



دانشگاه کردستان
University of Kurdistan
زانکۆی کوردستان

Department of Computer Engineering
University of Kurdistan

Neural Networks (Graduate level)
Single layer and multi layer perceptron
(Supervised learning)







By: Dr. Alireza Abdollahpouri



University of Kurdistan

Classification- Supervised learning

Given a **Training Set** of examples (pattern, label) the objective is to “learn” to classify a new pattern with the correct label.

	pattern	label
Example 1		cat
Example 2		not cat
Example 3		cat
Example 4		not cat
Example 5		not cat
Example 6		cat



cat

$$TS = \{(\mathbf{x}_k, t_k), k = 1, \dots, M\}$$

$$\mathbf{x}_k = \begin{pmatrix} x_{k1} \\ x_{k2} \\ \vdots \\ x_{kn} \end{pmatrix}$$

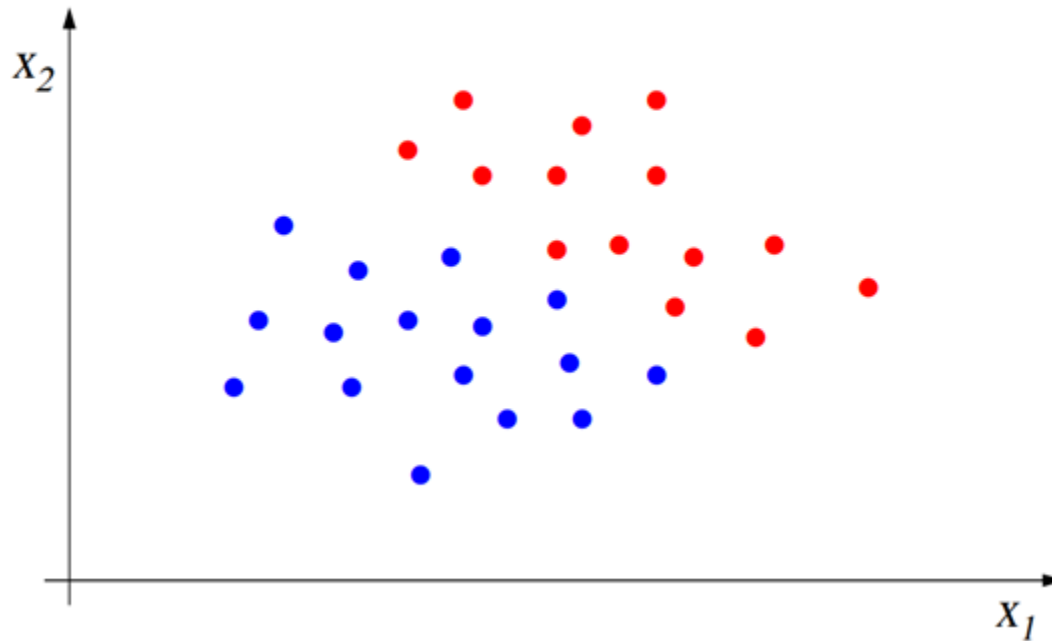
$$\mathbf{x}_k \in R^n$$

$$t_k \in [0,1]$$

Classification

Basically we want our system to **classify a set of patterns** as belonging to a given **class** or not.

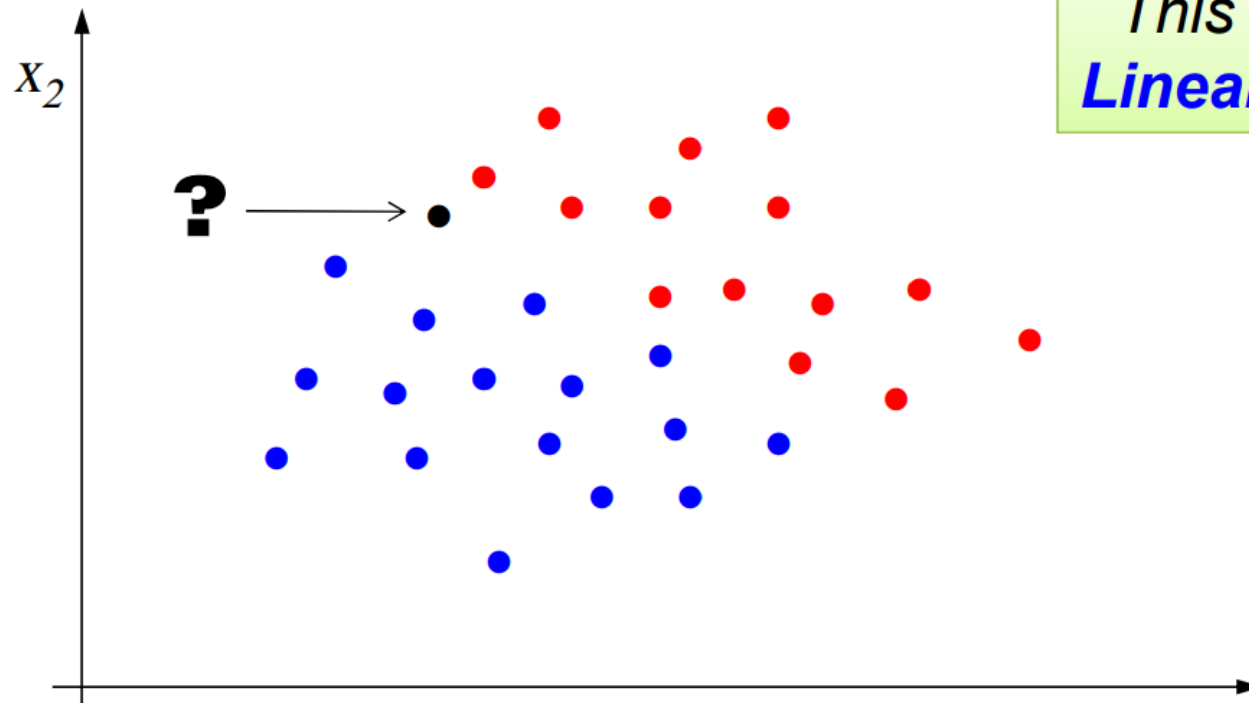
If $x \in \mathbf{R}^2$ we can represent the situation on a plane:



Classification

*A simple approach to classify a new pattern is to look at the **closest neighbor** and return its label.*

A better way is to find a line that best separates the data set:

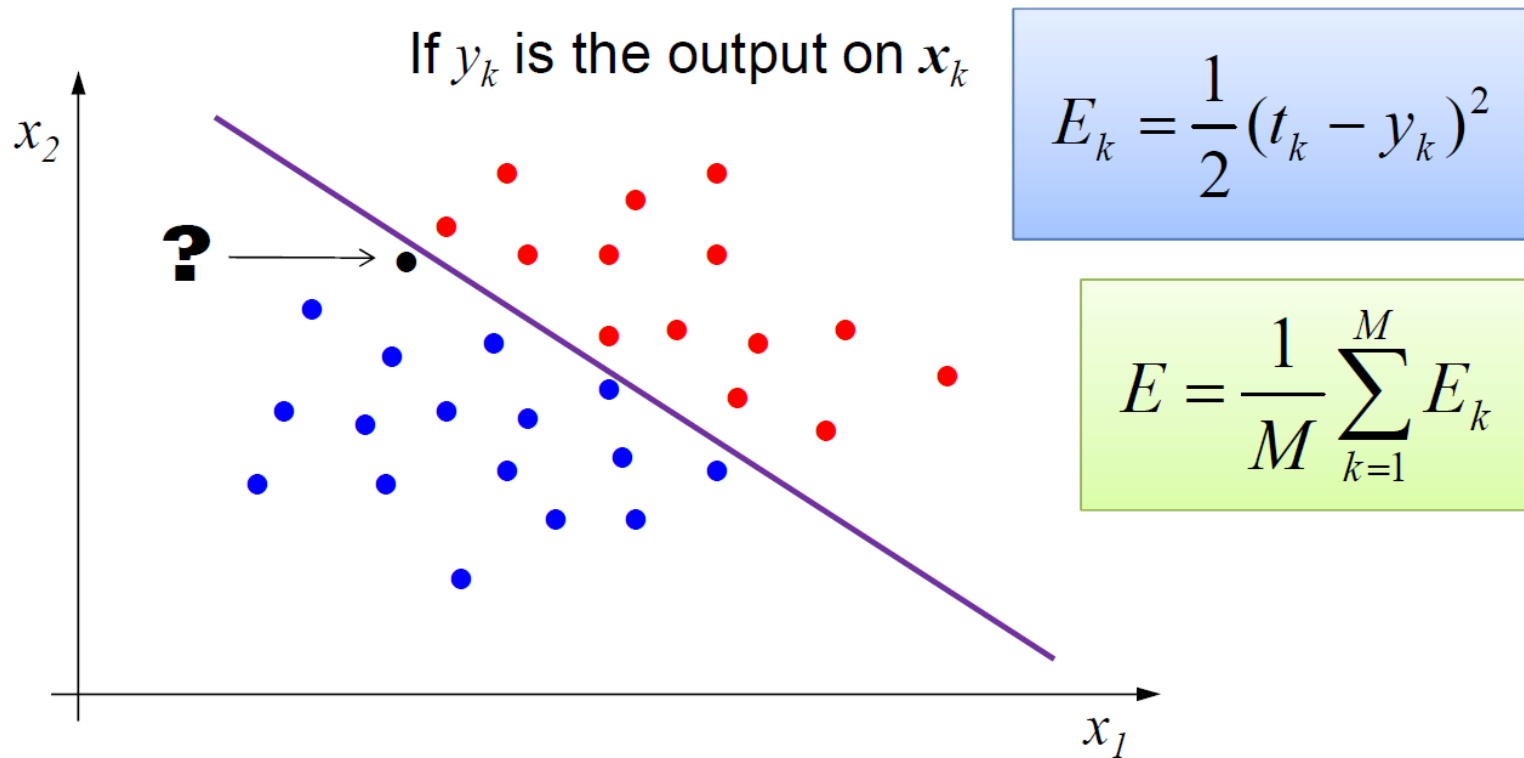


*This is called a
Linear Classifier*

Linear Classifier

GOAL: identify the line that “**best separates**” the two data sets.

Error: the quality of separation is measured by a loss function that measures the error of the classifier:



Supervised learning

In a supervised learning paradigm, a neural network operates in two distinct phases:

1. A learning phase

*The network is trained to classify the examples in the Training Set (**weights are modified based on errors**).*

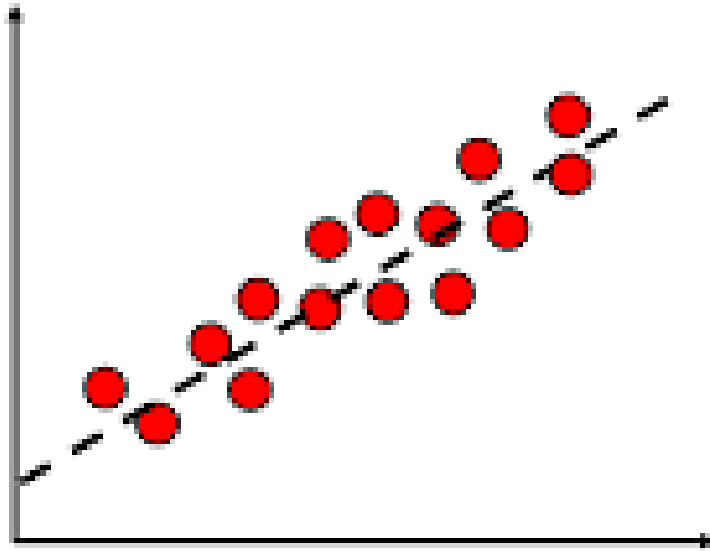
2. An operating phase

*The network is used on new data never seen before (**weights are kept fixed**).*



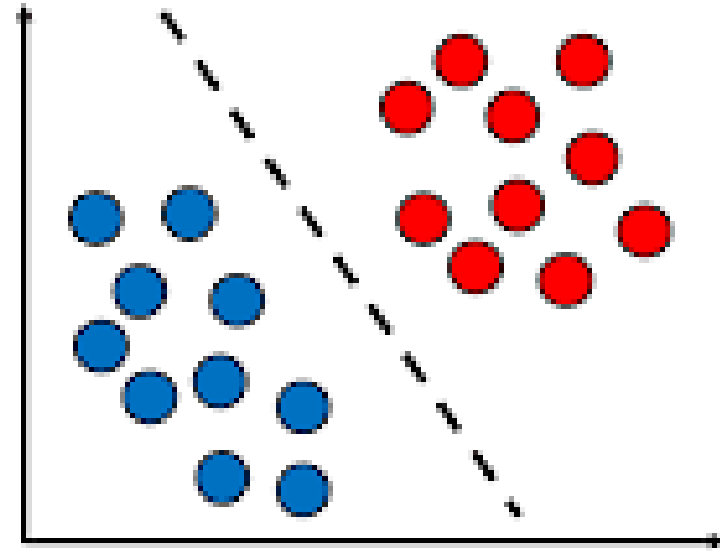
Supervised learning

Regression



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

Classification

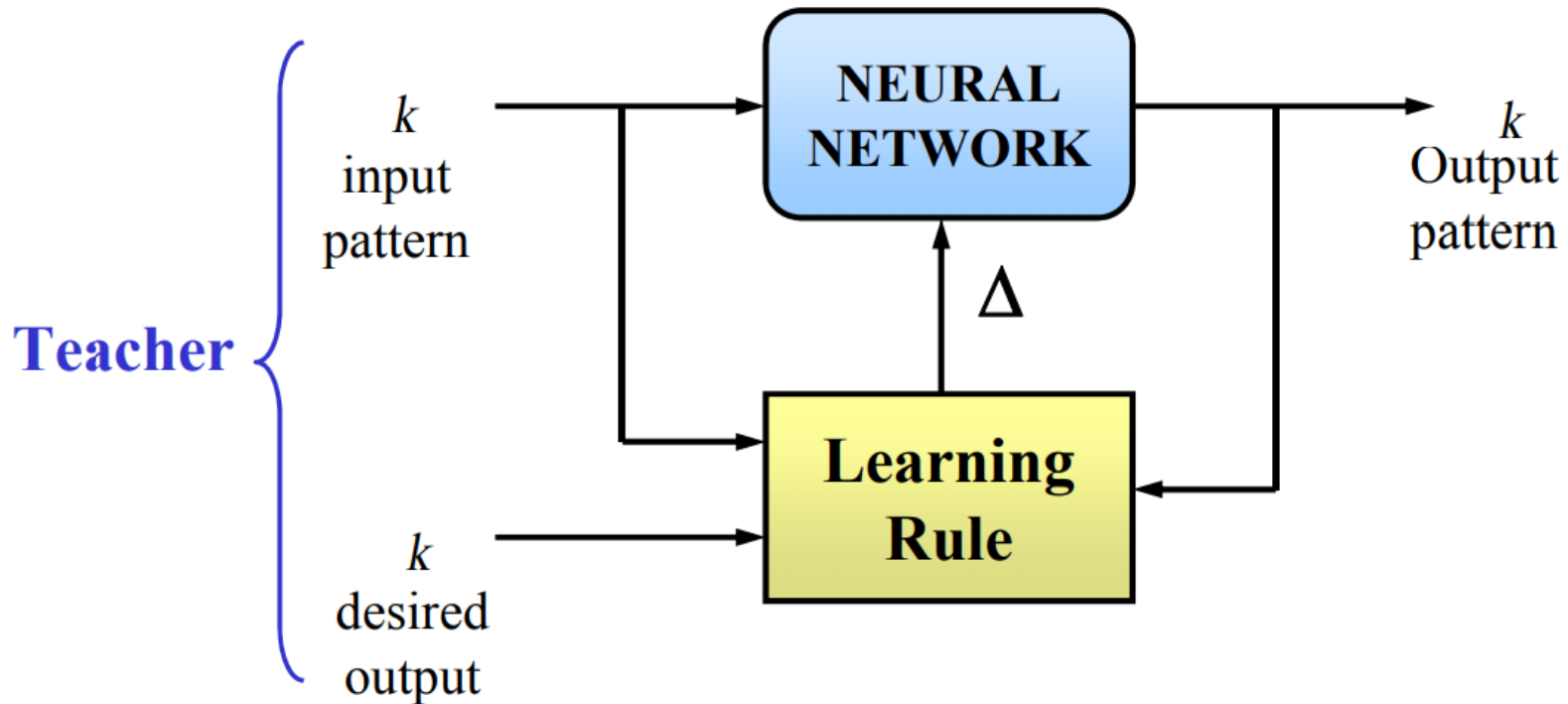


$$\text{Log Loss} = -\frac{1}{N} \sum_{i=1}^N y_i \log \hat{y}_i + (1-y_i) \log (1-\hat{y}_i)$$

$$\text{Loss} = -\sum_{j=1}^K y_j \log(\hat{y}_j)$$

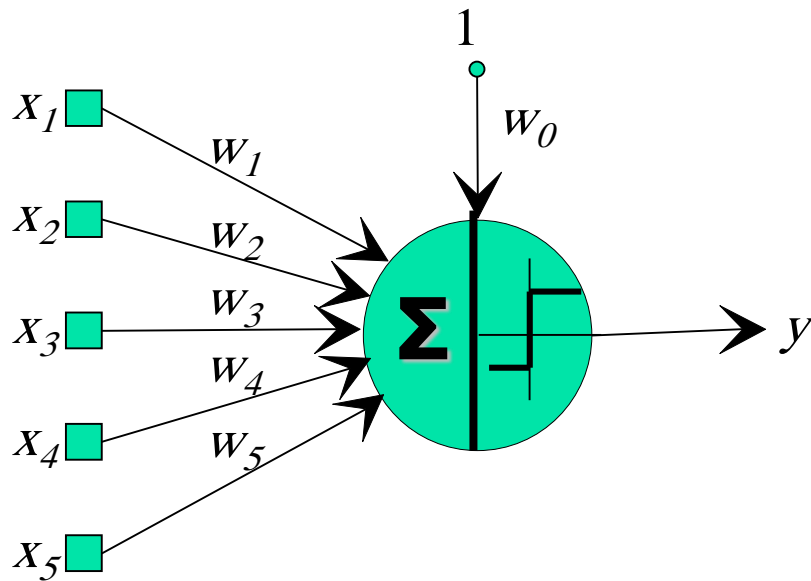
where k is number of classes in the data

Learning phase



The Perceptron

The first model of a biological neuron

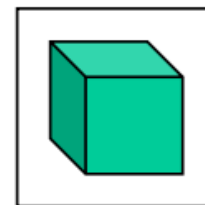
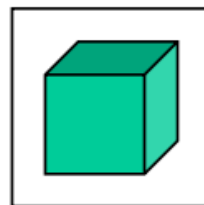
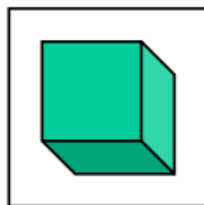
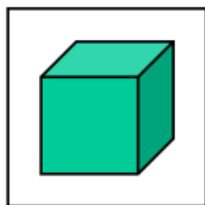


$$y = \text{sgn} \left(\sum_{i=1}^n w_i x_i + w_0 \right)$$

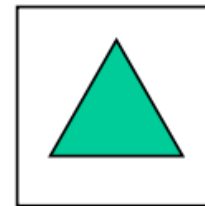
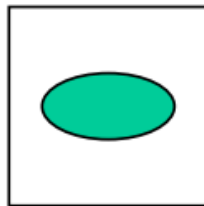
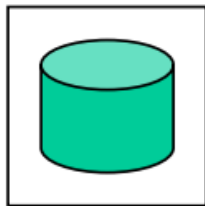
Perceptron for Classification

- A perceptron can be trained to recognize whether an input pattern X belongs or not to a class C :

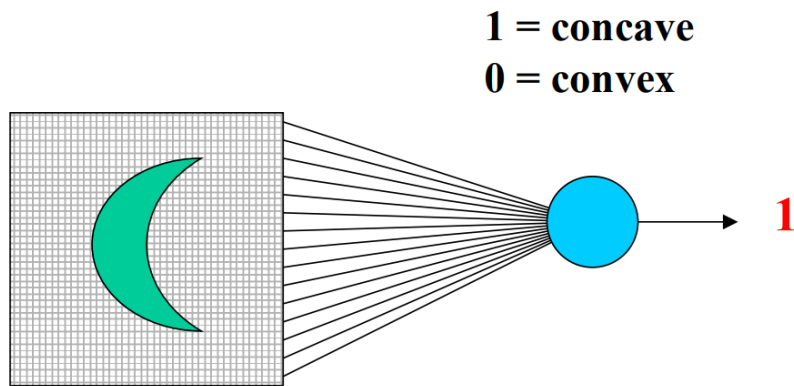
CUBE
(1)



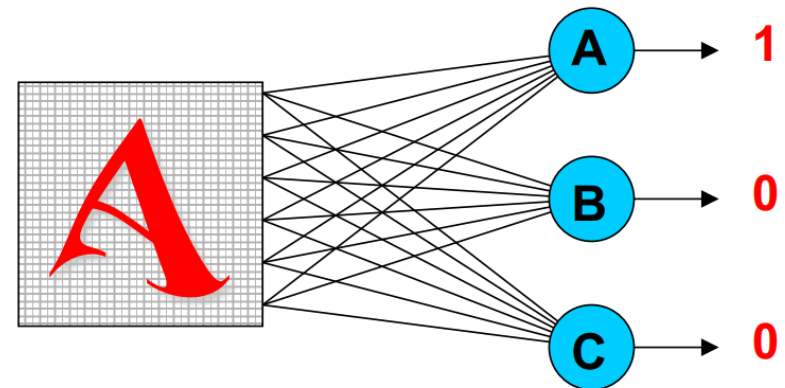
NOT
CUBE
(0)



Classification



Simple case
(two classes – one output neuron)

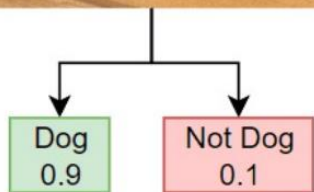


General case
(multiple classes – Several output neurons)

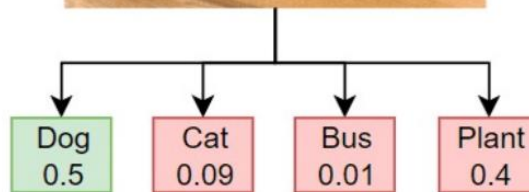
3 Types of Classification

- Binary Classification
- Multi-class Classification
- Multi-label Classification

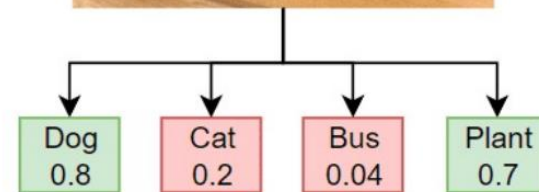
Binary Classification



Multiclass Classification



Multilabel Classification



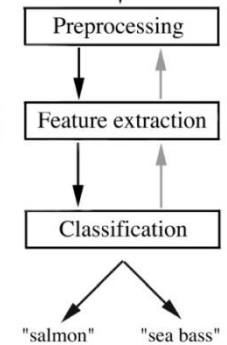
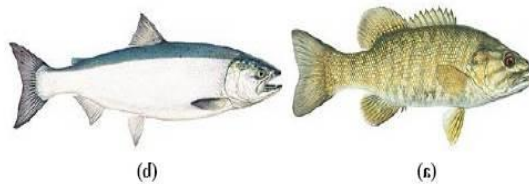
Complexity of PR – An Example

Problem: Sorting incoming fish on a conveyor belt.

Assumption: Two kind of fish:

(1) sea bass

(2) salmon



salmon

sea bass

salmon

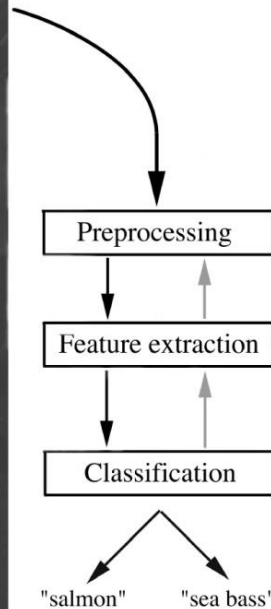
salmon

sea bass

sea bass



Pre-processing Step



Example

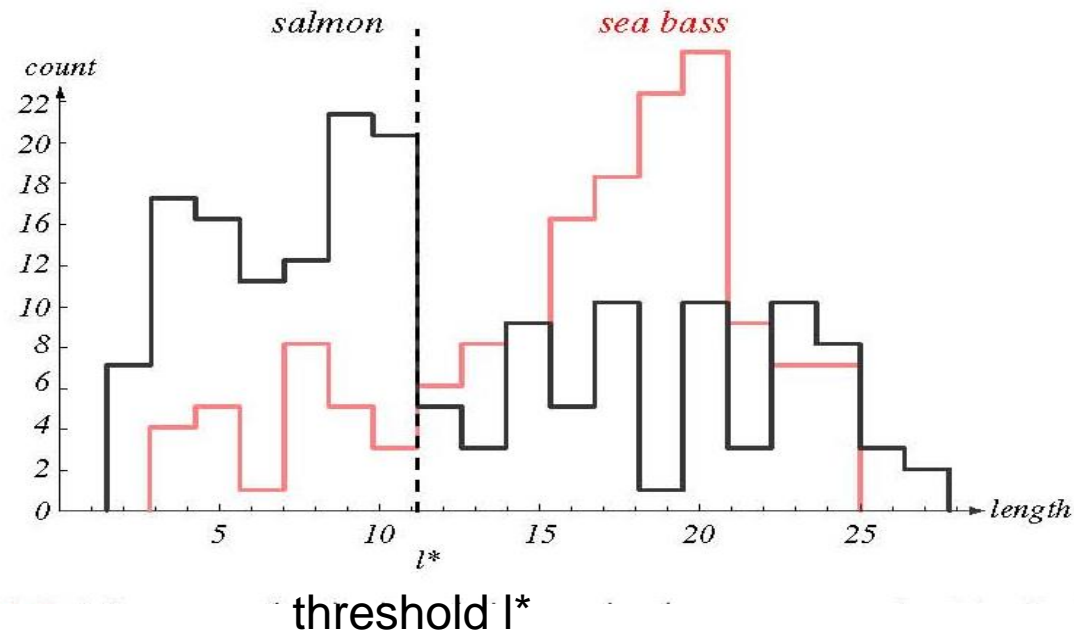
- (1) Image enhancement
- (2) Separate touching or occluding fish
- (3) Find the boundary of each fish

Feature Extraction

- Assume a fisherman told us that a sea bass is generally **longer** than a salmon.
- We can use **length** as a feature and decide between sea bass and salmon according to a **threshold** on length.
- **How** should we choose the threshold?



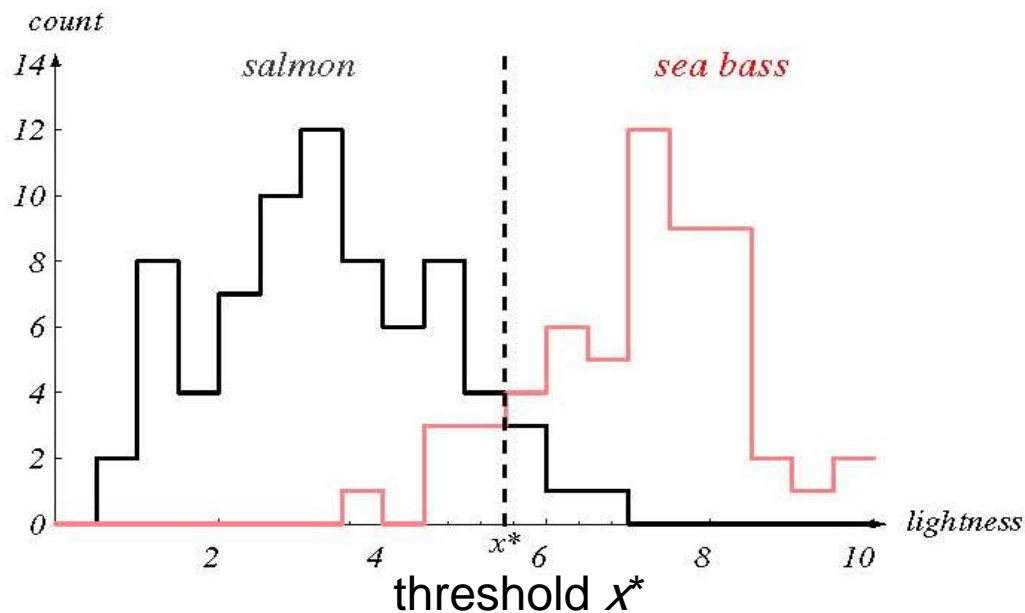
"Length" Histograms



- Even though sea bass is longer than salmon on the average, there are many examples of fish where this observation does not hold.

"Average Lightness" Histograms

- Consider a different feature such as "average lightness"



- It seems easier to choose the threshold x^* but we still cannot make a perfect decision.

Multiple Features

- To improve recognition accuracy, we might have to use more than one features at a time.
 - Single features might not yield the best performance.
 - Using combinations of features might yield better performance.

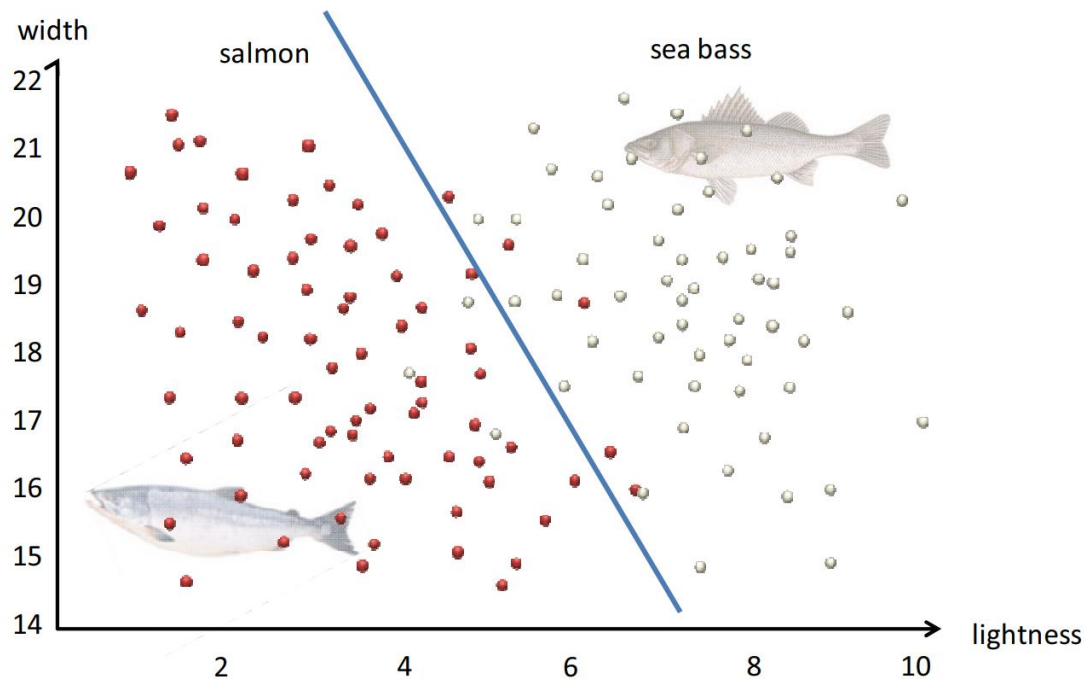
$$\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \begin{matrix} x_1 : \textit{lightness} \\ x_2 : \textit{width} \end{matrix}$$

- **How** many features should we choose?



Classification

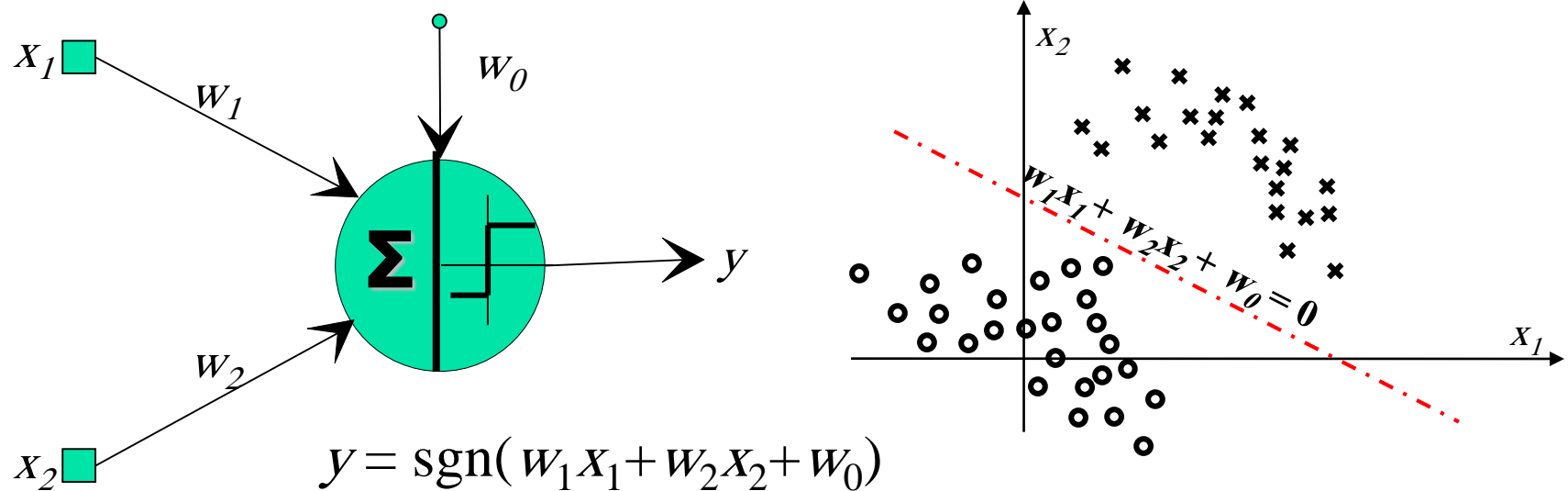
- Partition the *feature space* into two regions by finding the **decision boundary** that minimizes the error.



- How** should we find the optimal decision boundary?

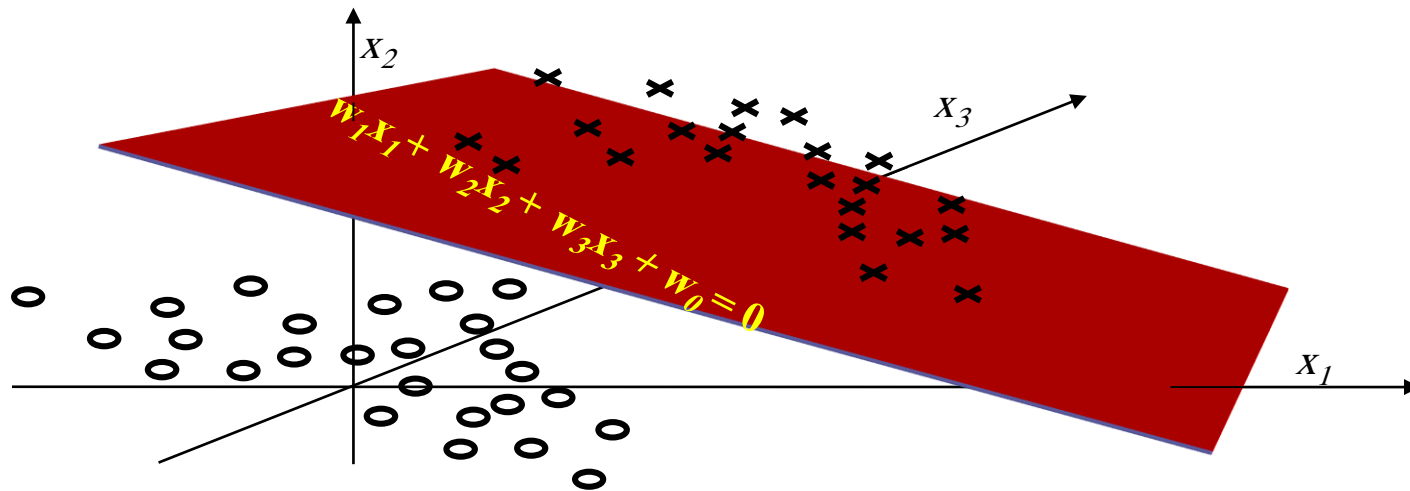
What a Perceptron does?

For a perceptron with 2 input variables namely x_1 and x_2
Equation $W^T X = 0$ determines a line separating positive from negative examples.



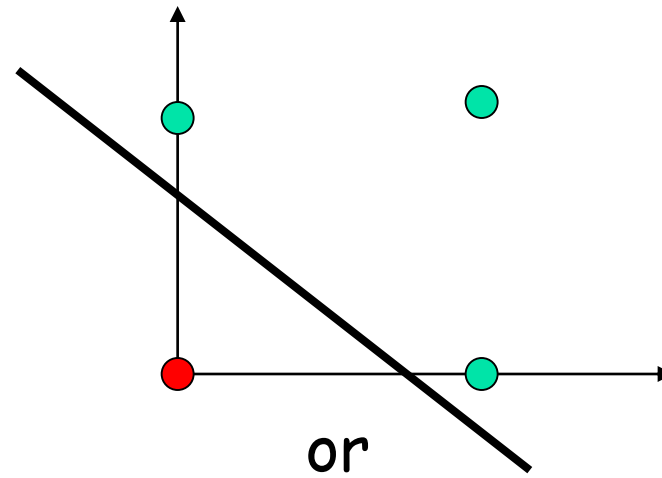
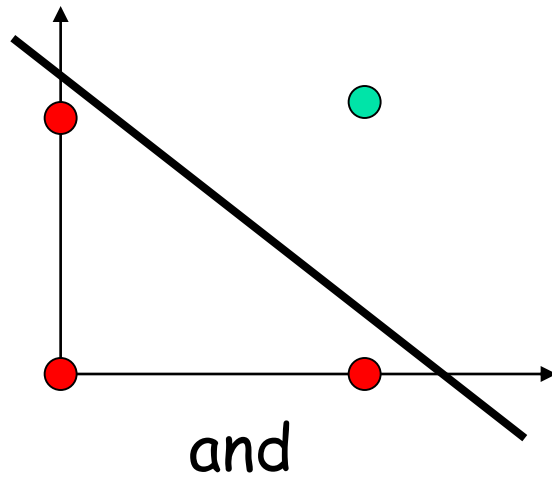
What a Perceptron does?

For a perceptron with n input variables, it draws a Hyper-plane as the decision boundary over the (n -dimensional) input space. It classifies input patterns into two classes.



The perceptron outputs 1 for instances lying on one side of the hyperplane and outputs -1 for instances on the other side.

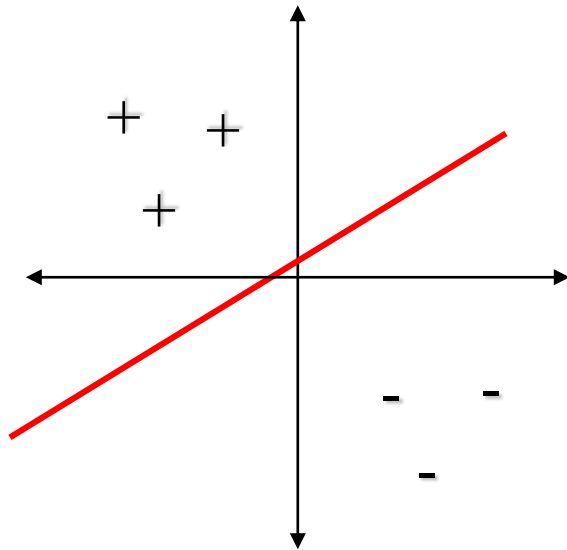
What can be represented using Perceptrons?



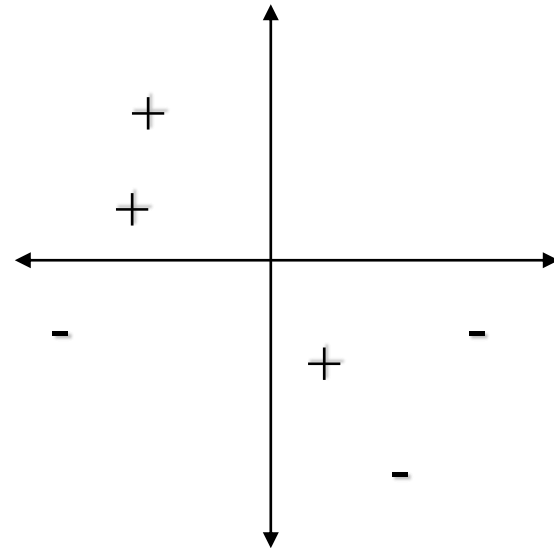
Representation Theorem: perceptrons can only represent linearly separable functions.
Examples: AND, OR, NOT.

Limits of the Perceptron

A perceptron can learn only examples that are called “linearly separable”. These are examples that can be perfectly separated by a hyperplane.



Linearly separable



Non-linearly separable

Learning Perceptrons

- Learning is a process by which the free parameters of a neural network are adapted through a process of stimulation by the environment in which the network is embedded. The type of learning is determined by the manner in which the parameters changes take place.
- In the case of Perceptrons, we use a supervised learning.
- Learning a perceptron means **finding the right values for W** that satisfy the input examples $\{(input_i, target_i)^*\}$
- The hypothesis space of a perceptron is the space of all weight vectors.

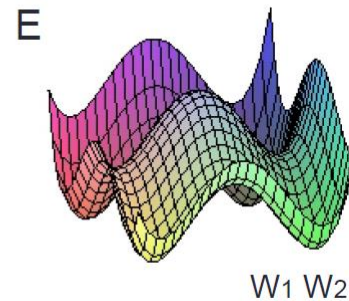
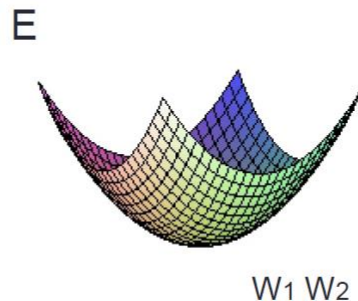
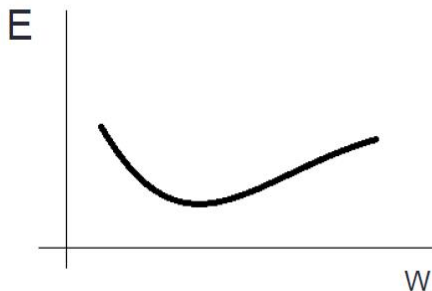


How to find the weights?

We want to find a set of weights that enable our perceptron to correctly classify our data.

We can define a cost function: $E = \frac{1}{2}(y-g(z))^2$ $\left\{ \begin{array}{l} y: \text{ is the correct output} \\ g(z): \text{ is the actual output} \end{array} \right.$

- we know $\partial E / \partial w_i$ then we can search for a minimum of E in **weight space**



Learning Perceptrons

Principle of learning using the perceptron rule:

1. A set of training examples is given: $\{(x, t)^*\}$ where x is the input and t the target output [supervised learning]
2. Examples are presented to the network.
3. For each example, the network gives an output \mathbf{o} .
4. If there is an error, the hyperplane is moved in order to correct the output error.
5. When all training examples are correctly classified, Stop learning.



Learning Perceptrons

More formally, the algorithm for learning Perceptrons is as follows:

1. Assign random values to the weight vector
2. Apply the **perceptron rule** to every training example
3. Are all training examples correctly classified?

Yes. Quit

No. Go Back to Step 2.



Perceptron Training Rule

The perceptron training rule:

For a new training example $[X = (x_1, x_2, \dots, x_n), t]$
update each weight according to this rule:

$$w_i = w_i + \Delta w_i$$

Where $\Delta w_i = \eta \cdot (t - o) \cdot x_i$

t: target output

o: output generated by the perceptron

η : constant called the **learning rate** (e.g., 0.1)



Perceptron Training Rule

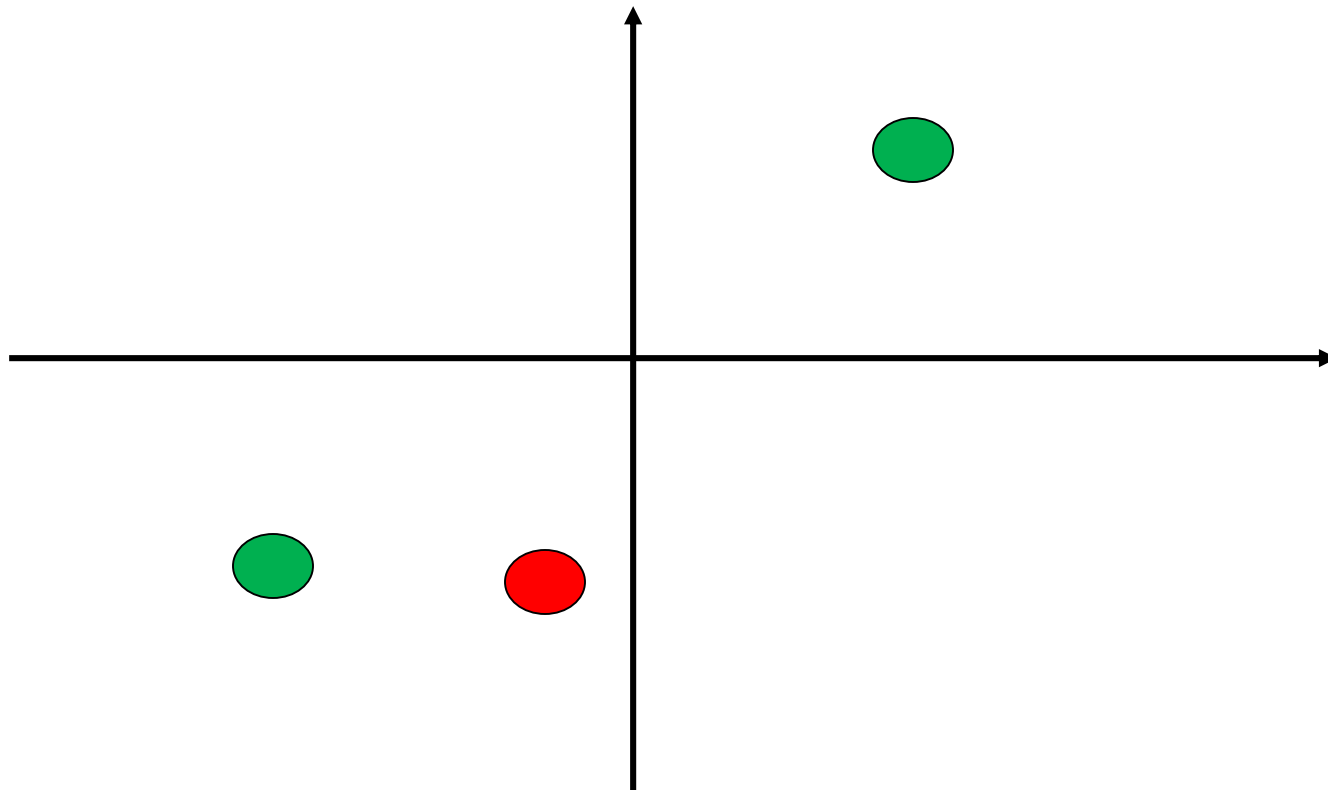
Comments about the perceptron training rule:

- If the example is correctly classified the term $(t-o)$ equals zero, and no update on the weight is necessary.
- If the perceptron outputs -1 and the real answer is 1 , the weight is increased.
- If the perceptron outputs a 1 and the real answer is -1 , the weight is decreased.
- Provided the examples are linearly separable and a small value for η is used, the rule is proved to classify all training examples correctly.



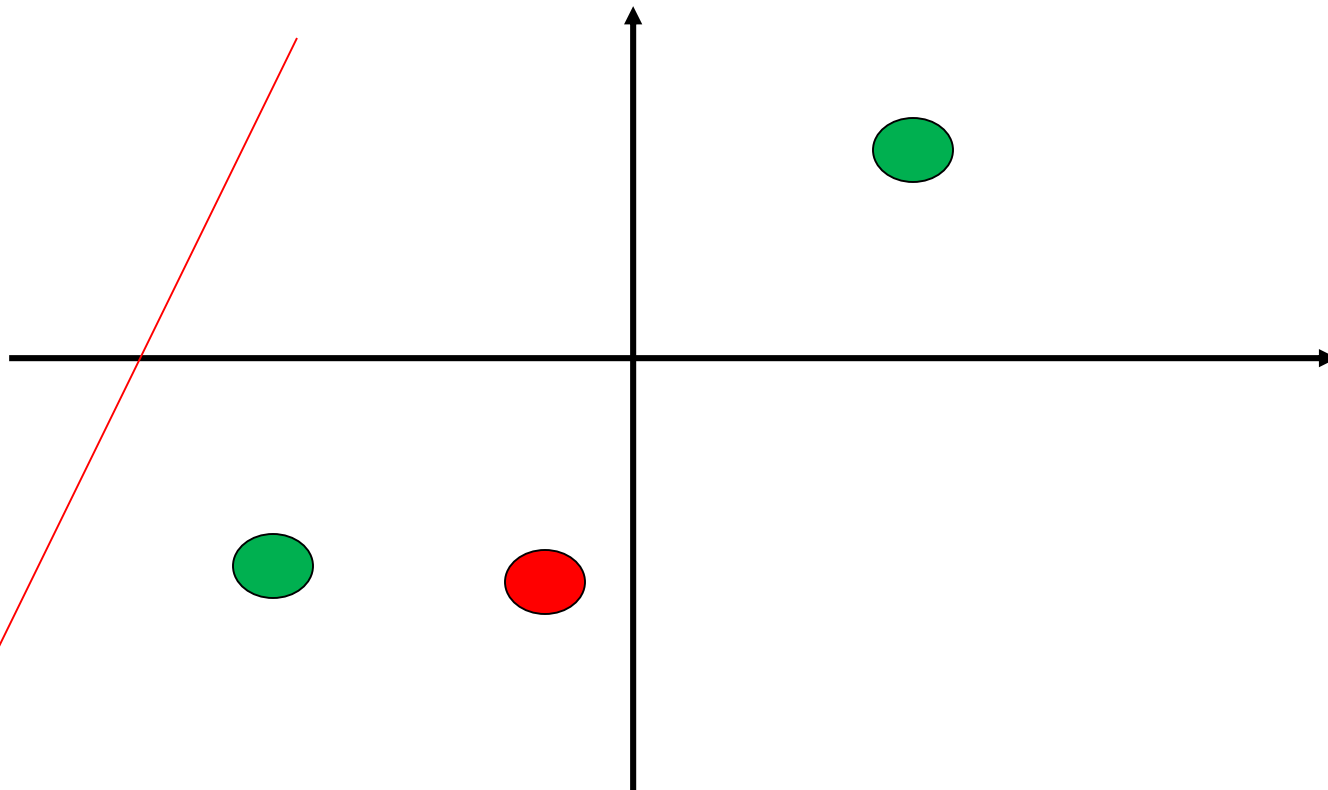
Perceptron Training Rule

Consider the following example: (two classes: Red and Green)



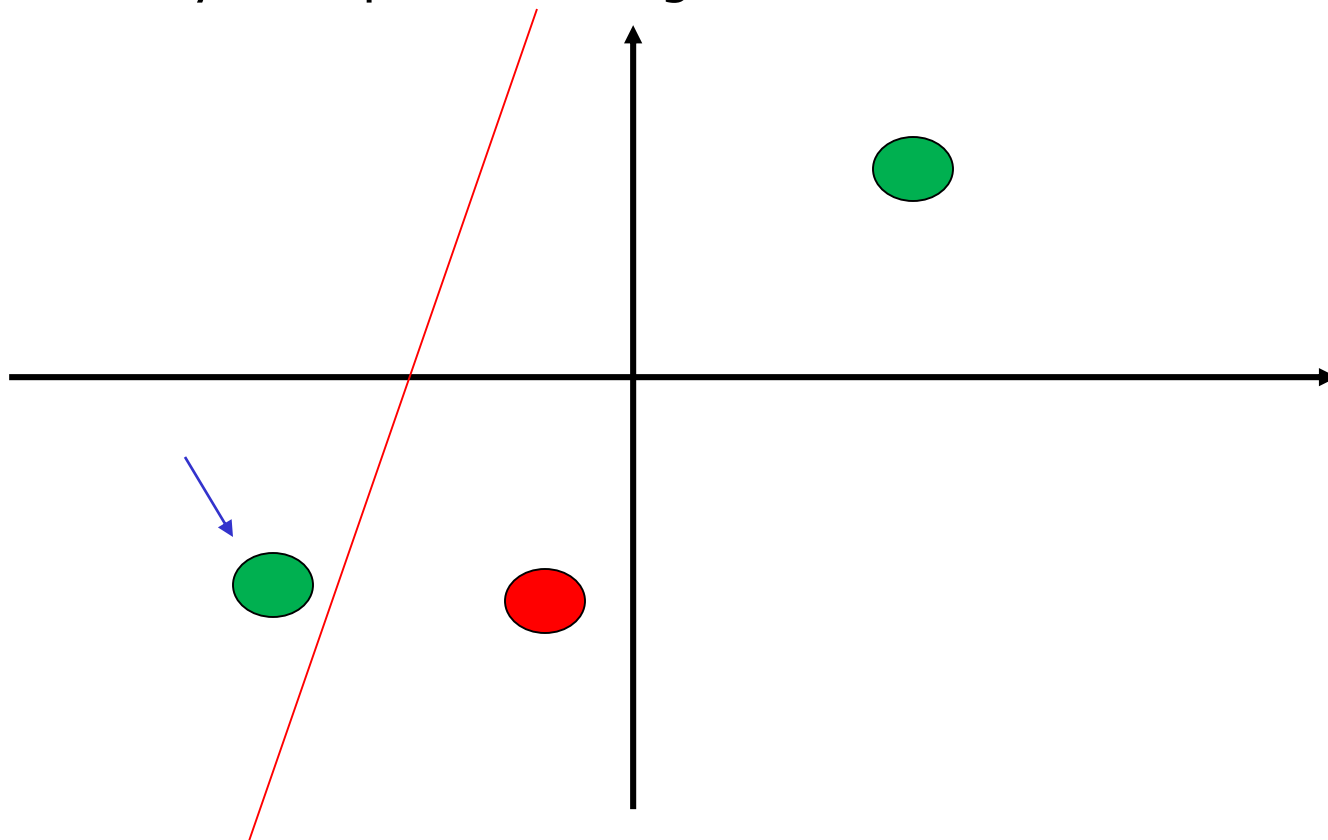
Perceptron Training Rule

Random Initialization of perceptron weights ...



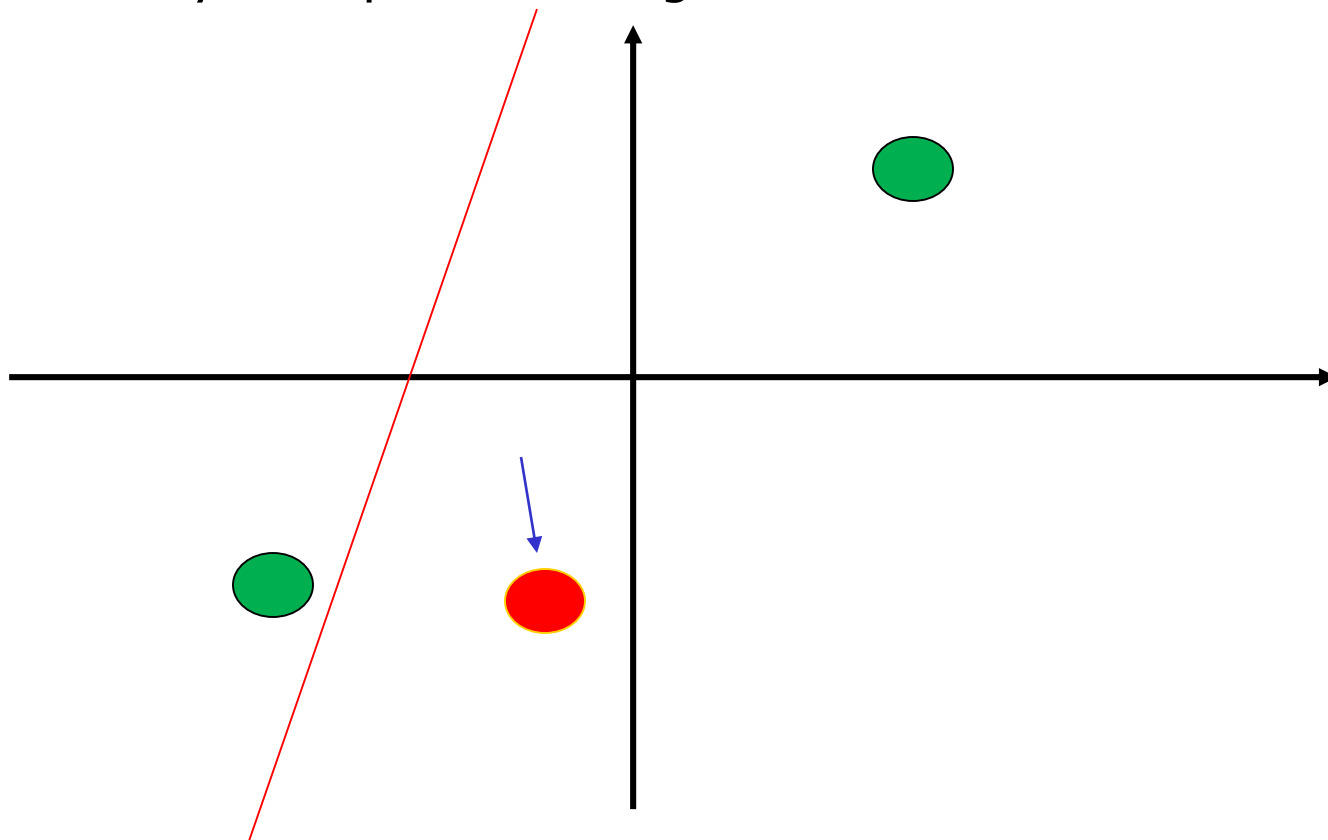
Perceptron Training Rule

Apply Iteratively Perceptron Training Rule on the different examples:



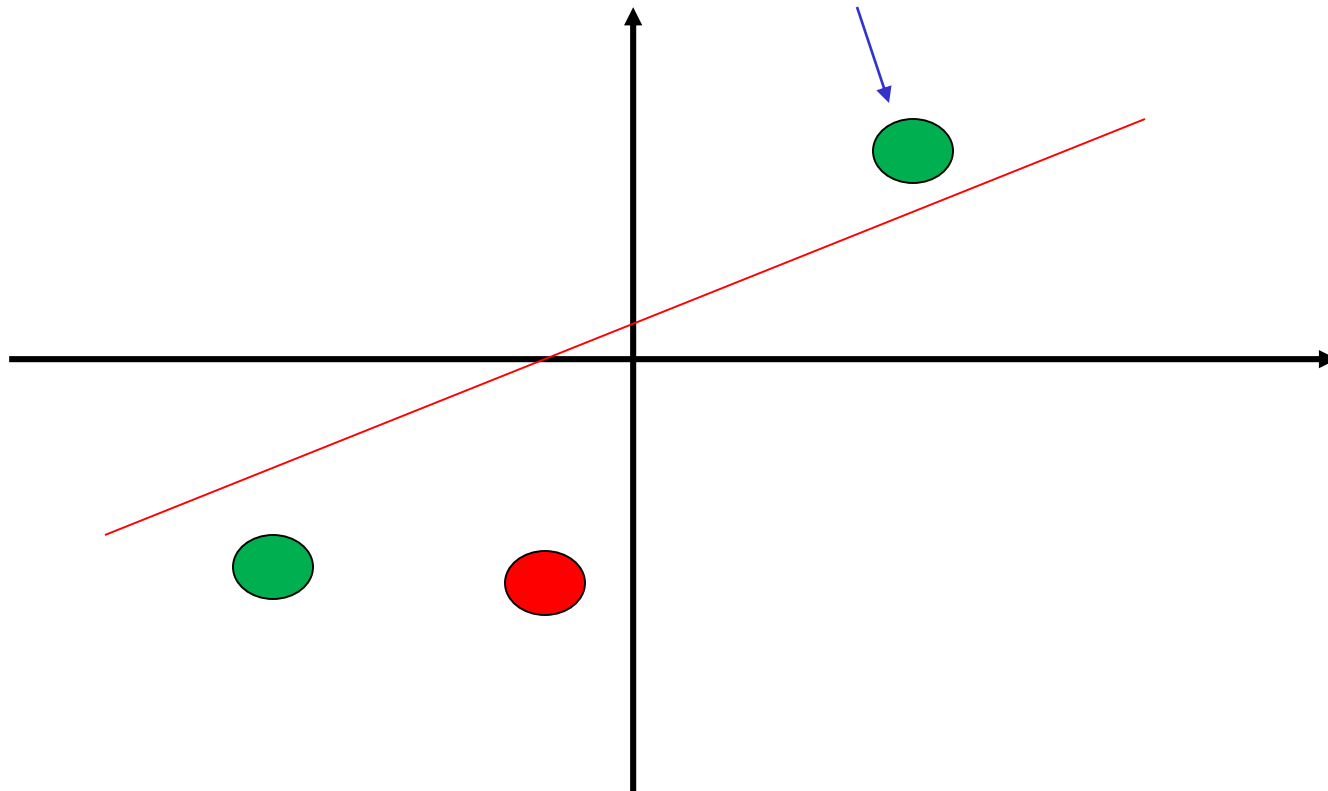
Perceptron Training Rule

Apply Iteratively Perceptron Training Rule on the different examples:



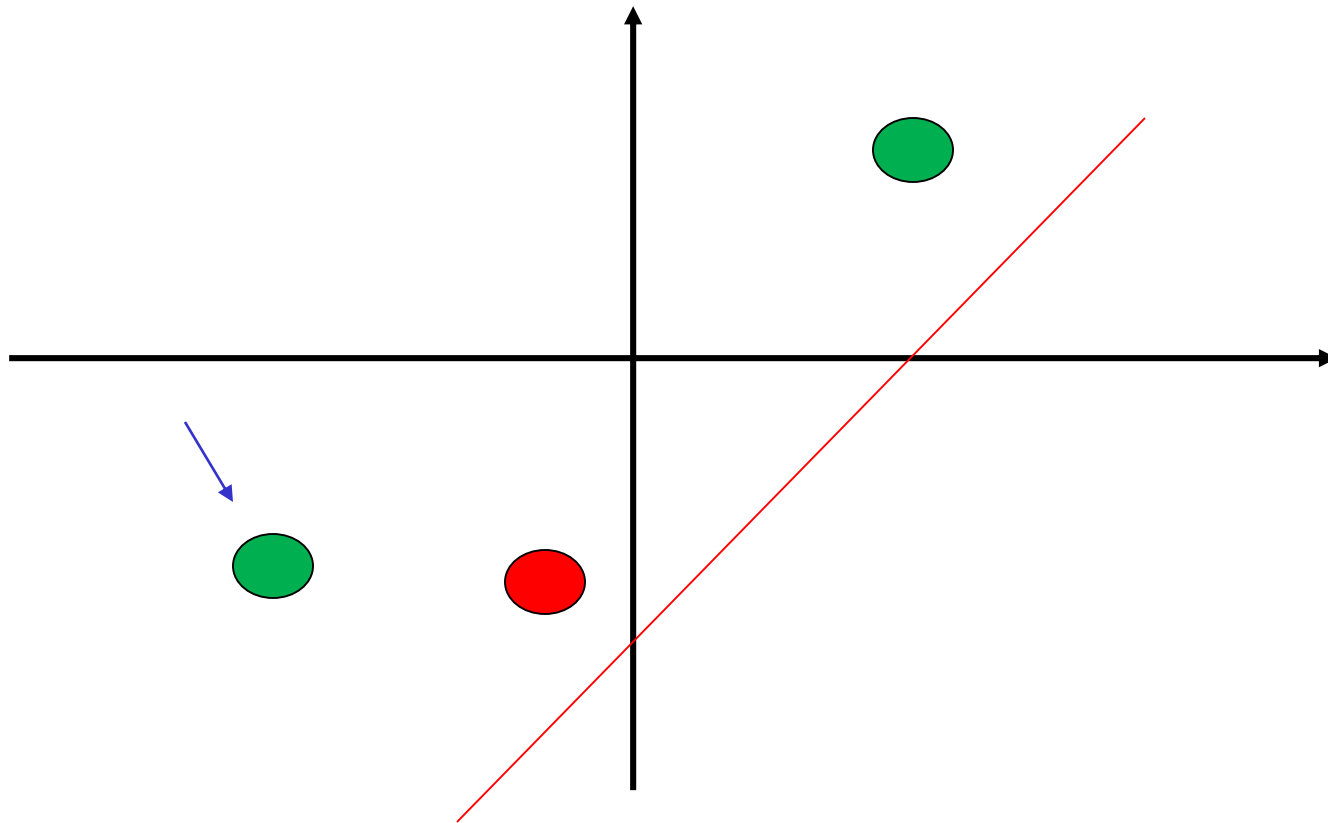
Perceptron Training Rule

Apply Iteratively Perceptron Training Rule on the different examples:



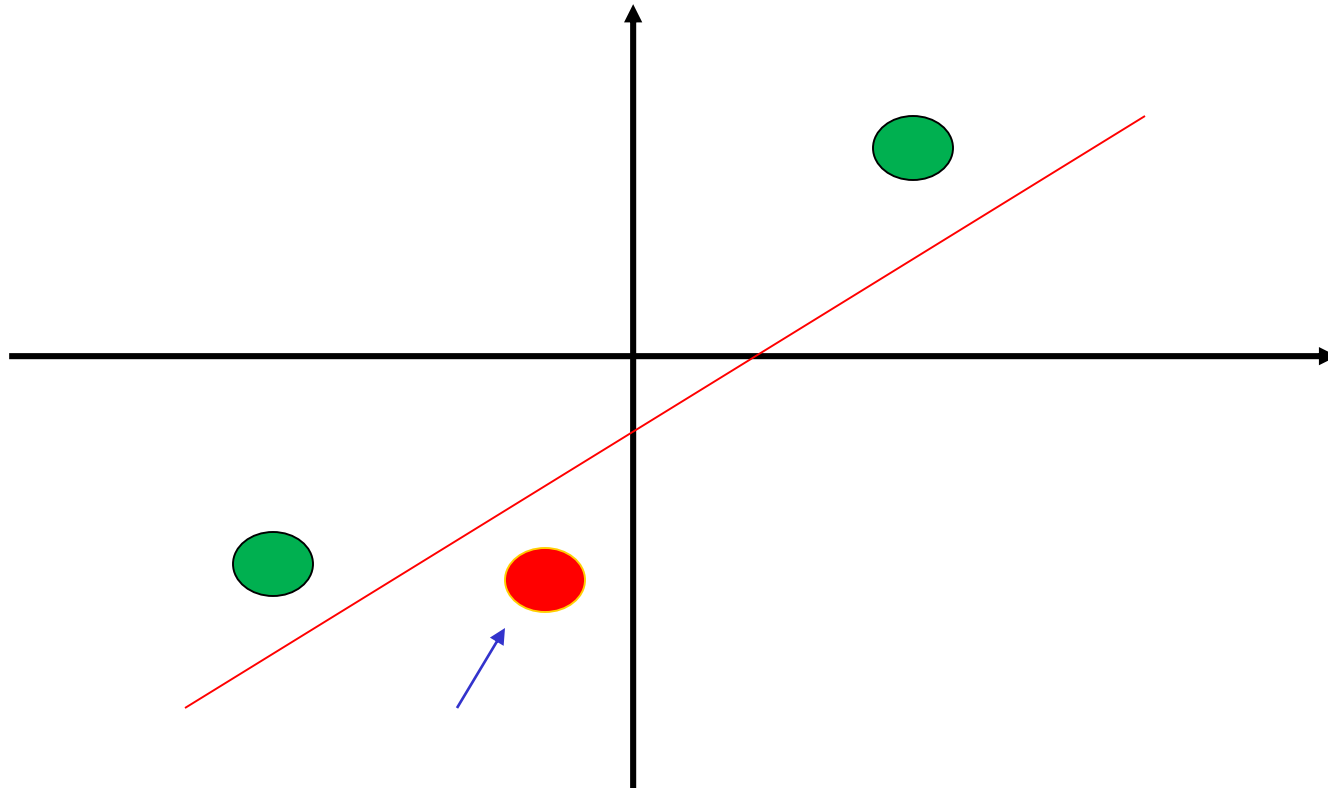
Perceptron Training Rule

Apply Iteratively Perceptron Training Rule on the different examples:



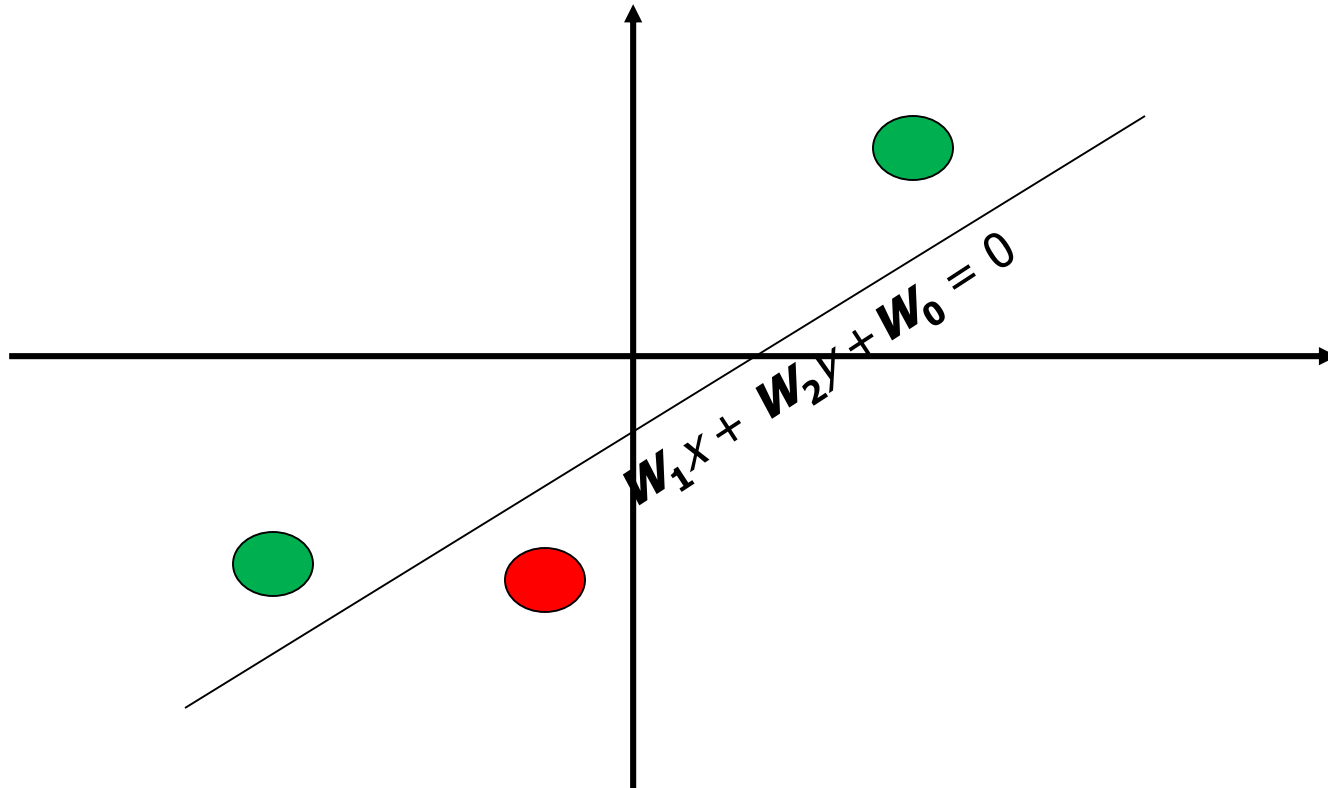
Perceptron Training Rule

All examples are correctly classified ... stop Learning



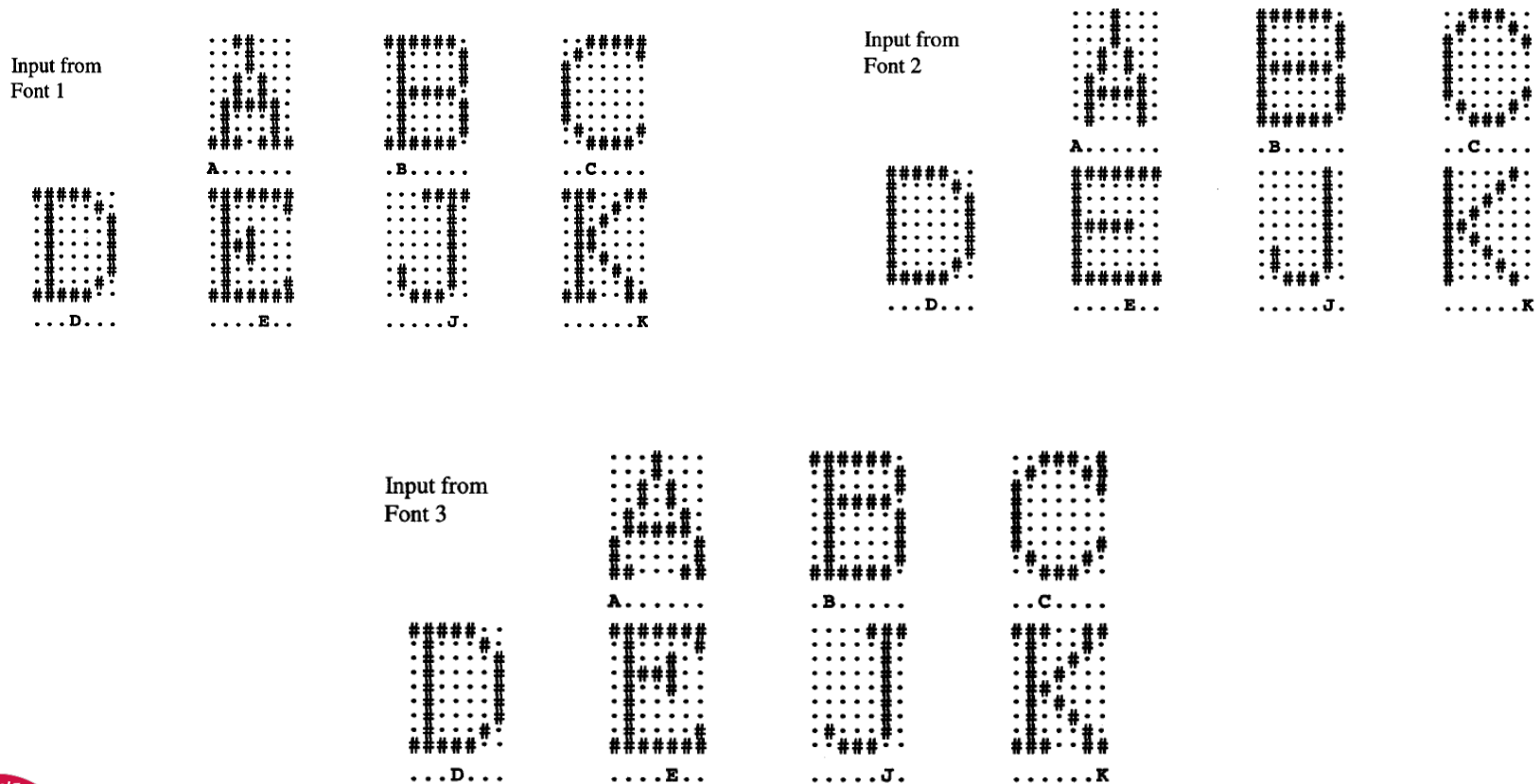
Perceptron Training Rule

The straight line $w_1x + w_2y + w_0 = 0$ separates the two classes



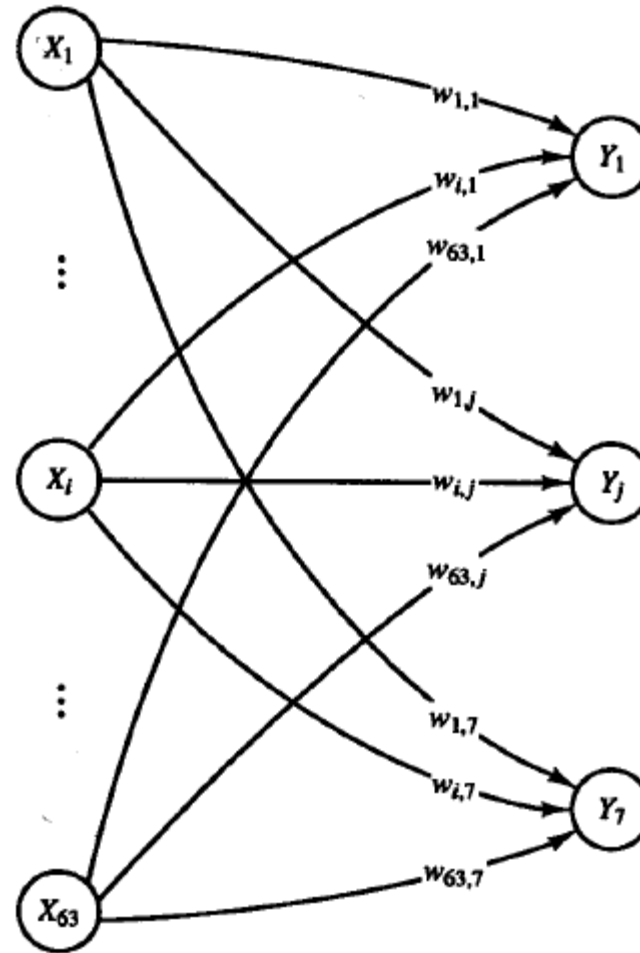
Example

- We want to classify the following 21 characters written by 3 fonts into 7 classes.



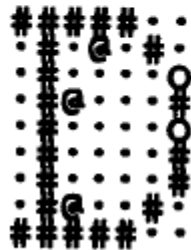
Example

Single-layer network

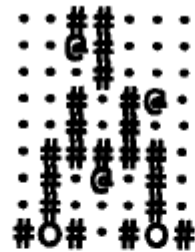


Example

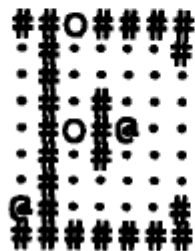
Input from
Font 1



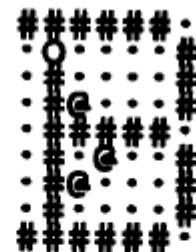
. . . D . . .



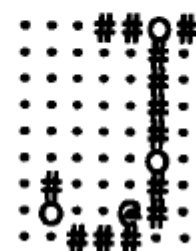
. . . A



. . . E . . .



. . . B



. J . . .



. . . C . E . K



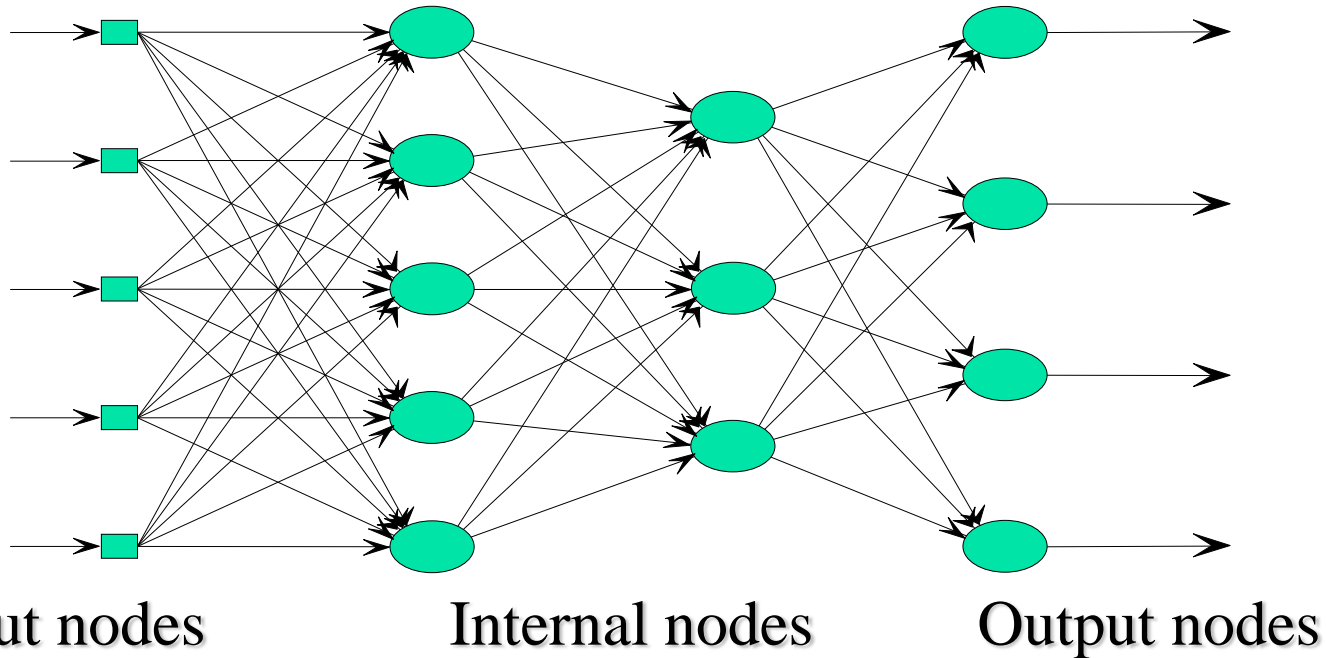
. K

Multi-Layer Perceptron (MLP)

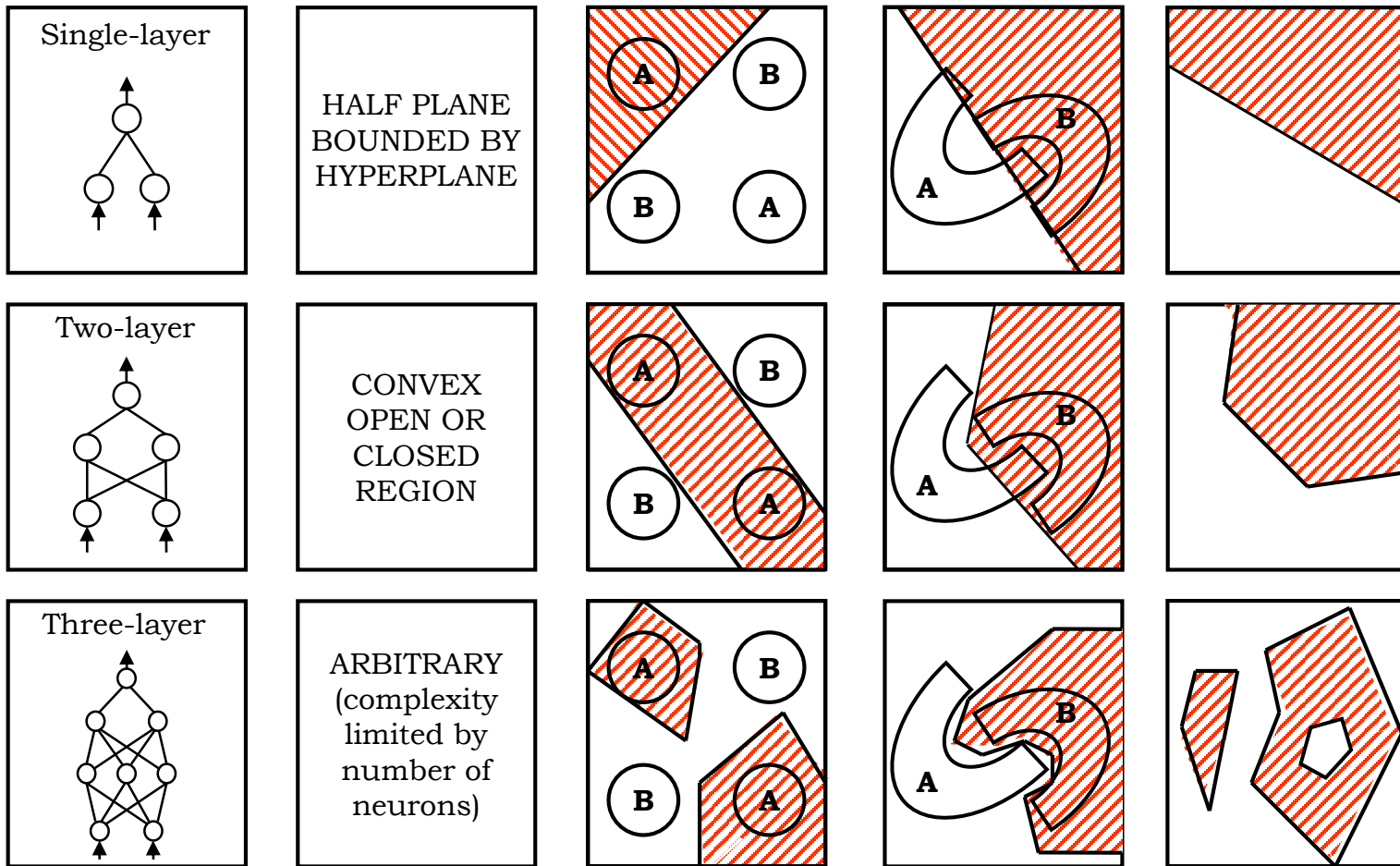


MultiLayer Perceptron

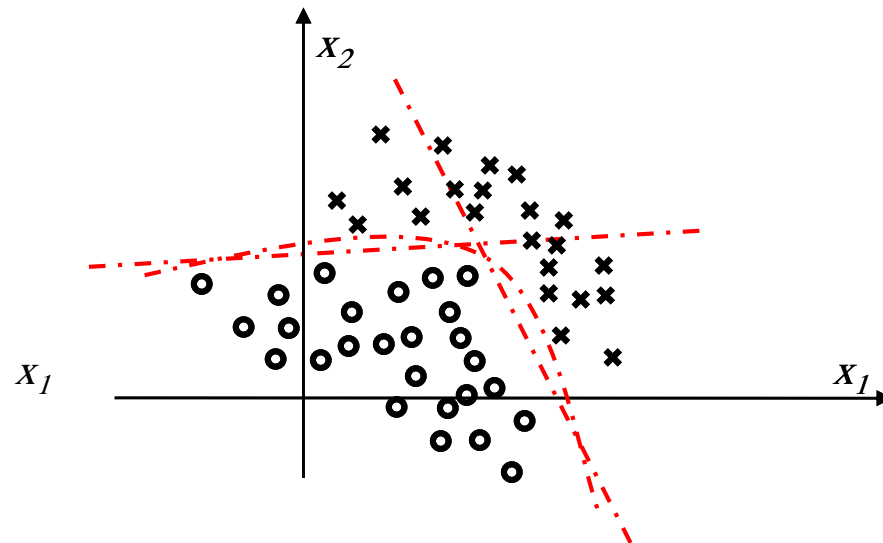
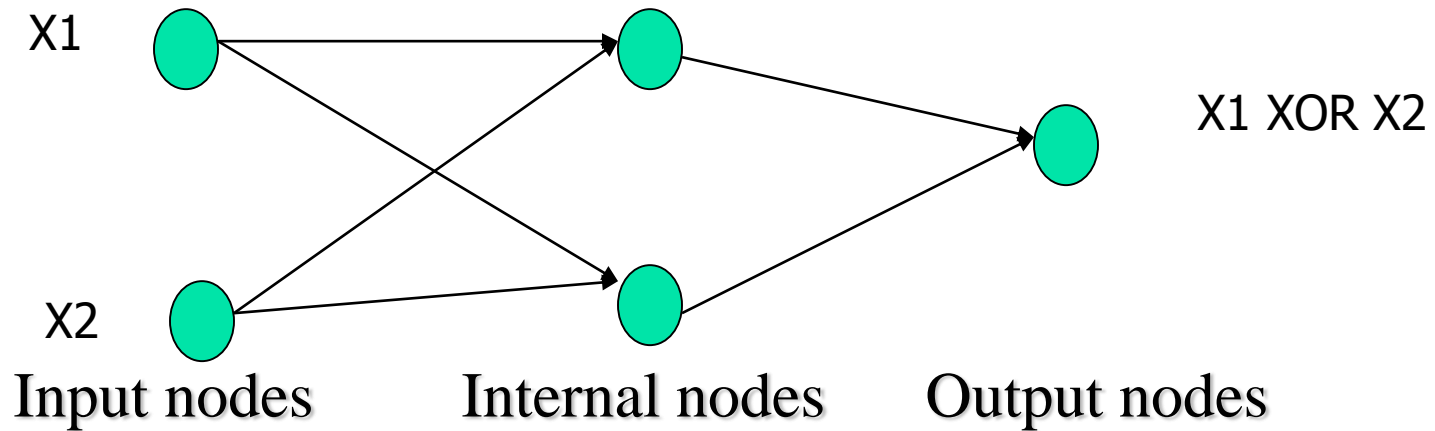
In contrast to perceptrons, multilayer networks can learn not only multiple decision boundaries, but the boundaries may be nonlinear.



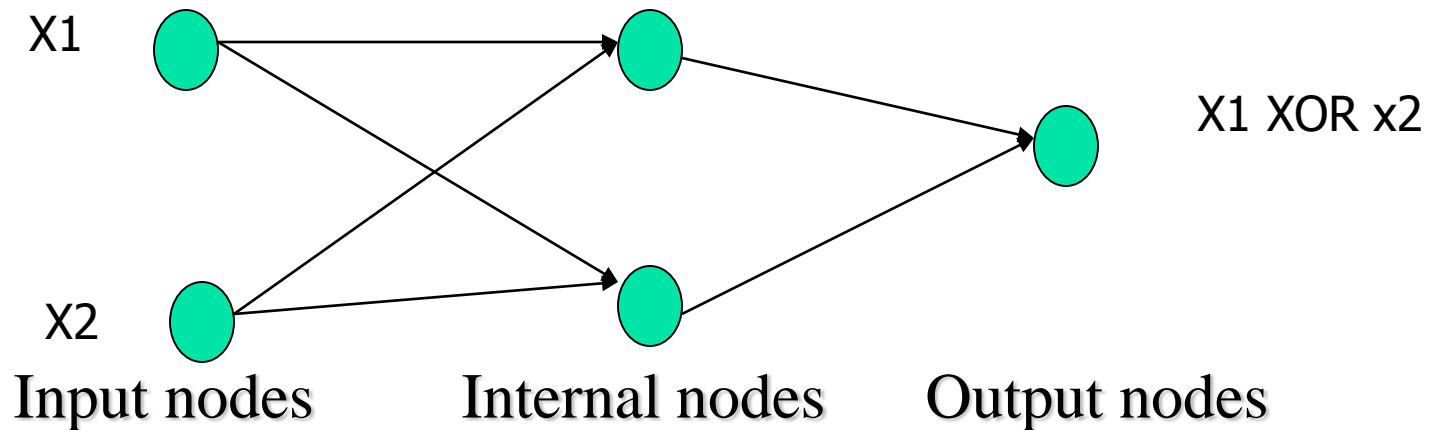
MultiLayer Perceptron- Decision Boundaries



Solution for XOR : Add a hidden layer !!



Solution for XOR : Add a hidden layer !!



The problem is: How to learn Multi Layer Perceptrons??

Solution: **Backpropagation Algorithm** invented by Rumelhart and colleagues in 1986

Problems

- How do we train a multi-layered network?
- What is the desired output of hidden neurons?

This problem remained unsolved for many years, causing the so called “**AI winter**”.



Rumelhart

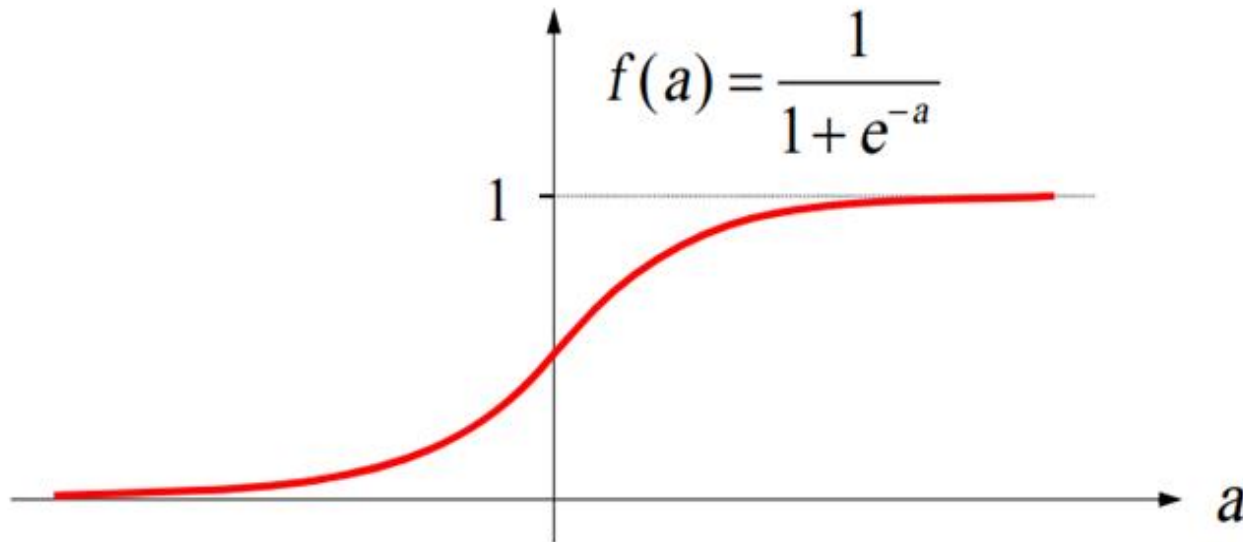


Hinton

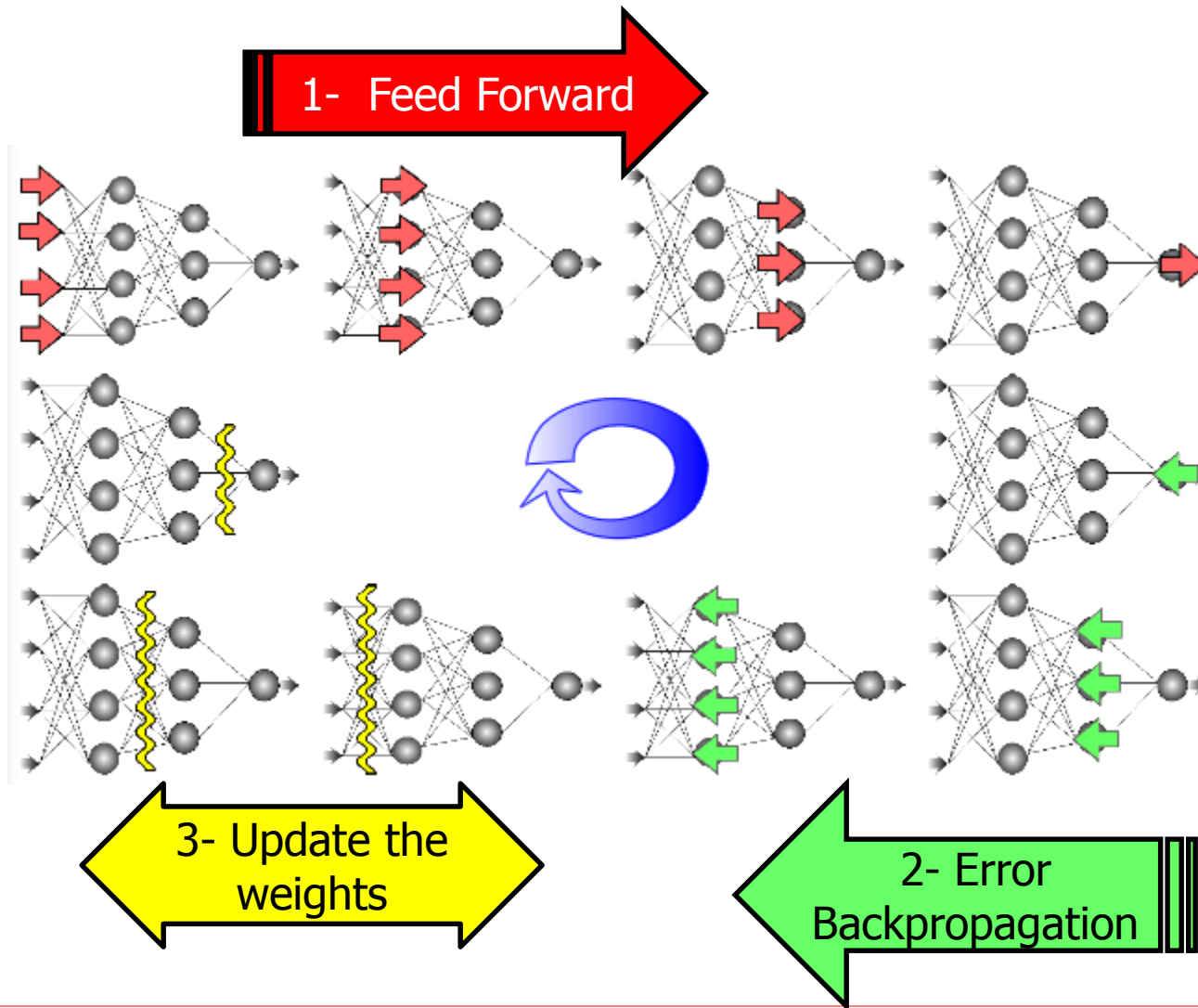
Backpropagation learning Algorithm

(Rumelhart-Hinton-Williams, 1986)

- *Multi-layer feedforward networks;*
- *Output is generated using a **sigmoid** function: $y_j = f(a_j)$*



Backpropagation- Algorithm



Backpropagation: Objectives

➤ Learning

Teach the network a set of desired associations $(\mathbf{x}_k, \mathbf{t}_k)$ provided by the **Training Set**.

➤ Convergence

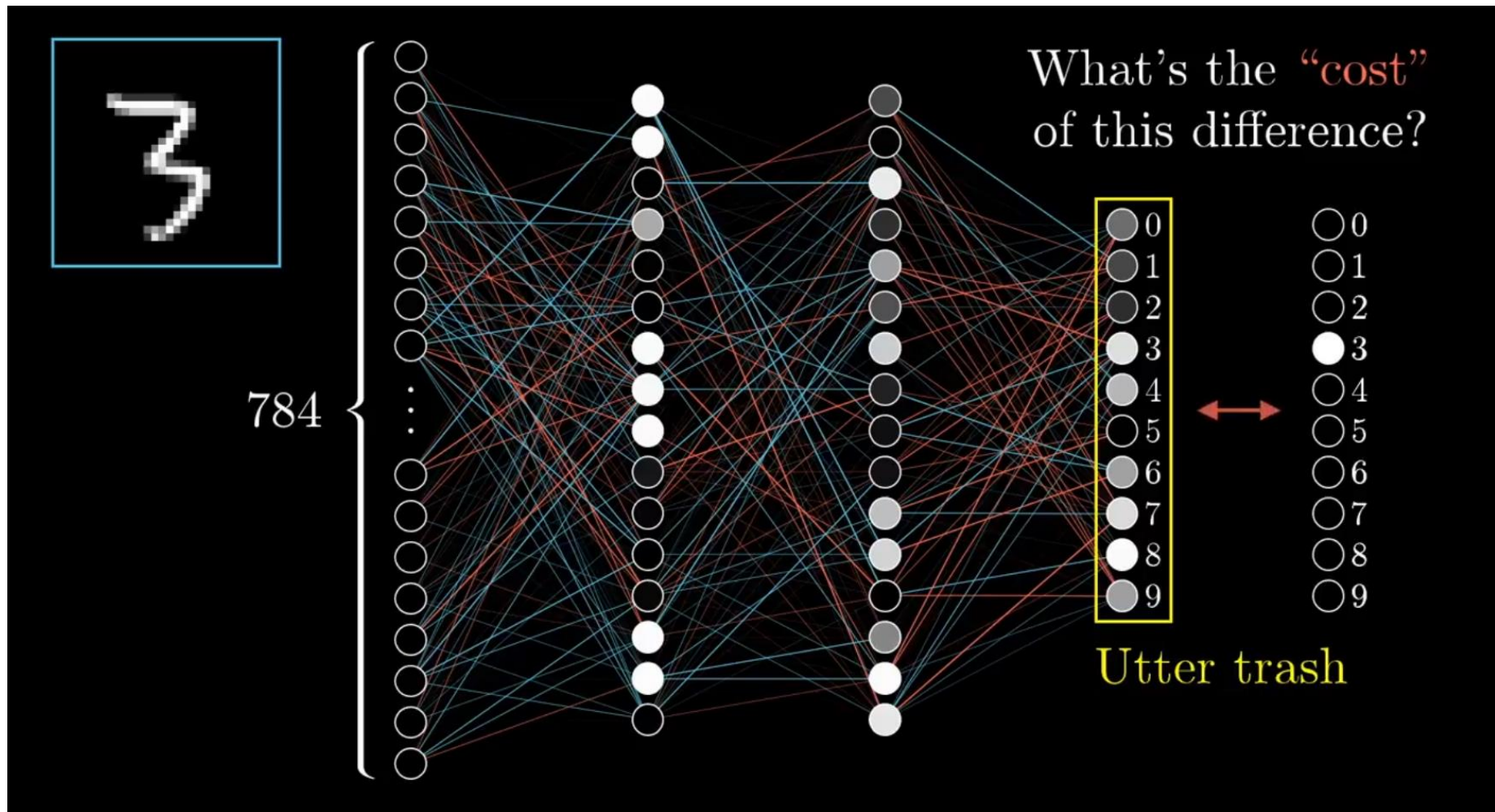
Reduce the global error E by changing weights, such that $E < \varepsilon$, in a finite amount of time.

➤ Generalization

Make the network to respond well on inputs that were never shown (i.e., not in the Training Set).



Backpropagation (Error or cost)



Backpropagation (Error or cost)

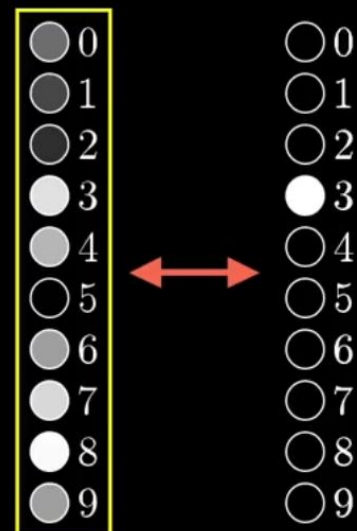
Cost of

3

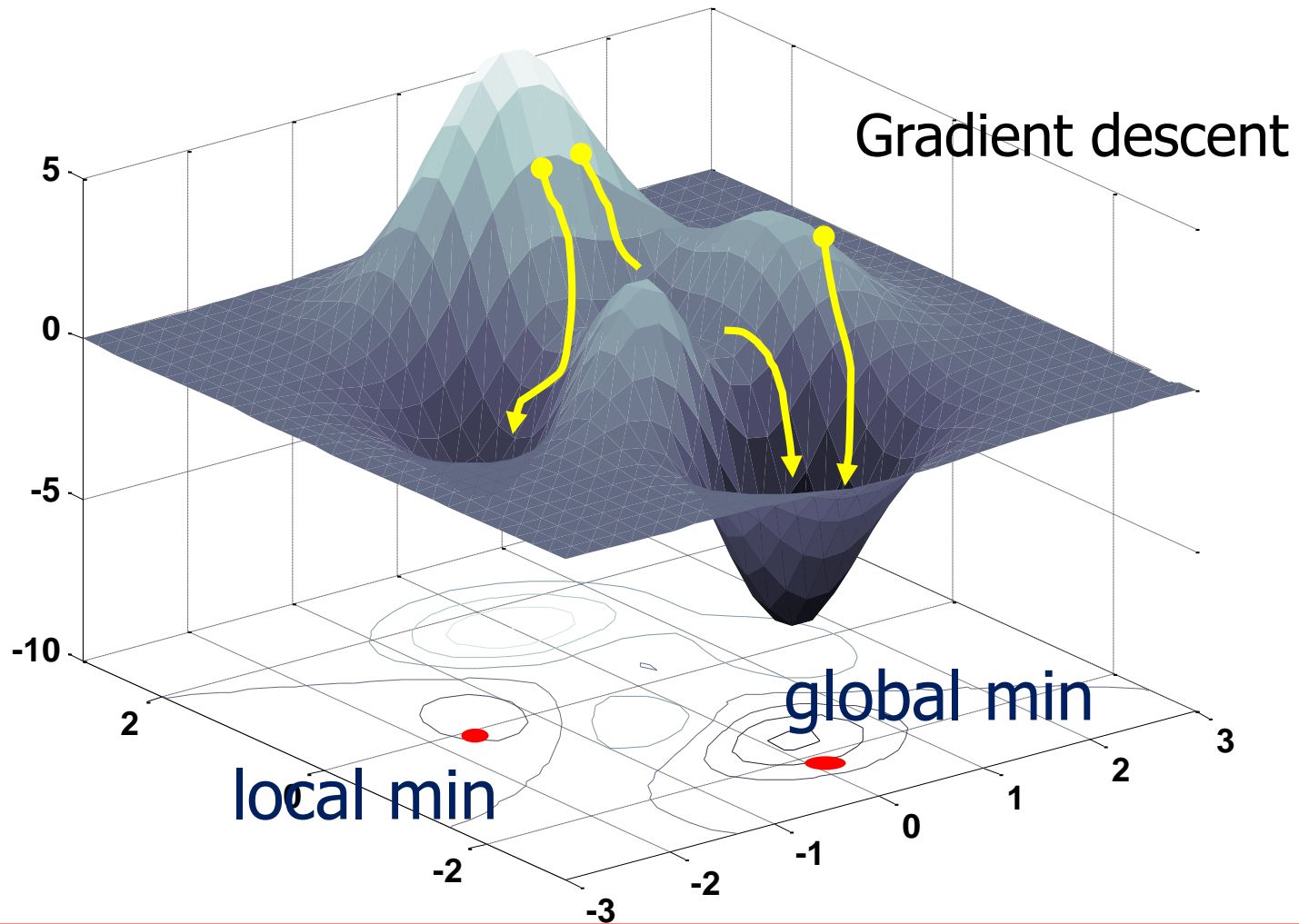
3.37

$$\begin{aligned} 0.1863 &\leftarrow (0.43 - 0.00)^2 + \\ 0.0809 &\leftarrow (0.28 - 0.00)^2 + \\ 0.0357 &\leftarrow (0.19 - 0.00)^2 + \\ 0.0138 &\leftarrow (0.88 - 1.00)^2 + \\ 0.5242 &\leftarrow (0.72 - 0.00)^2 + \\ 0.0001 &\leftarrow (0.01 - 0.00)^2 + \\ 0.4079 &\leftarrow (0.64 - 0.00)^2 + \\ 0.7388 &\leftarrow (0.86 - 0.00)^2 + \\ 0.9817 &\leftarrow (0.99 - 0.00)^2 + \\ 0.3998 &\leftarrow (0.63 - 0.00)^2 \end{aligned}$$

What's the "cost" of this difference?



Error space (Multi-Modal Cost Surface)

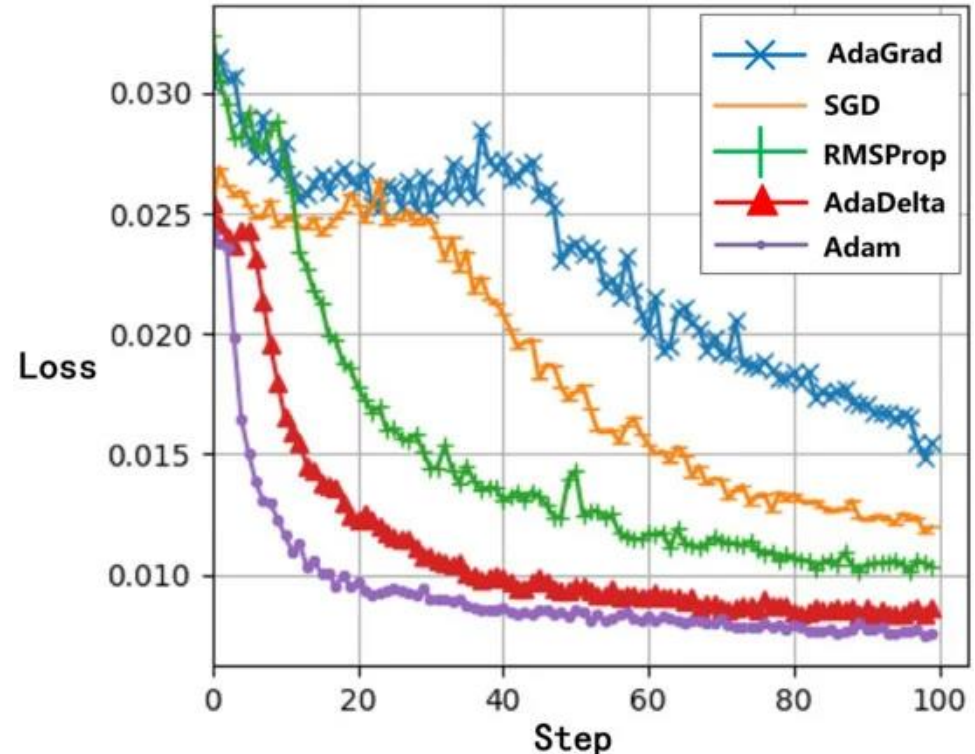


Optimizers

- **Gradient Descent:** most fundamental technique to train Neural Networks

- Variants:

- Momentum
- SGD
- AdaGrad
- AdaDelta
- RMSprop
- Adam



Local vs. global minimum

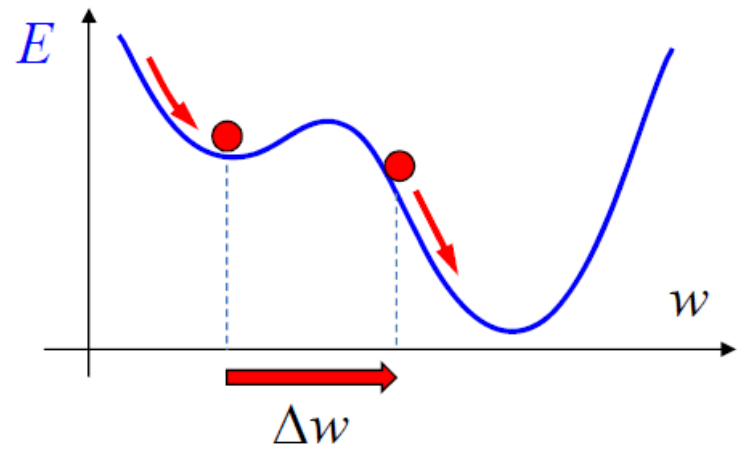
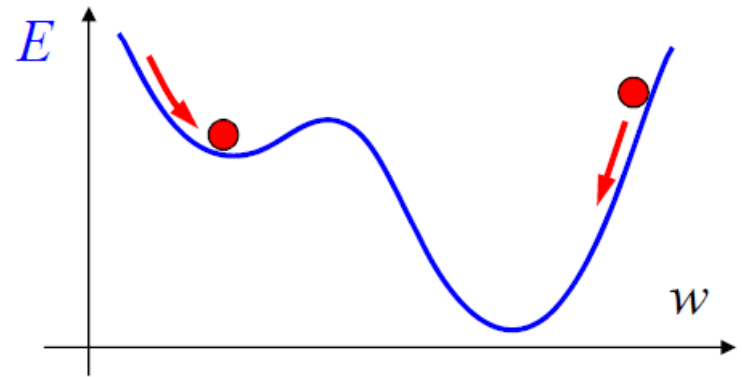
There are two approaches to escape local minima:

1. Reset the network

Restart training with new random weights. You may be luckier!

2. Make a random jump

Add a random value to the weights. It may be enough to escape.



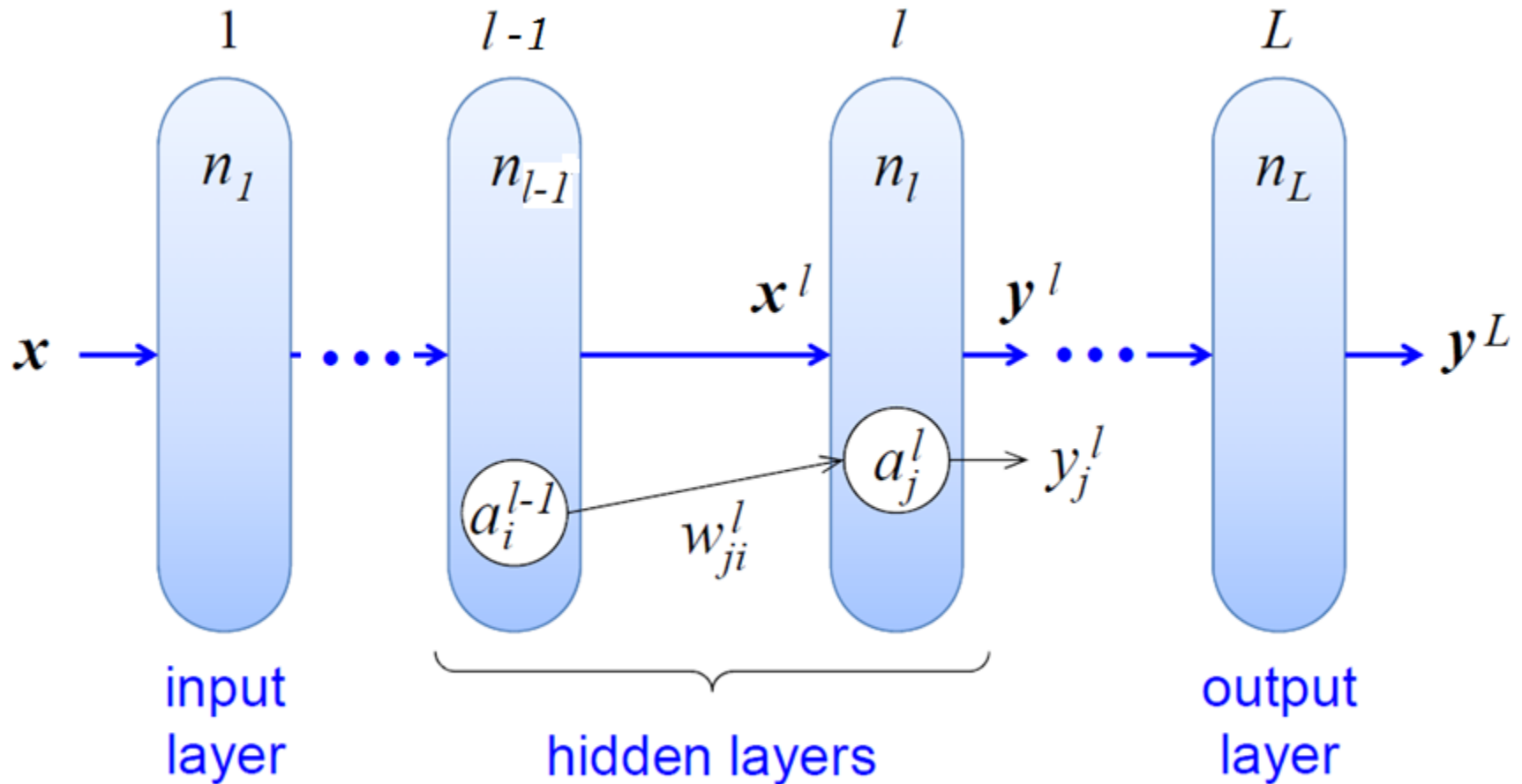
Weight updates

$$\vec{W} = \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_{13,000} \\ w_{13,001} \\ w_{13,002} \end{bmatrix}$$
$$-\nabla C(\vec{W}) = \begin{bmatrix} 0.31 \\ 0.03 \\ -1.25 \\ \vdots \\ 0.78 \\ -0.37 \\ 0.16 \end{bmatrix}$$

w_0 should increase somewhat
 w_1 should increase a little
 w_2 should decrease a lot

$w_{13,000}$ should increase a lot
 $w_{13,001}$ should decrease somewhat
 $w_{13,002}$ should increase a little

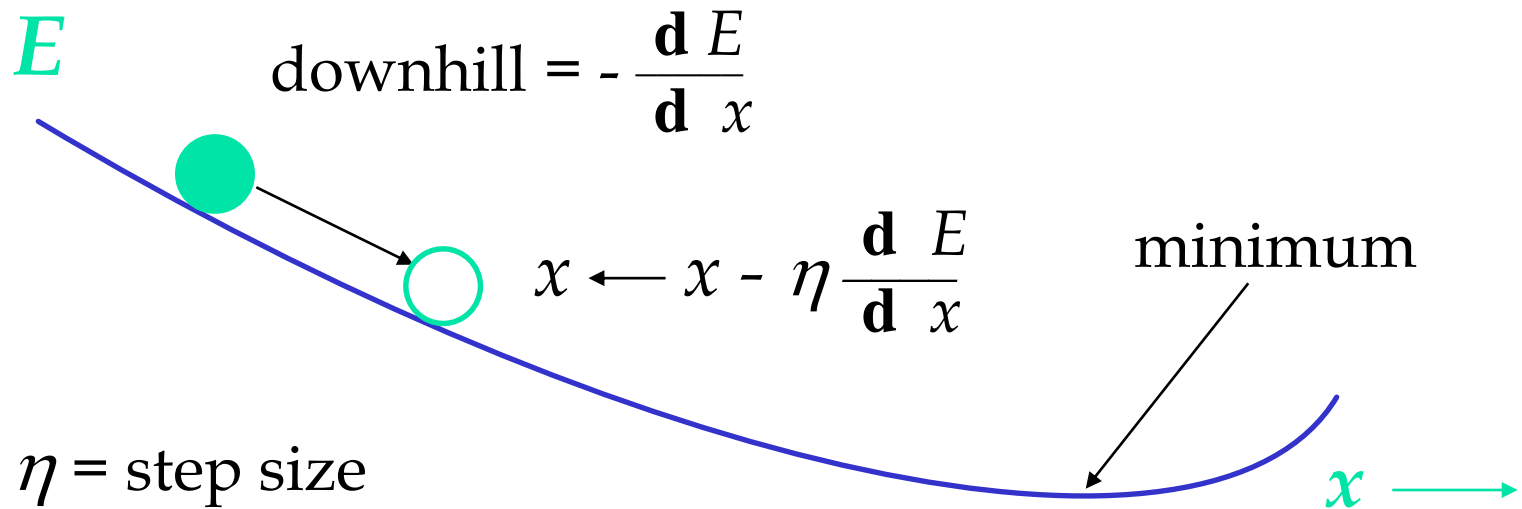
Backpropagation- Algorithm



Minimizing Error Using Steepest Descent

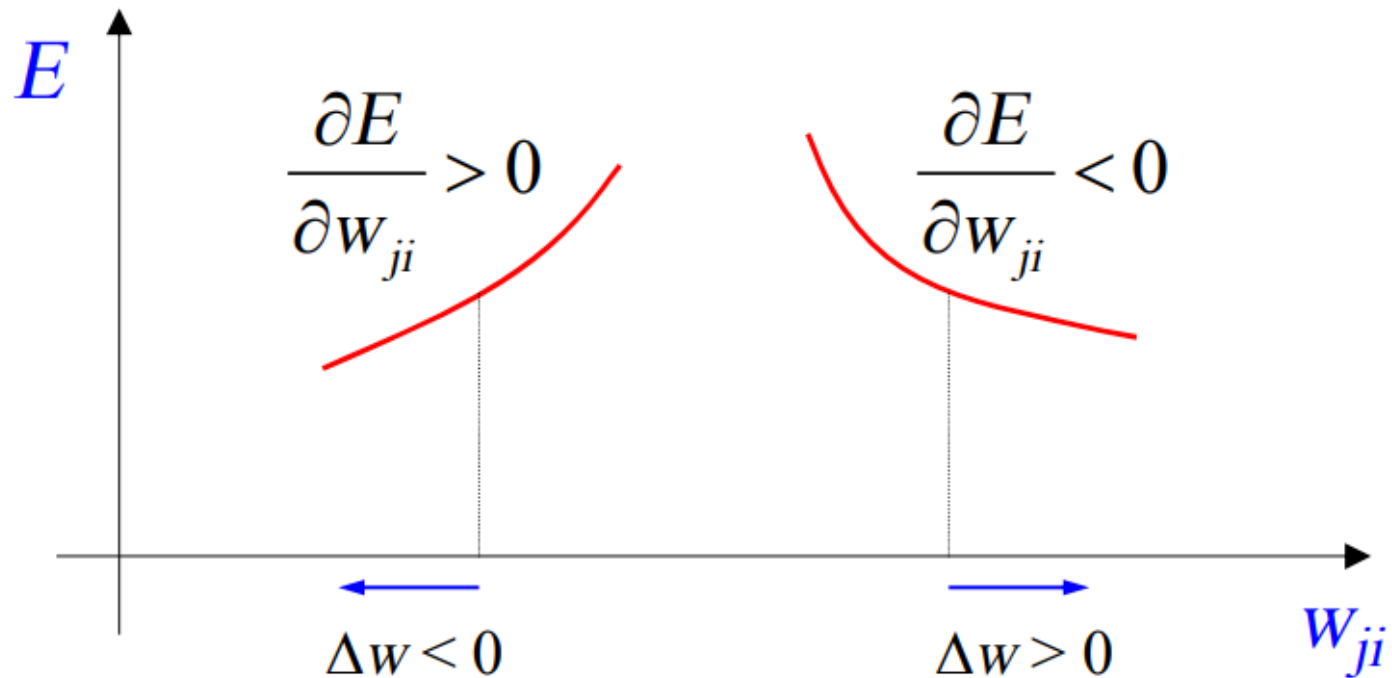
- The main idea:

Find the way downhill and take a step:



Convergence

To reduce the error by changing weights, the following strategy is adopted:



Updating weights

Therefore, the weights are changed according to the following law:

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}}$$

Gradient
rule

η = Learning coefficient (learning rate)

Back-propagating error

$$\Delta w_{ji}(k) = -\eta \frac{\partial E_k}{\partial w_{ji}} = -\eta \frac{\partial E_k}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

being $a_j = \sum_{i=1}^n w_{ji} x_i + b_j$ we have:

$$\frac{\partial a_j}{\partial w_{ji}} = x_i$$

and by
defining

$$\delta_j \triangleq -\frac{\partial E_k}{\partial a_j}$$

then:

$$\Delta w_{ji}(k) = \eta \delta_j x_i$$

So, now the problem is to compute δ_j for each neuron.

Back-propagating- computing δ_j (for output layer)

$$\delta_j^L = -\frac{\partial E_k}{\partial a_j} = -\frac{\partial E_k}{\partial y_j} \frac{\partial y_j}{\partial a_j}$$

being $E_k = \frac{1}{2} \sum_{j=1}^{n_L} (t_{kj} - y_j^L)^2$ we have: $\frac{\partial E_k}{\partial y_j} = -(t_{kj} - y_j^L)$

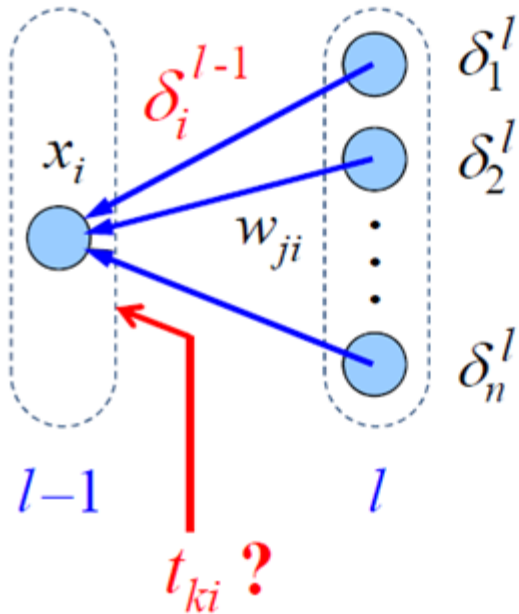
and since $\frac{\partial y_j}{\partial a_j} = f'(a_j)$ we obtain:

$$\delta_j^L = (t_{kj} - y_j^L) f'(a_j^L)$$



Back-propagating- computing δ_j (for hidden layers)

Note that the same formula $\delta_i^{l-1} = (t_{ki} - y_i^{l-1}) f'(a_i^{l-1})$ cannot be used for hidden neurons, since t_{ki} is unknown!



Hence the idea is to backpropagate the errors back through the weights to assign a blame to hidden neurons.



$$\delta_i^{l-1} = f'(a_i^{l-1}) \sum_{j=1}^{n_l} w_{ji}^l \delta_j^l$$



Updating weights

Generalized
Delta Rule:

$$\Delta w_{ji}(k) = \eta \delta_j x_i$$

For the output
neurons:

$$\delta_j^L = (t_{kj} - y_j^L) f'(a_j^L)$$

For the hidden
neurons

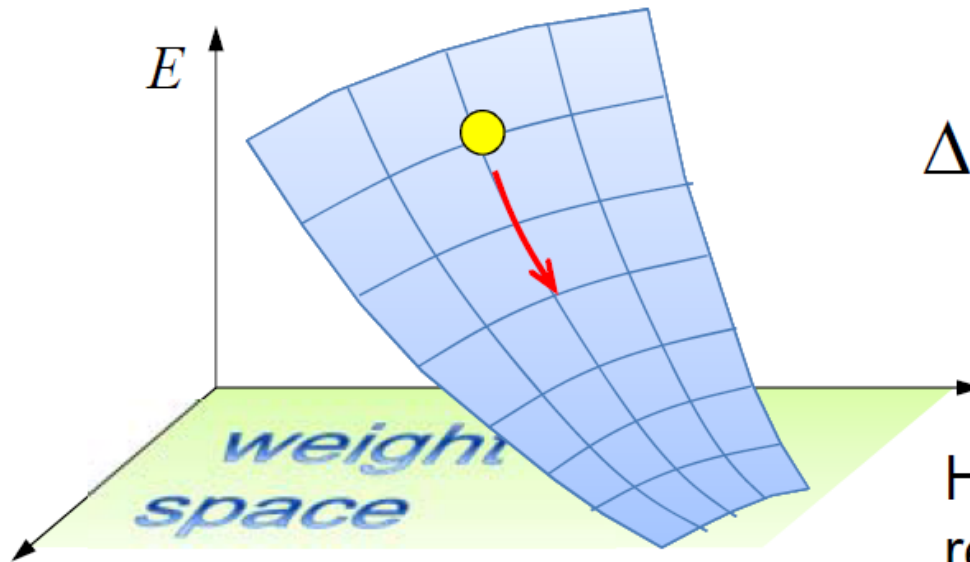
$$\delta_i^{l-1} = f'(a_i^{l-1}) \sum_{j=1}^{n_l} w_{ji}^l \delta_j^l$$

Back Propagation: Algorithm

1. randomly initialize the weights;
2. **do** {
3. initializes the global error $\mathbf{E} = \mathbf{0}$;
4. **for each** $(X_k, t_k) \in \text{TS}$ {
5. compute \mathbf{y}_k and error \mathbf{E}_k ;
6. compute δ_j on the output layer;
7. compute δ_{the} on the hidden layer;
8. update weights of the network: $\Delta \mathbf{w} = \eta \delta \mathbf{x}$;
9. updates the global error: $\mathbf{E} = \mathbf{E} + \mathbf{E}_k$; }
10. } **while** $(\mathbf{E} > \epsilon)$;



Minimizing the global error



$$\Delta w_{ji} = \frac{1}{M} \sum_{k=1}^M \Delta w_{ji}(k)$$

Hence **a single learning step** requires:

- show all the M examples;
- store all the weights variations for each example;
- update all the weights after completing the training set.

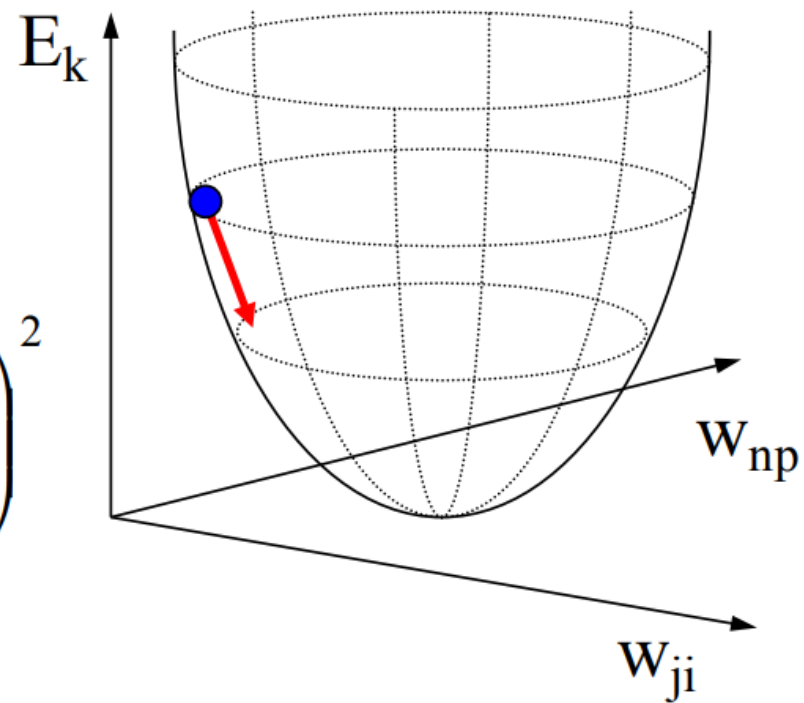
A pass of the entire training set is called an **epoch**.

Backpropagating- remarks

- The error has a quadratic form in the space of weights:

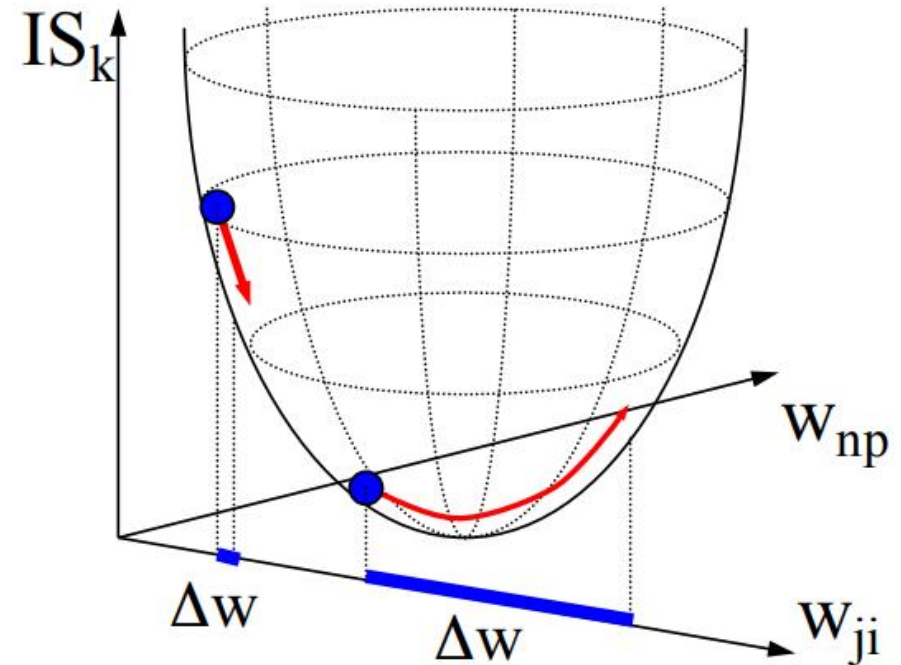
$$E_k = \sum_{j=1}^n (t_{kj} - y_{kj})^2$$

$$= \sum_{j=1}^n \left(t_{kj} - f \left(\sum_{i=1}^p w_{ji} x_{ki} \right) \right)^2$$



Backpropagating- remarks

$$\Delta w_{ji} = \eta \delta_j x_i$$



η too small \Rightarrow slow learning

η too big \Rightarrow fluctuations

Learning rate

Possible solutions

- Vary η as a function of the error, to speed up convergence at the beginning and reduce oscillations at the end.
- Attenuate oscillations with a low-pass filter on the weights:

$$\Delta w_{ji}(t) = \eta \delta_j x_i + \mu \Delta w_{ji}(t-1)$$

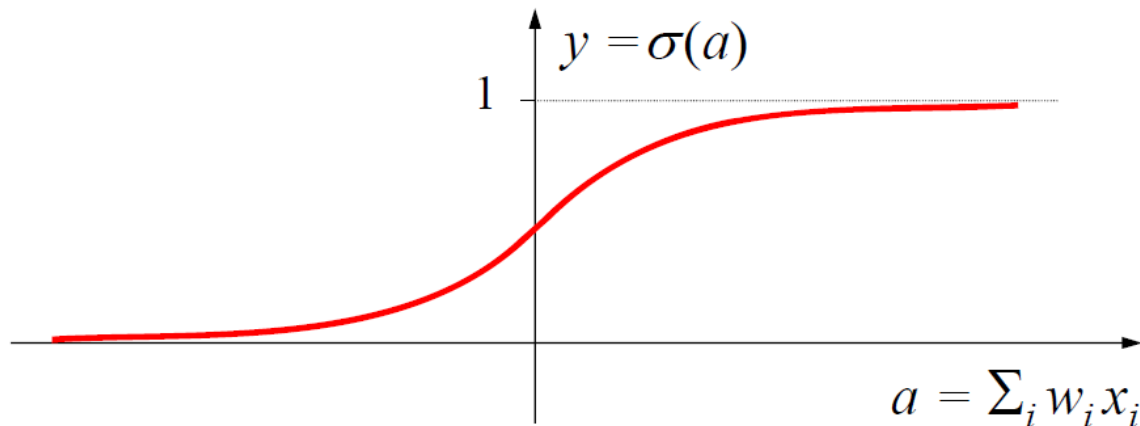
μ is called **momentum**

Initializing weights

To favor the learning phase, weights have to be initialized with random small values. In fact:

$$\Delta w = \eta \delta x \propto f'(a)$$

small $|w| \Rightarrow$ small $|a| \Rightarrow$ big $f'(a) \Rightarrow$ big Δw



Hence, weights are initialized as random variables with normal distribution with mean 0 and standard deviation $\sqrt{1/n_{in}}$ where n_{in} is the number of inputs of the neuron.

Hyper parameters

People distinguish two types of parameters in a neural network:

Model parameters

They are those that are found by learning:

- Weights
- Biases

Network hyper-parameters

They are those that have to be tuned to optimize learning:

- Number of hidden layers
- Number of hidden neurons
- Weight initialization range
- Training set size
- Mini-batch size
- ...
- Loss function (error)
- Activation function
- Number of epochs
- Learning rate
- Momentum
- ...



Type of data sets

Training
Set

Used for **training** the model parameters

Validation
Set

Used for **tuning** the hyper-parameters
(Used to decide when to stop training only by monitoring the error.)

Test
Set

Used for **assessing** the performance of
a **fully-trained** network









Consistency of the TS

If some examples are inconsistent, convergence of learning is not guaranteed:

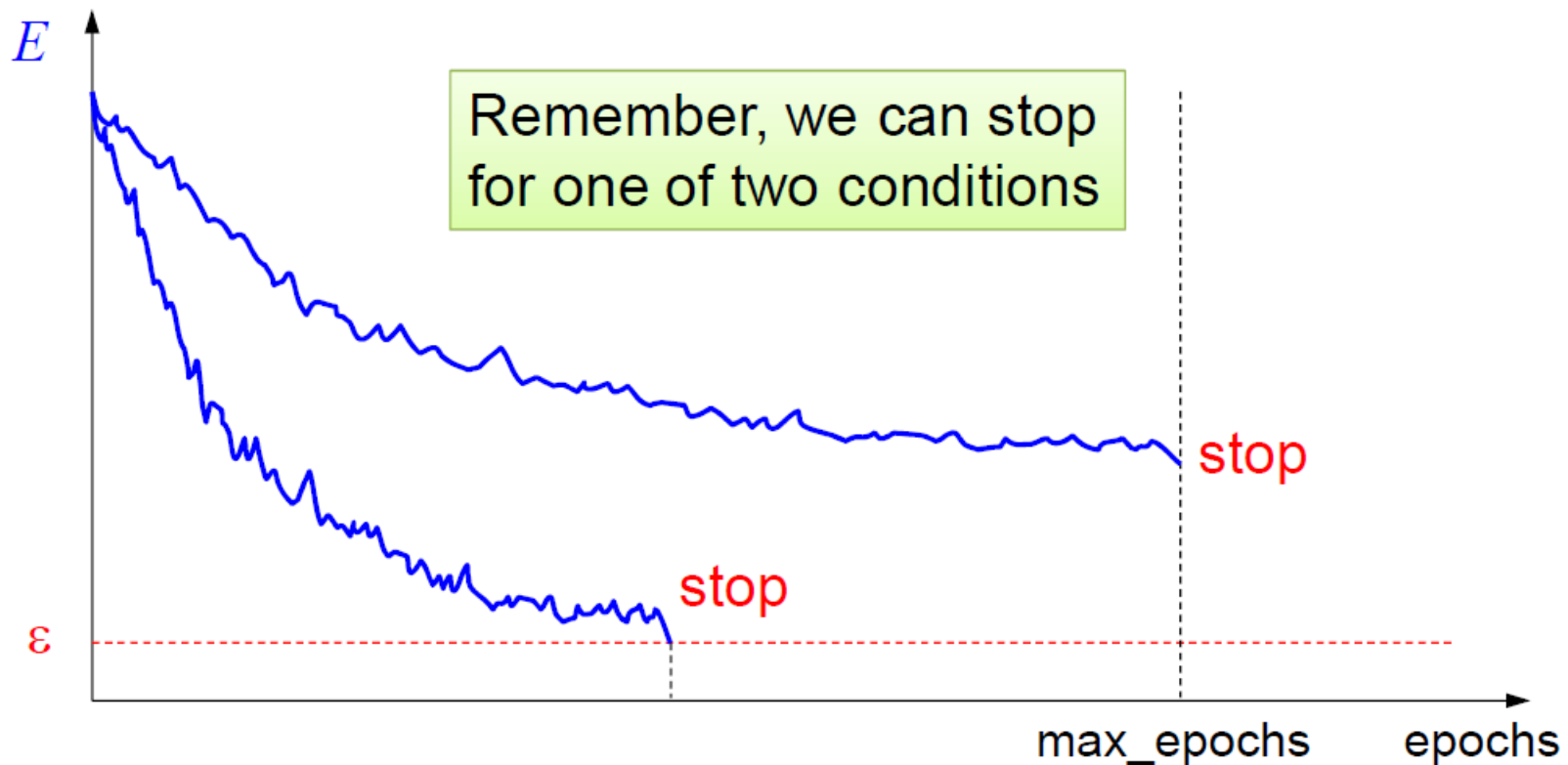
In real cases, inconsistencies can be introduced by similar noisy patterns belonging to different classes

Examples of problematic training patterns taken from images of handwritten characters:

sample	target
	0
	6
	3
	5
	7
	1

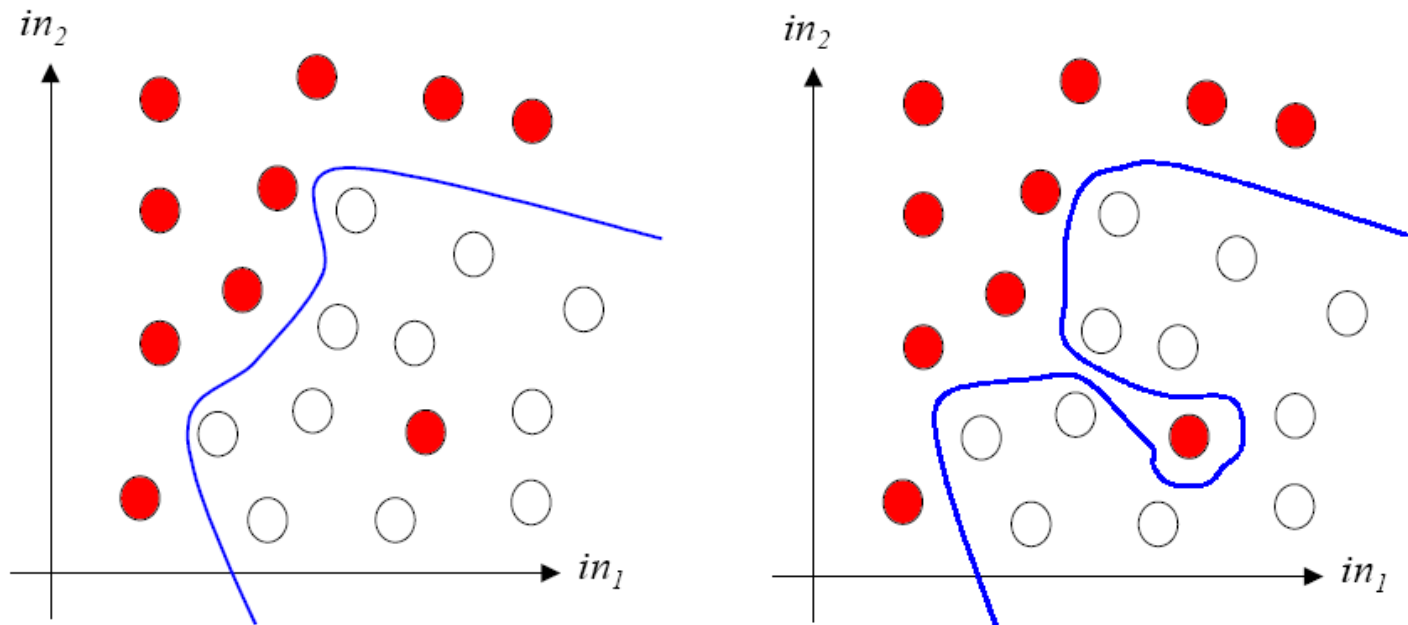
Stopping criteria

Once we have trained our network, how can we evaluate its performance?



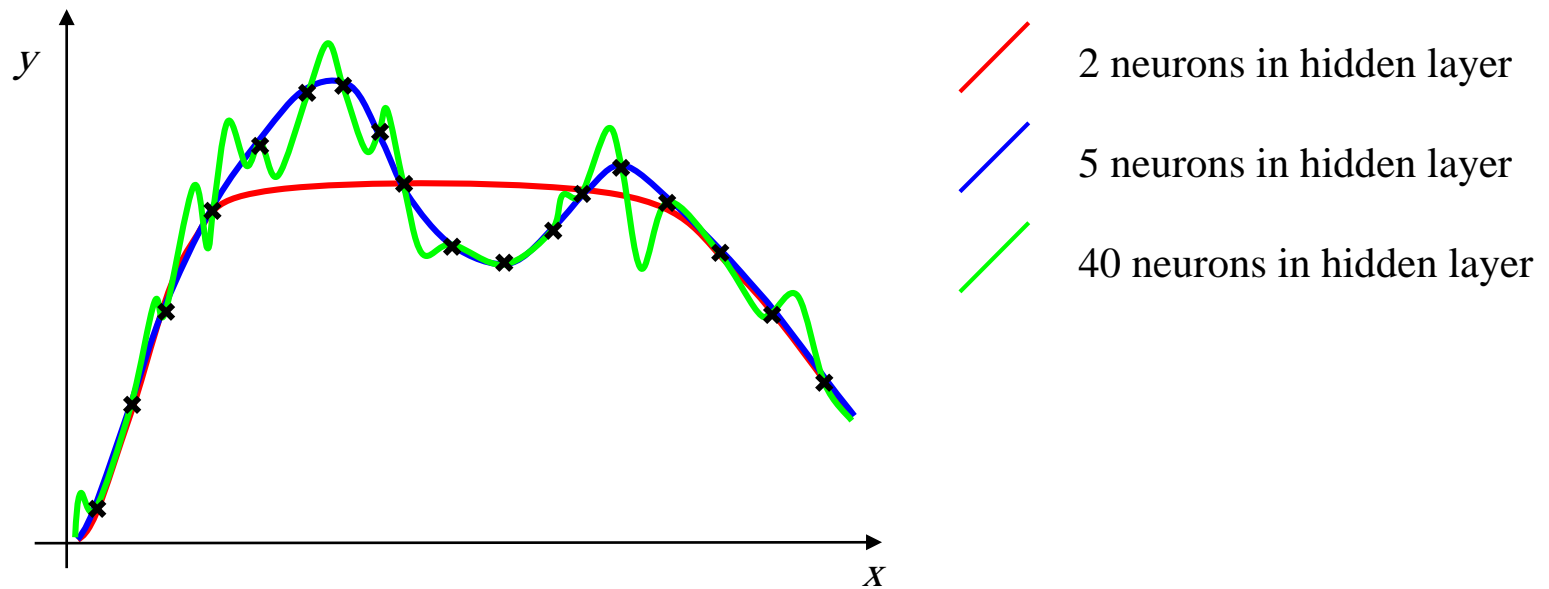
Generalization in Classification

- Suppose the task of our network is to learn a classification decision boundary
- Our aim is for the network to generalize to classify new inputs appropriately. If we know that the training data contains noise, we don't necessarily want the training data to be classified totally accurately, as that is likely to reduce the generalization ability.



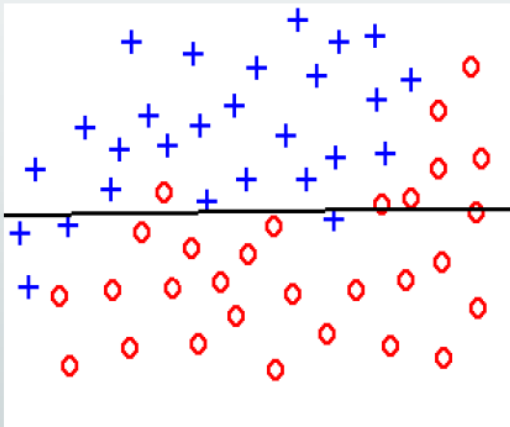
The problem of overfitting ...

- Approximation of the function $y = f(x)$:

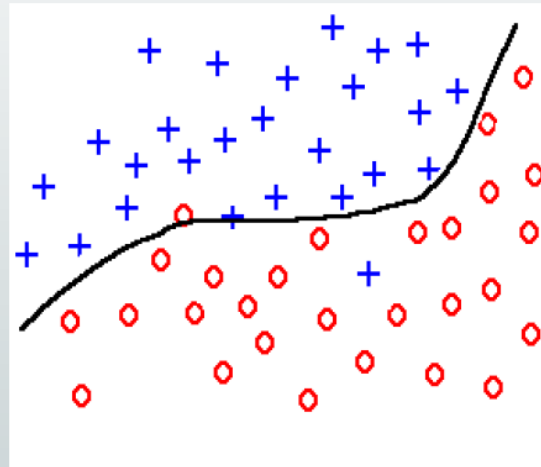


- The overfitting is not detectable in the learning phase ...
- So use **Cross-Validation** ...

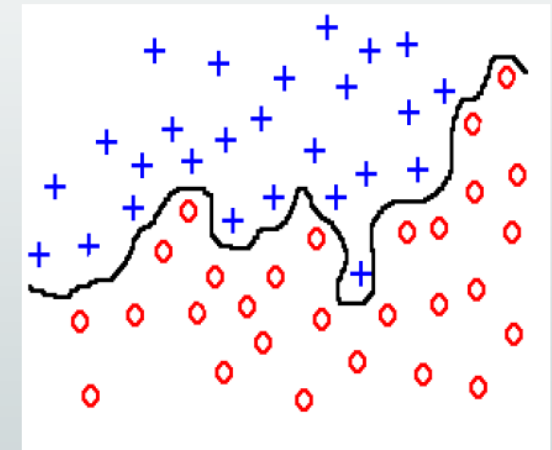
Overfitting and underfitting



underfitting



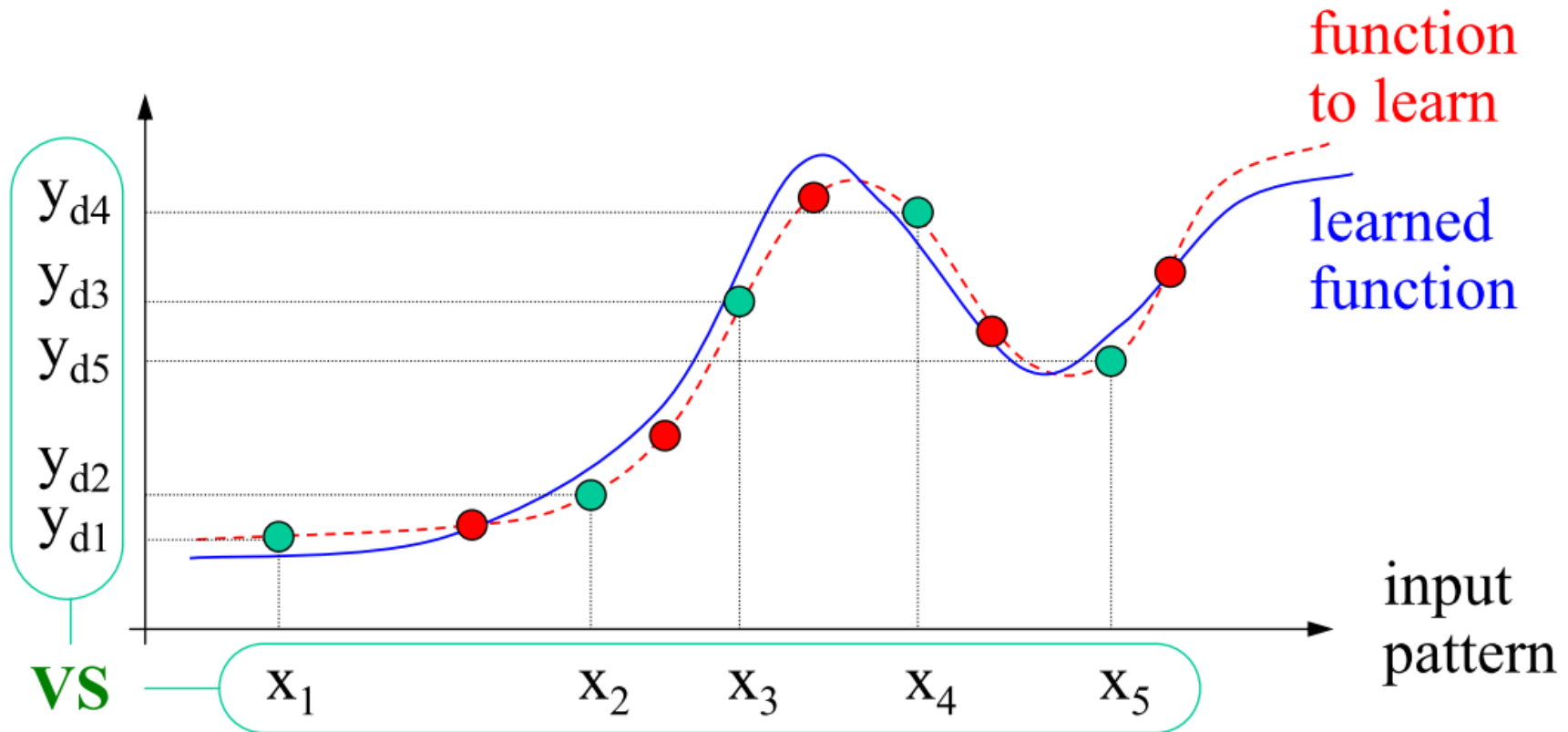
good fit



overfitting

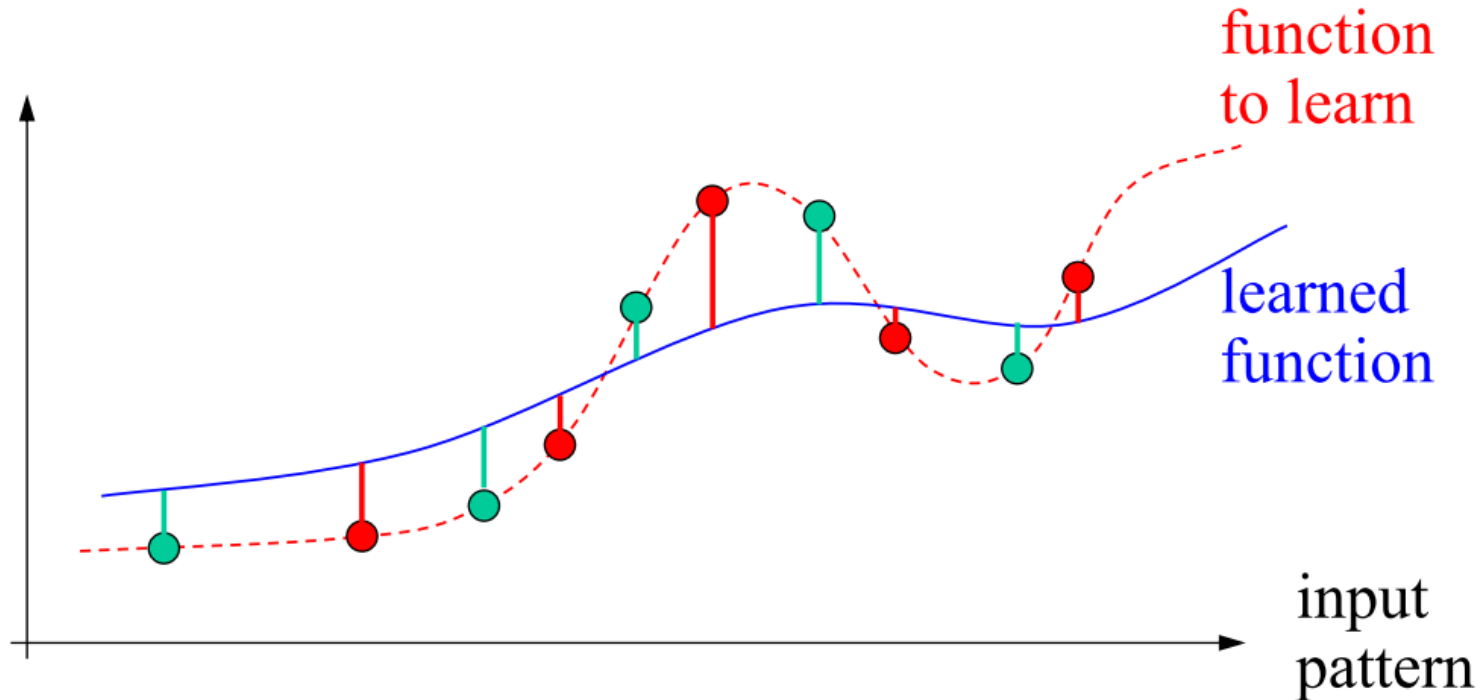
Generalization in Function Approximation

If the network is well dimensioned, both the errors E_{TS} and E_{VS} are small



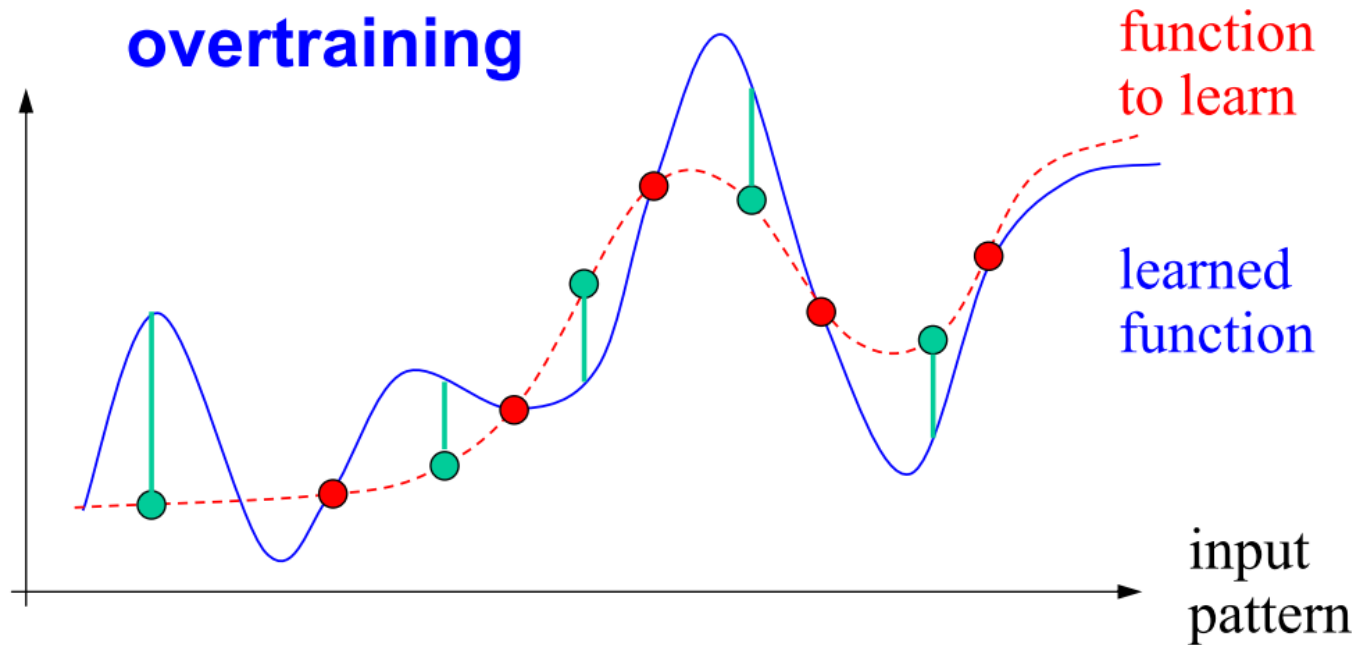
Generalization in Function Approximation

If the network does not have enough hidden neurons, it is not able to approximate the function and both errors are large:



Generalization in Function Approximation

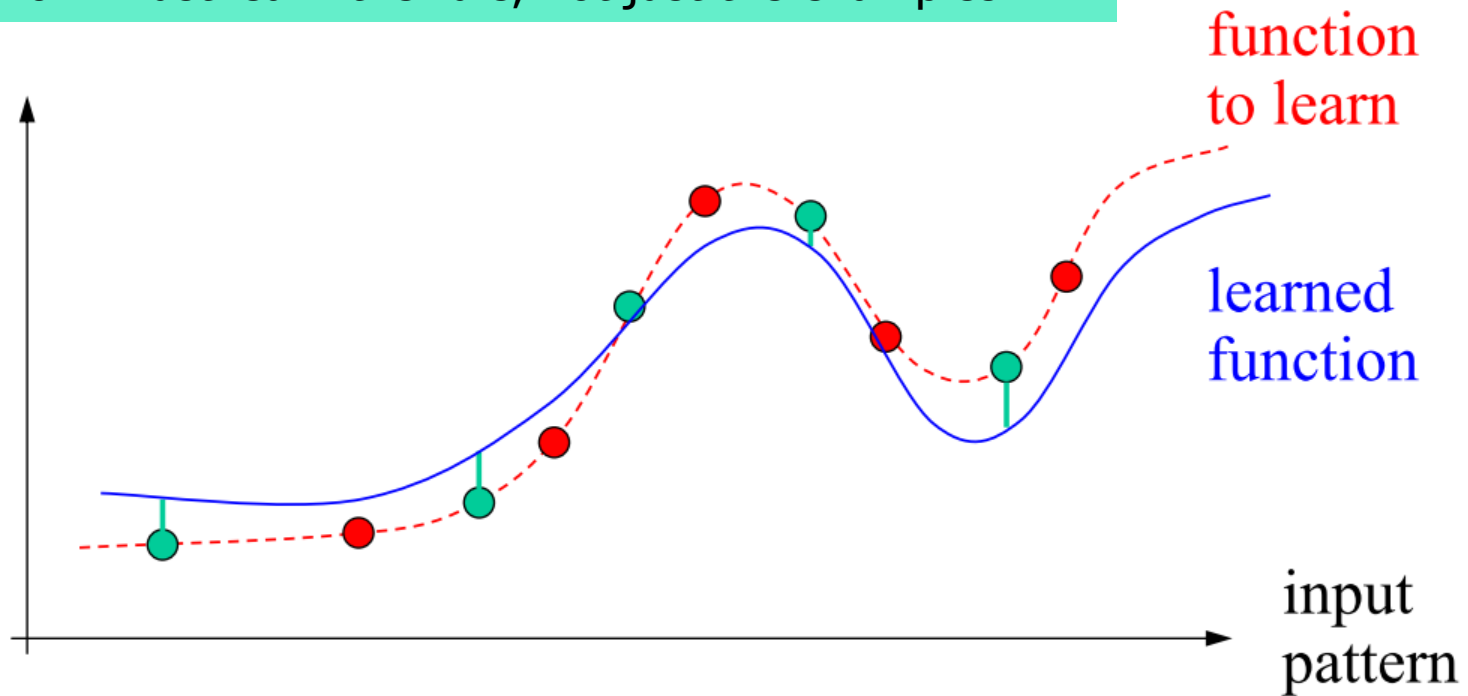
If the network has too many hidden neurons, it could respond correctly to the TS ($E_{TS} < \epsilon$), but could not generalize well (E_{VS} too large):



Generalization in Function Approximation

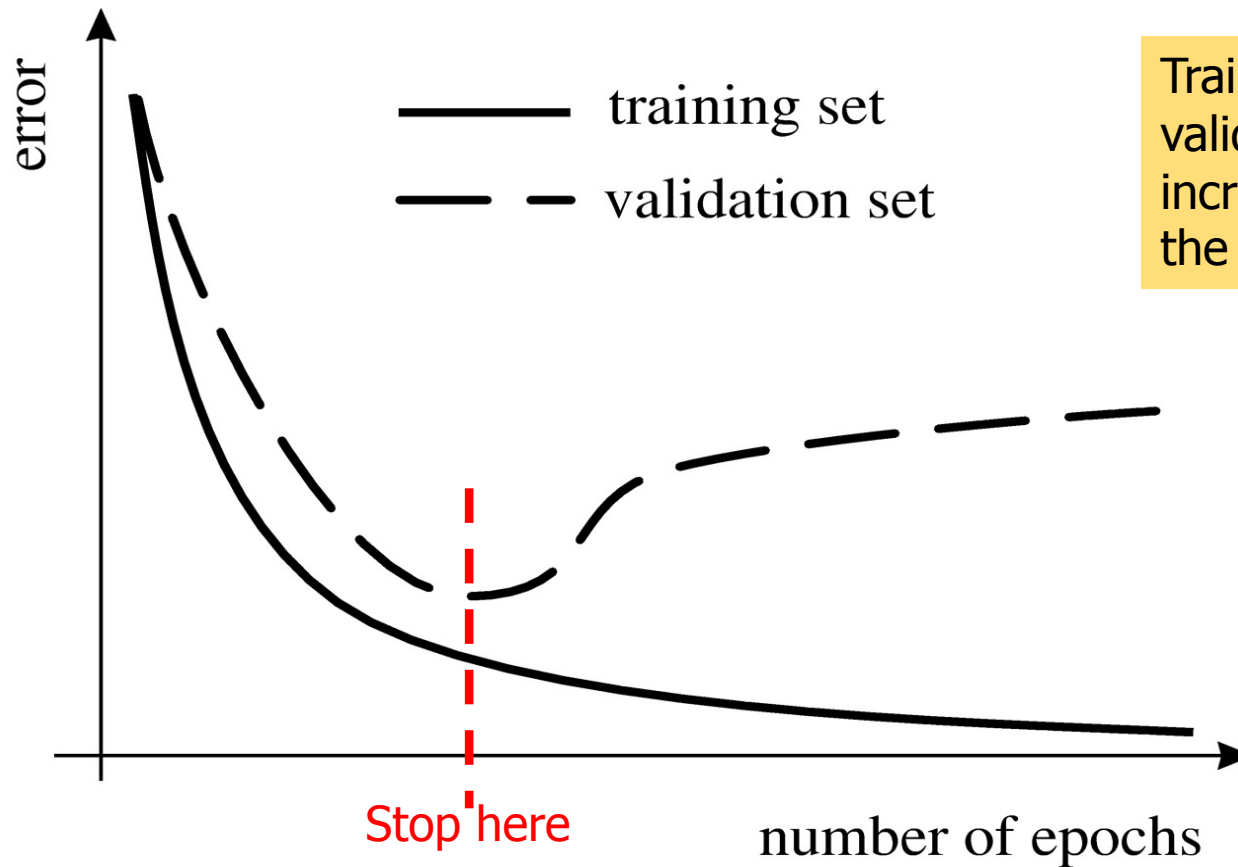
To avoid the overtraining effect, we can train the network using the TS, but monitor E_{VS} and stop the training when ($E_{VS} < \epsilon$):

(network must learn the rule, not just the examples)



Earlier Stopping - Good Generalization

Use of a validation set allows periodic testing to see whether the model has overfitted

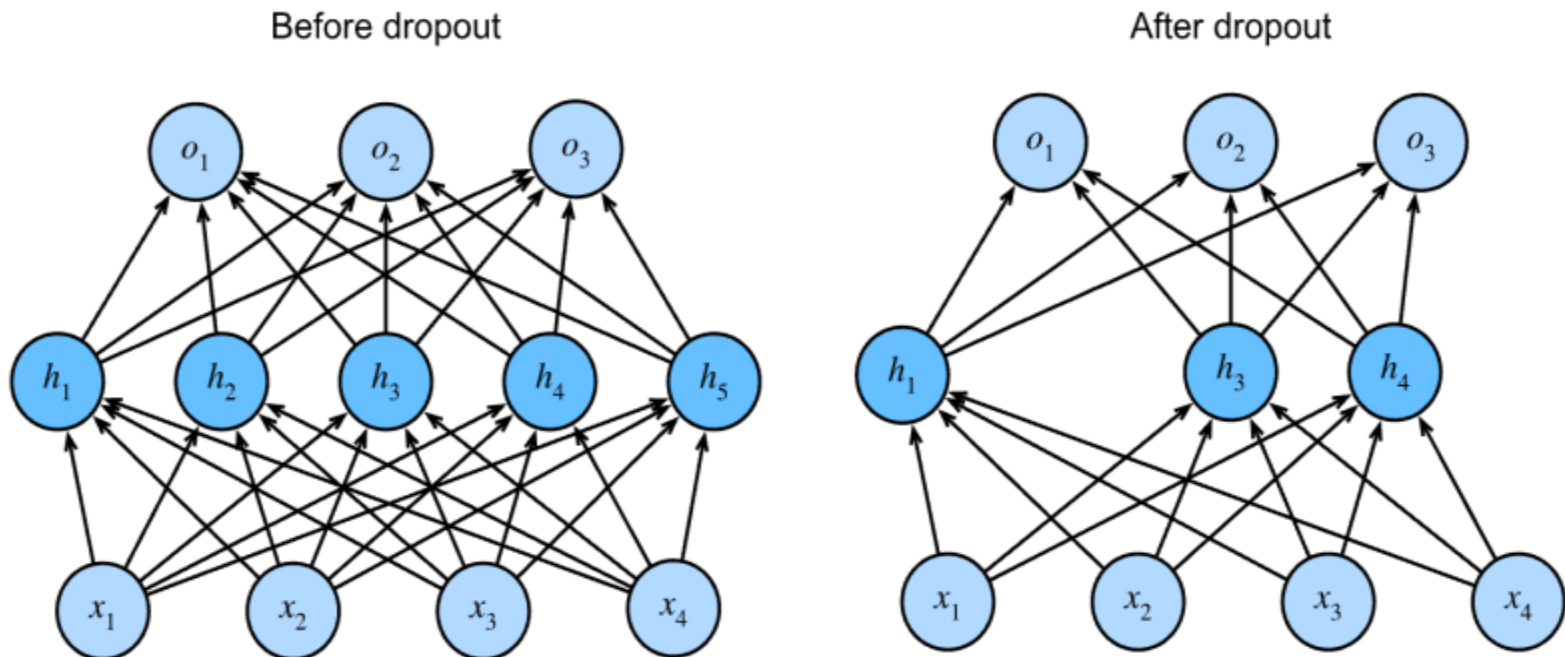


Training stops when the validation loss starts to increase, indicating that the model is overfitting.



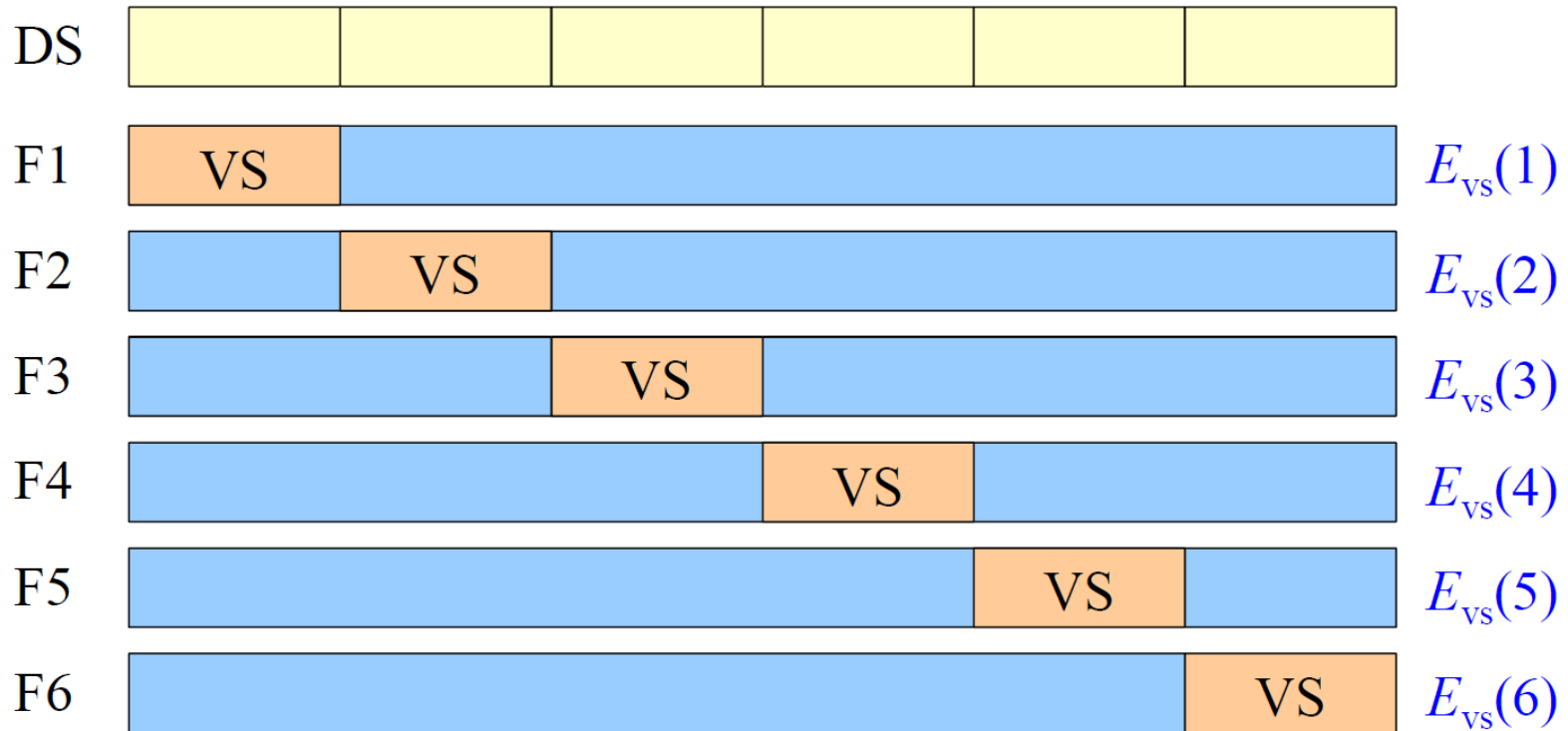
Regularization: Dropout

- Dropout regularizes the network by randomly dropping neurons from the neural network during **training**



K-Fold Cross Validation

K = 6

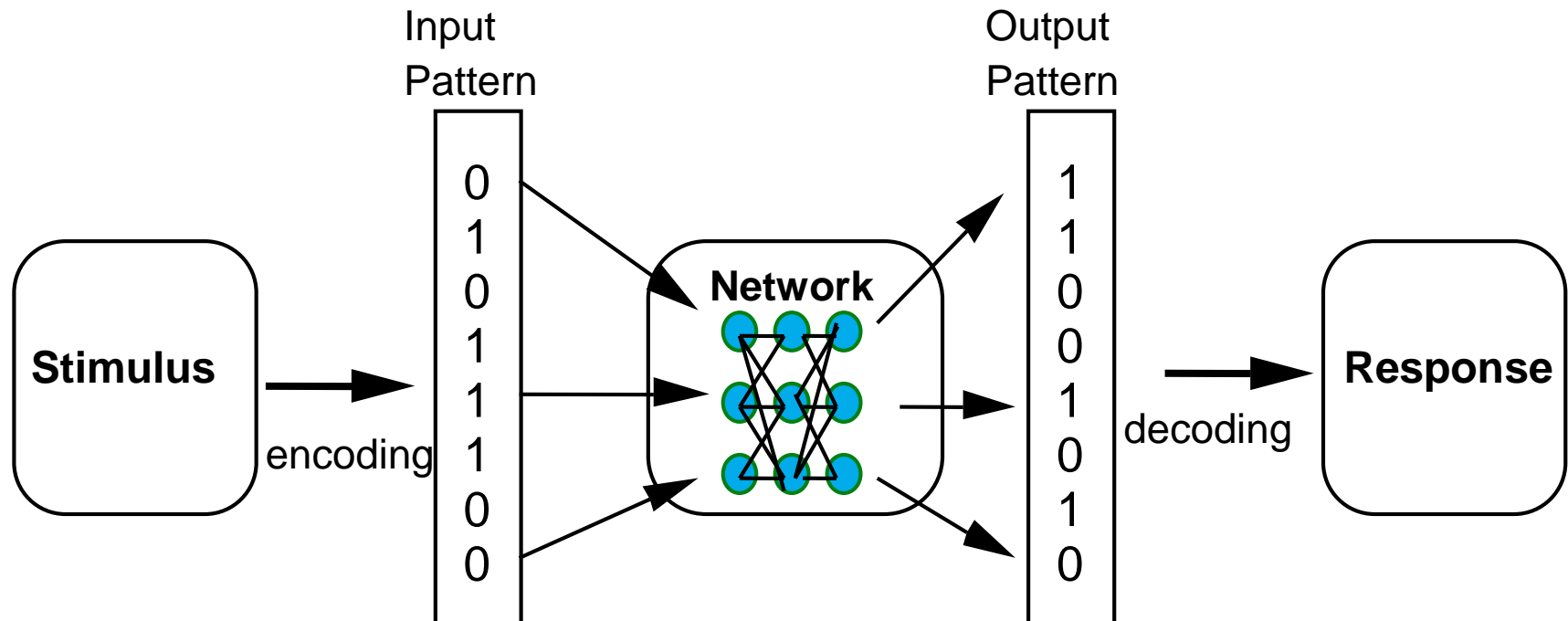


$$E_{vs} = \frac{1}{K} \sum_{i=1}^K E_{vs}(i)$$

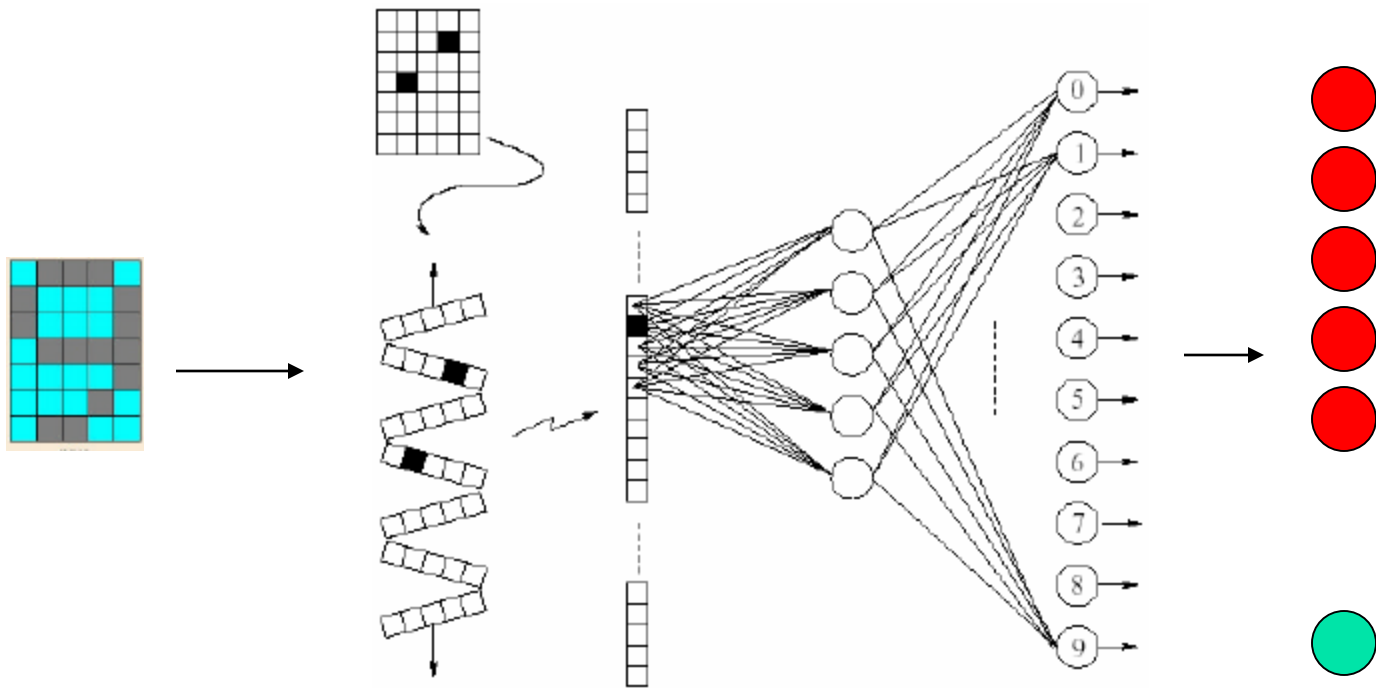
$$A_{vs} = \frac{1}{K} \sum_{i=1}^K A_{vs}(i)$$

Application of MLPs

The general scheme when using ANNs is as follows:



Application: Digit Recognition



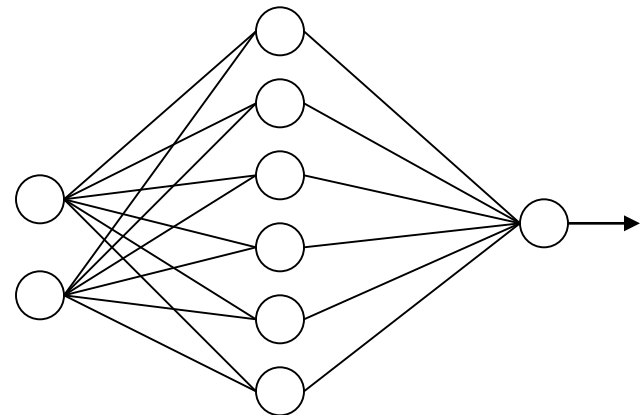
Learning XOR Operation: Matlab Code

```
P = [ 0 0 1 1; ...  
      0 1 0 1]  
T = [ 0 1 1 0];
```

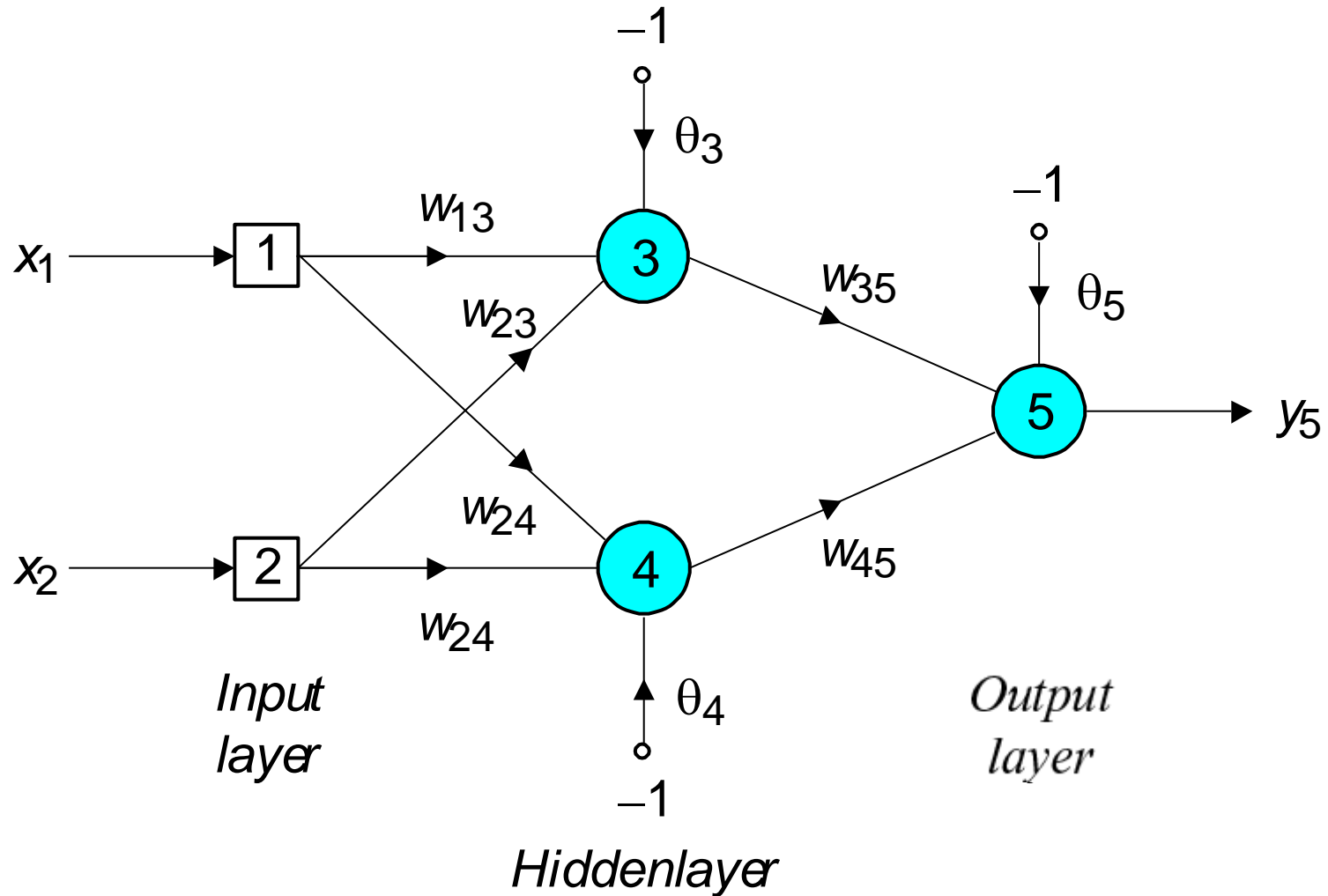
```
net = newff([0 1;0 1],[6 1],{'tansig' 'tansig'});
```

```
net.trainParam.epochs = 4850;  
net = train(net,P,T);
```

```
X = [0 1];  
Y = sim(net,X);  
display(Y);
```



Solving the XOR operation



Solving the XOR operation

- The effect of the threshold applied to a neuron in the hidden or output layer is represented by its weight, θ , connected to a fixed input equal to -1 .
- The initial weights and threshold levels are set randomly as follows:
 $w_{13} = 0.5, w_{14} = 0.9, w_{23} = 0.4, w_{24} = 1.0, w_{35} = -1.2, w_{45} = 1.1, \theta_3 = 0.8, \theta_4 = -0.1$ and $\theta_5 = 0.3$.



Solving the XOR operation

- We consider a training set where inputs x_1 and x_2 are equal to 1 and desired output $y_{d,5}$ is 0. The actual outputs of neurons 3 and 4 in the hidden layer are calculated as

$$y_3 = \text{sigmoid}(x_1 w_{13} + x_2 w_{23} - \theta_3) = 1 / \left[1 + e^{-(1 \cdot 0.5 + 1 \cdot 0.4 - 1 \cdot 0.8)} \right] = 0.5250$$

$$y_4 = \text{sigmoid}(x_1 w_{14} + x_2 w_{24} - \theta_4) = 1 / \left[1 + e^{-(1 \cdot 0.9 + 1 \cdot 1.0 + 1 \cdot 0.1)} \right] = 0.8808$$

- Now the actual output of neuron 5 in the output layer is determined as:

$$y_5 = \text{sigmoid}(y_3 w_{35} + y_4 w_{45} - \theta_5) = 1 / \left[1 + e^{-(0.5250 \cdot 1.2 + 0.8808 \cdot 1.1 - 1 \cdot 0.3)} \right] = 0.5097$$

- Thus, the following error is obtained:

$$e = y_{d,5} - y_5 = 0 - 0.5097 = -0.5097$$

Solving the XOR operation

- The next step is weight training. To update the weights and threshold levels in our network, we propagate the error, e , from the output layer backward to the input layer.
- First, we calculate the error gradient for neuron 5 in the output layer:

$$\delta_5 = y_5 (1 - y_5) e = 0.5097 \cdot (1 - 0.5097) \cdot (-0.5097) = -0.1274$$

- Then we determine the weight corrections assuming that the learning rate parameter, α , is equal to 0.1:

$$\Delta w_{35} = \alpha \cdot y_3 \cdot \delta_5 = 0.1 \cdot 0.5250 \cdot (-0.1274) = -0.0067$$

$$\Delta w_{45} = \alpha \cdot y_4 \cdot \delta_5 = 0.1 \cdot 0.8808 \cdot (-0.1274) = -0.0112$$

$$\Delta \theta_5 = \alpha \cdot (-1) \cdot \delta_5 = 0.1 \cdot (-1) \cdot (-0.1274) = -0.0127$$



Solving the XOR operation

- Next we calculate the error gradients for neurons 3 and 4 in the hidden layer:

$$\delta_3 = y_3(1 - y_3) \cdot \delta_5 \cdot w_{35} = 0.5250 \cdot (1 - 0.5250) \cdot (-0.1274) \cdot (-1.2) = 0.0381$$

$$\delta_4 = y_4(1 - y_4) \cdot \delta_5 \cdot w_{45} = 0.8808 \cdot (1 - 0.8808) \cdot (-0.1274) \cdot 1.1 = -0.0147$$

- We then determine the weight corrections:

$$\Delta w_{13} = \alpha \cdot x_1 \cdot \delta_3 = 0.1 \cdot 1 \cdot 0.0381 = 0.0038$$

$$\Delta w_{23} = \alpha \cdot x_2 \cdot \delta_3 = 0.1 \cdot 1 \cdot 0.0381 = 0.0038$$

$$\Delta \theta_3 = \alpha \cdot (-1) \cdot \delta_3 = 0.1 \cdot (-1) \cdot 0.0381 = -0.0038$$

$$\Delta w_{14} = \alpha \cdot x_1 \cdot \delta_4 = 0.1 \cdot 1 \cdot (-0.0147) = -0.0015$$

$$\Delta w_{24} = \alpha \cdot x_2 \cdot \delta_4 = 0.1 \cdot 1 \cdot (-0.0147) = -0.0015$$

$$\Delta \theta_4 = \alpha \cdot (-1) \cdot \delta_4 = 0.1 \cdot (-1) \cdot (-0.0147) = 0.0015$$

Solving the XOR operation

- At last, we update all weights and threshold:

$$w_{13} = w_{13} + \Delta w_{13} = 0.5 + 0.0038 = 0.5038$$

$$w_{14} = w_{14} + \Delta w_{14} = 0.9 - 0.0015 = 0.8985$$

$$w_{23} = w_{23} + \Delta w_{23} = 0.4 + 0.0038 = 0.4038$$

$$w_{24} = w_{24} + \Delta w_{24} = 1.0 - 0.0015 = 0.9985$$

$$w_{35} = w_{35} + \Delta w_{35} = -1.2 - 0.0067 = -1.2067$$

$$w_{45} = w_{45} + \Delta w_{45} = 1.1 - 0.0112 = 1.0888$$

$$\theta_3 = \theta_3 + \Delta\theta_3 = 0.8 - 0.0038 = 0.7962$$

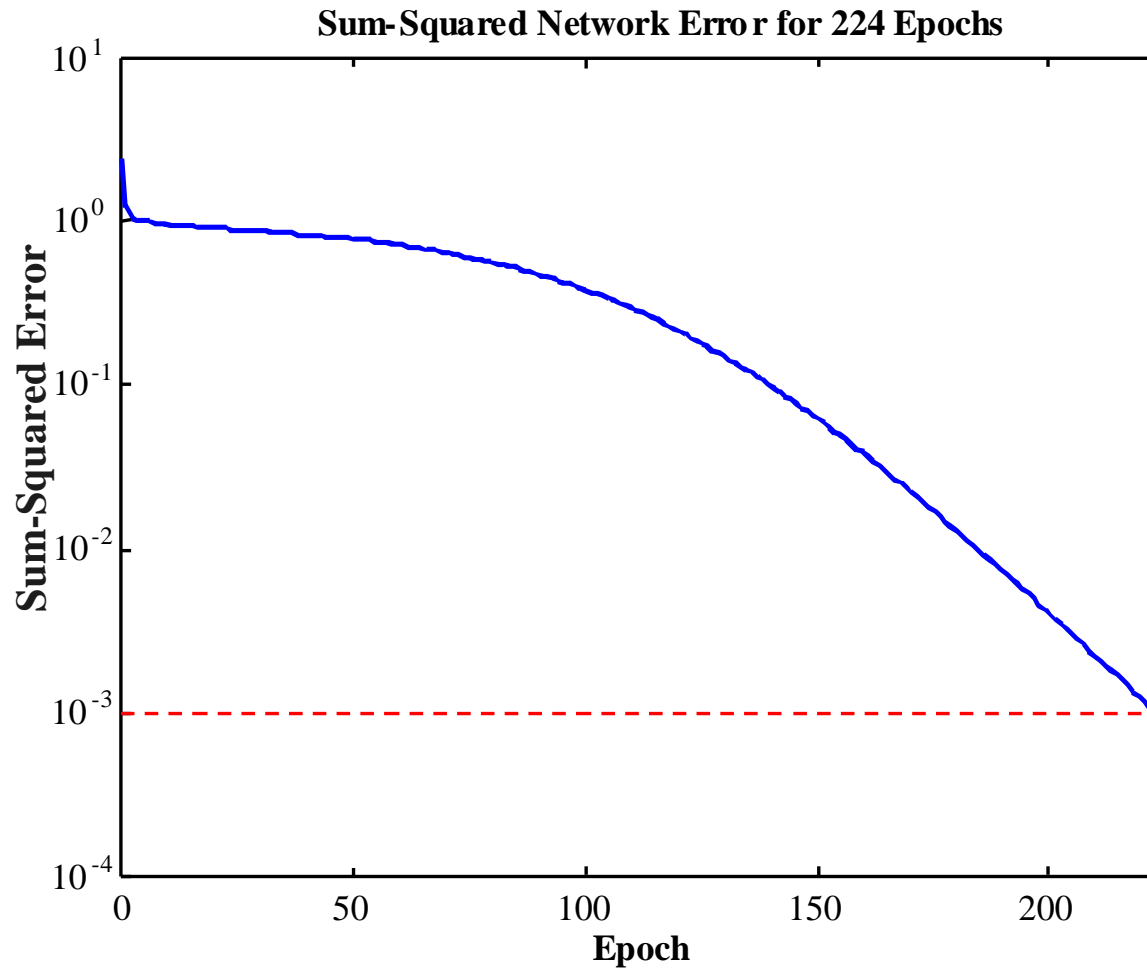
$$\theta_4 = \theta_4 + \Delta\theta_4 = -0.1 + 0.0015 = -0.0985$$

$$\theta_5 = \theta_5 + \Delta\theta_5 = 0.3 + 0.0127 = 0.3127$$

- The training process is repeated until the sum of squared errors is less than 0.001.



Learning curve for operation XOR



Final results of three-layer network learning

Inputs		Desired output y_d	Actual output y_5	Error e	Sum of squared errors
x_1	x_2				
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	

Training Modes

- **Incremental mode (on-line, sequential, stochastic, or per-observation):** Weights updated after each instance is presented
- **Batch mode (off-line or per-epoch):** Weights updated after all the patterns are presented



NN: Universal Approximator

- Any desired continuous function can be implemented by a three-layer network given sufficient number of hidden units, proper nonlinearities and weights (Kolmogorov)



NN DESIGN ISSUES

- Data representation
- Network Topology
- Network Parameters
- Training



Data Representation

- Data representation **depends on the problem.**
- In general ANNs work on continuous (real valued) attributes. Therefore symbolic attributes are encoded into continuous ones.
- Attributes of different types may have different ranges of values which affect the training process.
- **Normalization** may be used, like the following one which scales each attribute to assume values between 0 and 1.

$$x_i = \frac{x_i - \min_i}{\max_i - \min_i}$$

for each value x_i of i^{th} attribute, \min_i and \max_i are the minimum and maximum value of that attribute over the training set.



Network Topology

- The number of **layers** and **neurons** depend on the specific task.
- In practice this issue is solved by **trial and error**.
- Two types of adaptive algorithms can be used:
 - start from a large network and successively remove some neurons and links until network performance degrades.
 - begin with a small network and introduce new neurons until performance is satisfactory.



Network parameters

- How are the **weights** initialized?
- How is the **learning rate** chosen?
- How many **hidden layers** and how many **neurons**?
- How many examples in the **training set**?



Initialization of weights

- In general, initial weights are randomly chosen, with typical values between **-1.0** and **1.0** or **-0.5** and **0.5**.
- If some inputs are much larger than others, random initialization may bias the network to give much more importance to larger inputs.
- In such a case, weights can be initialized as follows:

$$W_{ij} = \pm \frac{1}{2N} \sum_{i=1, \dots, N} \frac{1}{|x_i|}$$

For weights from the input to the first layer

$$W_{jk} = \pm \frac{1}{2N} \sum_{i=1, \dots, N} \frac{1}{\varphi(\sum w_{ij} x_i)}$$

For weights from the first to the second layer



Choice of learning rate

- The right value of η depends on the application.
- Values between 0.1 and 0.9 have been used in many applications.
- Other heuristics is that adapt η during the training as described in previous slides.
- It is common to start with large values and decrease monotonically.
 - Start with 0.9 and decrease every 5 epochs
 - Use a Gaussian function
 - $\eta = 1/k$
 - ...



Size of Training set

- Rule of thumb:
 - the number of training examples should be at least five to ten times the number of weights of the network.
- Other rule:

$$N > \frac{|W|}{(1 - a)}$$

$|W|$ = number of weights
 a = expected accuracy on test set



MLP- Summary

- Design the **architecture**, choose **activation functions** (e.g. sigmoids)
- Choose a way to **initialize the weights** (e.g. random initialization)
- Choose a **loss function** (e.g. log loss) to measure how well the model fits training data
- Choose an **optimizer** (typically an SGD variant) to update the weights



