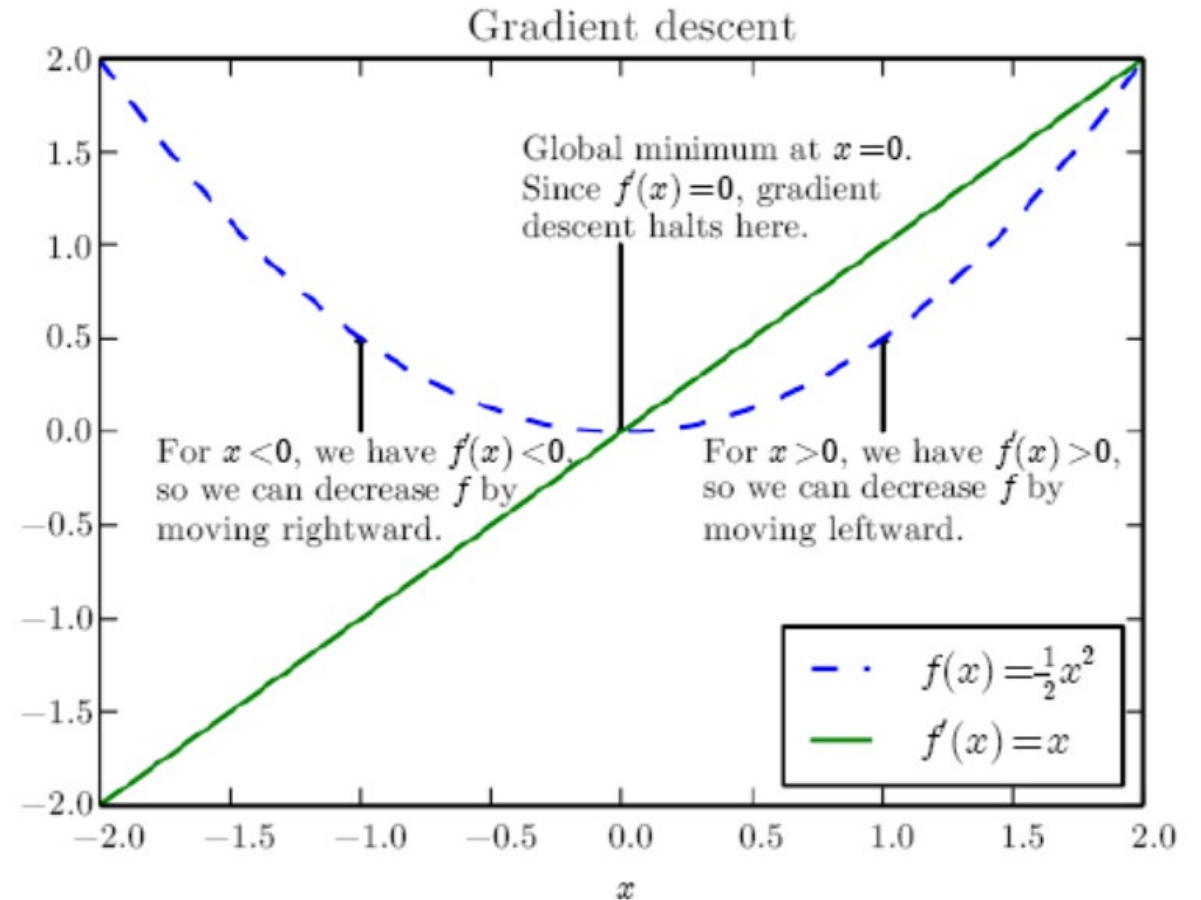
- The first derivative of a function  $f(x)$ , which we write as  $f'(x)$  or as  $df/dx$ , is the slope of the tangent line to the function at the point  $x$ .
- It specifies how to scale a small change in the input to obtain the corresponding change in the output:
  - $f(x+\epsilon) \approx f(x) + \epsilon * f'(x)$ 
    - $x$ : initial value of the input
    - $x+\epsilon$ : move the initial value by  $\epsilon$
- The derivative helps us to understand how a small change in the input  $x$  will affect the output  $y$  and therefore is useful for minimizing a cost function.



<https://qph.ec.quoracdn.net/main-qimg-30fd53287fe2f10d6f5d99492ffabf31>

# Back to basics: First derivative

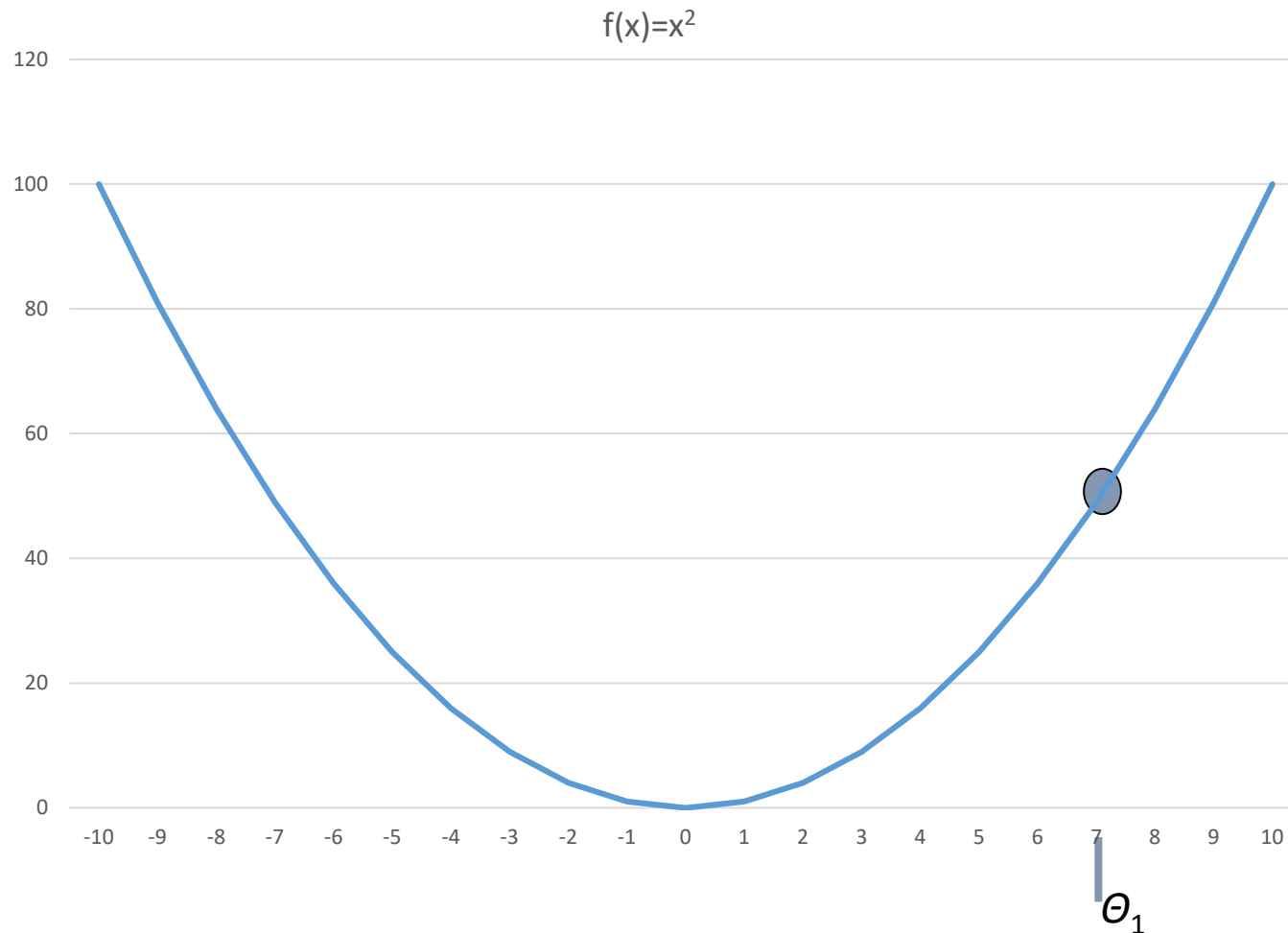
- the first derivative tells us how whether a function is increasing or decreasing, and by how much it is increasing or decreasing
- Positive slope:  $f'(x) > 0 \rightarrow$  the function  $f(x)$  is increasing
  - as  $x$  increases,  $f(x)$  also increases
- Negative slope:  $f'(x) < 0 \rightarrow$  the function  $f(x)$  is decreasing
  - as  $x$  increases,  $f(x)$  decreases.
- We can thus use the derivative to decide on how to move.
- This techniques is called gradient descent (Cauchy, 1847)



Source: Deep learning book, Goodfellow, Bengio and Courville,  
Chapter 4

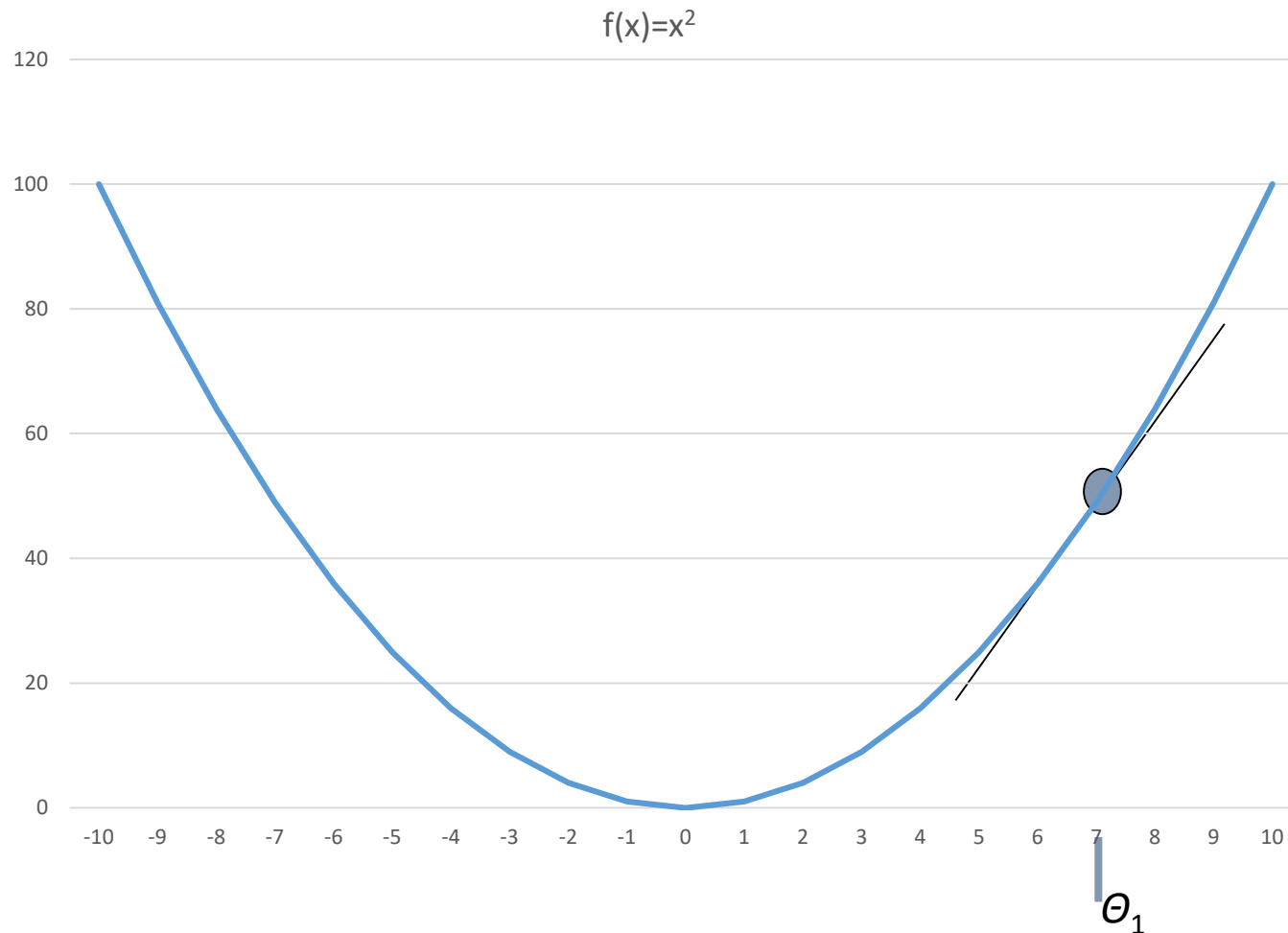
# A simple example with 1 variable

- Let  $f(x)=x^2$  and let assume we start at  $\Theta_1$



# A simple example with 1 variable

- Let  $f(x)=x^2$  and let assume we start at  $\Theta_1$

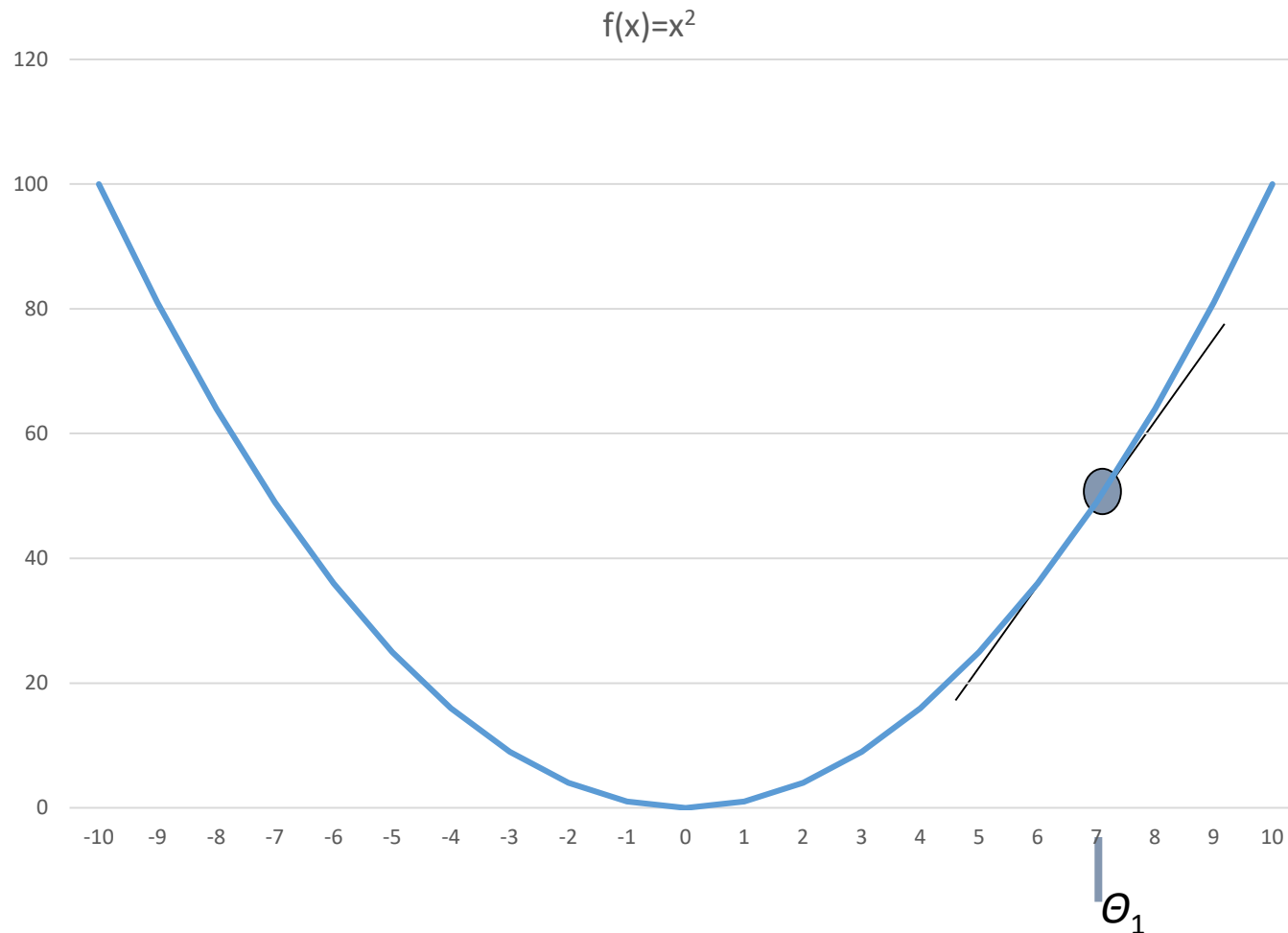


What is the slope of the line that is tangent to the function?



# A simple example with 1 variable

- Let  $f(x)=x^2$  and let assume we start at  $\Theta_1$



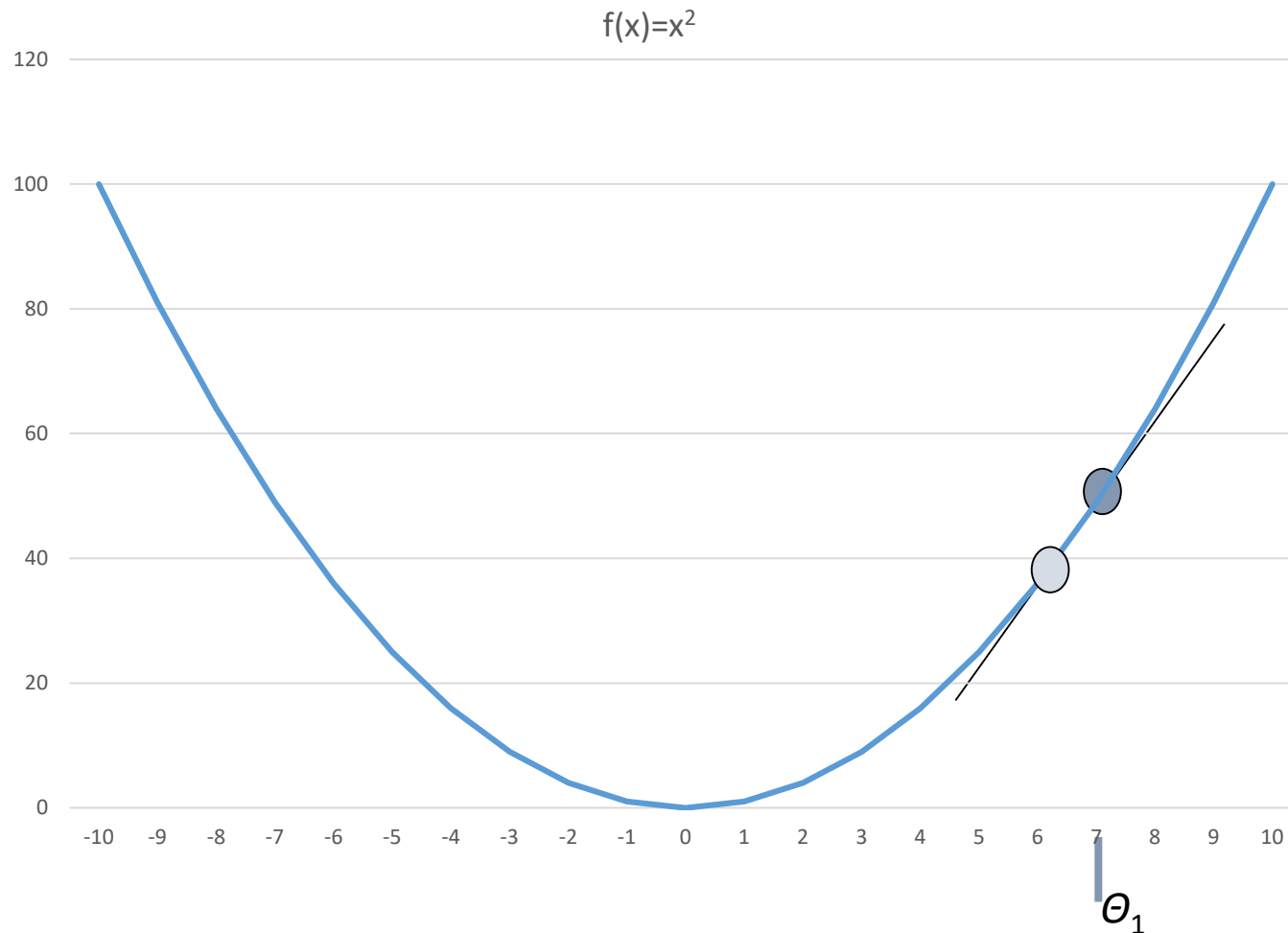
What is the slope of the line that is tangent to the function?

In this case the derivative is positive.

→ Move  $\Theta_1$  to the left

# A simple example with 1 variable

- Let  $f(x)=x^2$  and let assume we start at  $\Theta_1$



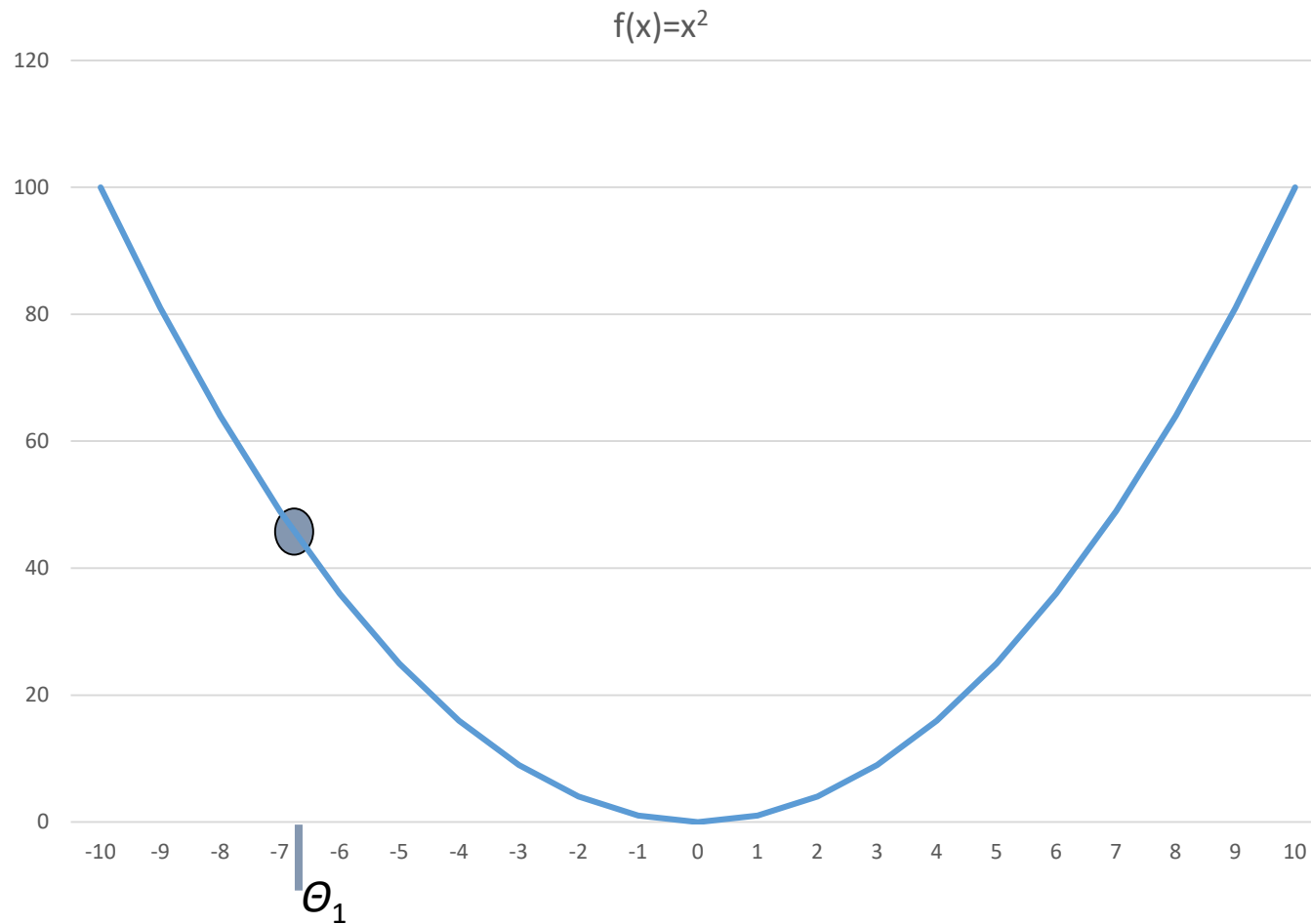
What is the slope of the line that is tangent to the function?

In this case the derivative is positive.

→ Move  $\Theta_1$  to the left

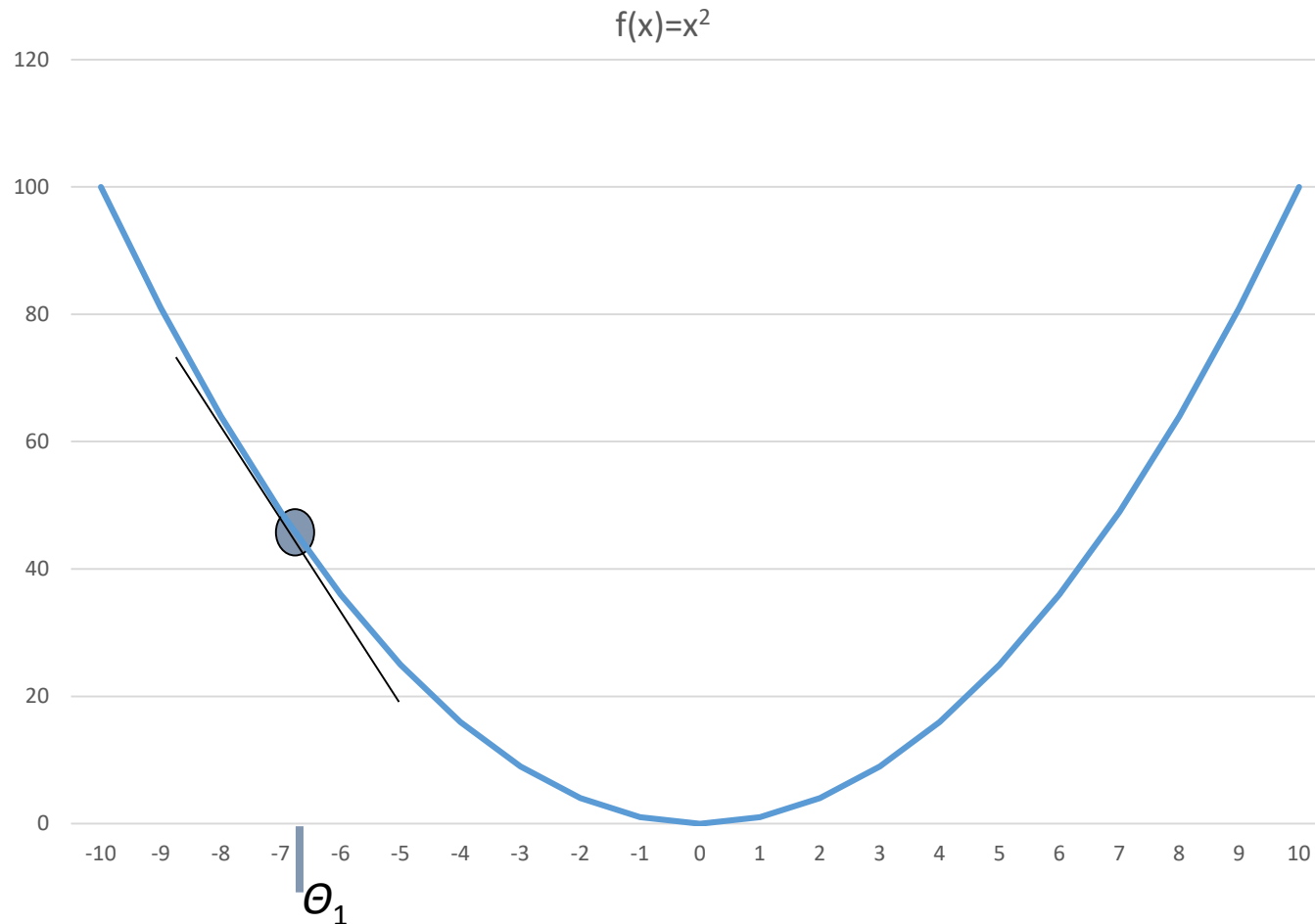
# A simple example with 1 variable

- Let us assume a different starting point  $\Theta_1$



# A simple example with 1 variable

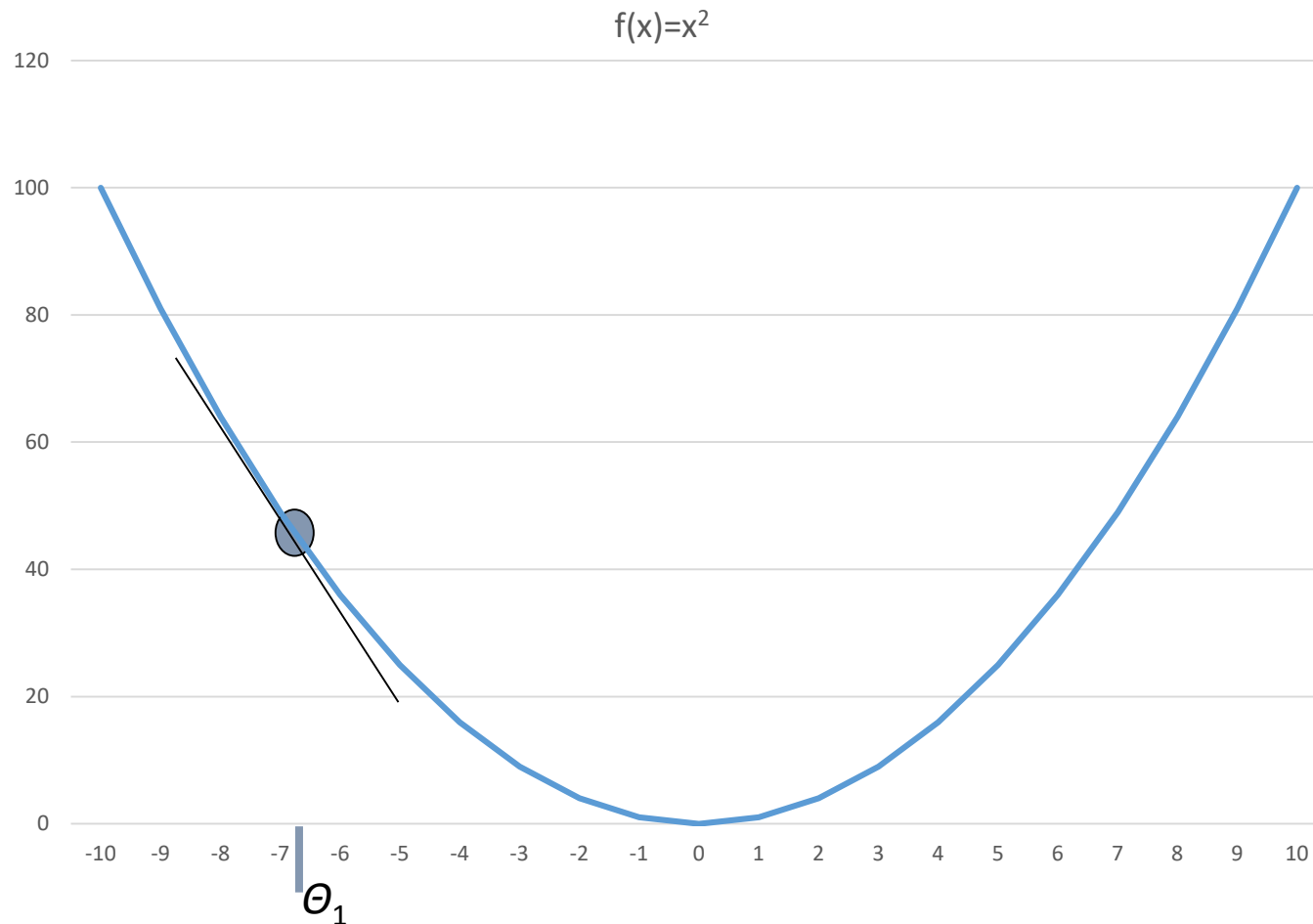
- Let us assume a different starting point  $\Theta_1$



What is the slope of the line that is tangent to the function?

# A simple example with 1 variable

- Let us assume a different starting point  $\Theta_1$



What is the slope of the line that is tangent to the function?

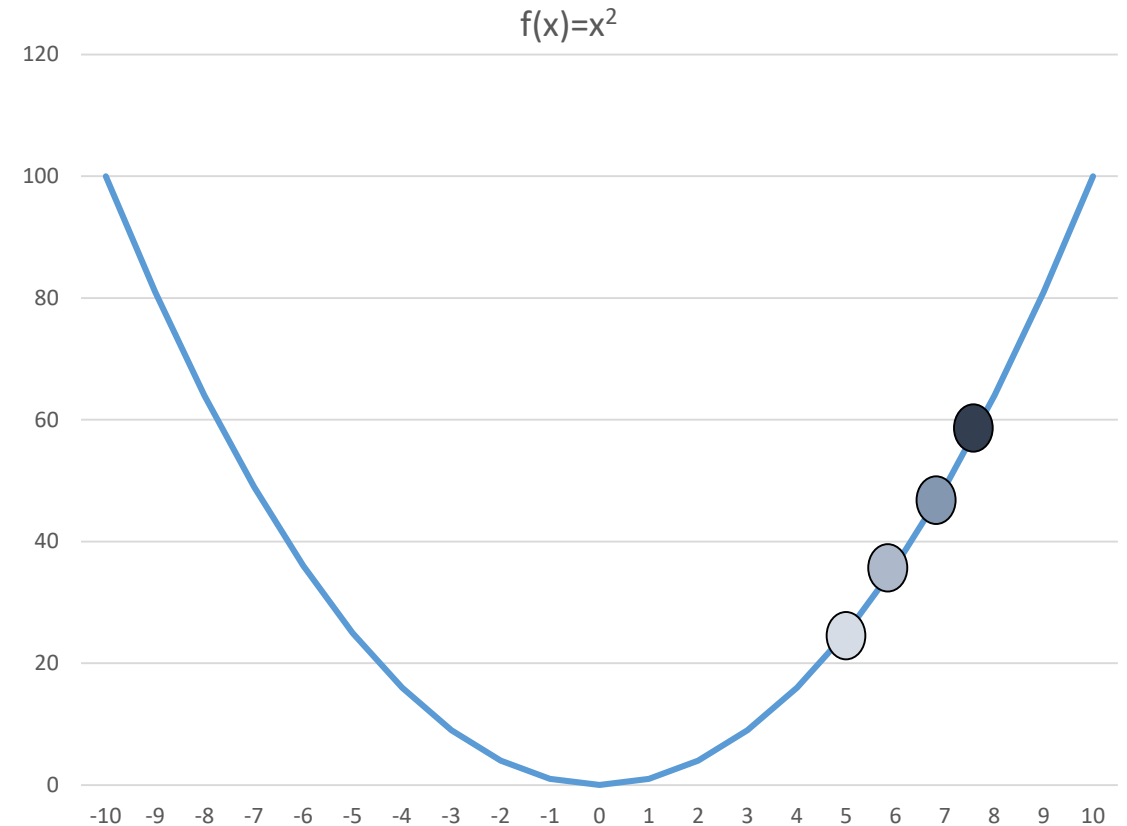
In this case the derivative is negative.

→ Move  $\Theta_1$  to the right

# The role of the learning rate

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

- If  $\eta$  is too small, gradient descent can be slow.
  - too many steps are needed.
- If  $\epsilon$  is too large, gradient descent can overshoot the minimum and it may fail to converge.



# Derivation of the gradient descent learning rule

- Training error:  $E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$

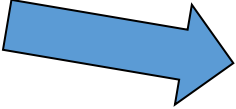
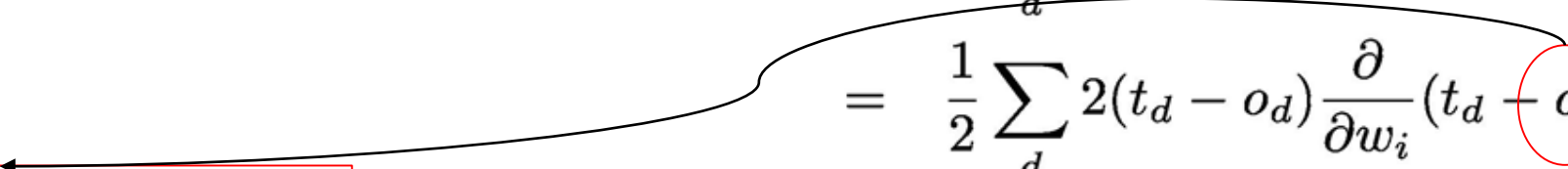
- Gradient of  $E$  w.r.t.  $w$ :  $\nabla E[\vec{w}] \equiv \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \dots, \frac{\partial E}{\partial w_n} \right]$

- Training rule:  $\vec{w} \leftarrow \vec{w} + \Delta \vec{w}$

- where
  - $\eta$ : the learning rate
- $$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$

- Each component  $w_i$  of the weight vector is altered in proportion to:  $\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$

## How to calculate the gradient at each step?

$$E[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_d (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_d 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\ &= \sum_d (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \end{aligned}$$


$o = w_0 + w_1 x_1 + \dots + w_n x_n$

$$\frac{\partial E}{\partial w_i} = \sum_d (t_d - o_d)(-x_{i,d})$$

- where  $x_{id}$ : the single input attribute value  $x_i$  for training example  $d$



# Perceptron training rule

$$w_i \leftarrow w_i + \Delta w_i$$

- where

$$\Delta \vec{w} = -\eta \nabla E[\vec{w}] \quad \Rightarrow \quad \Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id} \quad \Rightarrow \quad \Delta w_i = \eta (t - o) x_i$$

- $\eta$ : a small constant (e.g., 0.1) called the learning rate
- $t$ : the real class value of the training instance (target value)
- $o$ : perceptron's output for the instance
- $x_i$ : value of training instance for feature  $i$

## (batch) Gradient descent training

GRADIENT-DESCENT(*training\_examples*,  $\eta$ )

Initialize each  $w_i$  to some small random value

Until the termination condition is met, Do

- Initialize each  $\Delta w_i$  to zero.
- For each  $\langle \vec{x}, t \rangle$  in *training\_examples*, Do
  - Input instance  $\vec{x}$  to unit and compute output  $o$
  - For each linear unit weight  $w_i$ , Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

- For each linear unit weight  $w_i$ , Do

$$w_i \leftarrow w_i + \Delta w_i$$

# Incremental/Stochastic gradient descent

- In **batch gradient descent**, weight updates are computed after summing over all training examples  $D$
- **Stochastic gradient descent**: approximate the gradient descent search by updating weights incrementally, following the calculation of the error for each individual example

GRADIENT-DESCENT(*training\_examples*,  $\eta$ )

Initialize each  $w_i$  to some small random value

Until the termination condition is met, Do

- Initialize each  $\Delta w_i$  to zero.
- For each  $\langle \vec{x}, t \rangle$  in *training\_examples*, Do
  - Input instance  $\vec{x}$  to unit and compute output  $o$
  - For each linear unit weight  $w_i$ , Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$



$$w_i \leftarrow w_i + \eta(t - o) x_i$$

We update the weights for each training example according to the gradient w.r.t.  $E_d(\vec{w})$



- ~~For each linear unit weight  $w_i$ , Do~~

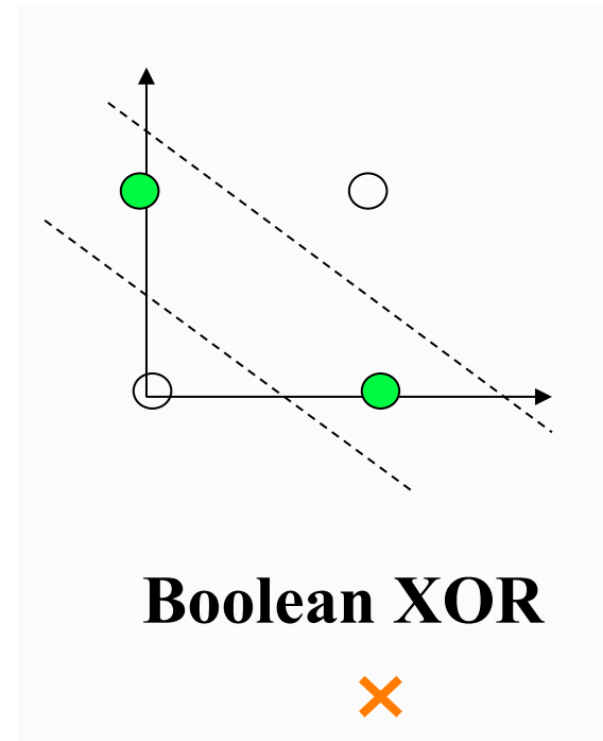
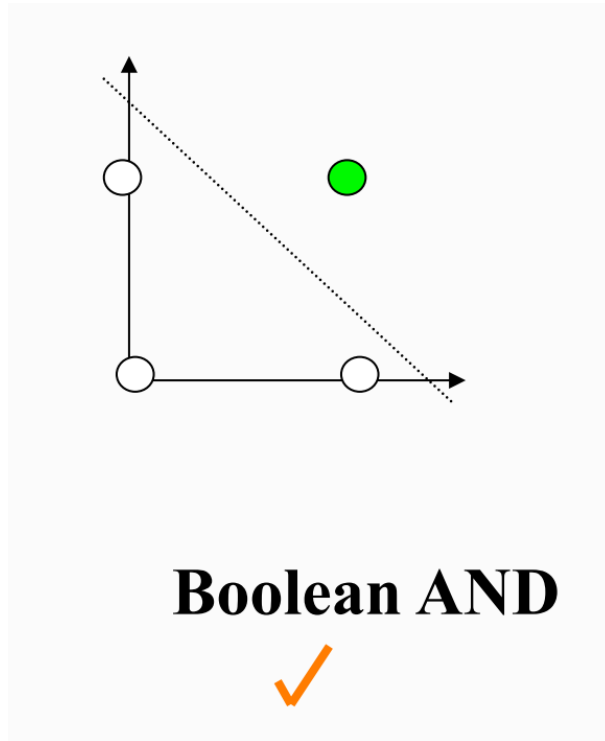
$$~~w_i \leftarrow w_i + \Delta w_i~~$$

# Outline

- Neural Networks: Basic intuition and basic notions
- Perceptron
- Beyond simple perceptron
- Things you should know from this lecture & reading material

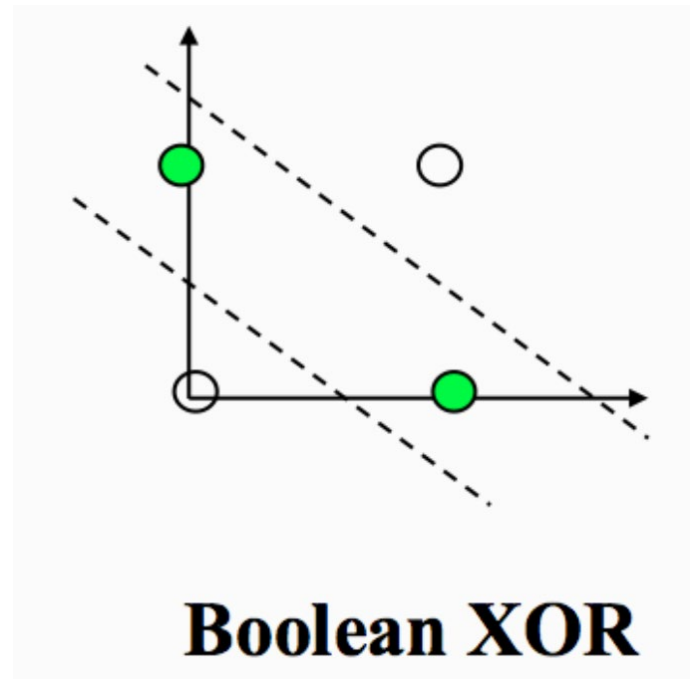
# Perceptron limitations

- A single layer perceptron can only learn linearly separable problems.
- Boolean AND function is linearly separable, whereas Boolean XOR function is not.



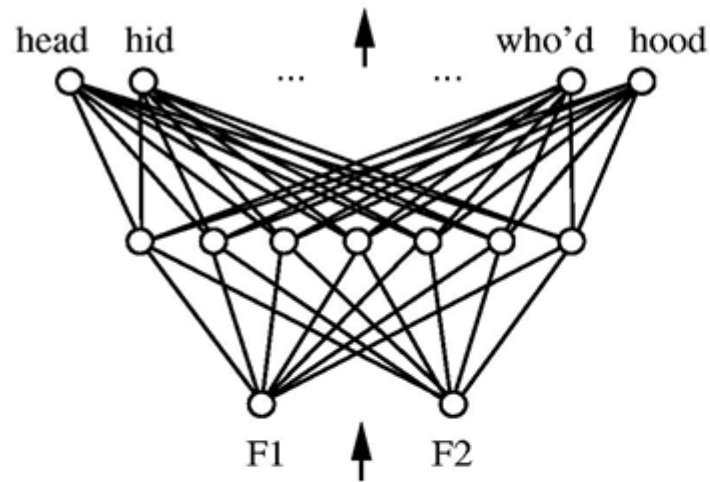
# Perceptron limitations

- For non-linearly separable problems
  - ▣ Add more layers → multi-layer networks

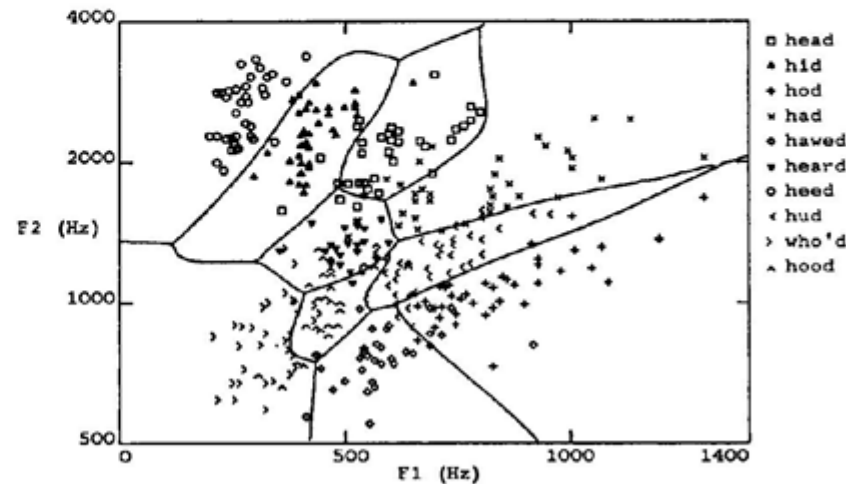


# From single perceptron's to multilayer networks

- Single perceptron models can represent only linear decision surfaces
- For more complex functions, we need different networks
- Multi-layer networks represent highly non-linear decision surfaces

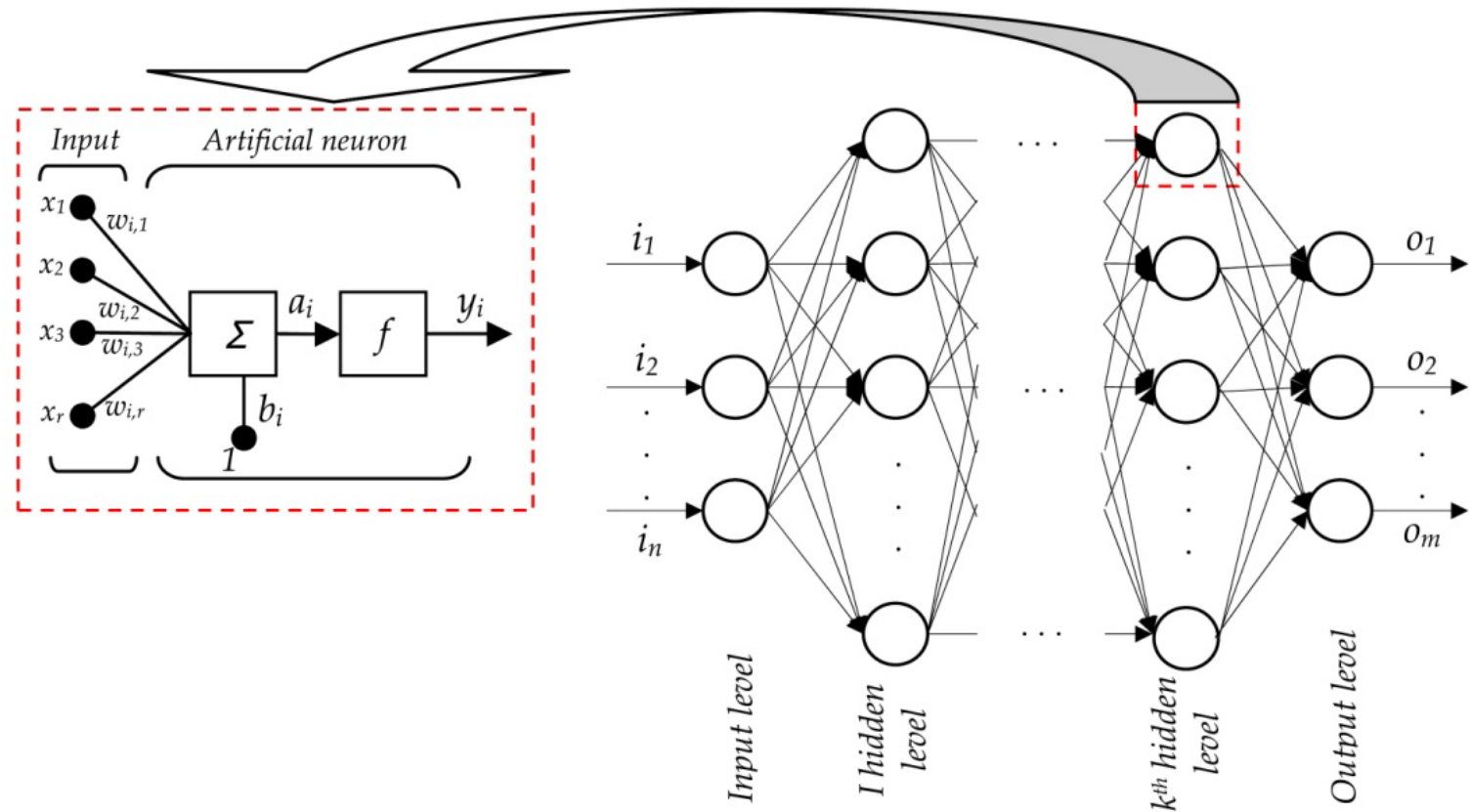


An example of a multilayer network (for speech recognition) and its decision surface



# Multilayer networks

- Input layer
- Hidden layers
- Output layer

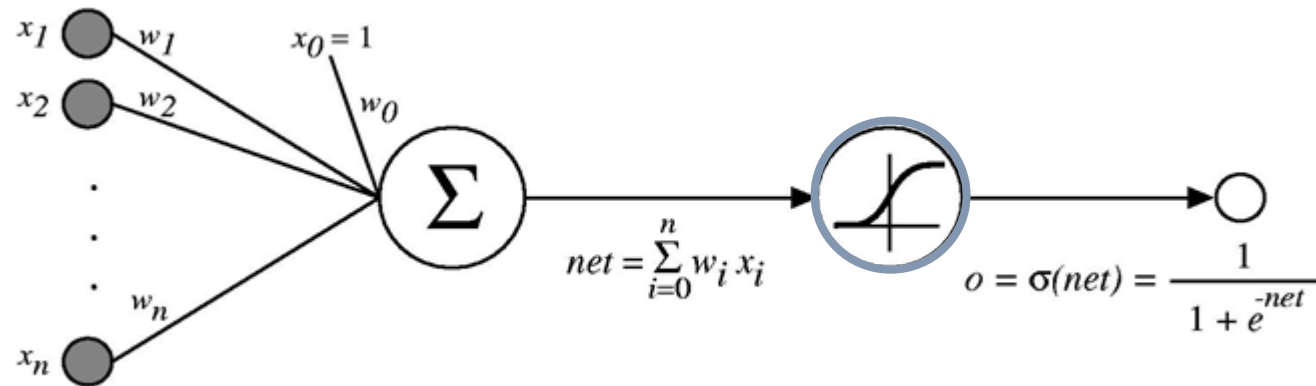


Source: <https://www.analyticsvidhya.com/blog/2016/03/introduction-deep-learning-fundamentals-neural-networks/>



# Sigmoid unit

- Very much like the perceptron unit but based on a smoothed, differentiable threshold function



- The threshold output is a continuous function of its input

□  $\sigma()$ : is the sigmoid function

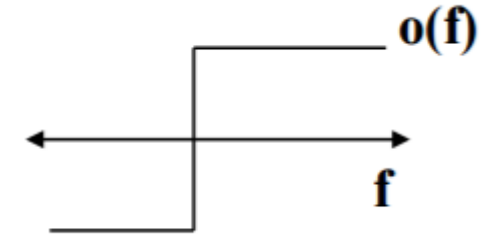
$$o = \sigma(\vec{w} \cdot \vec{x})$$

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

# Step function vs sigmoid function

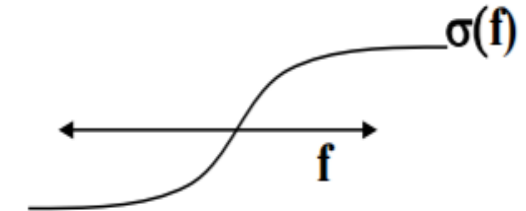
- Step function

- Thresholded output: takes values +1 or -1



- Sigmoid function

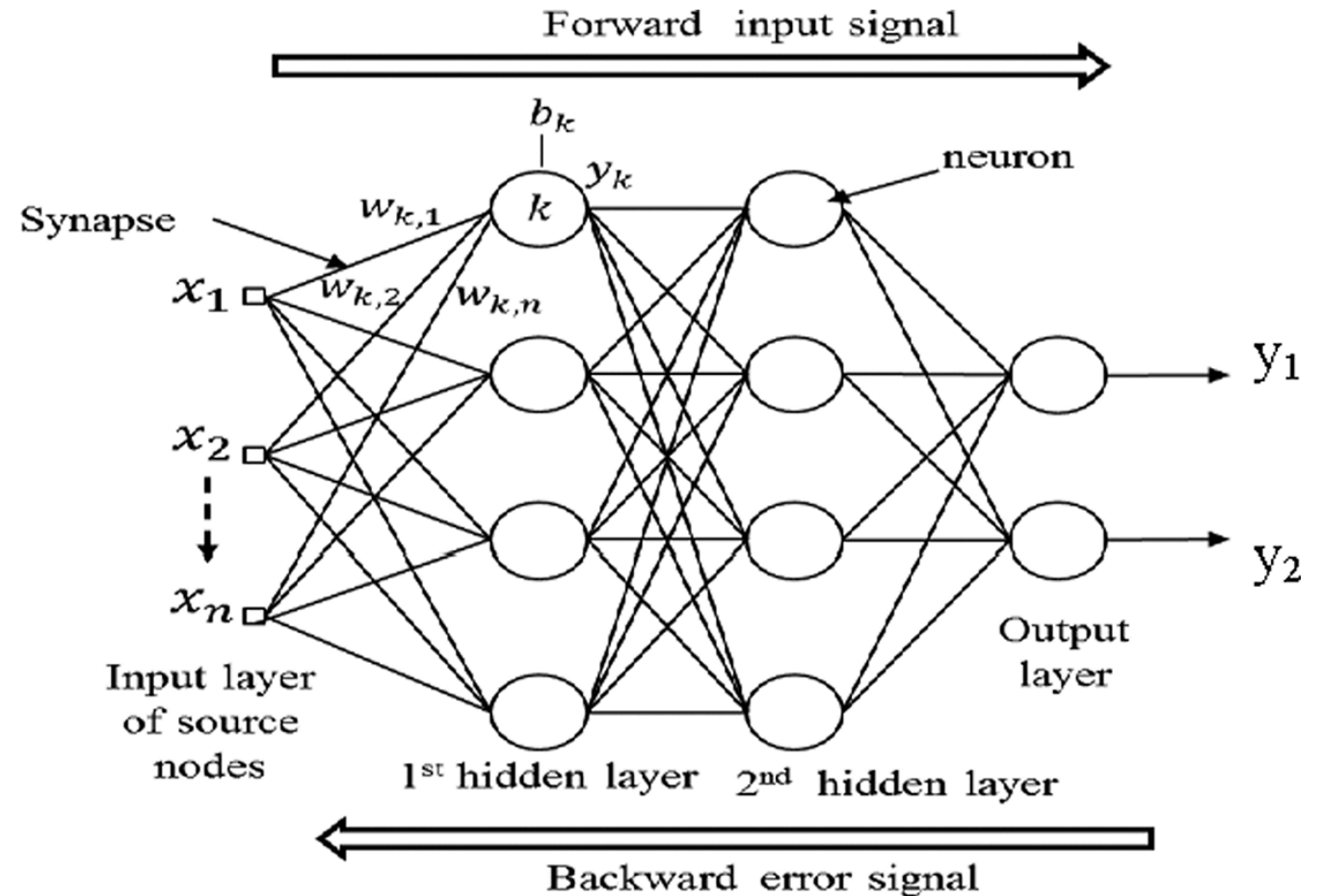
- Sigmoid output: takes real values between -1 and +1
- The sigmoid is in effect an approximation to the threshold function above, but has a gradient that we can use for learning
  - Nice property



$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

# Learning the weights: Backpropagation

- The backpropagation algorithm learns the weights of a multilayer network, given a network with a fixed set of units and interconnections.
- It uses gradient descent in its attempt to minimize the squared error between the network output values and the target values for these outputs.



# Backpropagation algorithm

## Backpropagation Algorithm

Initialize all weights to small random numbers

Until convergence, Do

For each training example, Do

1. Input it to network and compute network outputs

2. For each output unit  $k$

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

3. For each hidden unit  $h$

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k$$

4. Update each network weight  $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

where  $\Delta w_{i,j} = \eta \delta_j x_{i,j}$

$\delta_n$ : the error term associated with unit  $n$ .

Before we had:

$$w_i \leftarrow w_i + \eta(t - o) x_i.$$

So, we replace the  $(t-o)$  error with a more complex definition of error

# Backpropagation algorithm

## Backpropagation Algorithm

Initialize all weights to small random numbers

Until convergence, Do

For each training example, Do

1. Input it to network and compute network outputs

2. For each output unit  $k$

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

The derivative of the  
sigmoid activation  
function

3. For each hidden unit  $h$

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k$$

4. Update each network weight  $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

$$\text{where } \Delta w_{i,j} = \eta \delta_j x_{i,j}$$

# Backpropagation algorithm

## Backpropagation Algorithm

Initialize all weights to small random numbers

Until convergence, Do

For each training example, Do

1. Input it to network and compute network outputs

2. For each output unit  $k$

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

Target values are directly available in the output which indicate the error of an output unit

3. For each hidden unit  $h$

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k$$

No target values are directly available in the hidden units to indicate their error.

Therefore we sum up the (weighted) error terms  $\delta_k$  of the output units influenced by  $h$

4. Update each network weight  $w_{i,j}$

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

The weight characterizes the degree to which  $h$  is responsible for the error in  $k$

$$\text{where } \Delta w_{i,j} = \eta \delta_j x_{i,j}$$

# More on backpropagation

- Gradient descent over entire network weight vector
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Often include weight momentum  $\alpha$  to making the weight update on the  $n^{\text{th}}$  iteration depend partially on the update that occurred during the  $(n-1)$ th iteration:

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Intuitively, this helps to keep the ball rolling in the same direction from one iteration to the next.
- Minimizes error over training examples.
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations → too slow
- Once trained, using the network is very fast.

# Convergence of Backpropagation

- Gradient descent to some local minimum
  - Perhaps not global minimum...
  - Add momentum
  - Stochastic gradient descent
  - Train multiple nets with different initial weights
- Nature of convergence
  - Initialize weights near zero
  - Therefore, initial networks near-linear
  - Increasingly non-linear functions possible as training progresses (weights grow)



# Properties of ANNs

- Learning from examples
  - labeled or unlabeled
- Adaptivity
  - changing the connection strengths to learn things
- Non-linearity
  - the non-linear activation functions are essential
- Fault tolerance
  - if one of the neurons or connections is damaged, the whole network still works quite well
- Thus, they might be better alternatives than classical solutions for problems characterised by:
  - high dimensionality, noisy, imprecise or imperfect data; and
  - a lack of a clearly stated mathematical solution or algorithm

# Outline

- Class of linear classifiers
- Neural Networks - Perceptron
- Neural Networks - Multilayer perceptron
- Things you should know from this lecture & reading material

# Overview and Reading

- Overview

- Simple perceptron
- Multilayer perceptron
- Backpropagation

- Reading

- Machine learning book by T. Mitchell (Chapter 4).
  - How the backpropagation rule is derived: Section 4.5.3
- Artificial Intelligence, A Modern Approach. Stuart Russell and Peter Norvig.
- Deep learning, Goodfellow, Bengio and Courville, Chapter 4.

# Hands on experience

- Implement the backpropagation algorithm on your own
  - ▣ for a simple network (1 input layer, 1 hidden layer, 1 output layer)



# Acknowledgements

- The slides are based on
  - ❑ KDD I lecture at LMU Munich (Johannes Aßfalg, Christian Böhm, Karsten Borgwardt, Martin Ester, Eshref Januzaj, Karin Kailing, Peer Kröger, Eirini Ntoutsi, Jörg Sander, Matthias Schubert, Arthur Zimek, Andreas Züfle)
  - ❑ Introduction to Data Mining book slides at <http://www-users.cs.umn.edu/~kumar/dmbook/>
  - ❑ Pedro Domingos Machine Lecture course slides at the University of Washington
  - ❑ Machine Learning book by T. Mitchel slides at <http://www.cs.cmu.edu/~tom/mlbook-chapter-slides.html>
  - ❑ C. Burges, [A Tutorial on Support Vector Machines for Pattern Recognition](#), DMKD 1998
  - ❑ Thank you to all TAs contributing to their improvement, namely Vasileios Iosifidis, Damianos Melidis, Tai Le Quy, Han Tran.

Thank you

Questions/Feedback/Wishes?