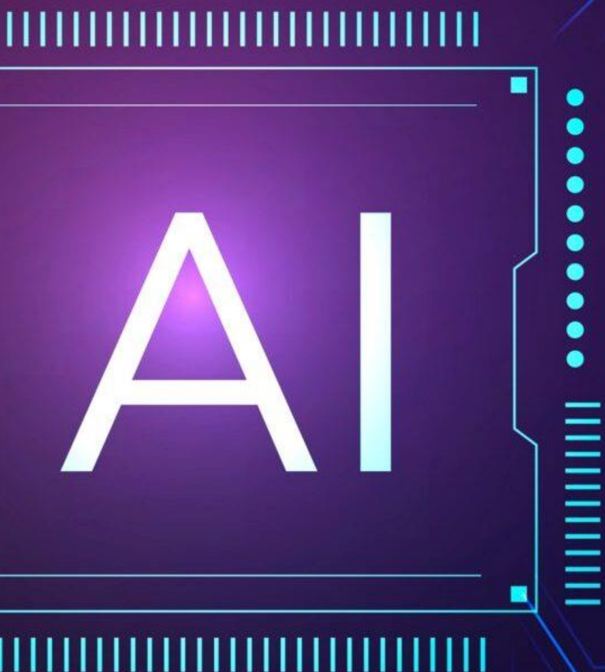


MULTI-LAYER PERCEPTRON

Nguyễn Ngọc Thảo – Nguyễn Hải Minh
{nnthao, nhminh}@fit.hcmus.edu.vn

Outline

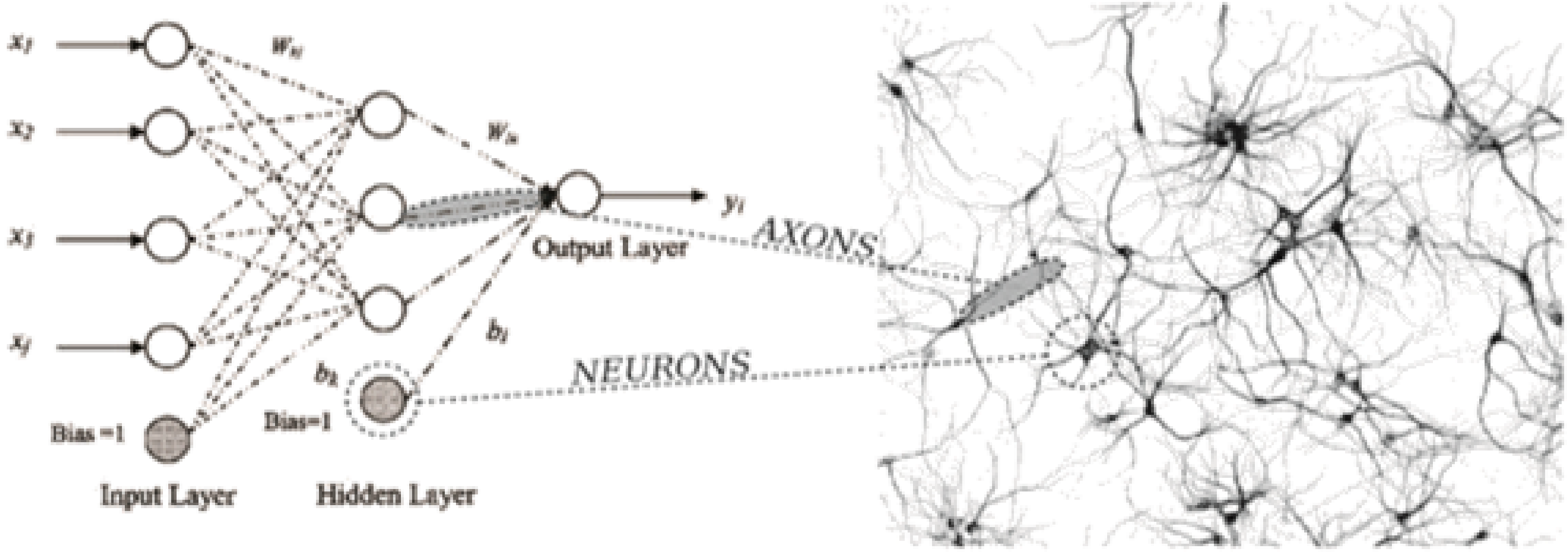
- Artificial neural networks
- Perceptron
- Multi-layer perceptron

What is a neural network?

- The biological neural network (NN) is a reasoning model based on the human brain.
  - There are approximately 86 billion neurons. Estimates of connections vary for an adult, ranging from 100 to 500 trillion.
- It is a system that is highly complex, nonlinear and parallel information-processing.
- Learning through experience is an essential characteristic.
  - **Plasticity**: connections leading to the "right answer" are enhanced, while those to the "wrong answer" are weakened.

Biological neural network

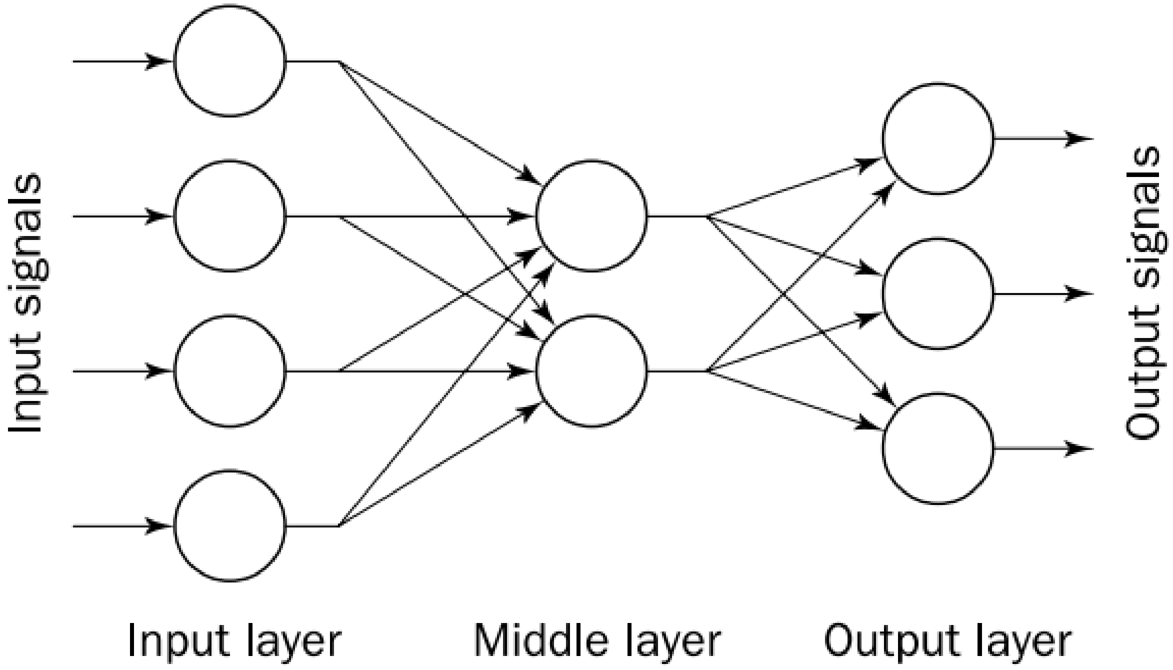
- There are attempts to emulate biological neural network in the computer, resulting artificial neural networks (ANNs).



- Just resemble the learning mechanisms, not the architecture
  - Megatron-Turing's NLG: 530 billion parameters, GPT-3: 175 billion

ANN: Network architecture

- An ANN has many neurons, arranging in a hierarchy of layers.
- Each neuron is an elementary information-processing unit.

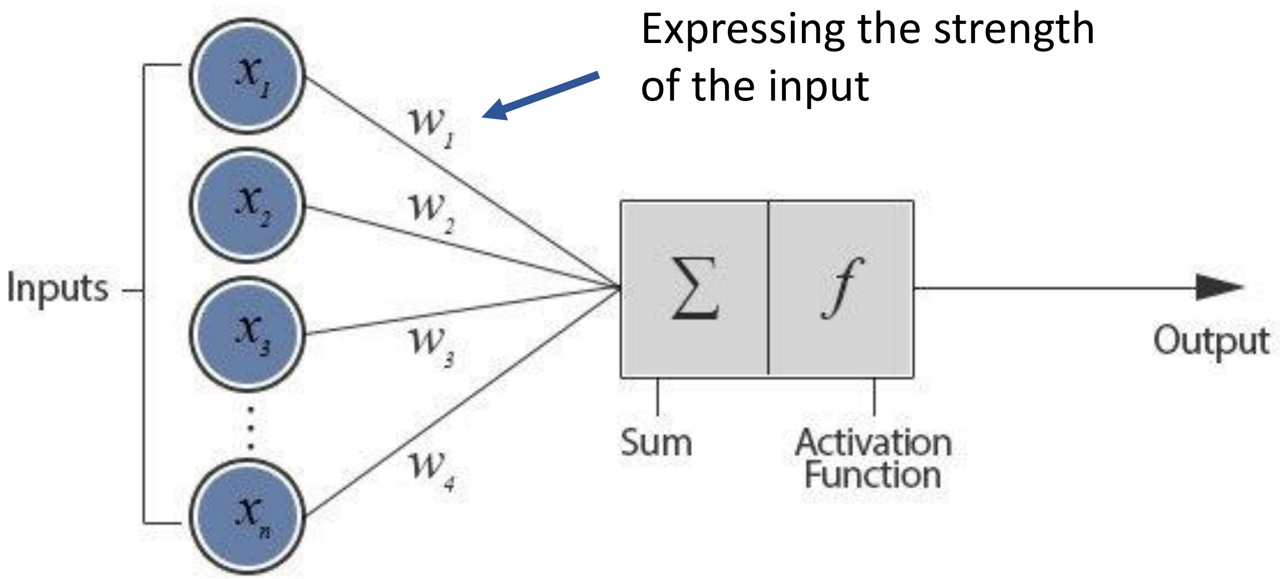


- ANN improve performance via experience and generalization.

ANN: Applications

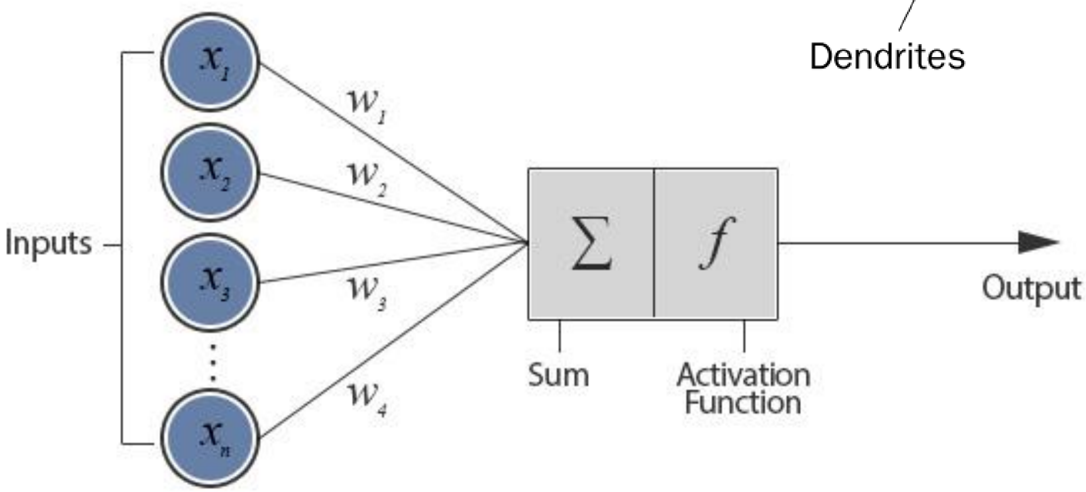
ANN: Neurons and Signals

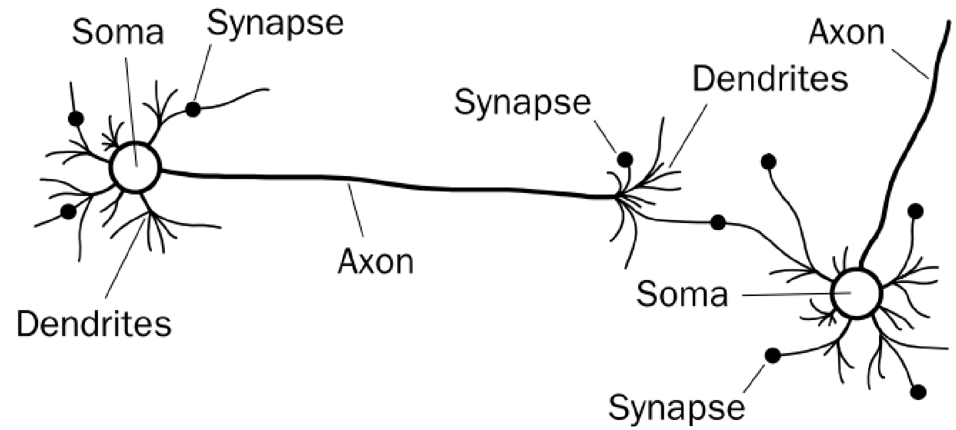
- Each neuron receives several input signals through its connections and produces at most a single output signal.



- The set of weights is the long-term memory in an ANN $\rightarrow$ the learning process iteratively adjusts the weights.

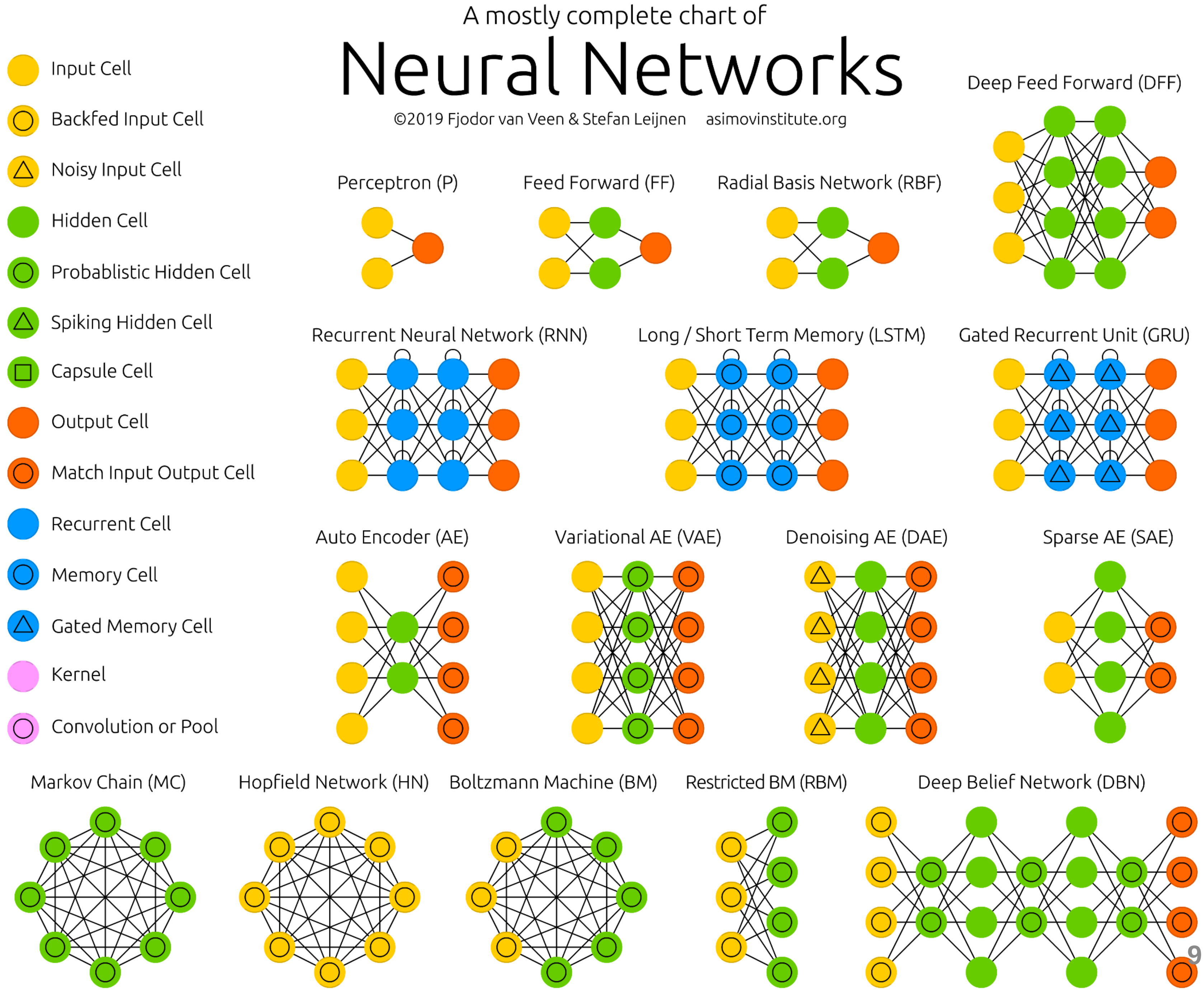
Artificial neuron



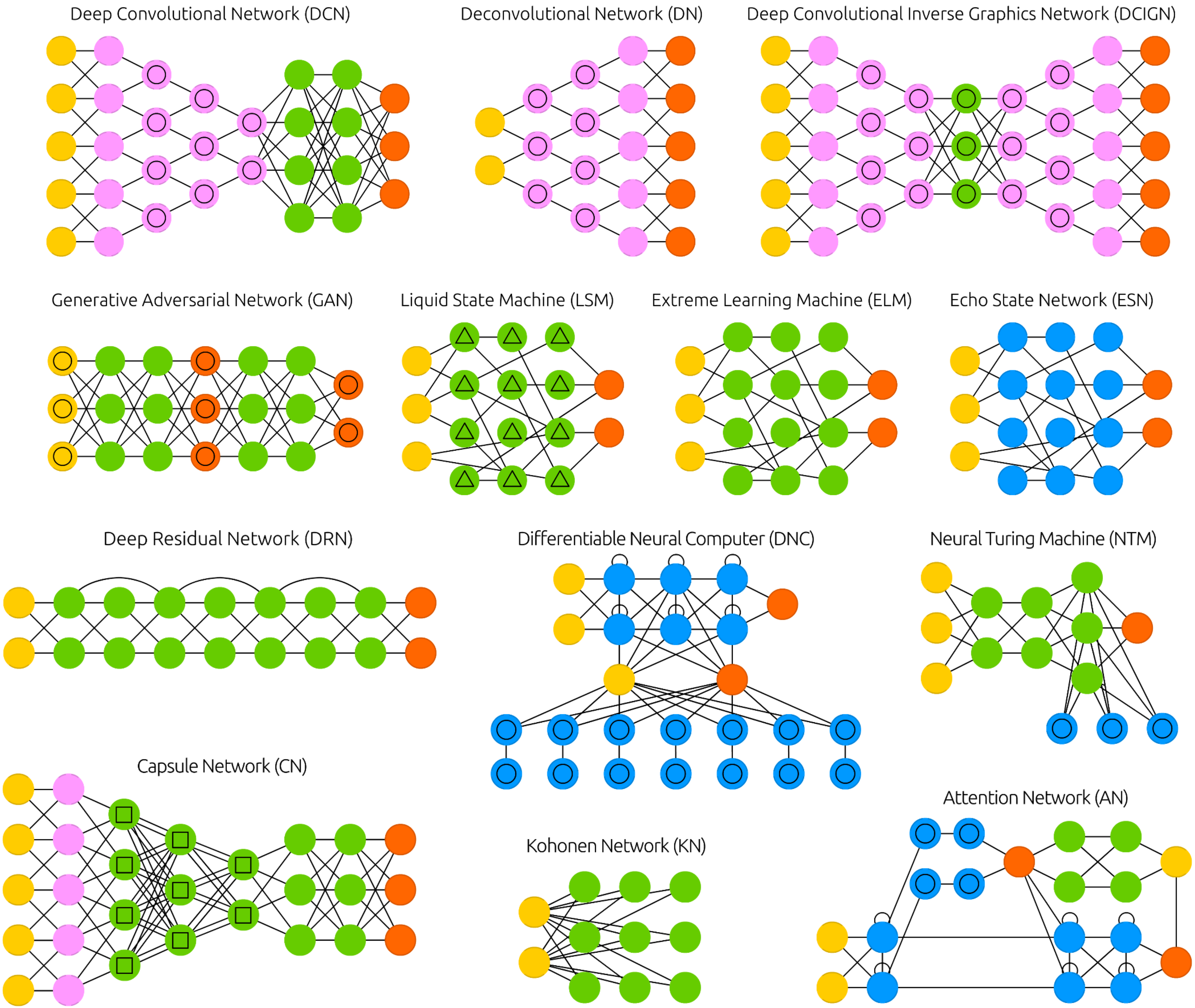


Biological neuron

Biological neuron	Artificial neuron
Soma	Neuron
Dendrite	Input
Axon	Output
Synapse	Weight


A mostly complete chart of
Neural Networks
©2019 Fjodor van Veen & Stefan Leijnen asimovinstitute.org
Input Cell
Backfed Input Cell
Noisy Input Cell
Hidden Cell
Probablistic Hidden Cell
Spiking Hidden Cell
Capsule Cell
Output Cell
Match Input Output Cell
Recurrent Cell
Memory Cell
Gated Memory Cell
Kernel
Convolution or Pool
Perceptron (P)
Feed Forward (FF)
Radial Basis Network (RBF)
Deep Feed Forward (DFF)
Recurrent Neural Network (RNN)
Long / Short Term Memory (LSTM)
Gated Recurrent Unit (GRU)
Auto Encoder (AE)
Variational AE (VAE)
Denoising AE (DAE)
Sparse AE (SAE)
Markov Chain (MC)
Hopfield Network (HN)
Boltzmann Machine (BM)
Restricted BM (RBM)
Deep Belief Network (DBN)

9



Source: [The Asimov Institute]

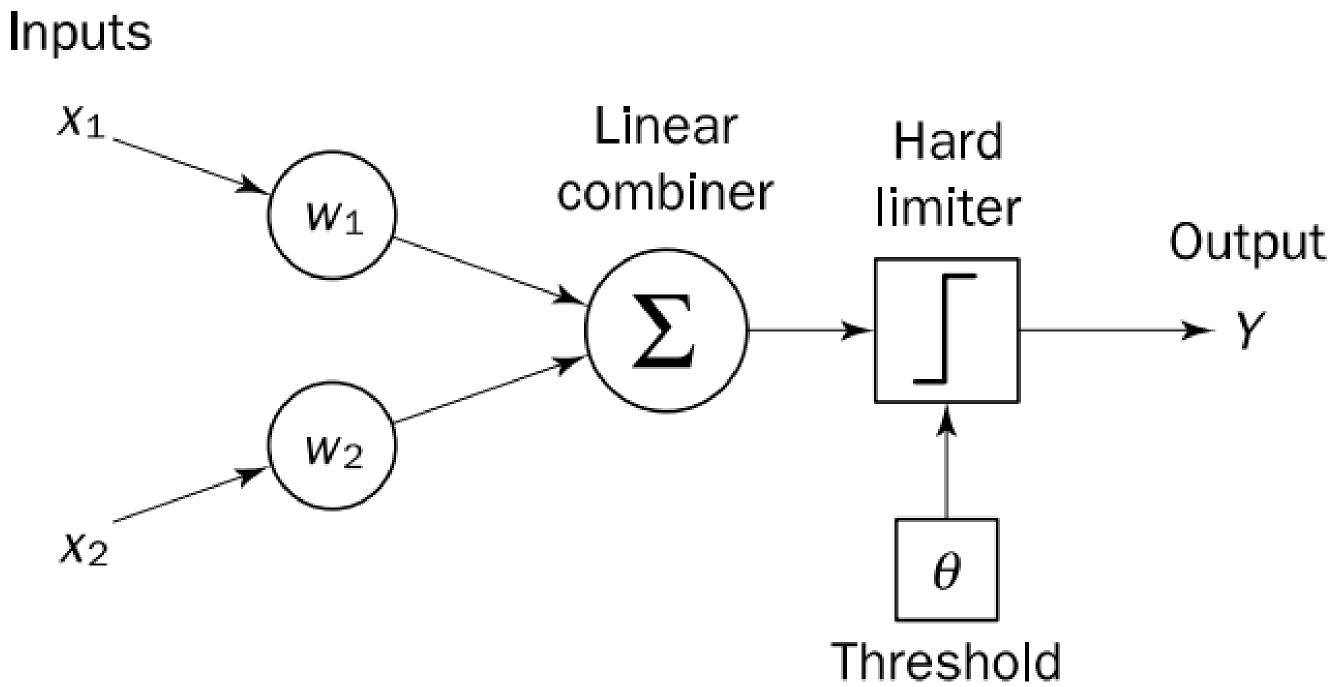
How to build an ANN?

- The network architecture must be decided first.
  - How many neurons are to be used?
  - How the neurons are to be connected to form a network?
- Then determine which learning algorithm to use,
  - Supervised /semi-supervised / unsupervised / reinforcement learning
- And finally train the neural network
  - How to initialize the weights of the network?
  - How to update them from a set of training examples.

Perceptron

Perceptron (Frank Rosenblatt, 1958)

- A perceptron has a single neuron with adjustable synaptic weights and a hard limiter.

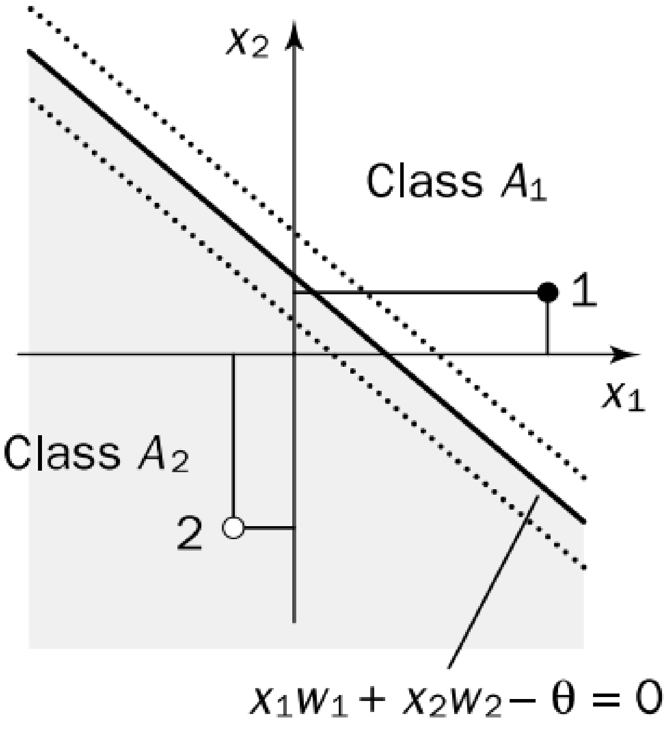


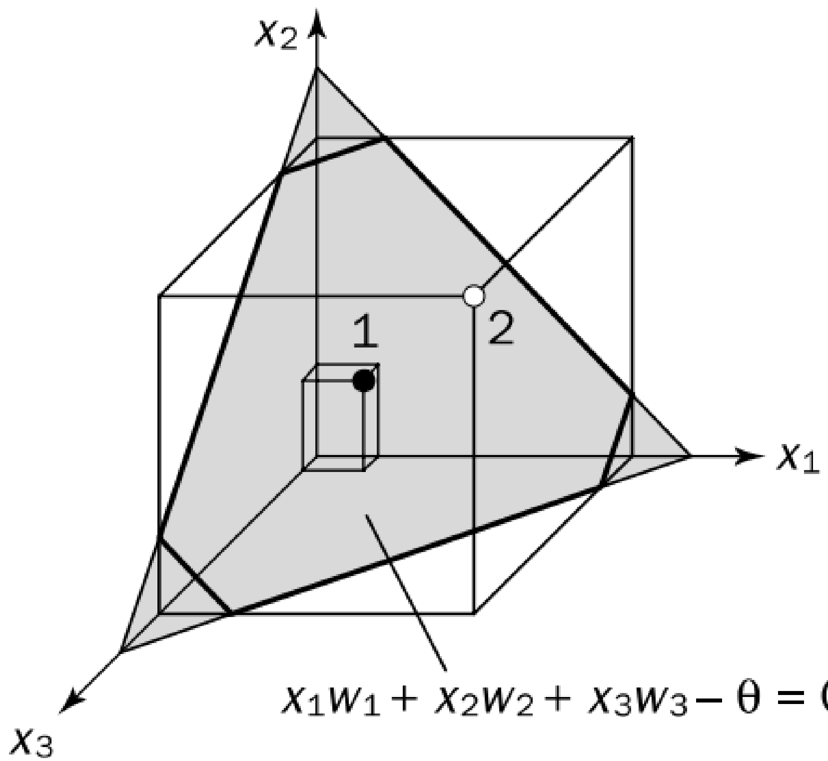
A single-layer two-input perceptron

How does a perceptron work?

- Divide the n-dimensional space into two decision regions by a hyperplane defined by the linearly separable function

$$\boldsymbol{y = \sum_{i=1}^{n} x_i w_i - \theta}$$





Perceptron learning rule

- **Step 1 – Initialization:** Initial weights $\boldsymbol{w_1}, \boldsymbol{w_2}, \dots, \boldsymbol{w_n}$ and threshold $\boldsymbol{\theta}$ are randomly assigned to small numbers (usually in $[-0.5, 0.5]$, but not restricted to).
- **Step 2 – Activation:** At iteration $\boldsymbol{p}$, apply the $\boldsymbol{p}^{th}$ example, which has inputs $\boldsymbol{x_1}(\boldsymbol{p}), \boldsymbol{x_2}(\boldsymbol{p}), \dots, \boldsymbol{x_n}(\boldsymbol{p})$ and desired output $\boldsymbol{Y_d}(\boldsymbol{p})$, and calculate the actual output

$$\boldsymbol{Y(p) = \sigma\left(\sum_{i=1}^{n} x_i(p) w_i(p) + (-1)\theta\right)}$$

$$\boxed{\sigma(x) = \begin{cases} 1 & if\ x \geq 0 \\ 0 & if\ x < 0 \end{cases}}$$

  where n is the number of perceptron inputs and $step$ is the activation function

- **Step 3 – Weight training**
  - Update the weights $\boldsymbol{w_i}$: $\boldsymbol{w_i(p+1) = w_i(p) - \Delta w_i(p)}$

    where $\Delta \boldsymbol{w_i}(\boldsymbol{p})$ is the weight correction at iteration $\boldsymbol{p}$
  - The **delta rule** determines how to adjust the weights: $\boldsymbol{\Delta w_i(p) = \eta \times x_i(p) \times e(p)}$

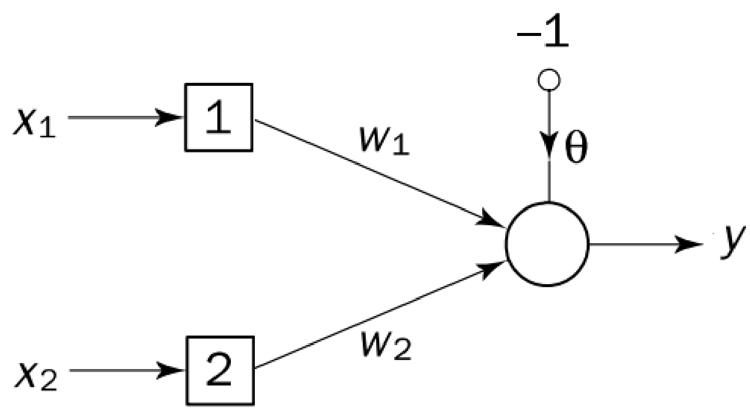
    where $\boldsymbol{\eta}$ is the learning rate $(0 < \eta < 1)$ and $\boldsymbol{e(p) = Y(p) - Y_d(p)}$
- **Step 4 – Iteration:** Increase iteration $\boldsymbol{p}$ by one, go back to Step 2 and repeat the process until convergence.

Perceptron for the logical AND/OR

- A single-layer perceptron **can** learn the **AND/OR** operations.

	Inputs		Desired output	Initial weights		Actual output	Error	Final weights	
Epoch	x_1	x_2	Y_d	w_1	w_2	Y	e	w_1	w_2
1	0	0	0	0.3	−0.1	0	0	0.3	−0.1
	0	1	0	0.3	−0.1	0	0	0.3	−0.1
	1	0	0	0.3	−0.1	1	1	0.2	−0.1
	1	1	1	0.2	−0.1	0	−1	0.3	0.0
2	0	0	0	0.3	0.0	0	0	0.3	0.0
	0	1	0	0.3	0.0	0	0	0.3	0.0
	1	0	0	0.3	0.0	1	1	0.2	0.0
	1	1	1	0.2	0.0	1	0	0.2	0.0
3	0	0	0	0.2	0.0	0	0	0.2	0.0
	0	1	0	0.2	0.0	0	0	0.2	0.0
	1	0	0	0.2	0.0	1	1	0.1	0.0
	1	1	1	0.1	0.0	0	−1	0.2	0.1
4	0	0	0	0.2	0.1	0	0	0.2	0.1
	0	1	0	0.2	0.1	0	0	0.2	0.1
	1	0	0	0.2	0.1	1	1	0.1	0.1
	1	1	1	0.1	0.1	1	0	0.1	0.1
5	0	0	0	0.1	0.1	0	0	0.1	0.1
	0	1	0	0.1	0.1	0	0	0.1	0.1
	1	0	0	0.1	0.1	0	0	0.1	0.1

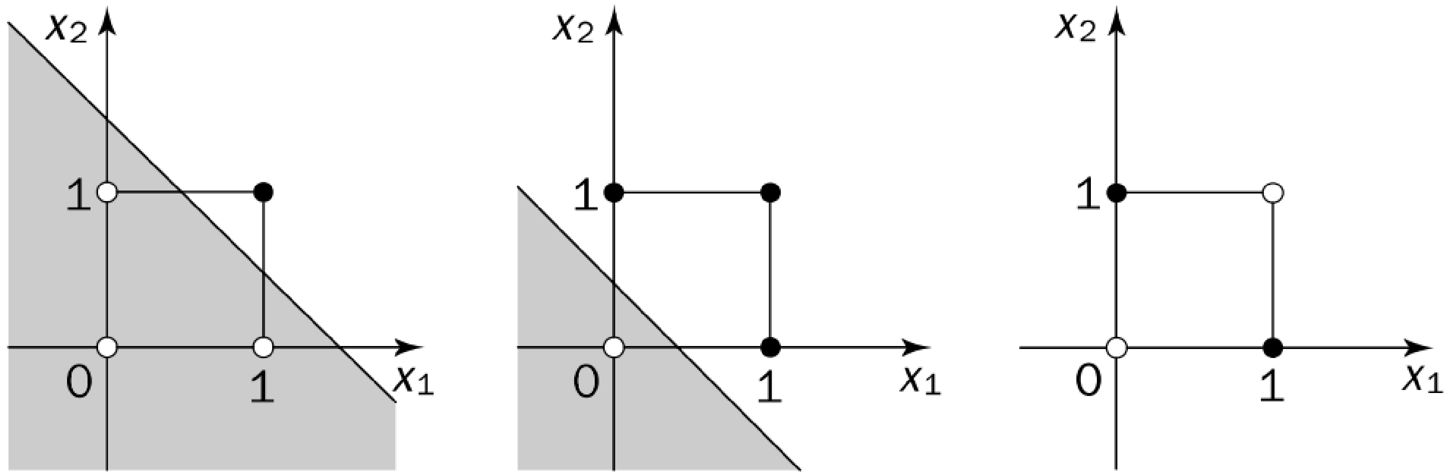
Threshold $\theta = 0.2$, learning rate $\eta = 0.1$



The learning of logical AND converged after several iterations

Perceptron for the logical XOR

- It **cannot** be trained to perform the **Exclusive-OR**.



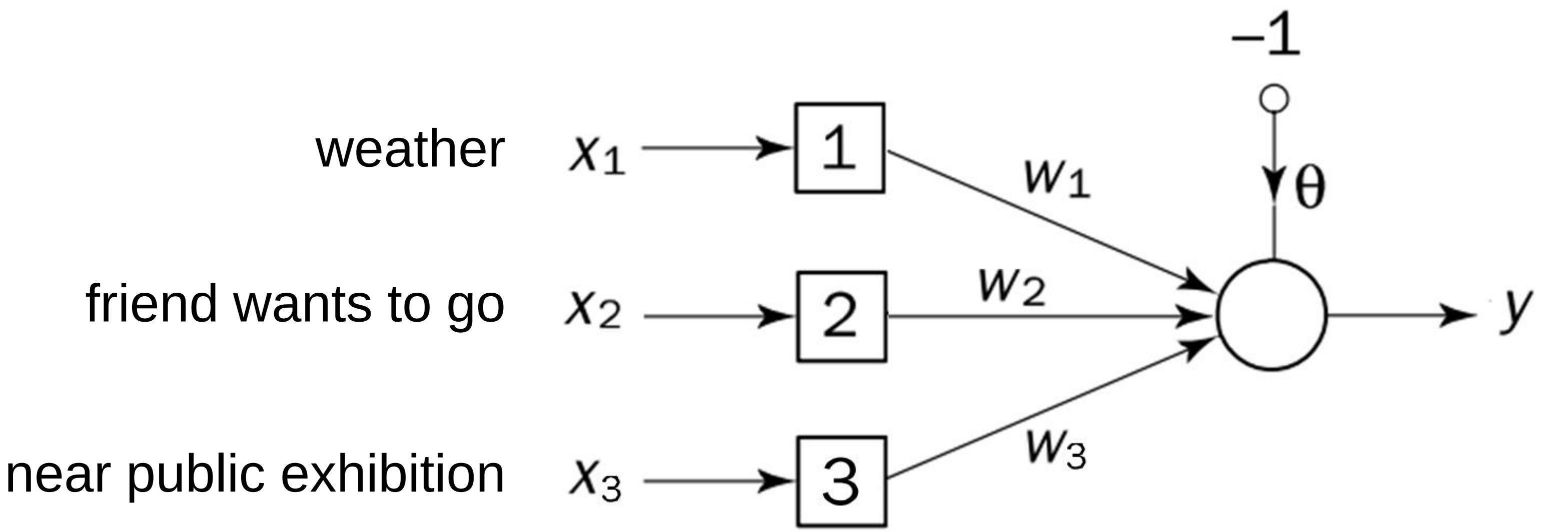
(a) AND ($x_1 \cap x_2$) (b) OR ($x_1 \cup x_2$) (c) Exclusive-OR ($x_1 \oplus x_2$)

- Generally, perceptron can classify only linearly separable patterns regardless of the activation function used.
  - Research works: Shynk, 1990 and Shynk and Bershad, 1992.

Perceptron: An example

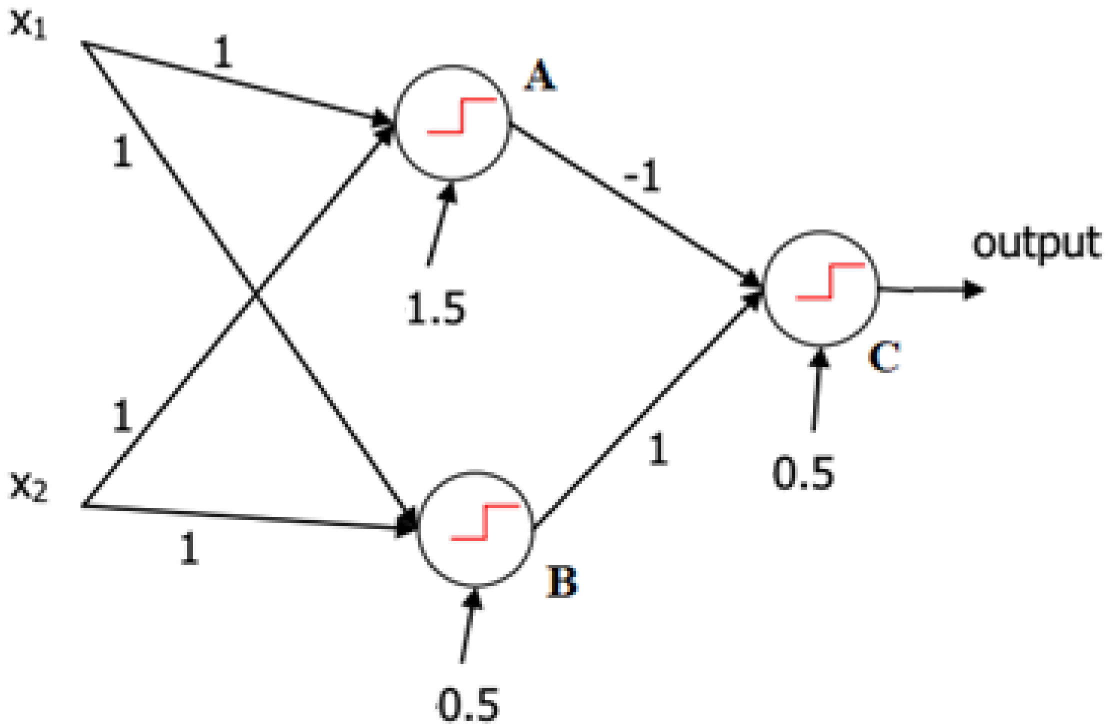
Suppose there is a high-tech exhibition in the city, and you are thinking about whether to go there. Your decision relies on the below factors:

- Is the weather good?
- Does your friend want to accompany you?
- Is the exhibition near public transit? (You don't own a car).



Quiz 03: Perceptron

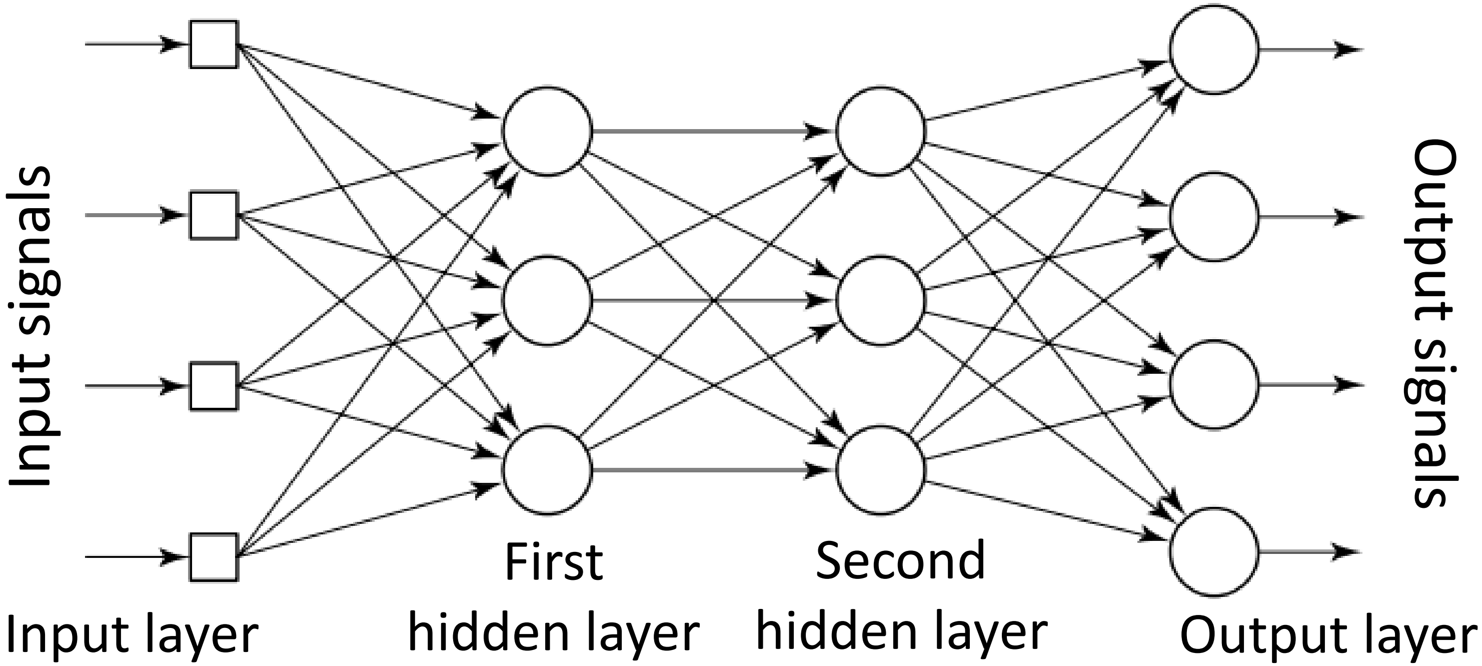
- Consider the following neural network which receives binary input values, x_1 and x_2, and produces a single binary value.



- For every combination (x_1, x_2), what are the output values at neurons, A, B and C?

Multi-layer perceptron

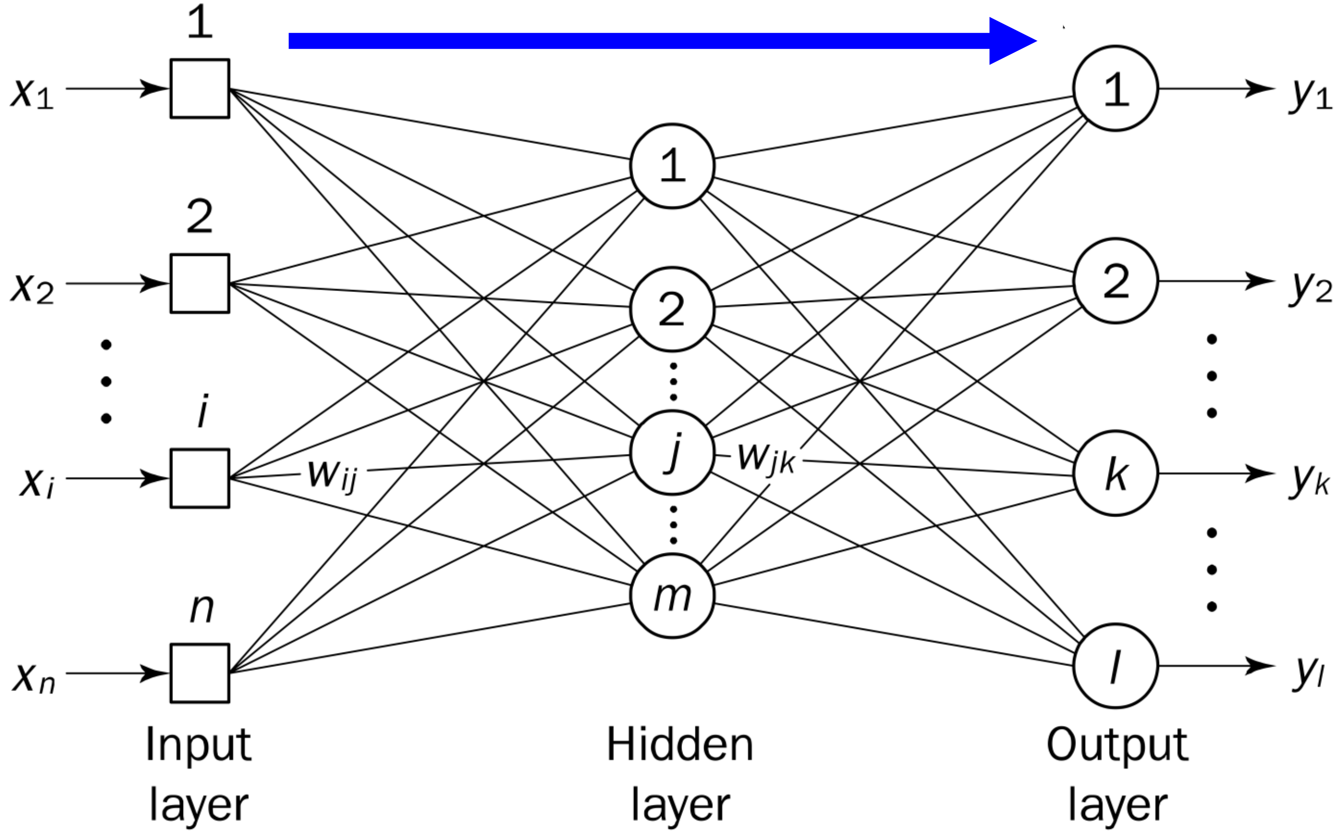
Multi-layer perceptron (MLP)



- A fully connected feedforward network with at least three layers.
- **Idea:** Map certain input(s) to a specified target value by using a cascade of nonlinear transformations.

Learning algorithm: Back-propagation

The input signals are propagated forwardly on a layer-by-layer basis.

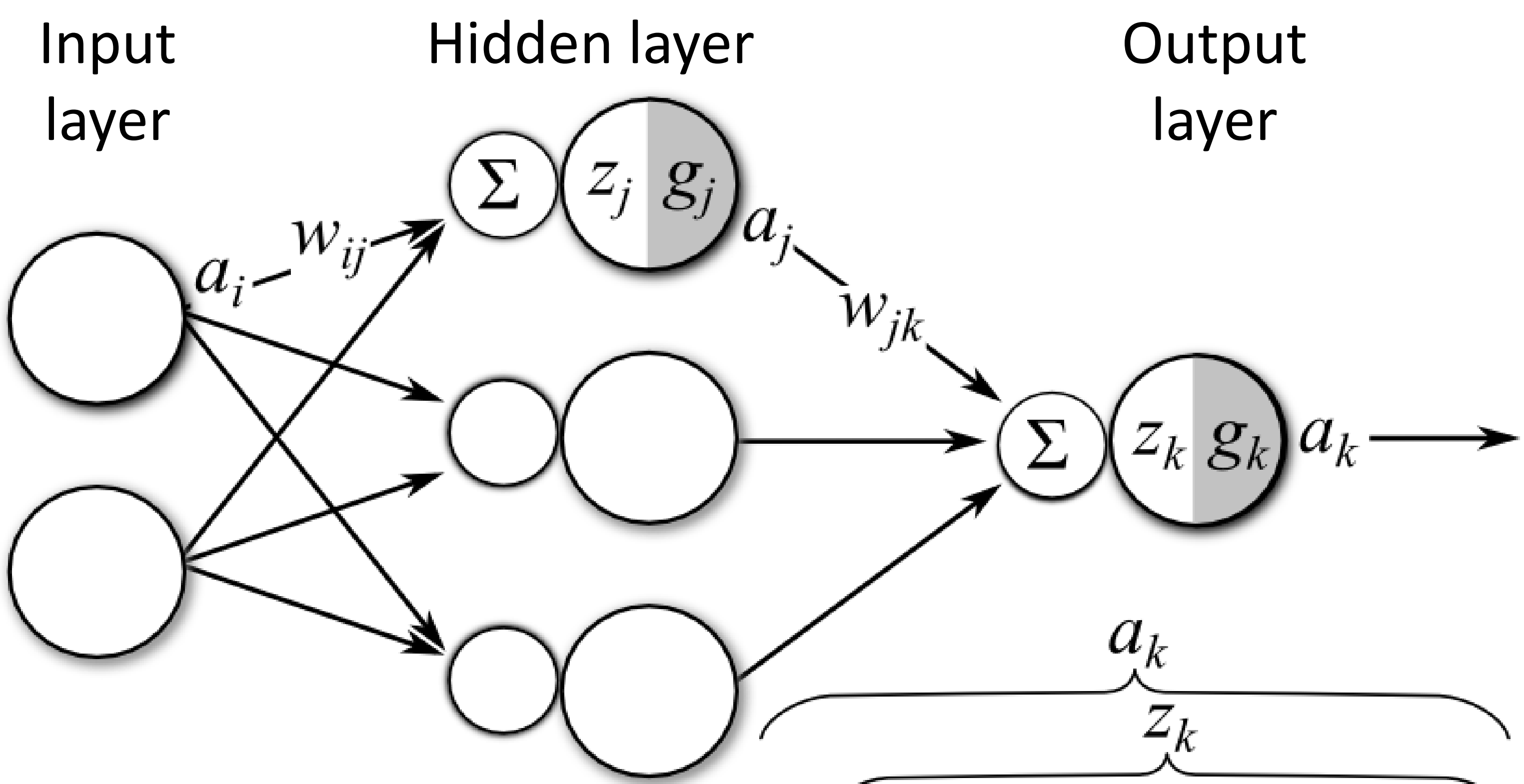


The error signals are propagated backwards from the output layer to the input layer.

Back-propagation algorithm

- Consider a MLP with one hidden layer.
- Note the following notations
  - a_i: the output value of node i in the input layer
  - z_j: the input value to node j in the layer h
  - g_j: the activation function for node j in the layer h (applied to z_j)
  - $a_j = g_j(z_j)$: the output value of node j in the layer h
  - b_j: the bias/offset for unit j in the layer h
  - w_{ij}: weights connecting node i in layer $(h-1)$ to node j in layer h
  - t_k: target value for node k in the output layer

Back-propagation algorithm



$$a_k = \overbrace{g_k(\overbrace{b_k + \Sigma_j \underbrace{g_j(\underbrace{b_j + \Sigma_i a_i w_{ij}}_{z_j})}_{a_j} w_{jk}}^{z_k})}^{a_k}$$

* Bias units are not shown.

BP algorithm: The error function

- Training a neural network entails finding parameters $\theta = \{\mathbf{W}, \mathbf{b}\}$ that minimize the errors.
- The error function is usually the sum of the squared errors between the target values t_k and the network outputs a_k.

$$\boldsymbol{E = \frac{1}{2}\sum_{k=1}^{l}(a_k - t_k)^2}$$

  - l is the dimensionality of the target for a single observation.

- This parameter optimization problem can be solved using gradient descent, computing $\frac{\partial E}{\partial \theta}$ for all θ.

BP algorithm: Output layer params

- Calculating the gradient of the error function with respect to those parameters is straightforward with the chain rule.

$$\frac{\partial E}{\partial w_{jk}} = \frac{1}{2}\sum_k (a_k - t_k)^2$$

$$= (a_k - t_k)\frac{\partial}{\partial w_{jk}}(a_k - t_k)$$

- Then, $\frac{\partial E}{\partial w_{jk}} = (a_k - t_k)\frac{\partial}{\partial w_{jk}} a_k$ since $\frac{\partial}{\partial w_{jk}} t_k = 0$

$$= (a_k - t_k)\frac{\partial}{\partial w_{jk}} g_k(z_k)$$ since $a_k = g(z_k)$

$$= (a_k - t_k) g_k'(z_k)\frac{\partial}{\partial w_{jk}} z_k$$

BP algorithm: Output layer params

- Recall that $z_k = b_k + \sum_j g_j(z_j) w_{jk}$, and hence, $\frac{\partial}{\partial w_{jk}} z_k = g_j(z_j) = a_j$.
- Then, $$\frac{\partial \boldsymbol{E}}{\partial \boldsymbol{w_{jk}}} = \boxed{(\boldsymbol{a_k} - \boldsymbol{t_k})}\boxed{\boldsymbol{g'_k(z_k)}}\boxed{\boldsymbol{a_j}}$$

the difference between the network output a_k and the target value t_k

the derivative of the activation function at z_k

the output of node j from the hidden layer feeding into the output layer

- The common activation function is the sigmoid function

$$g(z) = \frac{1}{1 + e^{-z}}$$

whose derivative is

$$g'(z) = g(z)(1 - g(z))$$

BP algorithm: Output layer params

- Let $\delta_k = (a_k - t_k) g'_k(z_k)$ be the error signal after being back-propagated through the output activation function g_k.
- The delta form of the error function gradient for the output layer weights is

$$\frac{\partial E}{\partial w_{jk}} = \delta_k a_j$$

- The gradient descent update rule for the output layer weights is

$$\boldsymbol{w_{jk} \leftarrow w_{jk} - \eta \frac{\partial E}{\partial w_{jk}}}$$

η is the learning rate

$$\leftarrow w_{jk} - \eta\, \delta_k\, a_j$$

$$\boldsymbol{\leftarrow w_{jk} - \eta\, (a_k - t_k)\, g_k(z_k) \big(1 - g_k(z_k)\big) a_j}$$

- Apply similar update rules for the remaining parameters w_{jk}.

BP algorithms: Output layer biases

- The gradient for the biases is simply the back-propagated error signal δ_k.

$$\frac{\partial E}{\partial b_k} = (a_k - t_k) g_k'(z_k)(1) = \delta_k$$

- Each bias is updated as $\boldsymbol{b_k \leftarrow b_k - \eta\ \delta_k}$

- Note that $\frac{\partial}{\partial b_k} z_k = \frac{\partial}{\partial b_k}\left[b_k + \sum_j g_j\left(z_j\right)\right] = 1$
  - The biases are weights on activations that are always equal to one, regardless of the feed-forward signal.
  - Thus, the bias gradients aren't affected by the feed-forward signal, only by the error.

BP algorithm: Hidden layer params

- The process starts just the same as for the output layer.

$$\frac{\partial E}{\partial w_{ij}} = \frac{1}{2}\sum_k (a_k - t_k)^2$$
$$= \sum_k (a_k - t_k)\frac{\partial}{\partial w_{ij}} a_k$$

- Apply the chain rule again, we obtain:

$$\frac{\partial E}{\partial w_{ij}} = \sum_k (a_k - t_k)\frac{\partial}{\partial w_{ij}} g_k(z_k) \qquad \text{since } a_k = g_k(z_k)$$
$$= \sum_k (a_k - t_k) g'_k(z_k)\frac{\partial}{\partial w_{ij}} z_k$$

BP algorithm: Hidden layer params

- The term z_k can be expanded as follows.

$$z_k = b_k + \sum_j a_j w_{jk} = b_k + \sum_j g_j(z_j) w_{jk} \qquad \text{since } a_j = g_j(z_j)$$

$$= b_k + \sum_j g_j\left(b_j + \sum_i a_i w_{ij}\right) w_{jk} \qquad \text{since } z_j = b_j + \textstyle\sum_i a_i w_{ij}$$

- Again, use the chain rule to calculate $\frac{\partial}{\partial w_{ij}} z_k$

$$\frac{\partial}{\partial w_{ij}} z_k = \frac{\partial z_k}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} = \frac{\partial}{\partial a_j}\left(b_k + \sum_j a_j w_{jk}\right)\frac{\partial a_j}{\partial w_{ij}} = w_{jk}\frac{\partial a_j}{\partial w_{ij}} = w_{jk}\frac{\partial g_j(z_j)}{\partial w_{ij}}$$

$$= w_{jk}\, g_j'(z_j)\frac{\partial z_j}{\partial w_{ij}} = w_{jk}\, g_j'(z_j)\frac{\partial}{\partial w_{ij}}\left(b_j + \textstyle\sum_j a_i w_{ij}\right)$$

$$= w_{jk}\, g_j'(z_j) a_i$$

BP algorithm: Hidden layer params

- Thus, $$\frac{\partial E}{\partial w_{ij}} = \sum_k (a_k - t_k) g_k'(z_k) w_{jk}\, g_j'(z_j) a_i$$

$$= \boxed{\left(\sum_k \delta_k w_{jk}\right)}\boxed{g_j'(z_j)}\boxed{a_i}$$

the output activation signal from the layer below a_i

error term

the derivative of the activation function at z_j

- Let $\delta_j = g_j'(z_j) \sum_k \delta_k w_{jk}$ denote the resulting error signal back to layer j.
- The error function gradient for the hidden layer weights is

$$\frac{\partial E}{\partial w_{ij}} = \delta_j a_i$$

To calculate the weight gradients at any layer l, we calculate the backpropagated error signal δ_l that reaches that layer from the "afterward" layers, and weight it by the feed-forward signal at $l-1$ *feeding into that layer.*

BP algorithm: Hidden layer params

- The gradient descent update rule for the hidden layer weights is

$$\boldsymbol{w_{ij} \leftarrow w_{ij} - \eta \frac{\partial E}{\partial w_{ij}}}$$

$$\leftarrow w_{ij} - \eta\, \delta_j\, a_i \quad \leftarrow w_{ij} - \eta\, g_j'(z_j)\left(\sum_k \delta_k w_{jk}\right) a_i$$

$$\boldsymbol{\leftarrow w_{ij} - \eta \left(\sum_k (a_k - t_k) g_k(z_k)(1 - g_k(z_k))\, w_{jk}\right) g_j(z_j)\big(1 - g_j(z_j)\big) a_i}$$

- Apply similar update rules for the remaining parameters w_{ij}.

BP algorithms: Hidden layer biases

- Calculating the error gradients with respect to the hidden layer biases b_j follows a very similar procedure to that for the hidden layer weights.

$$\frac{\partial E}{\partial b_j} = \sum_k (a_k - t_k) \frac{\partial}{\partial b_j} g_k(z_k) \ = \sum_k (a_k - t_k) g'_k(z_k) \frac{\partial z_k}{\partial b_j}$$

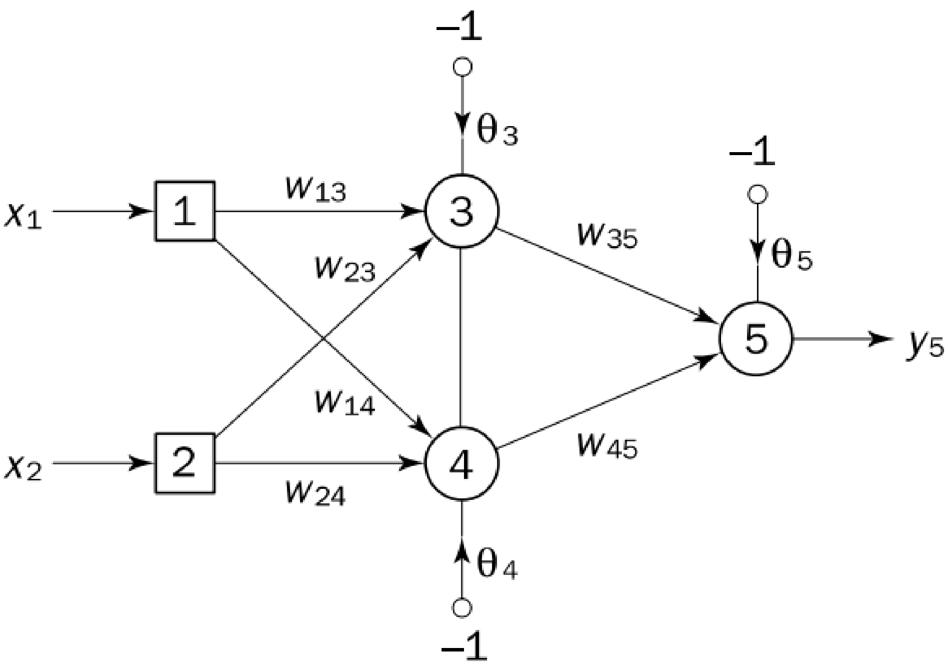
- Apply chain rule to solve $\frac{\partial z_k}{\partial b_j} = w_{jk}\ g'_j\left(z_j\right)(1)$
- The gradient for the biases is the back-propagated error signal δ_j.

$$\frac{\partial E}{\partial b_j} = \sum_k (a_k - t_k) g'_k(z_k) w_{jk}\ g'_j\left(z_j\right) = g'_j\left(z_j\right)\left(\sum_k \delta_k w_{jk}\right) = \delta_j$$

- Each bias is updated as $\boldsymbol{b_j \leftarrow b_j - \eta\ \delta_j}$

Back-propagation network for XOR

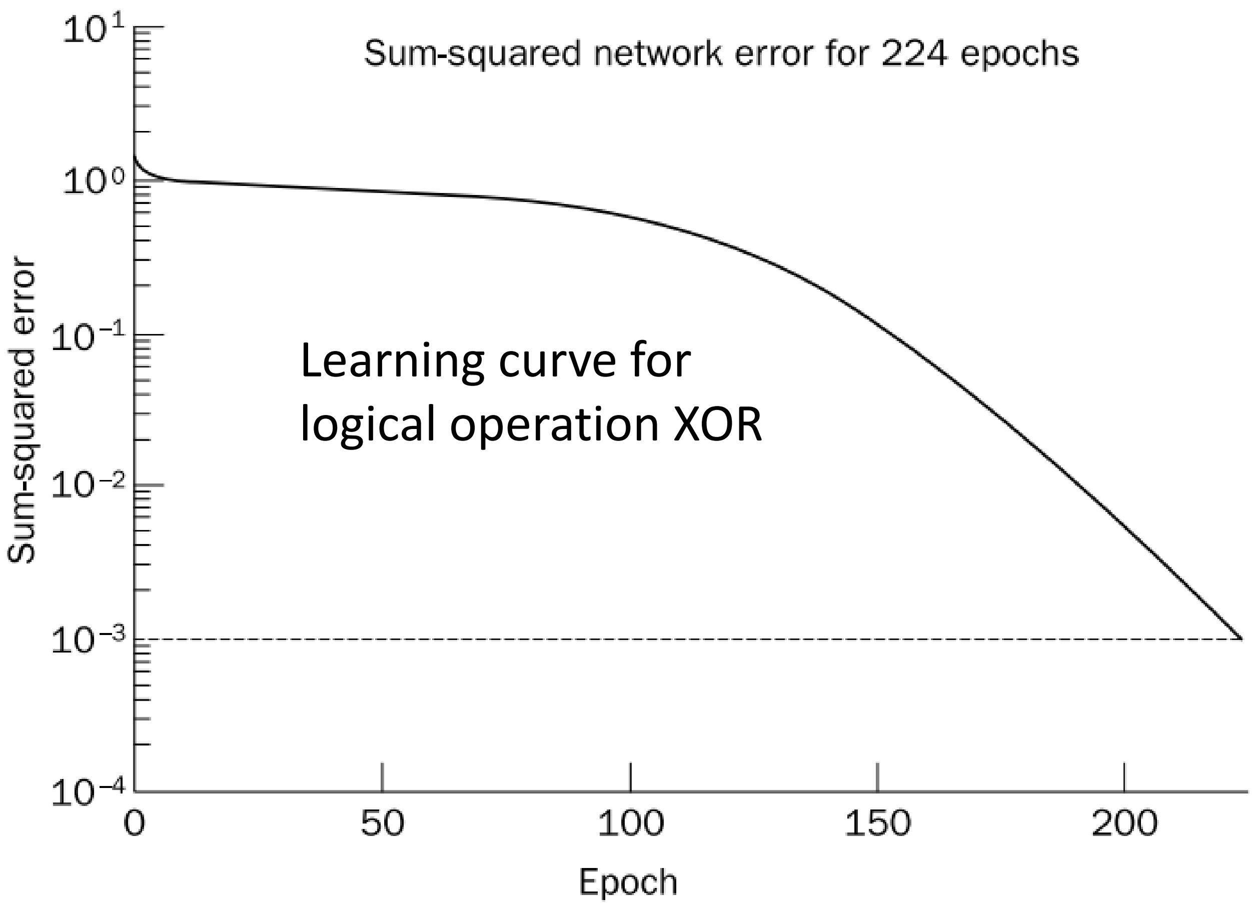
- The logical XOR problem took 224 epochs or 896 iterations for network training.



Inputs		Desired output	Actual output	Error	Sum of squared errors
x_1	x_2	y_d	y_5	e	
1	1	0	0.0155	–0.0155	0.0010
0	1	1	0.9849	0.0151	
1	0	1	0.9849	0.0151	
0	0	0	0.0175	–0.0175	

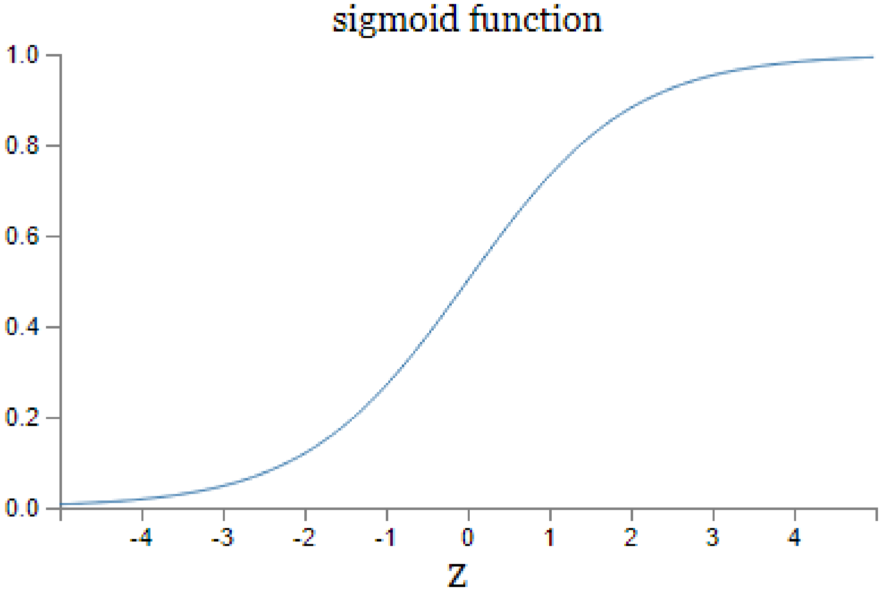
Sum of the squared errors (SSE)

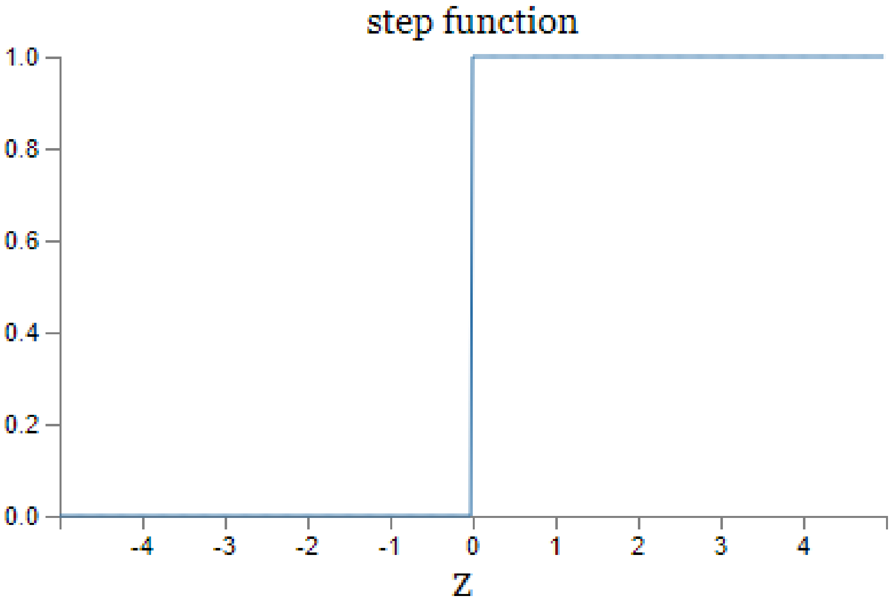
- When the SSE in an entire pass through all training sets is sufficiently small, a network is deemed to have converged.



Sigmoid neuron vs. Perceptron

- Sigmoid neuron better reflects the fact that small changes in weights and bias cause only a small change in output.





A sigmoidal function is a smoothed-out version of a step function.

About back-propagation learning

- *Are randomly initialized weights and thresholds leading to different solutions?*
  - Starting with different initial conditions will obtain different weights and threshold values. The problem will always be solved within different numbers of iterations.
- Back-propagation learning cannot be viewed as emulation of brain-like learning.
  - Biological neurons do not work backward to adjust the strengths of their interconnections, synapses.
- The training is slow due to extensive calculations.
  - Improvements: Caudill, 1991; Jacobs, 1988; Stubbs, 1990

Gradient descent

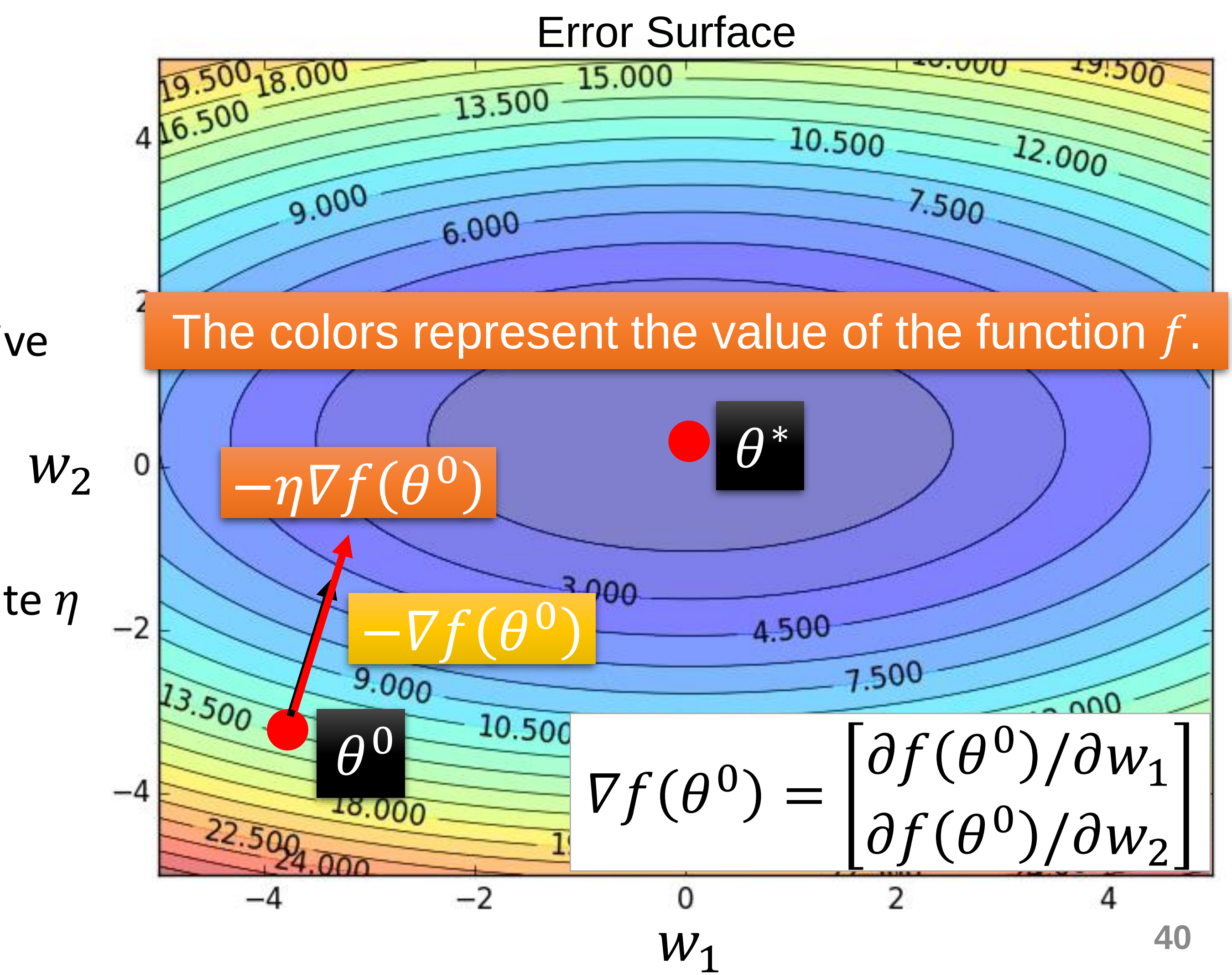
Gradient descent: Idea

- Consider two parameters, w_1 and w_2, in a network.

Randomly pick a starting point θ^0

Compute the negative gradient at θ^0
$\rightarrow -\nabla f(\theta^0)$

Time the learning rate η
$\rightarrow -\eta\nabla f(\theta^0)$



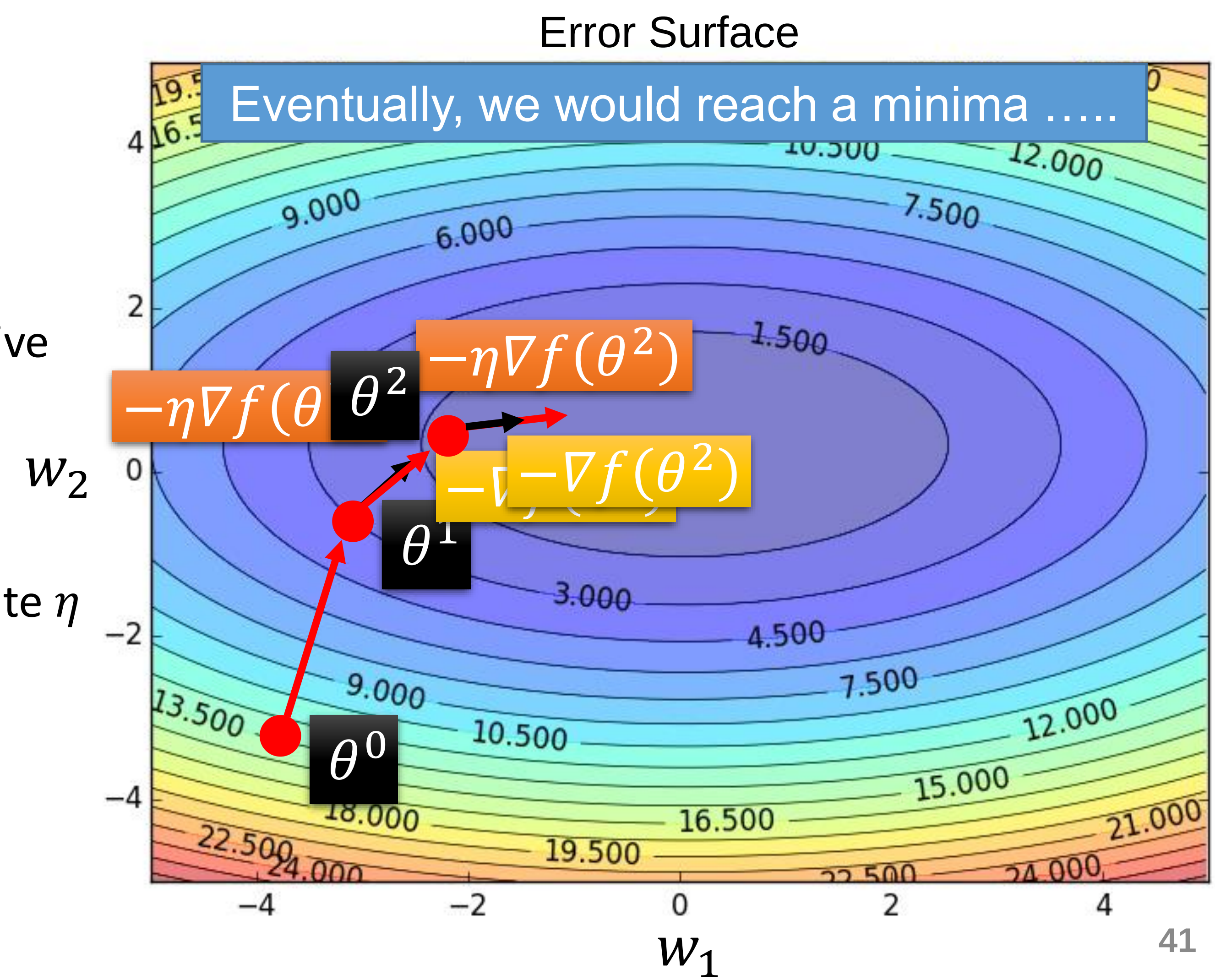
Gradient descent: Idea

- Consider two parameters, w_1 and w_2, in a network.

Randomly pick a starting point θ^0

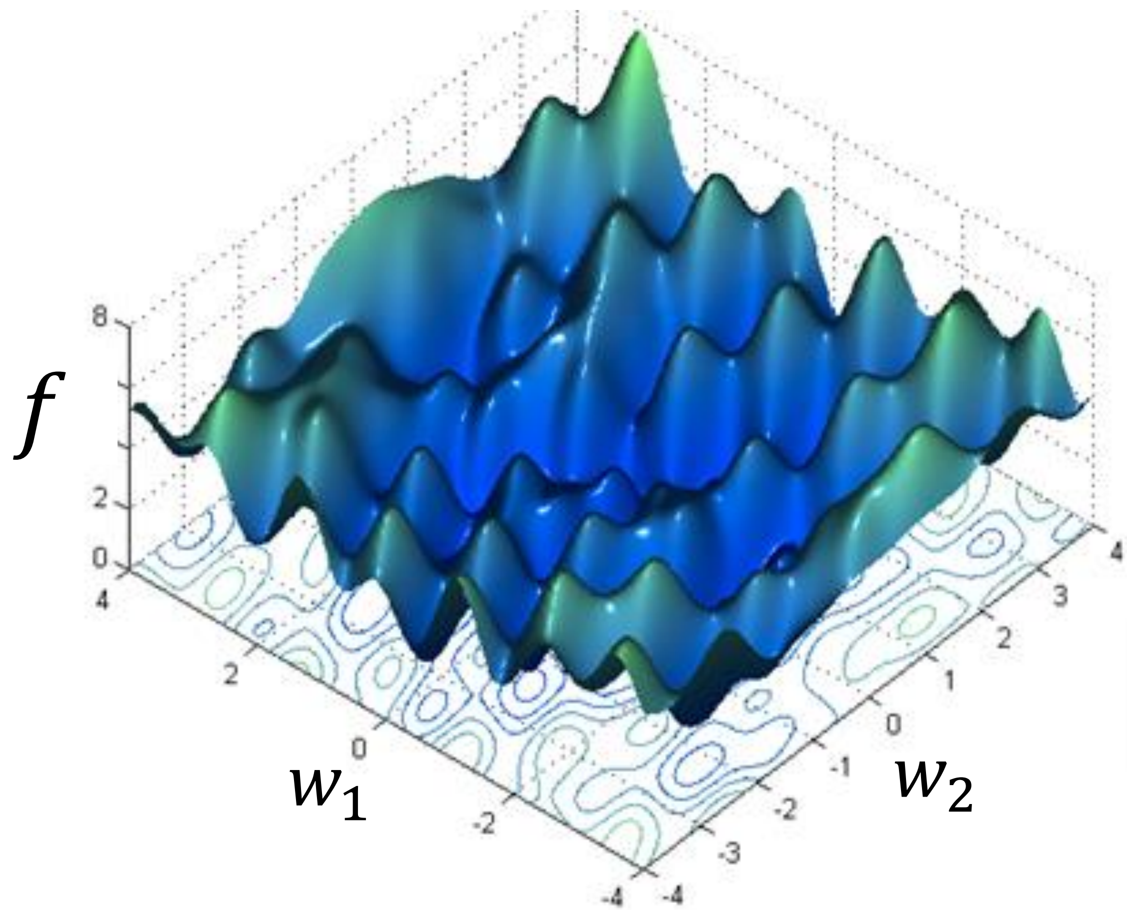
Compute the negative gradient at θ^0
$\rightarrow -\nabla f(\theta^0)$

Time the learning rate η
$\rightarrow -\eta\nabla f(\theta^0)$



Gradient descent: Optimality

- Gradient descent never guarantees global minima.



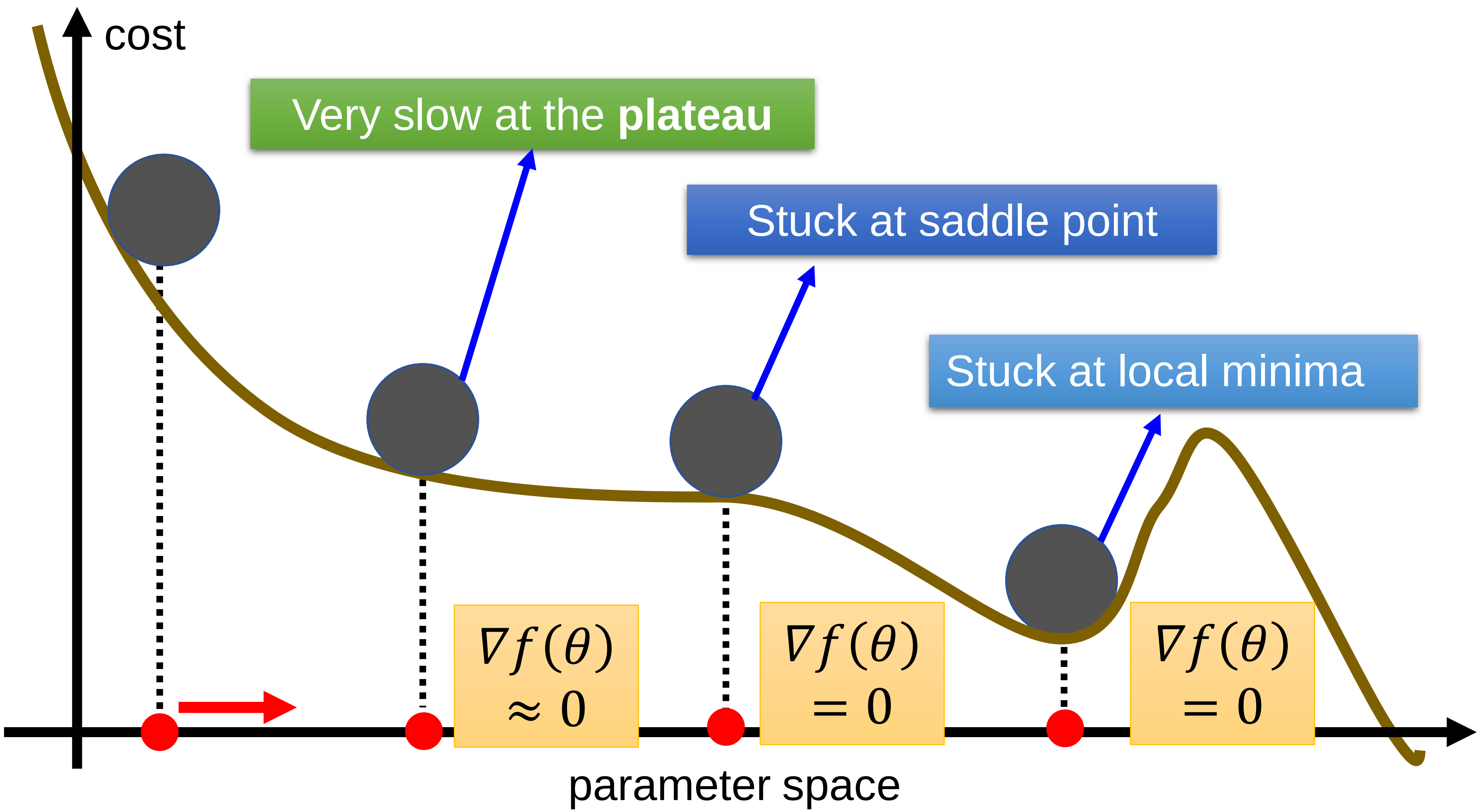
Different initial point θ^0

⇩

Reach different minima, so different results

Gradient descent: Optimality

- It also has issues at plateau and saddle point.



Accelarating the MLP learning

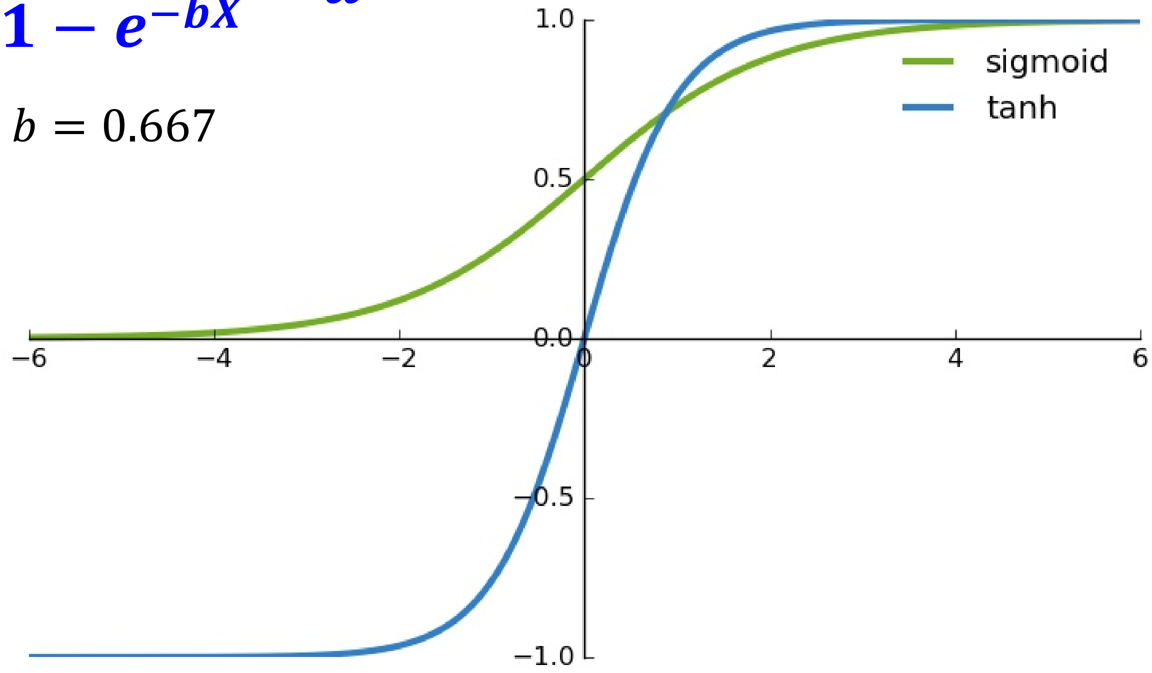
1. Use tanh instead of sigmoid

- We use a hyperbolic tangent to model the sigmoidal function.

$$\boldsymbol{Y = tanh(X) = \frac{2a}{1 - e^{-bX}} - a}$$

where $a = 1.716$ and $b = 0.667$

(Guyon, 1991)



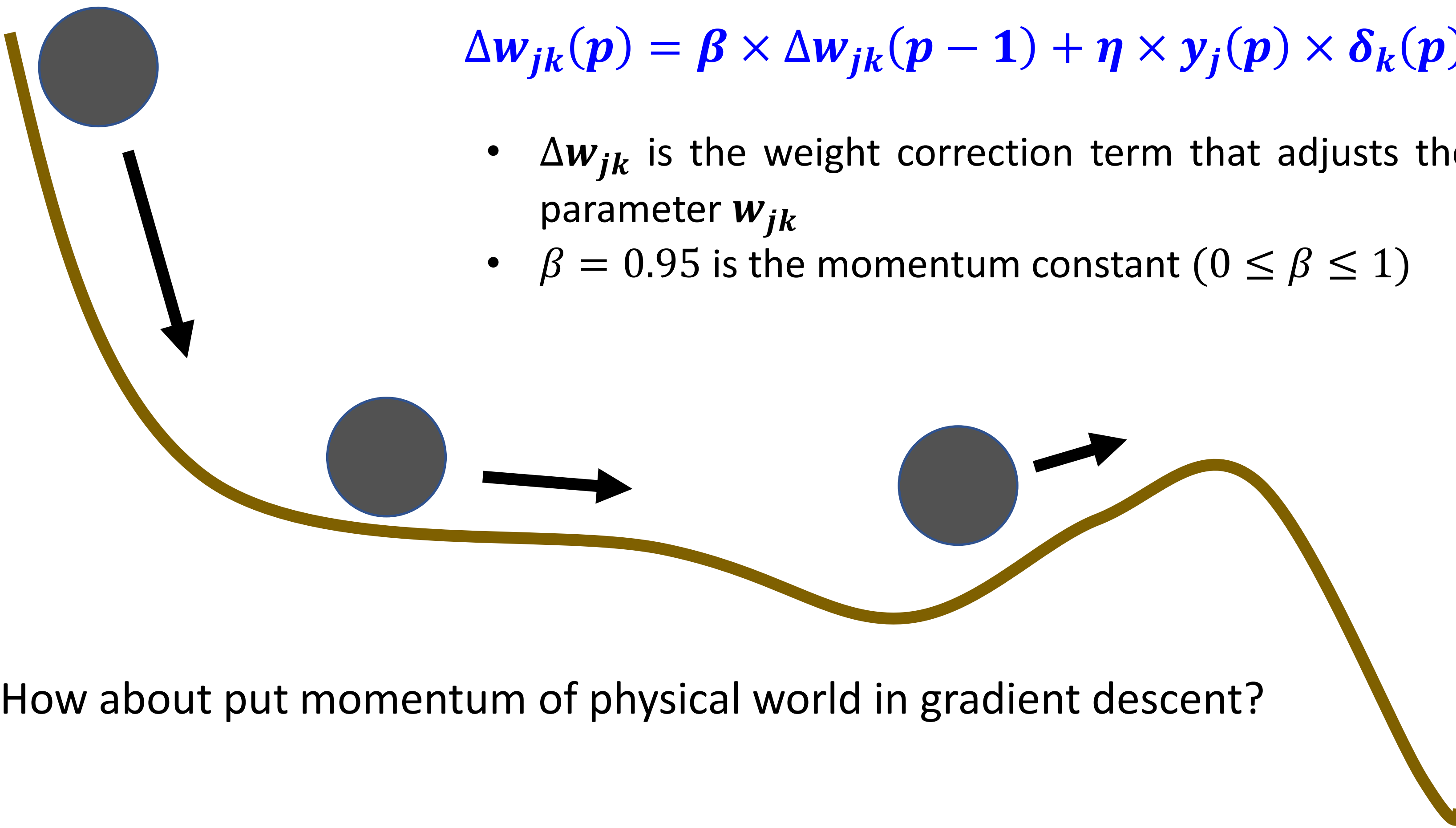
- Why is tanh better than sigmoid in MLP?

2. Generalized delta rule

- The delta rule further includes a momentum term (Rumelhart et al., 1986)

$$\Delta \boldsymbol{w}_{\boldsymbol{jk}}(\boldsymbol{p}) = \boldsymbol{\beta} \times \Delta \boldsymbol{w}_{\boldsymbol{jk}}(\boldsymbol{p}-\mathbf{1}) + \boldsymbol{\eta} \times \boldsymbol{y}_{\boldsymbol{j}}(\boldsymbol{p}) \times \boldsymbol{\delta}_{\boldsymbol{k}}(\boldsymbol{p})$$

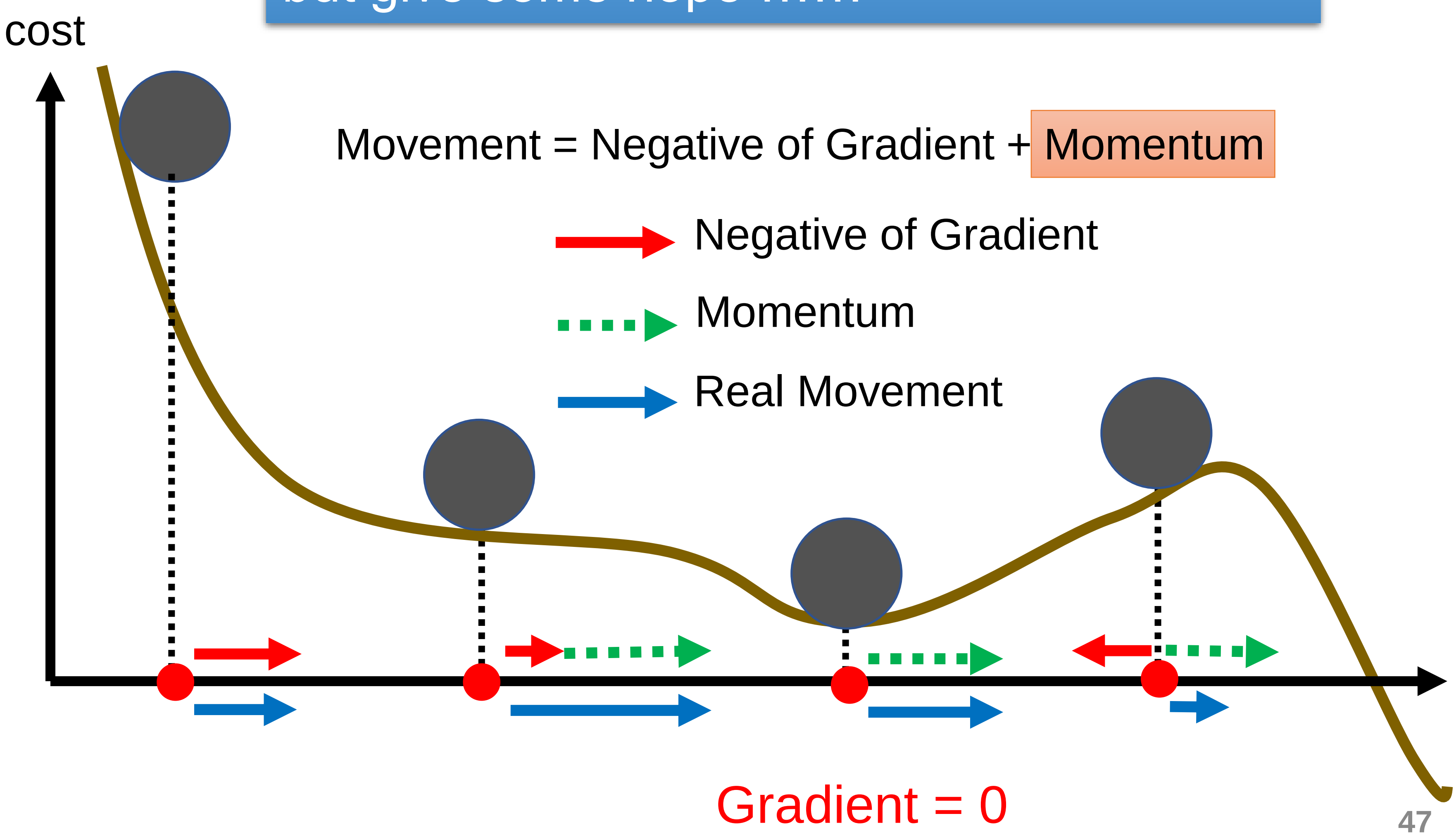
  - $\Delta \boldsymbol{w}_{\boldsymbol{jk}}$ is the weight correction term that adjusts the parameter $\boldsymbol{w}_{\boldsymbol{jk}}$
  - $\beta = 0.95$ is the momentum constant $(0 \le \beta \le 1)$



How about put momentum of physical world in gradient descent?

2. Generalized delta rule

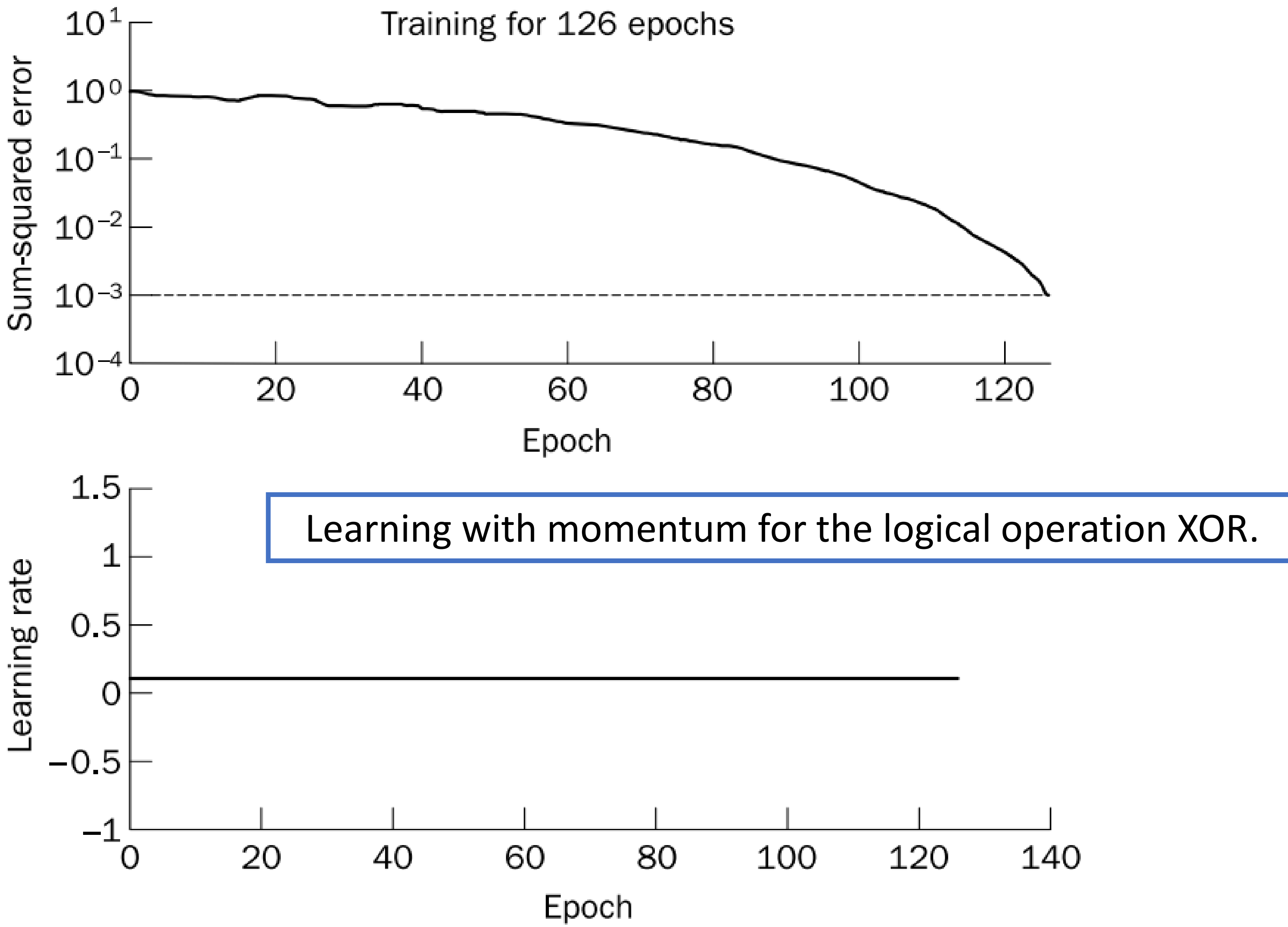
Still not guarantee reaching global minima, but give some hope ……



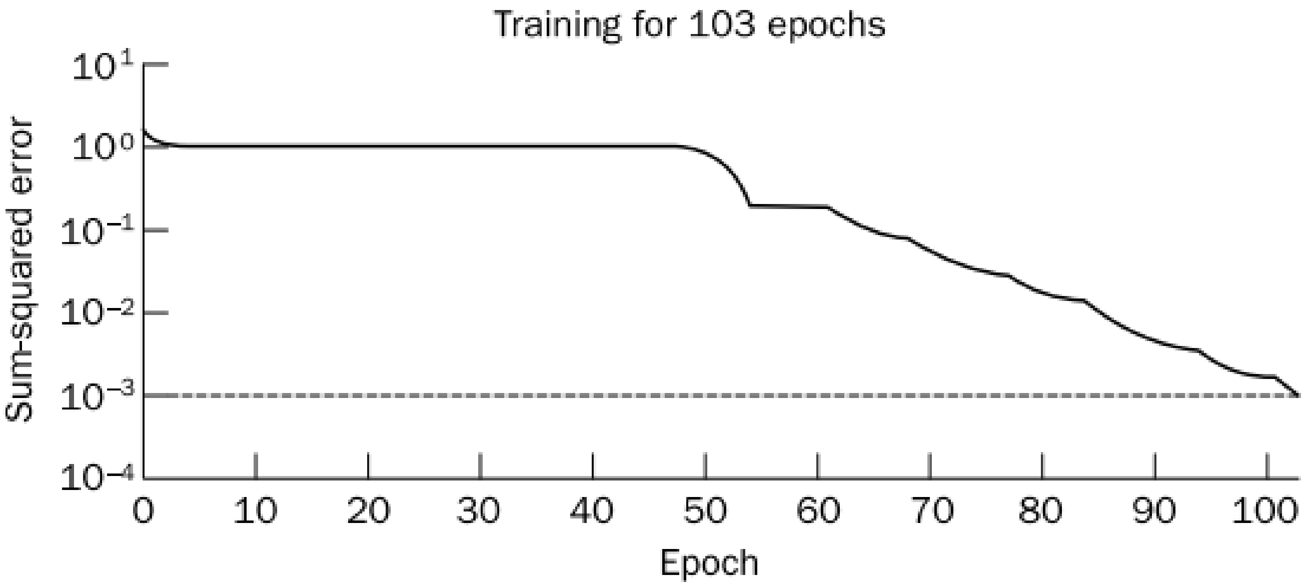
3. Adaptive learning rate

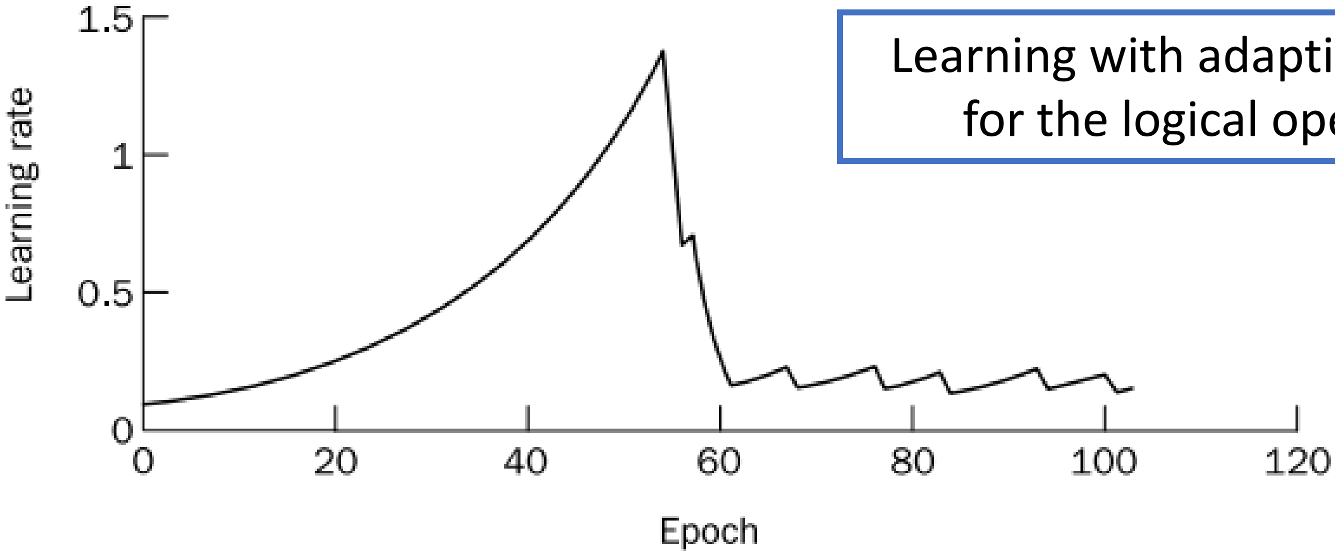
- Adjust the learning rate parameter $\boldsymbol{\eta}$ during training
  - Small η $\rightarrow$ small weight changes through iterations $\rightarrow$ smooth learning curve
  - Large η $\rightarrow$ speed up the training process with larger weight changes $\rightarrow$ possible instability and oscillatory
- Heuristic-like approaches for adjusting η
  - The algebraic sign of the SSE change remains for several consequent epochs $\rightarrow$ increase η.
  - The algebraic sign of the SSE change alternates for several consequent epochs $\rightarrow$ decrease η.
- One of the most effective acceleration means

Learning with momentum only



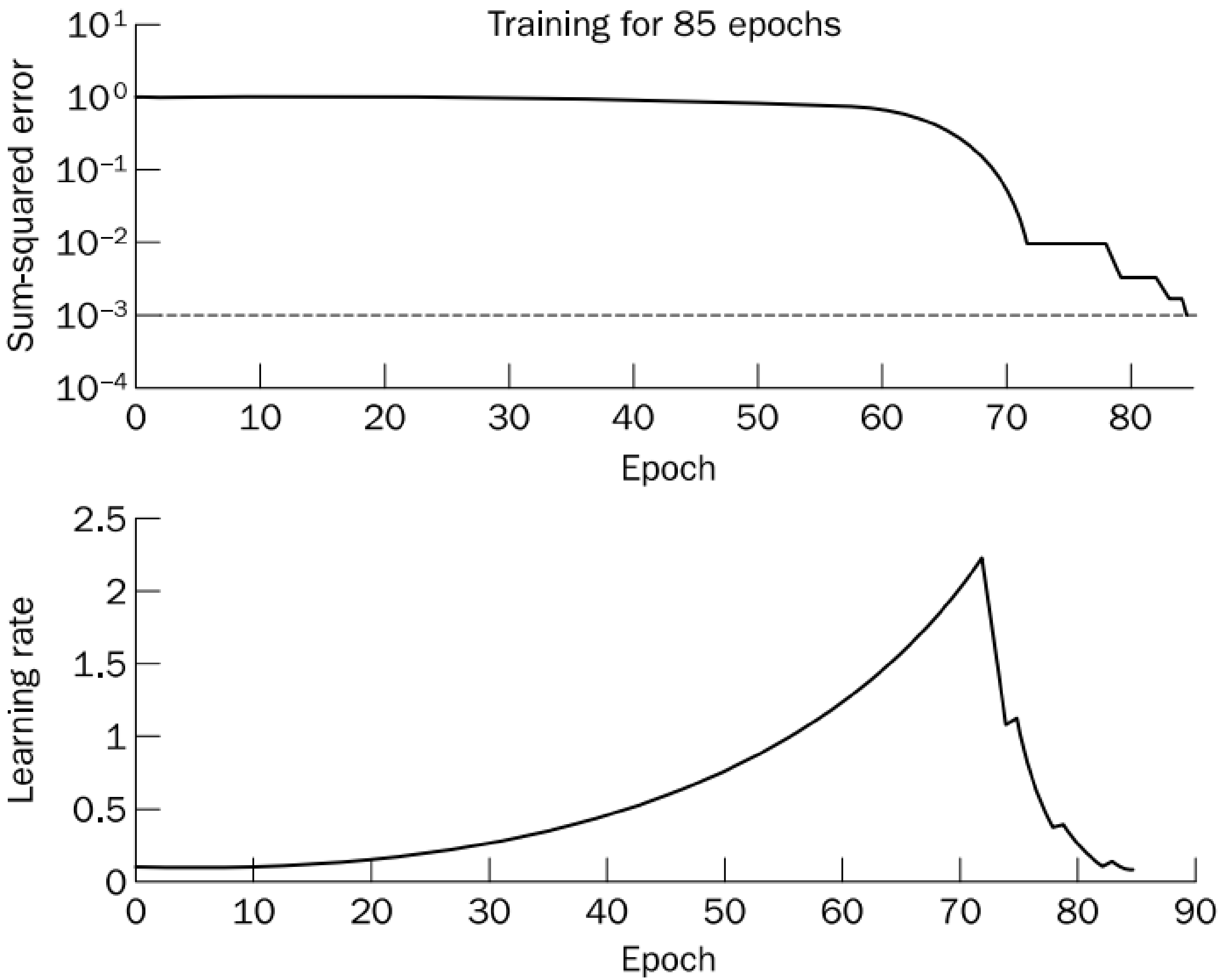
Learning with adaptive η only





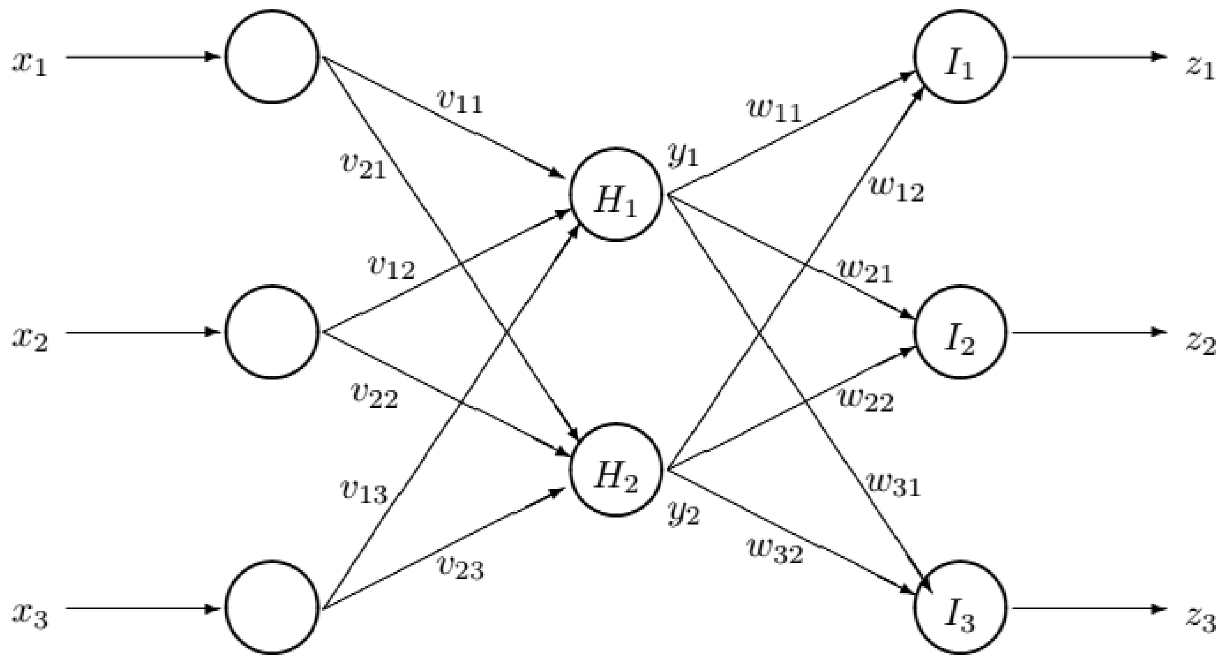
Learning with adaptive learning rate for the logical operation XOR.

Learning with adaptive η and momentum



Quiz 01: Multi-layer perceptron

- Consider the below feedforward network with one hidden layer of units.



- If the network is tested with an input vector $x = [1.0, 2.0, 3.0]$ then what are *the activation* H_1 *of the first hidden neuron* and the *activation* I_3 *of the third output neuron*?

Quiz 02: Forward the input signals

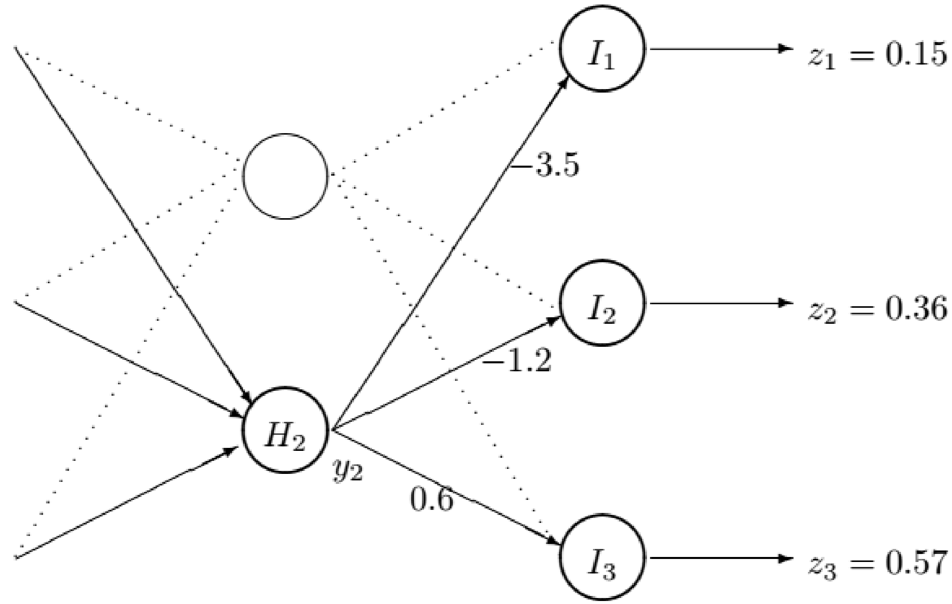
- The input vector to the network is $x = [x_1, x_2, x_3]^T$
- The vector of hidden layer outputs is $y = [y_1, y_2]^T$
- The vector of actual outputs is $z = [z_1, z_2, z_3]^T$
- The vector of desired outputs is $t = [t_1, t_2, t_3]^T$
- The network has the following weight vectors

$$v_1 = \begin{bmatrix} -2.0 \\ 2.0 \\ -2.0 \end{bmatrix} \qquad v_2 = \begin{bmatrix} 1.0 \\ 1.0 \\ -1.0 \end{bmatrix} \qquad w_1 = \begin{bmatrix} 1.0 \\ -3.5 \end{bmatrix} \qquad w_2 = \begin{bmatrix} 0.5 \\ -1.2 \end{bmatrix} \qquad w_3 = \begin{bmatrix} 0.3 \\ 0.6 \end{bmatrix}$$

- Assume that all units use sigmoid activation function and zero biases.

Quiz 03: Backpropagation error signals

- The figure shows part of the network described in the previous slide.
- Use the same weights, activation functions and bias values as described.



- A new input pattern is presented to the network and training proceeds as follows. The actual outputs are given by $z = [0.15, 0.36, 0.57]^T$ and the corresponding target outputs are given by $t = [1.0, 1.0, 1.0]^T$.
- The weights w_{12}, w_{22} and w_{32} are also shown.
- What is the error for each of the output units?

Acknowledgements

- Some parts of the slide are adapted from
  - Derivation: Error Backpropagation & Gradient Descent for Neural Networks (github.io link)
  - Negnevitsky, Michael. Artificial intelligence: A guide to intelligent systems. Pearson, 2005. Chapter 6.


...the end.