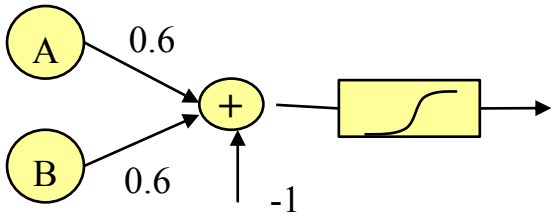
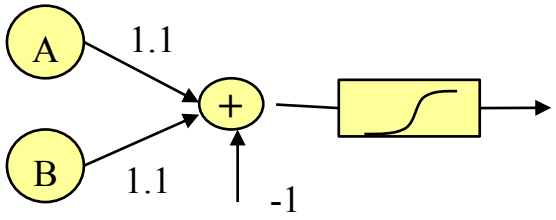
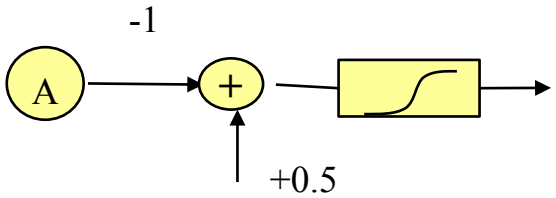
Lecture 4
Non-Linear Classifiers

Perceptrons and logic operations

➔ **AND, OR and NOT operations can be implemented by means of neurons**







A	B	g(x)	out
0	0	-1	0
0	1	-0.4	0
1	0	-0.4	0
1	1	0.2	1

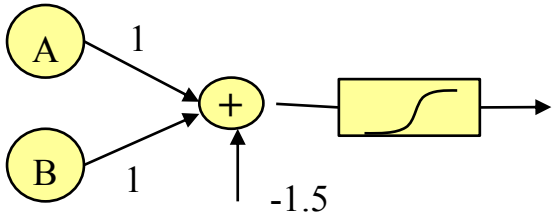
A	B	g(x)	out
0	0	-1	0
0	1	0.1	1
1	0	0.1	1
1	1	1.2	1

B	g(x)	out
0	0.5	1
1	-0.5	0

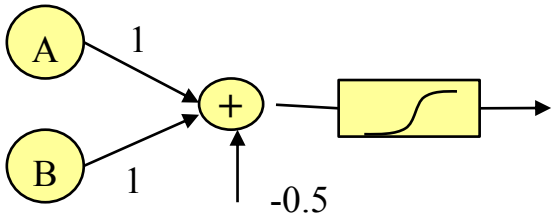
Perceptrons and logic operations

➔ Many solutions are possible..

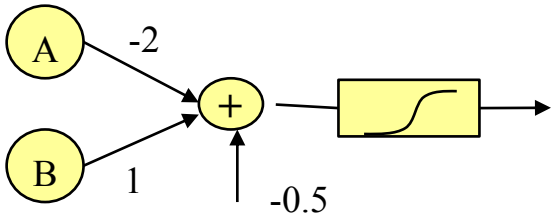
⇨ It is reassuring that logics can be expressed as neurons, if our mind works similarly to perceptrons!



A	B	g(x)	out
0	0	-1.5	0
0	1	-0.5	0
1	0	-0.5	0
1	1	0.5	1



A	B	g(x)	out
0	0	-0.5	0
0	1	0.5	1
1	0	0.5	1
1	1	1.5	1

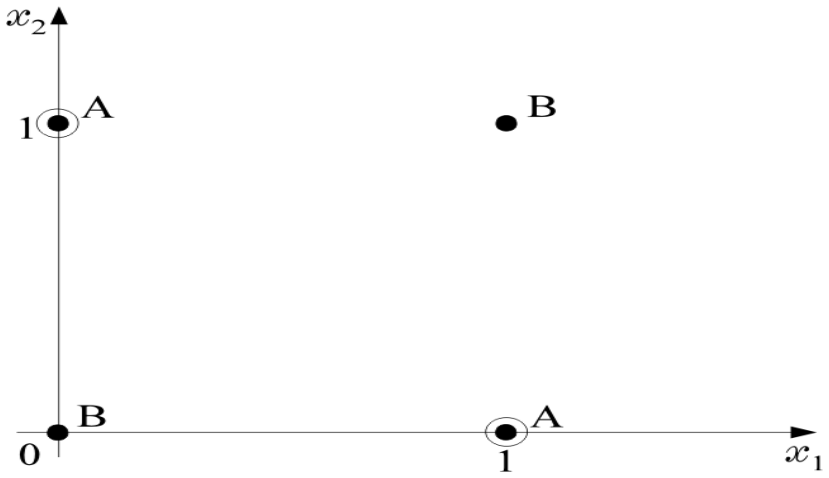


A	B	g(x)	out
0	0	-0.5	0
0	1	0.5	1
1	0	-2.5	0
1	1	-1.5	0

B and $\bar{A}$

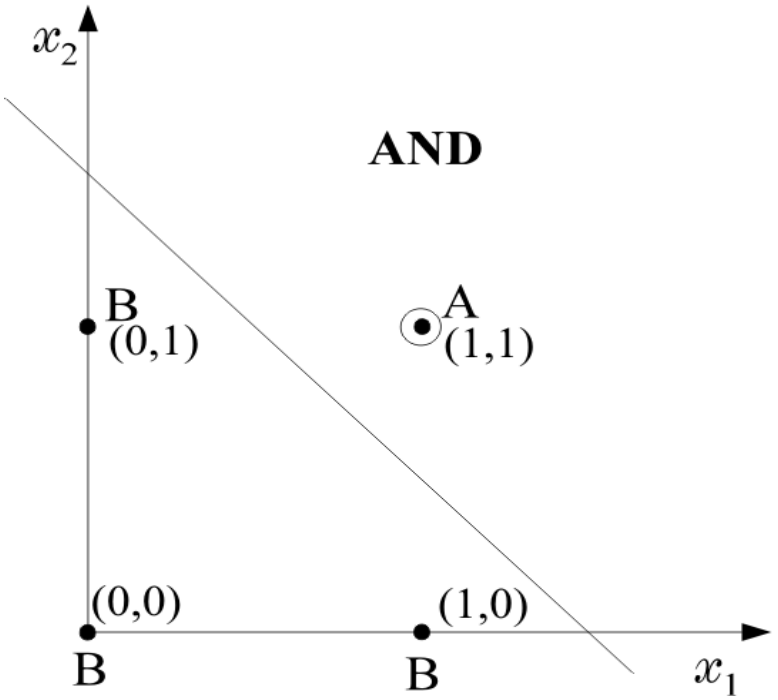
The XOR problem

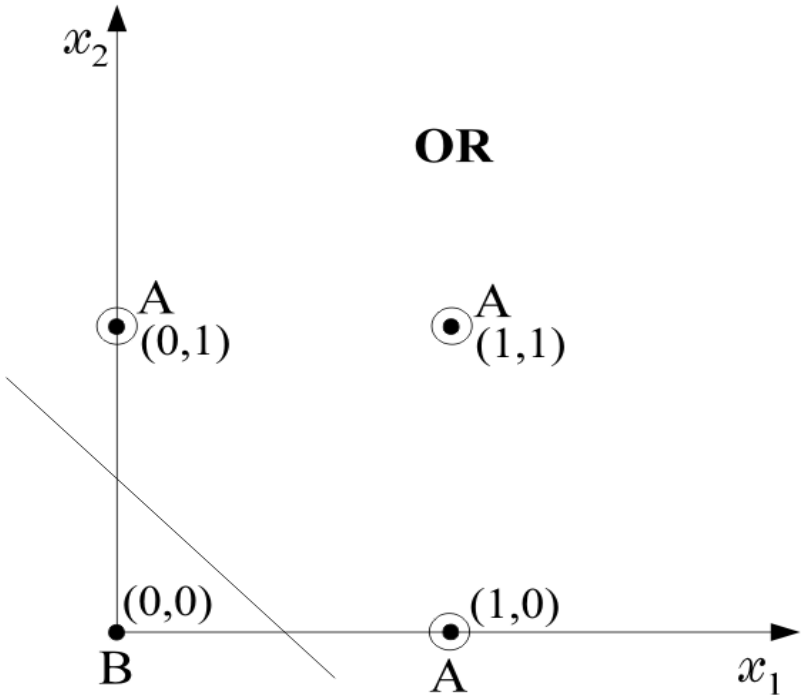
x_1	x_2	XOR	Class
0	0	0	B
0	1	1	A
1	0	1	A
1	1	0	B



Which decision region?

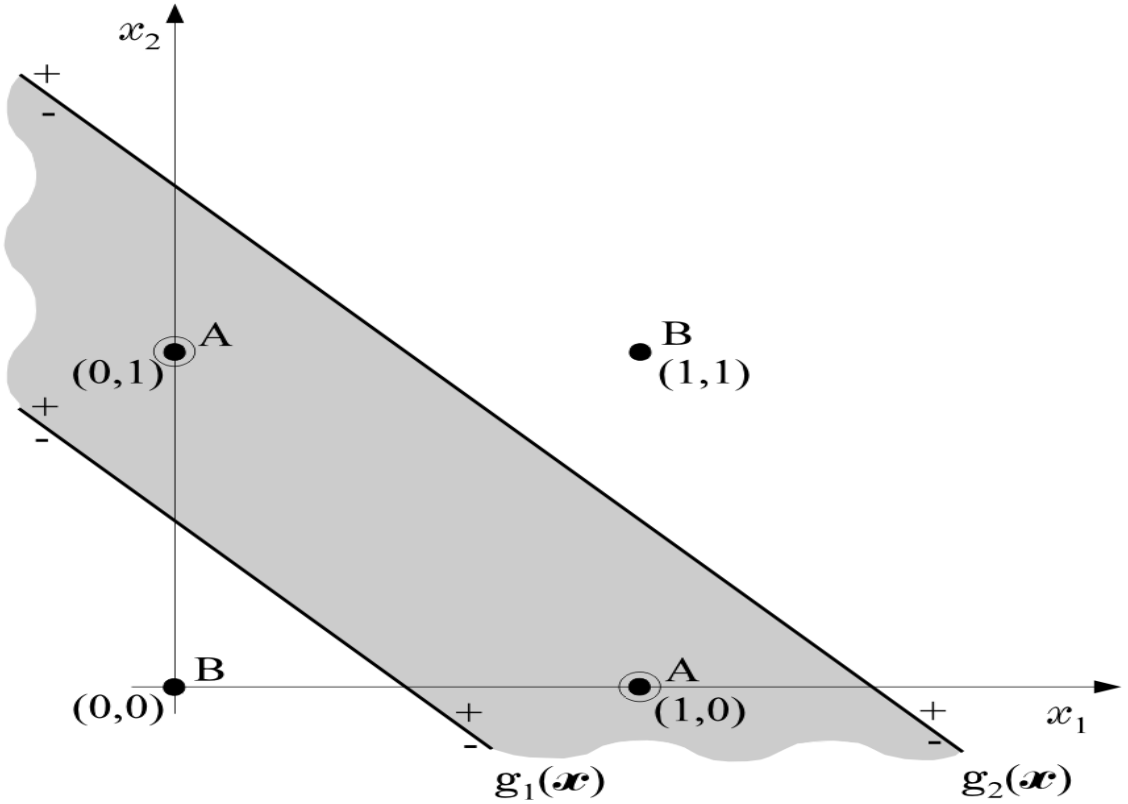
➔ **There is no single line (hyperplane) that separates class A from class B. On the contrary, AND and OR operations are linearly separable problems**





The two layer perceptron

⇨ For the XOR problem, draw **two**, instead, of one lines



⇨ Then class B is located outside the shaded area and class A inside. This is a two-phase design.

→Phase 1: Draw two lines (hyperplanes)

$$g_1(\underline{x}) = g_2(\underline{x}) = 0$$

Each of them is realized by a perceptron. The outputs of the perceptrons will be

$$y_i = f(g_i(\underline{x})) = \begin{cases} 0 \\ 1 \end{cases} \quad i = 1, 2$$

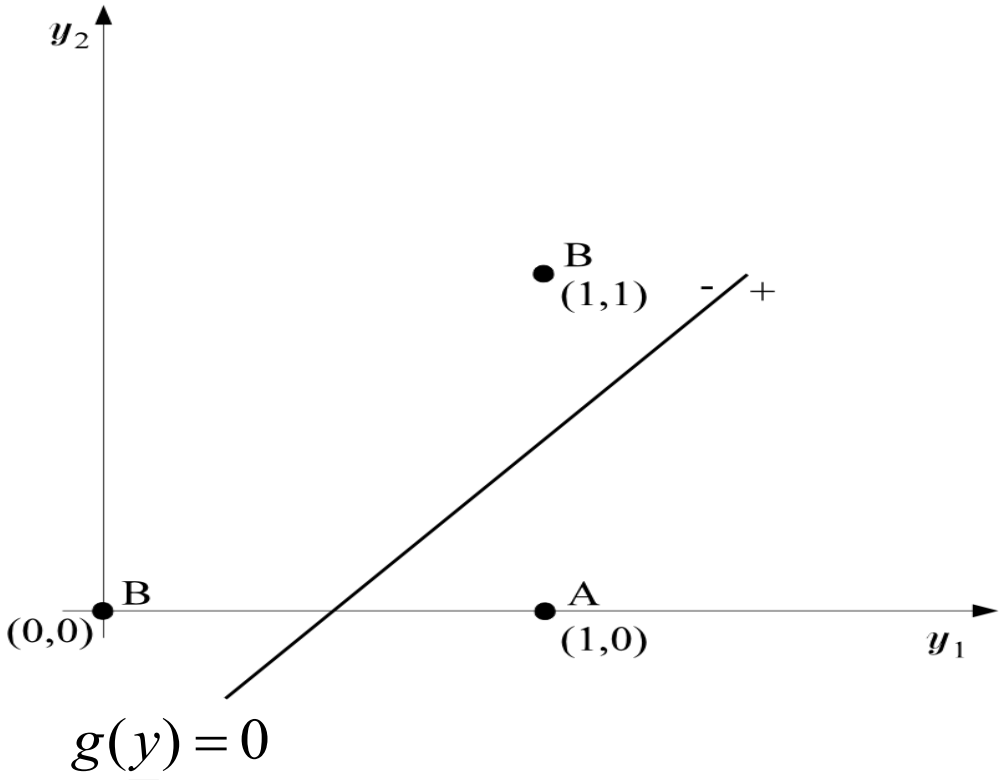
depending on the position of $\underline{x}$.

→Phase 2: Find the position of $\underline{x}$ *w.r.t.* both lines, based on the values of y_1, y_2.

1st phase				2nd phase
x_1	x_2	y_1	y_2	
0	0	0(-)	0(-)	B(0)
0	1	1(+)	0(-)	A(1)
1	0	1(+)	0(-)	A(1)
1	1	1(+)	1(+)	B(0)

Equivalently: The computations of the first phase perform a mapping $\underline{x} \rightarrow \underline{y} = [y_1, y_2]^T$

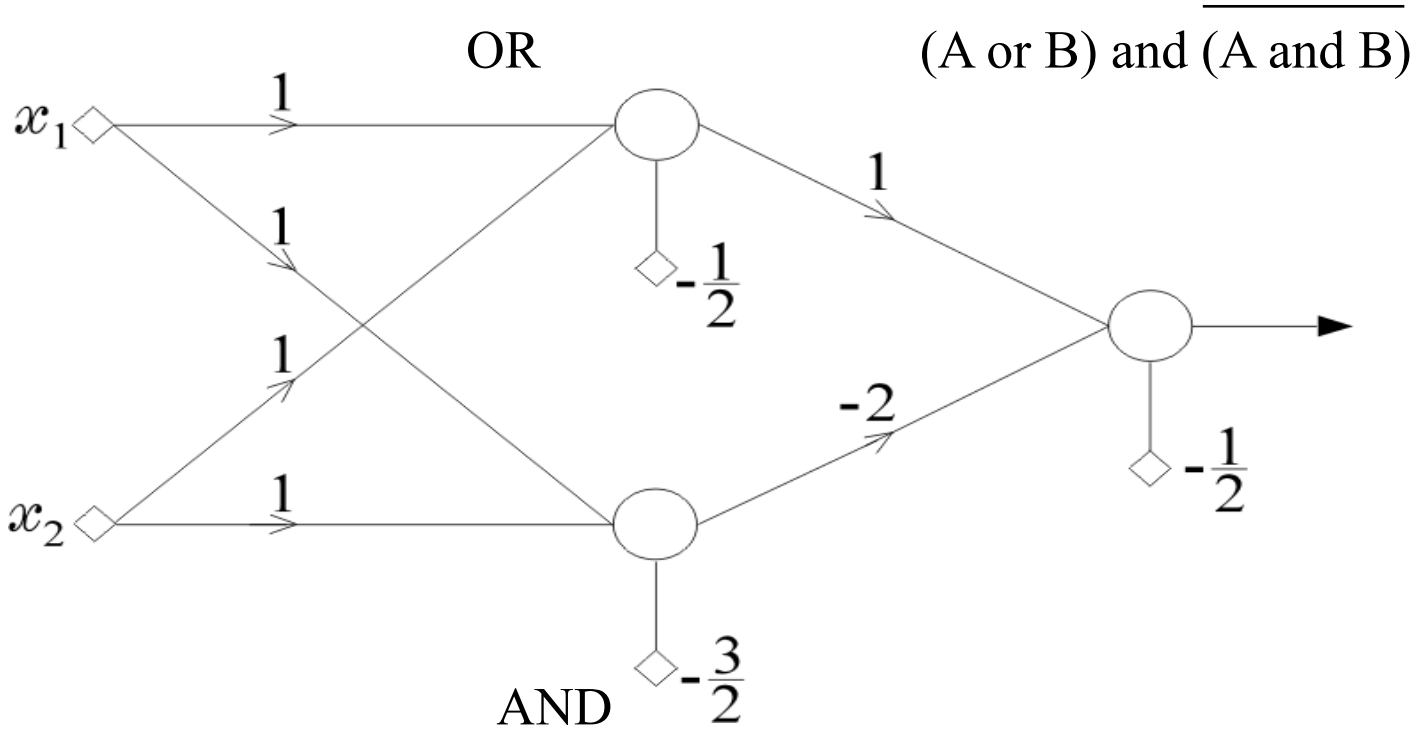
The decision is now performed on the transformed data. $\underline{y}$



This can be performed via a second line, which can also be realized by a perceptron.

⇨ Computations of the first phase perform a mapping that transforms the nonlinearly separable problem to a linearly separable one.

⇨ The architecture



Two Layer Network

→This is known as the two layer perceptron with one hidden and one output layer. The activation functions are

$$f(.) = \begin{cases} 0 \\ 1 \end{cases}$$

→The neurons (nodes) of the figure realize the following lines (hyperplanes)

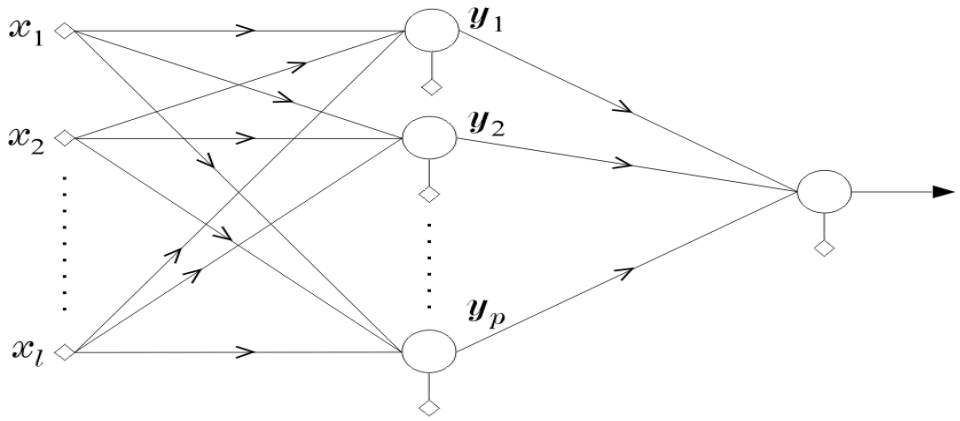
$$g_1(\underline{x}) = x_1 + x_2 - \frac{1}{2} = 0$$

$$g_2(\underline{x}) = x_1 + x_2 - \frac{3}{2} = 0$$

$$g(\underline{y}) = y_1 - 2y_2 - \frac{1}{2} = 0$$

➔Classification capabilities of the two-layer perceptron

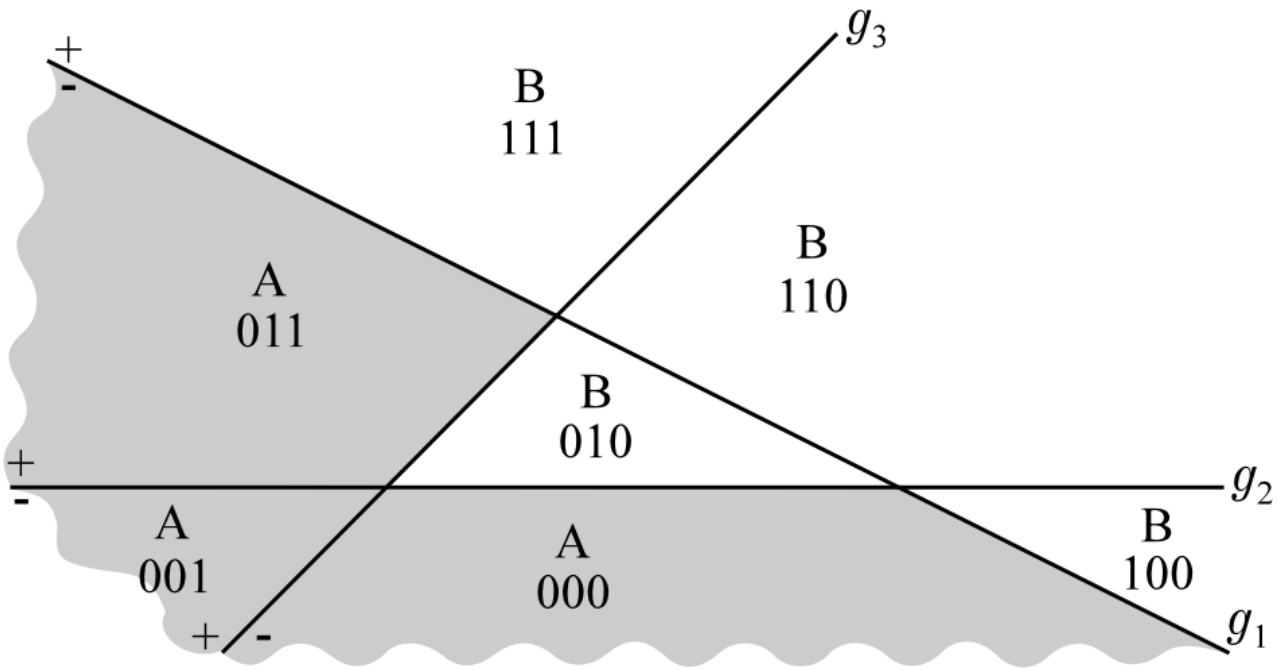
⇨ The mapping performed by the first layer neurons is onto the vertices of the unit side square



⇨ The more general case,

$$\underline{x} \rightarrow \underline{y} = [y_1, \ldots y_p]^T, \; y_i \in \{0,1\} \; i = 1, 2, \ldots p$$

⇨ Intersections of these hyperplanes form regions in the l-dimensional space. Each region corresponds to a vertex of the H_p unit hypercube.



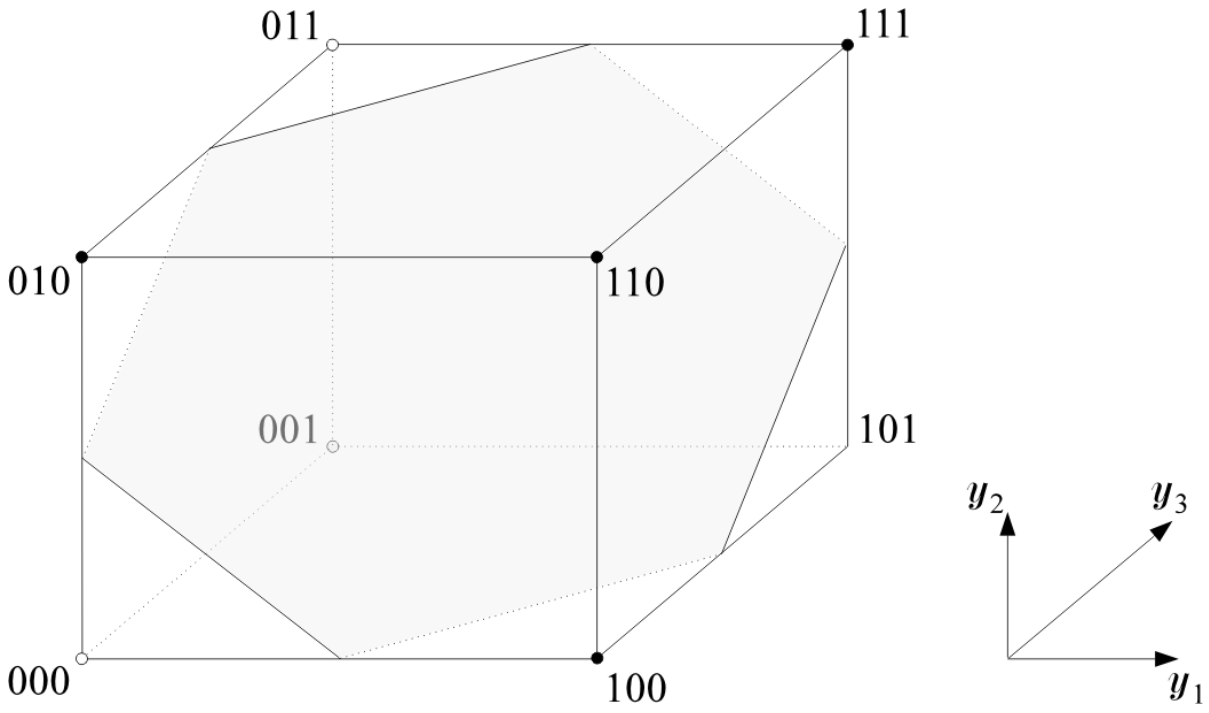
For example, the 001 vertex corresponds to the region which is located

to the (-) side of $g_1(\underline{x})=0$

to the (-) side of $g_2(\underline{x})=0$

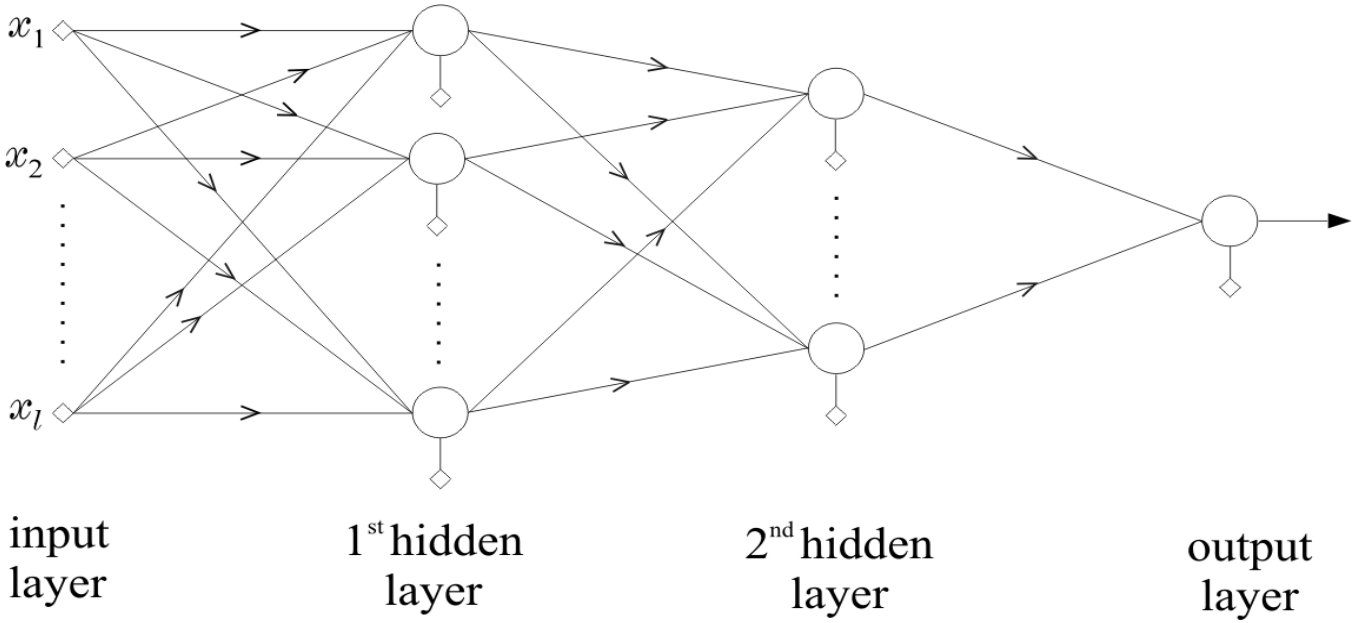
to the (+) side of $g_3(\underline{x})=0$

Not all the polyhedral regions can be separated with one hidden layer!



Three Layer-perceptrons

⇨ The architecture



⇨ This is capable to classify vectors into classes consisting of **ANY** union of polyhedral regions.

⇨ The idea is similar to the XOR problem. It realizes more than one planes in the $\underline{y} \in R^p$ space.

⇨ The reasoning

→For each vertex, corresponding to class, say A, construct a hyperplane which leaves THIS vertex on one side (+) and ALL the others to the other side (-).

→The output neuron realizes an OR gate

⇨ Overall:

The first layer of the network forms the hyperplanes, the second layer forms the regions and the output neuron forms the classes.

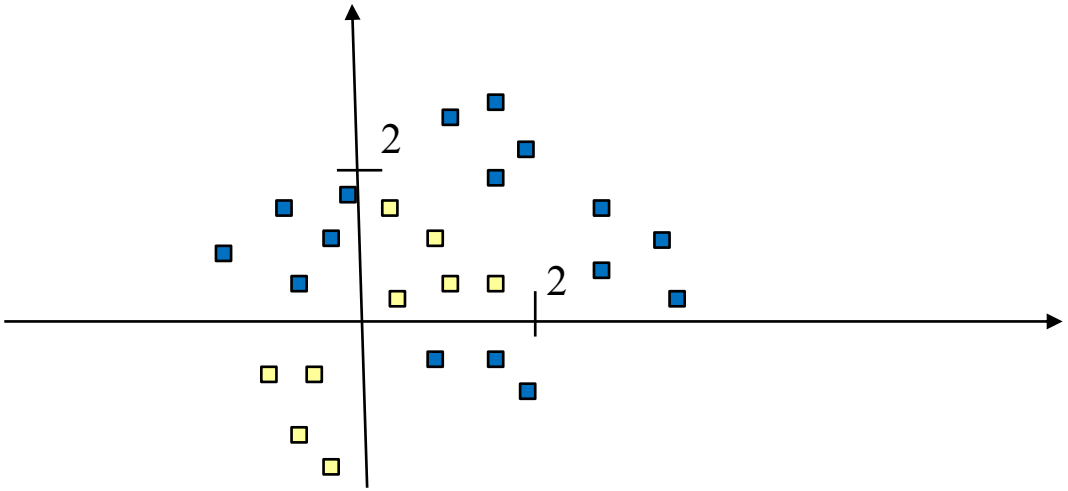
→Designing Multilayer Perceptrons

⇨ One direction is to adopt the above rationale and develop a structure that classifies **correctly all** the training patterns.

⇨ The other direction is to choose a structure and compute the synaptic weights to **optimize a cost function**.

Example

➔ **Find a MLP structure for solving the following classification problem**



And when regions cannot be visualized?

- ➔ Rather than finding an exact solution, the other approach is based on usual cost function minimizations
- ➔ But..
  - ⇨ How many layers?
  - ⇨ How many neurons for each layer?
  - ⇨ How optimizing a cost function based on combinations of non-differentiable functions?
- ➔ Backpropagation algorithm – proposed in 1983
  - ⇨ Many variants after this..

The Backpropagation Algorithm

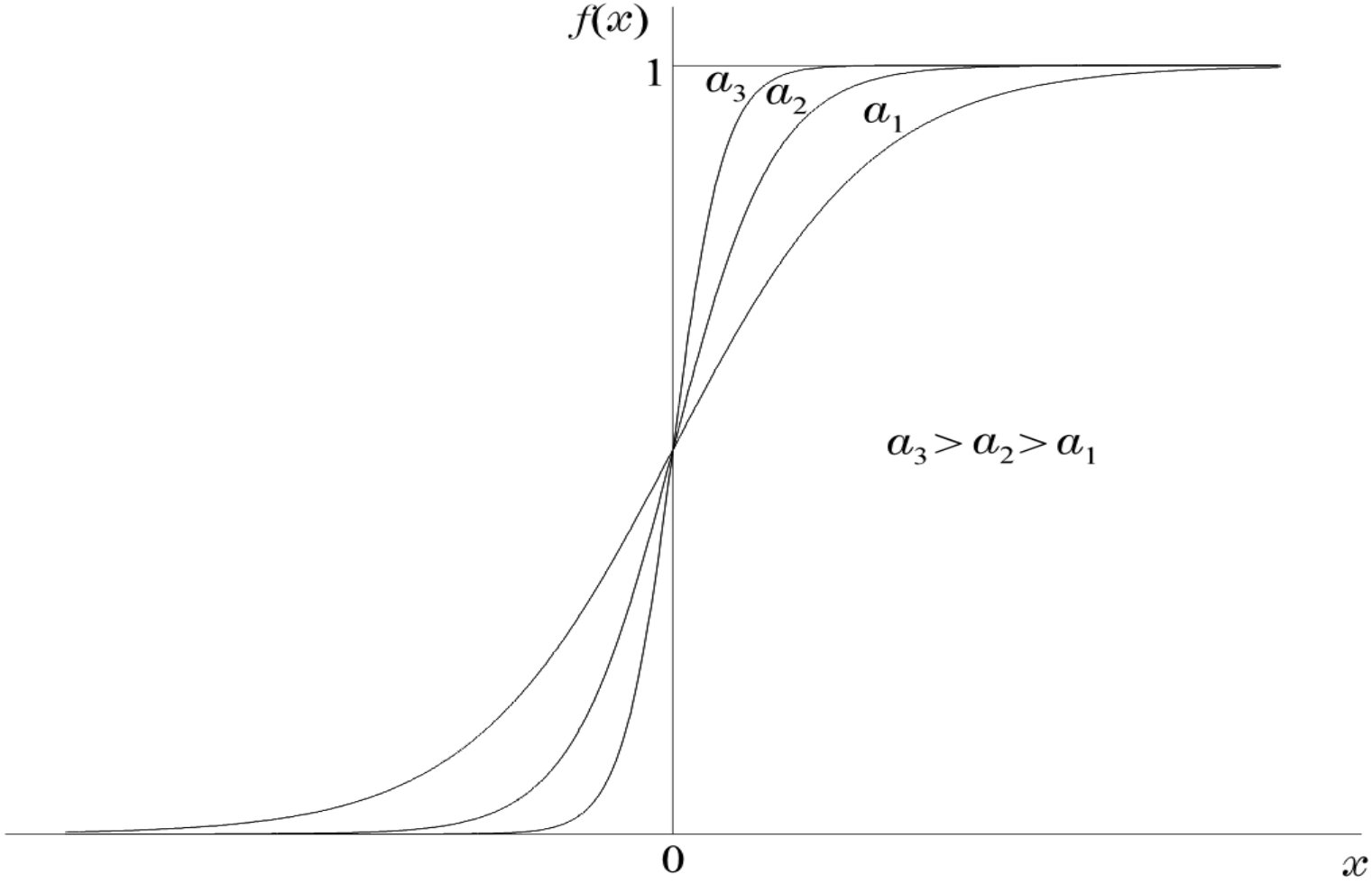
- ⇨ This is an algorithmic procedure that computes the synaptic weights iteratively, so that an adopted cost function is minimized (optimized)
- ⇨ In a large number of optimizing procedures, computation of derivatives are involved. Hence, discontinuous activation functions pose a problem, i.e.,

$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

- ⇨ There is always an escape path!!! The logistic function

$$f(x) = \frac{1}{1 + \exp(-ax)}$$

is an example. Other functions are also possible and in some cases more desirable.


$f(x)$
1
a_3
a_2
a_1
$a_3 > a_2 > a_1$
0
x

⇨ The steps:

→Adopt an optimizing cost function, e.g.,

» Least Squares Error

» Relative Entropy

between desired responses and actual responses of the network for the available training patterns. That is, from now on we have to live with errors. We only try to minimize them, using certain criteria.

→Adopt an algorithmic procedure for the optimization of the cost function with respect to the synaptic weights, e.g.,

» Gradient descent

» Newton's algorithm

» Conjugate gradient

→The task is a **nonlinear** optimization one. For the gradient descent method

$$\underline{w}_1^r(\text{new}) = \underline{w}_1^r(\text{old}) + \Delta\underline{w}_1^r$$

$$\Delta\underline{w}_1^r = -\mu\frac{\partial J}{\partial w_1^r}$$

⇨ The Procedure:

→Initialize unknown weights randomly with small values.

→Compute the gradient terms **backwards**, starting with the weights of the last (3rd) layer and then moving towards the first

→Update the weights

→Repeat the procedure until a termination procedure is met

⇨ Two major philosophies:

→Batch mode: The gradients of the last layer are computed once **ALL training data** have appeared to the algorithm, i.e., by summing up all error terms.

→Pattern mode: The gradients are computed every time a new **training data pair appears**. Thus gradients are based on successive individual errors.

How to compute the gradient?

➔ **Consider for simplicity a single input-output path**

⇨ Let output=act(w3*hidden2), hidden2=act(w2*hidden1), hidden1=act(w1*input)

→ i.e. output=act(w3*act(w2*act(w1*input)))

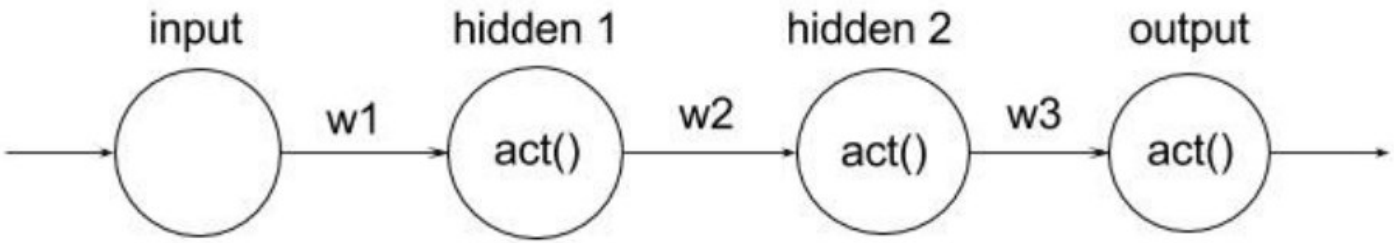
➔ **How does the coefficients affect the error on the desired output?**

⇨ Let's assume that the it exists the derivative of the act function

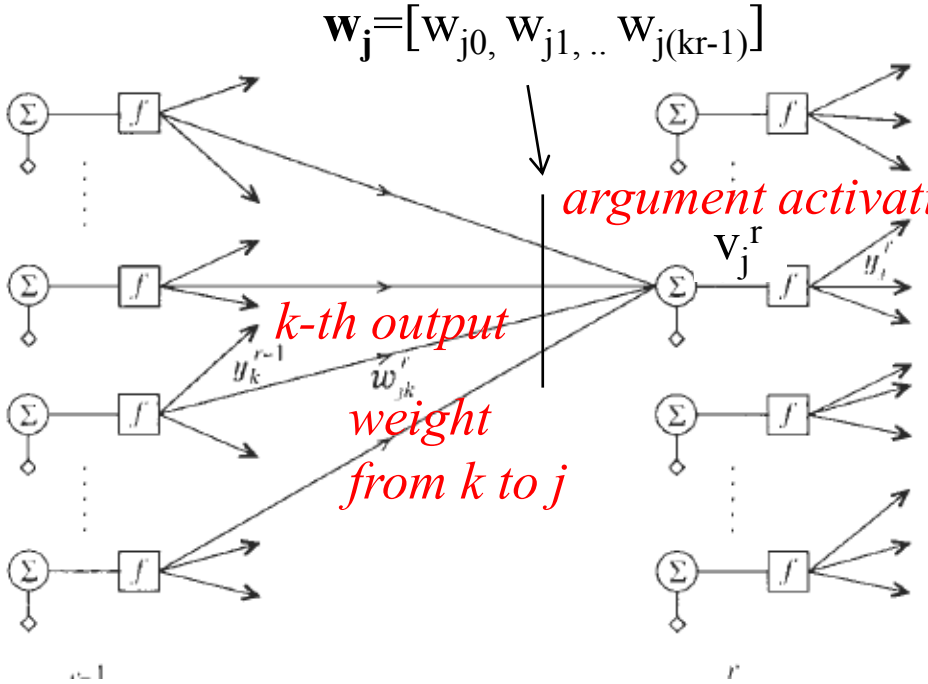
⇨ $\frac{\partial output}{\partial w1} = \frac{\delta output}{\delta hidden2} \cdot \frac{\delta hidden2}{\delta hidden1} \cdot \frac{\delta hidden1}{\delta w1}$

⇨ $\frac{\partial error}{\partial w1} = \frac{\delta error}{\delta output} \cdot \frac{\delta output}{\delta hidden2} \cdot \frac{\delta hidden2}{\delta hidden1} \cdot \frac{\delta hidden1}{\delta w1}$

➔ **Can we apply the usual gradient method?**



Definitions of variable involved



i-th training pair $(\underline{x_i}, \underline{y_i})$

$$v_j^r(\mathrm{i})=\sum_{k=1}^{kr-1} w_{jk}^r y_k^{r-1}(i) + w_{j0}^r = \sum_{k=0}^{kr-1} w_{jk}^r y_k^{r-1}(i)$$

$$y_k^r(i) = f\left(v_j^r(\mathrm{i})\right) = \hat{y}(k,i) \quad with\ r\ output\ layer$$

Error and cost function

➔ The error on the i-th training point depends on w_j^r by means of $v_j^r(i)$

$$\mathcal{E}(i) = \frac{1}{2}\sum_{m=1}^{k_L} e_m^2(i) \equiv \frac{1}{2}\sum_{m=1}^{k_L}(y_m(i) - \hat{y}_m(i))^2,$$

$$y_m^L(i) = f\left(v_j^L(\mathrm{i})\right) = \hat{y}_m(i)$$

$$\frac{\partial \mathcal{E}(i)}{\partial \boldsymbol{w}_j^r} = \frac{\partial \mathcal{E}(i)}{\partial \upsilon_j^r(i)} \frac{\partial \upsilon_j^r(i)}{\partial \boldsymbol{w}_j^r}$$

$\delta_j^r(\mathrm{i})$

$\mathbf{y}^{r-1}(\mathrm{i})$

➔ The cost function can be the total error on all the training points:

$$J = \sum_{i=1}^{N} \mathcal{E}(i)$$

Gradient computation

$$\mathcal{E}(i) \equiv \frac{1}{2}\sum_{m=1}^{k_L} e_m^2(i) \equiv \frac{1}{2}\sum_{m=1}^{k_L}(f(v_m^L(i)) - y_m(i))^2$$

$$\delta_j^L(i) = e_j(i) f'(v_j^L(i))$$

easy for the output layer!

and for the others??

$$\frac{\partial \mathcal{E}(i)}{\partial v_j^{r-1}(i)} = \sum_{k=1}^{k_r} \frac{\partial \mathcal{E}(i)}{\partial v_k^r(i)} \frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)}$$

$$\delta_j^{r-1}(i) = \sum_{k=1}^{k_r} \delta_k^r(i) \frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)}$$

$$\frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)} = \frac{\partial\left[\sum_{m=0}^{k_{r-1}} w_{km}^r y_m^{r-1}(i)\right]}{\partial v_j^{r-1}(i)}$$

$$\frac{\partial v_k^r(i)}{\partial v_j^{r-1}(i)} = w_{kj}^r f'(v_j^{r-1}(i))$$

Gradient computation

$$\delta_j^{r-1}(i) = \left[\sum_{k=1}^{k_r} \delta_k^r(i) w_{kj}^r\right] f'(v_j^{r-1}(i))$$

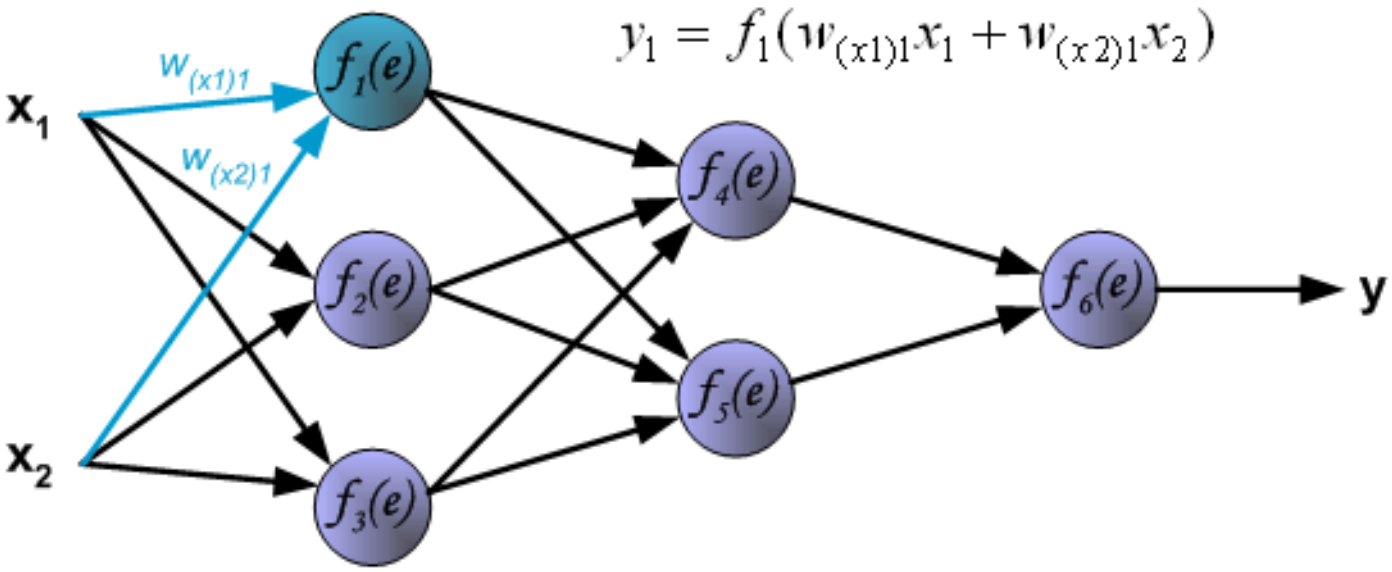
$$\delta_j^{r-1}(i) = e_j^{r-1}(i) f'(v_j^{r-1}(i))$$

we only miss the computation of f'(x)..

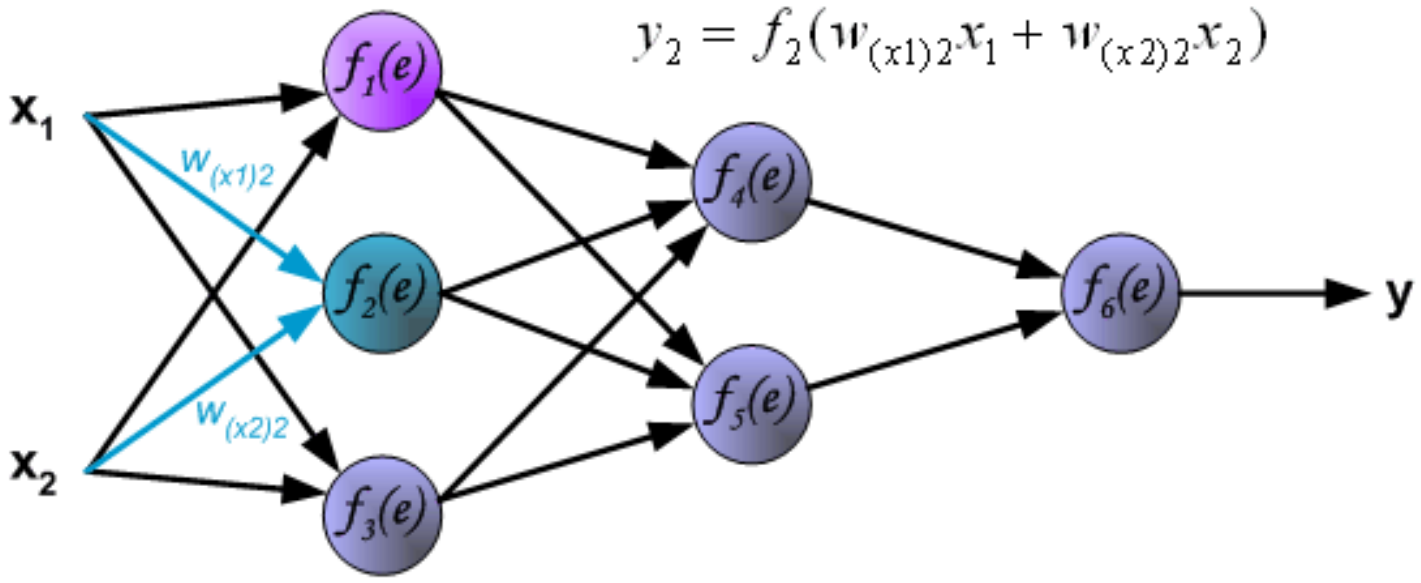
$$f'(x) = \alpha f(x)(1 - f(x))$$

Learning algorithm summary

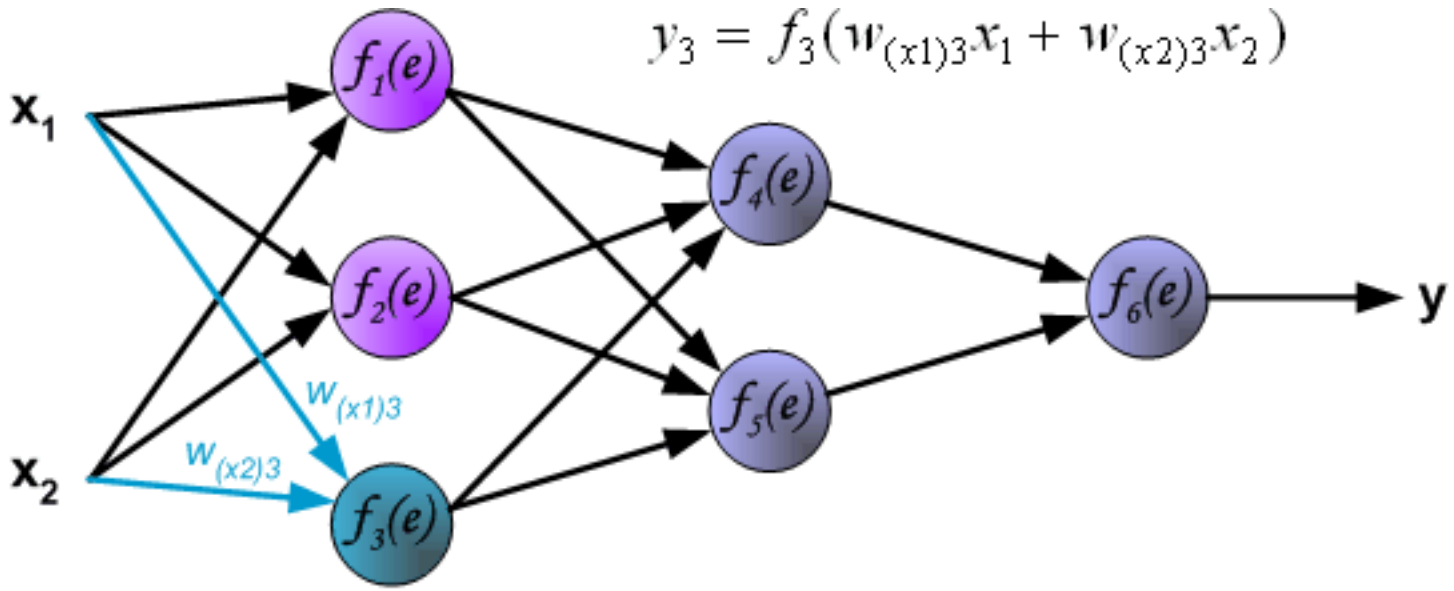
Training point:
features **(x1, x2)**, desired output **z**



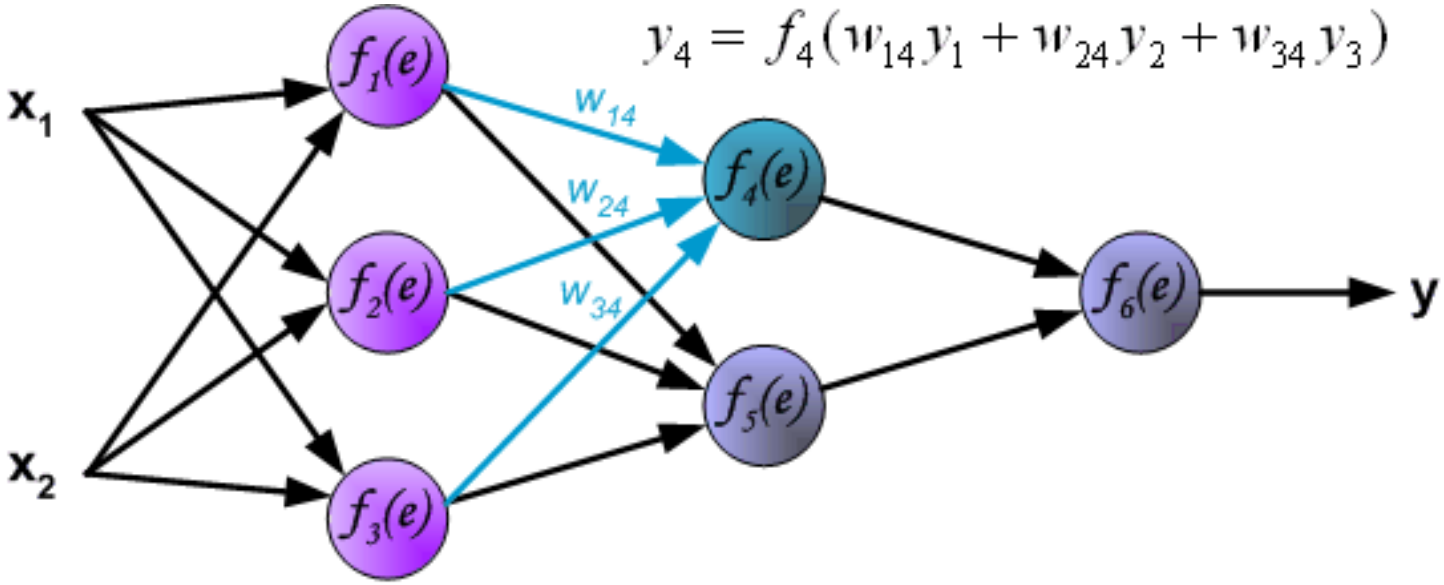
Forward phase



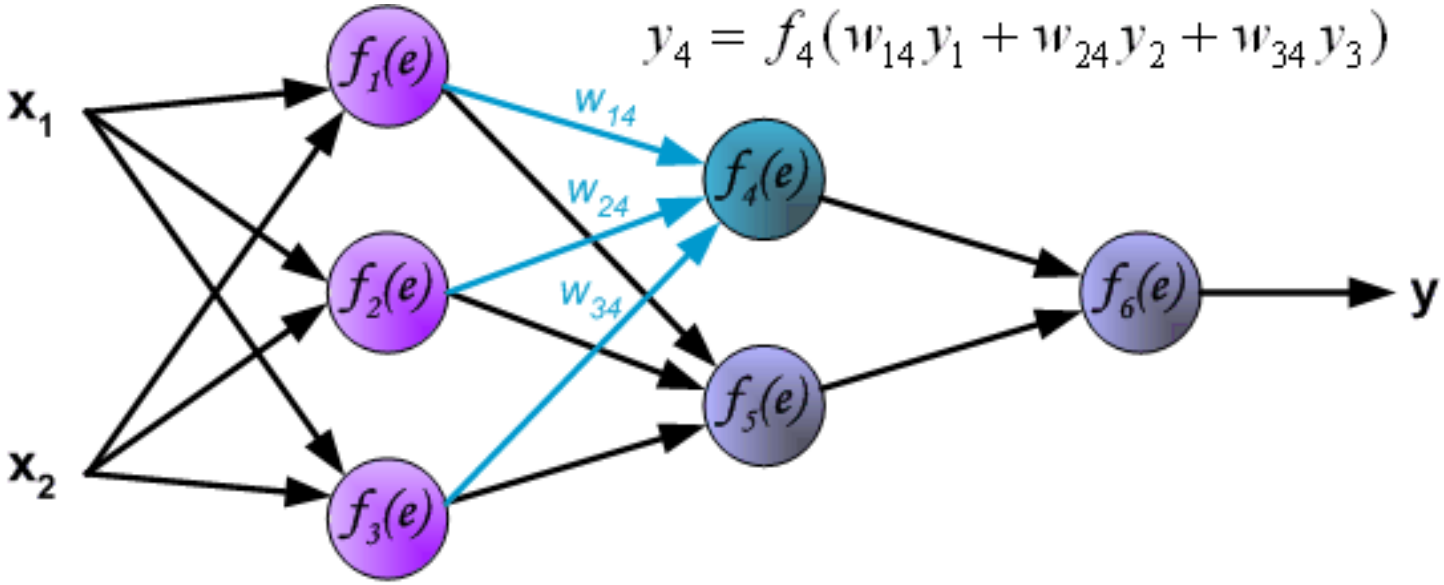
Forward phase



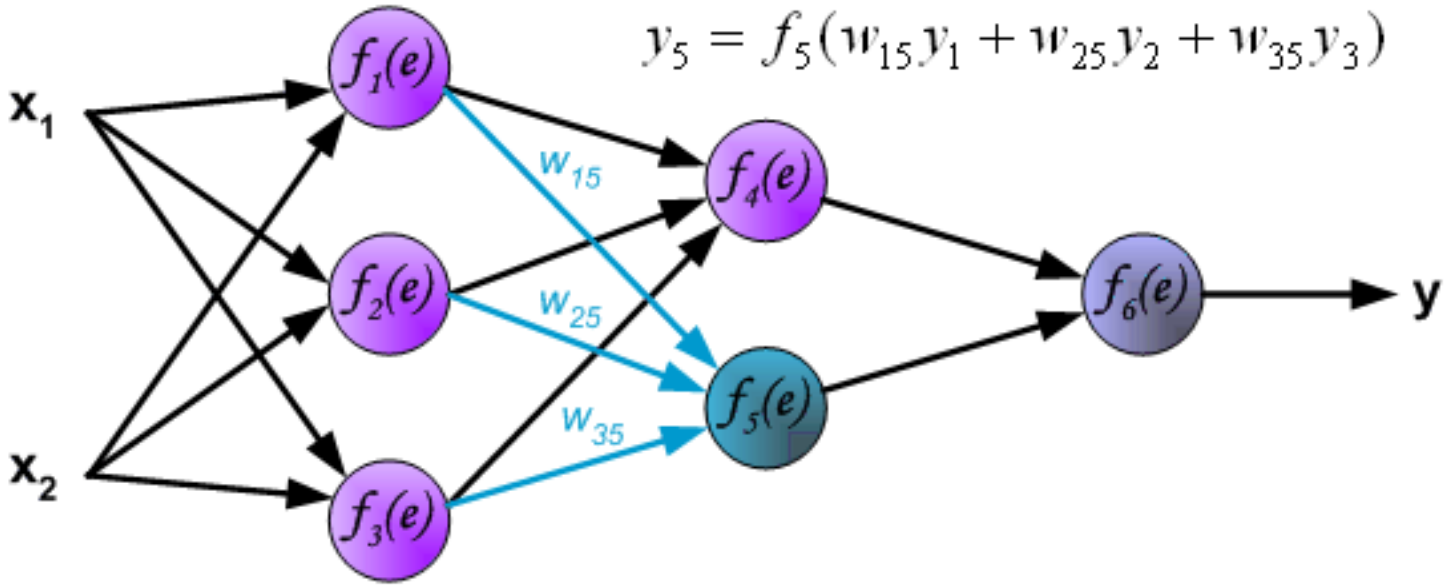
Forward phase



Forward phase

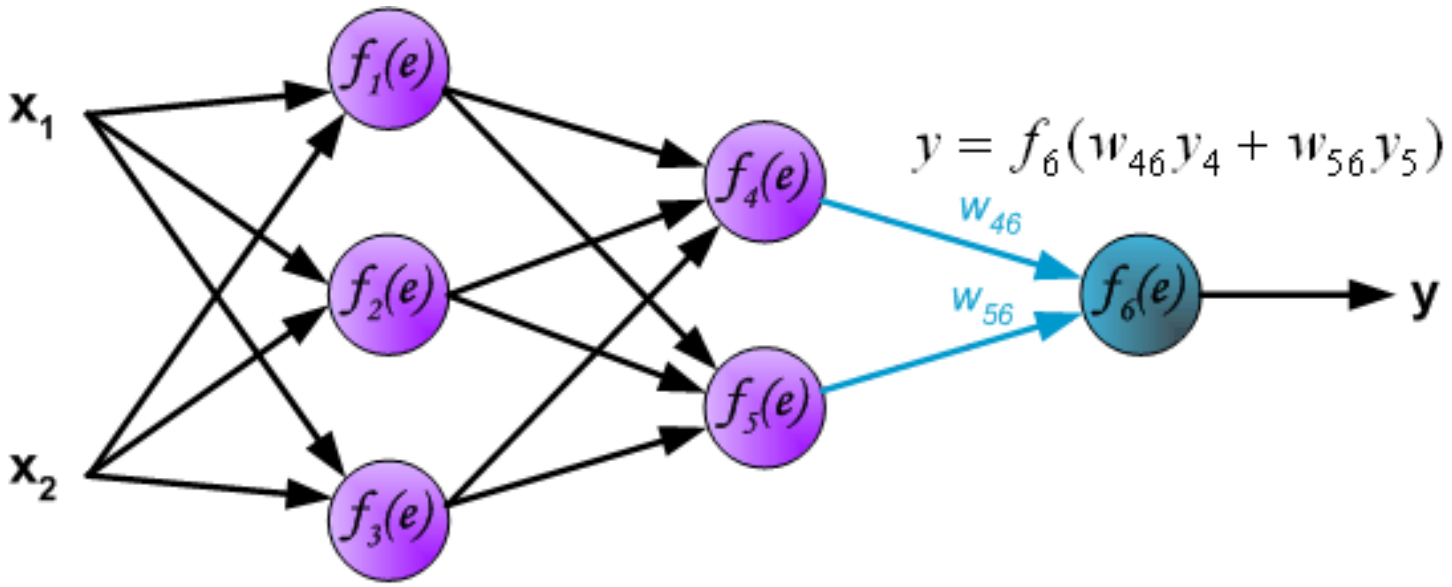


Forward phase



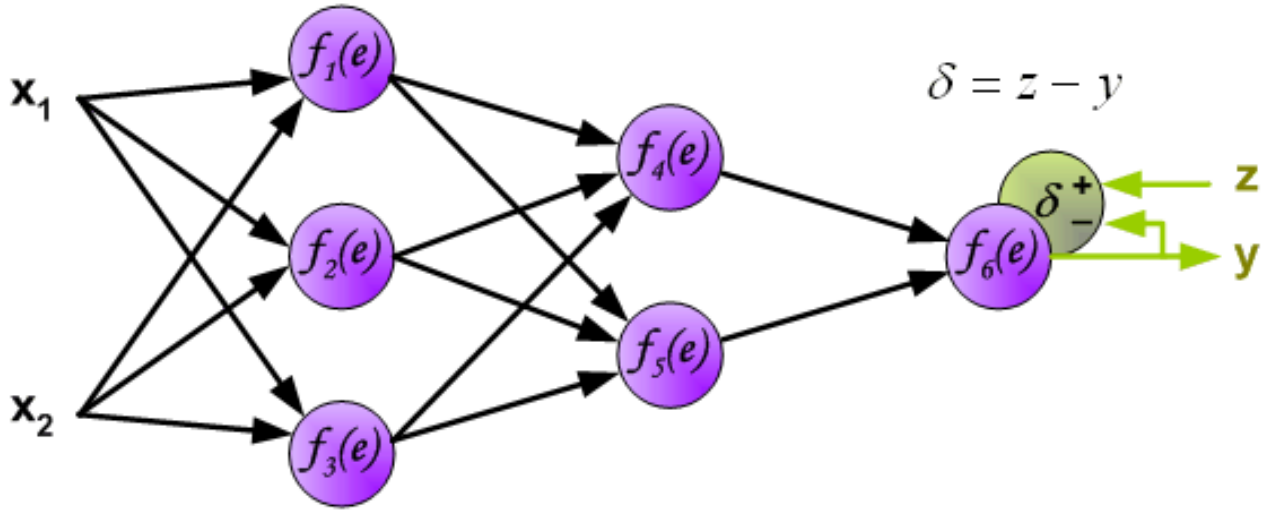
End of forward phase

Propagation of signals through the output layer.



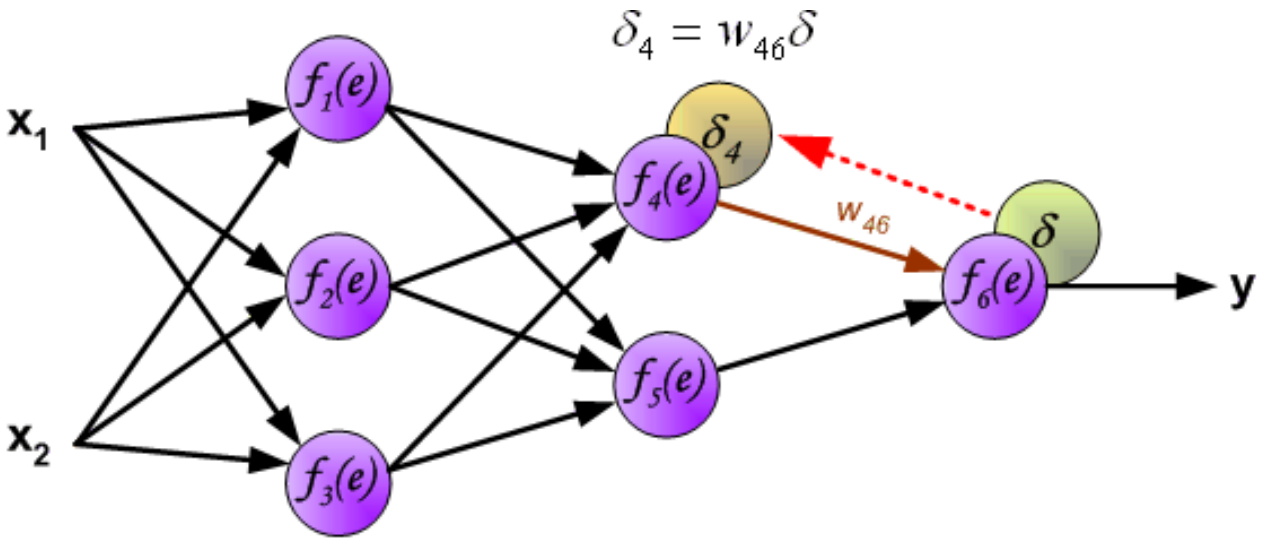
Backpropagation of errors

In the next algorithm step the output signal of the network *y* is compared with the desired output value (the target), which is found in training data set. The difference is called error signal *d* of output layer neuron

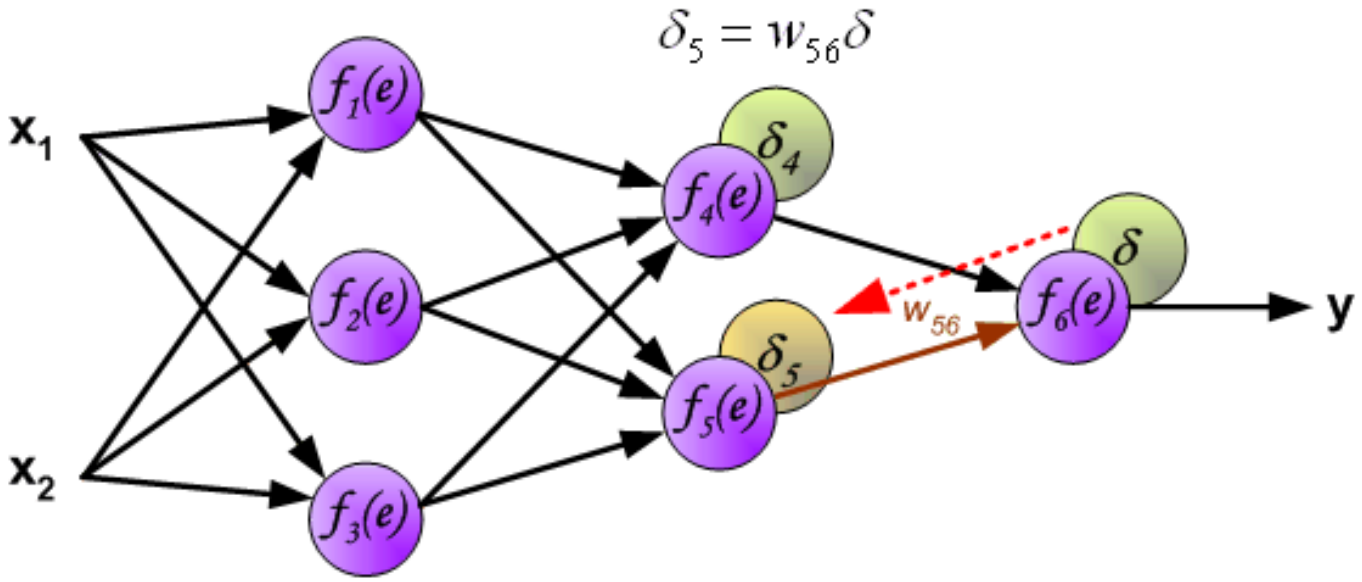


Backpropagation

The idea is to propagate error signal d (computed in single teaching step) back to all neurons, which output signals were input for discussed neuron.

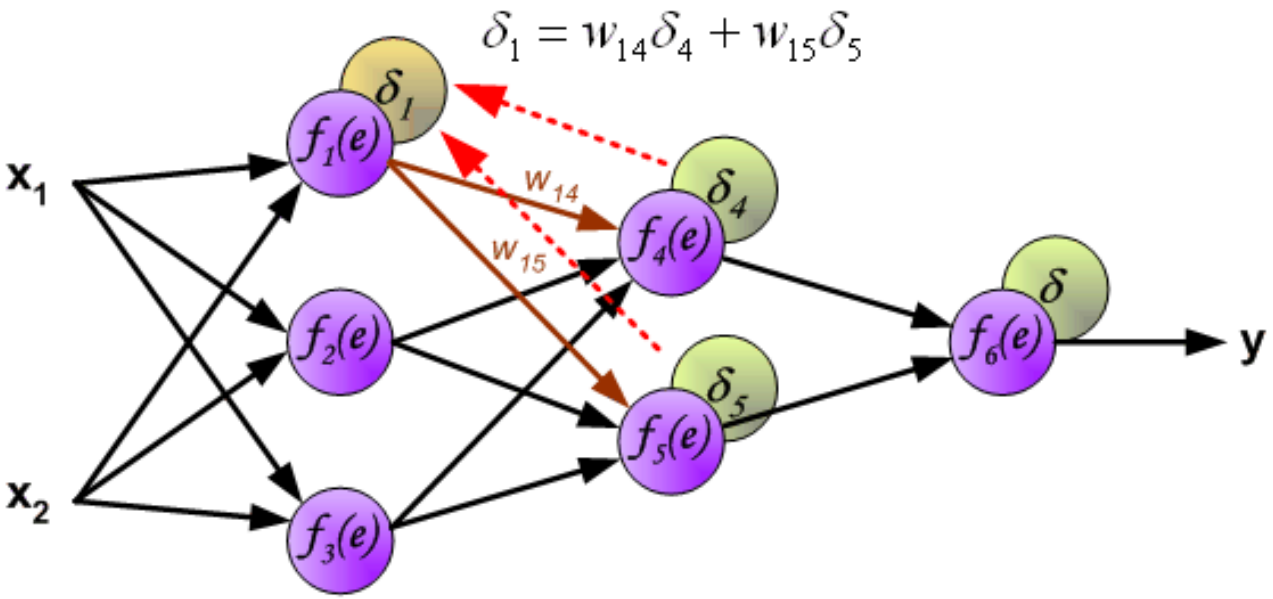


Backpropagation



Backpropagation

The weights' coefficients w_{mn} used to propagate errors back are equal to this used during computing output value. Only the direction of data flow is changed (signals are propagated from output to inputs one after the other). This technique is used for all network layers. If propagated errors came from few neurons they are added. The illustration is below:

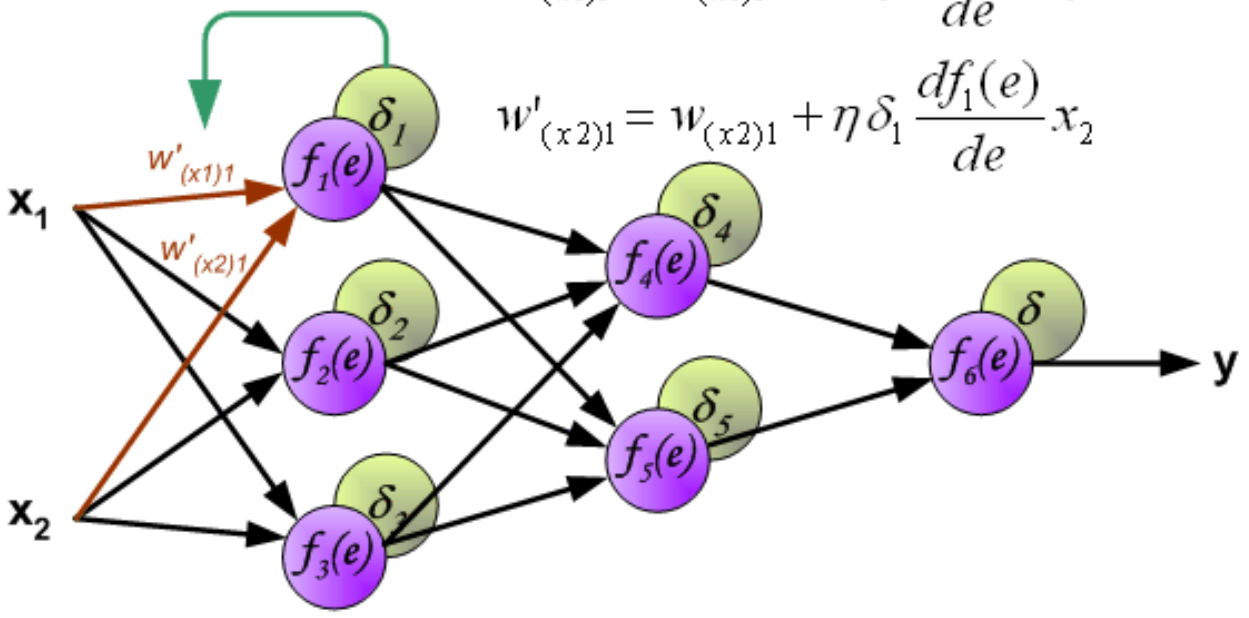


Backpropagation end

When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified. In formulas below *df(e)/de* represents derivative of neuron activation function (which weights are modified).

$$w'_{(x1)1} = w_{(x1)1} + \eta \delta_1 \frac{df_1(e)}{de} x_1$$

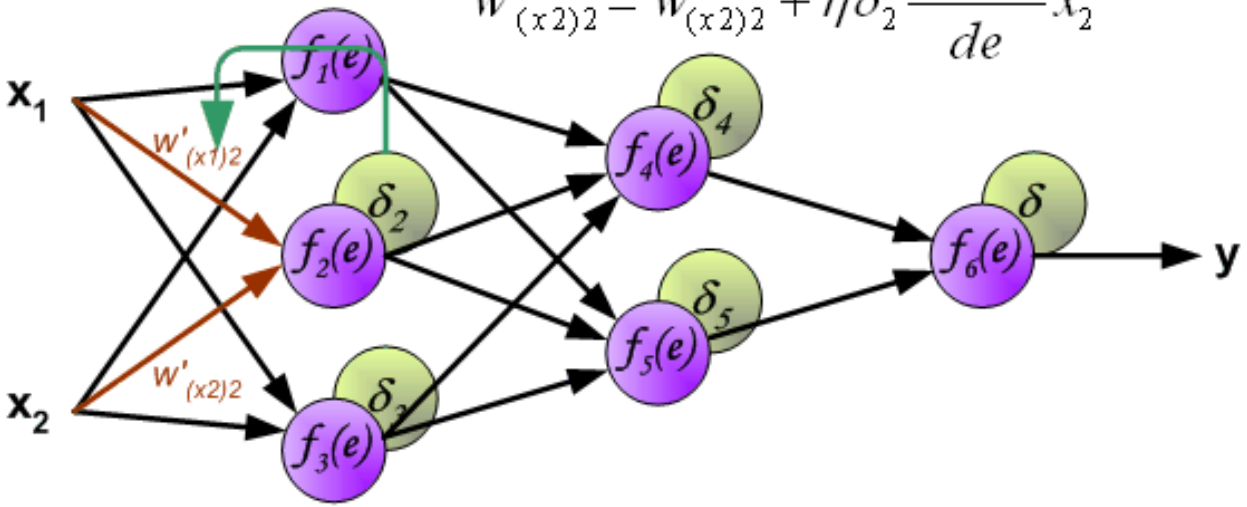
$$w'_{(x2)1} = w_{(x2)1} + \eta \delta_1 \frac{df_1(e)}{de} x_2$$



Coefficient updates

$$w'_{(x1)2} = w_{(x1)2} + \eta \delta_2 \frac{df_2(e)}{de} x_1$$

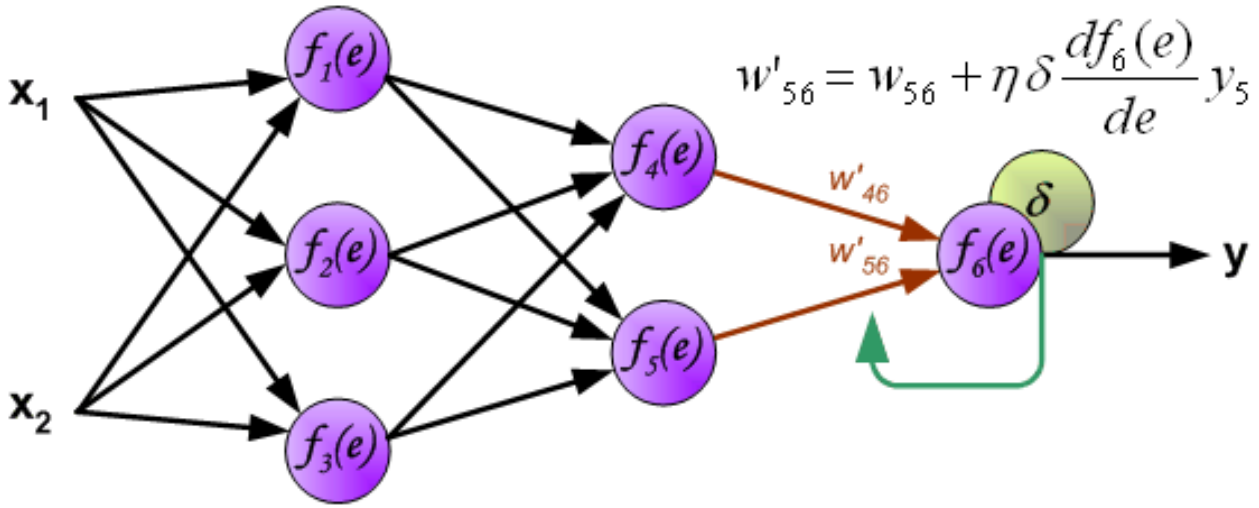
$$w'_{(x2)2} = w_{(x2)2} + \eta \delta_2 \frac{df_2(e)}{de} x_2$$

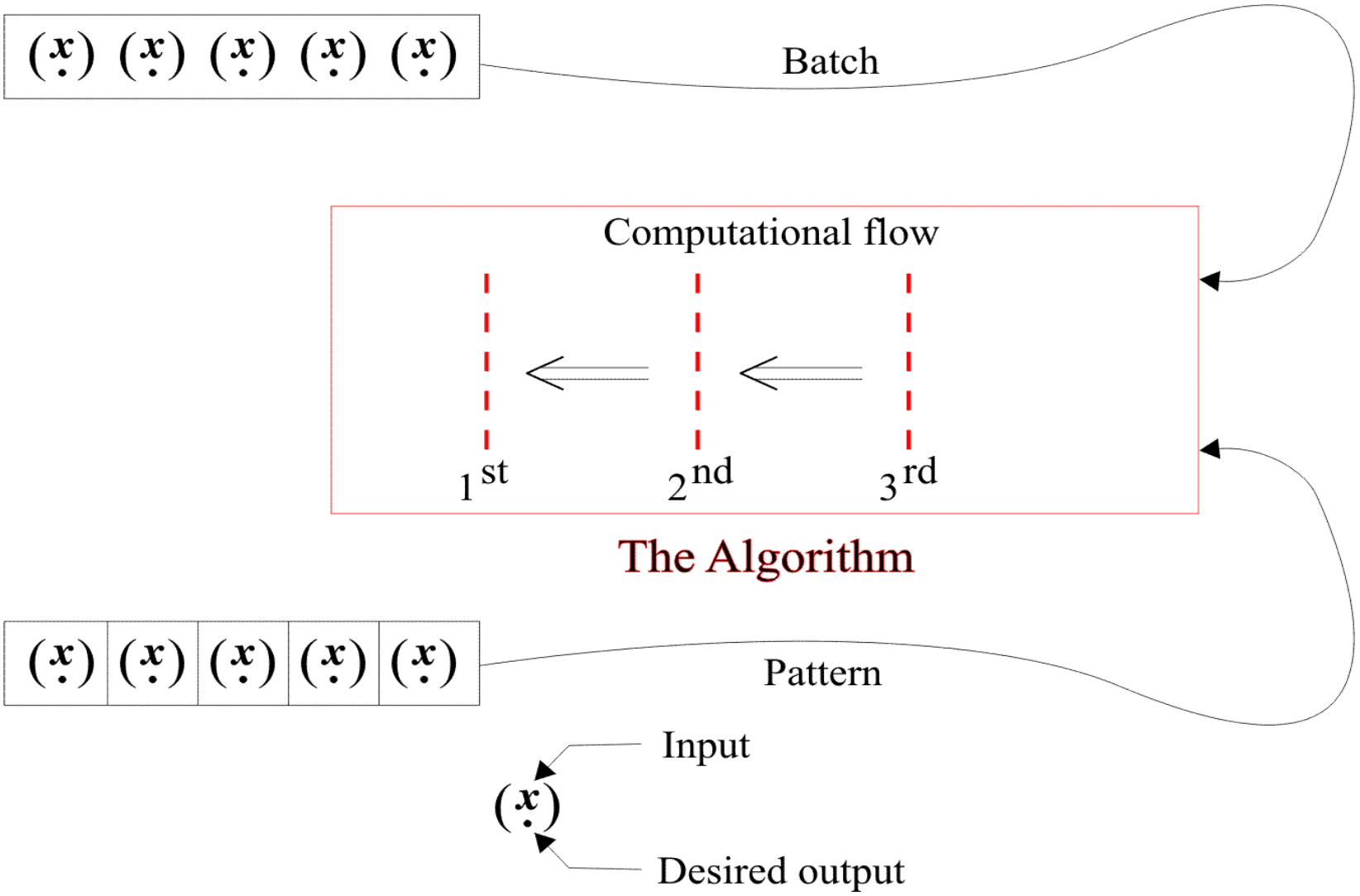


Coefficient updates

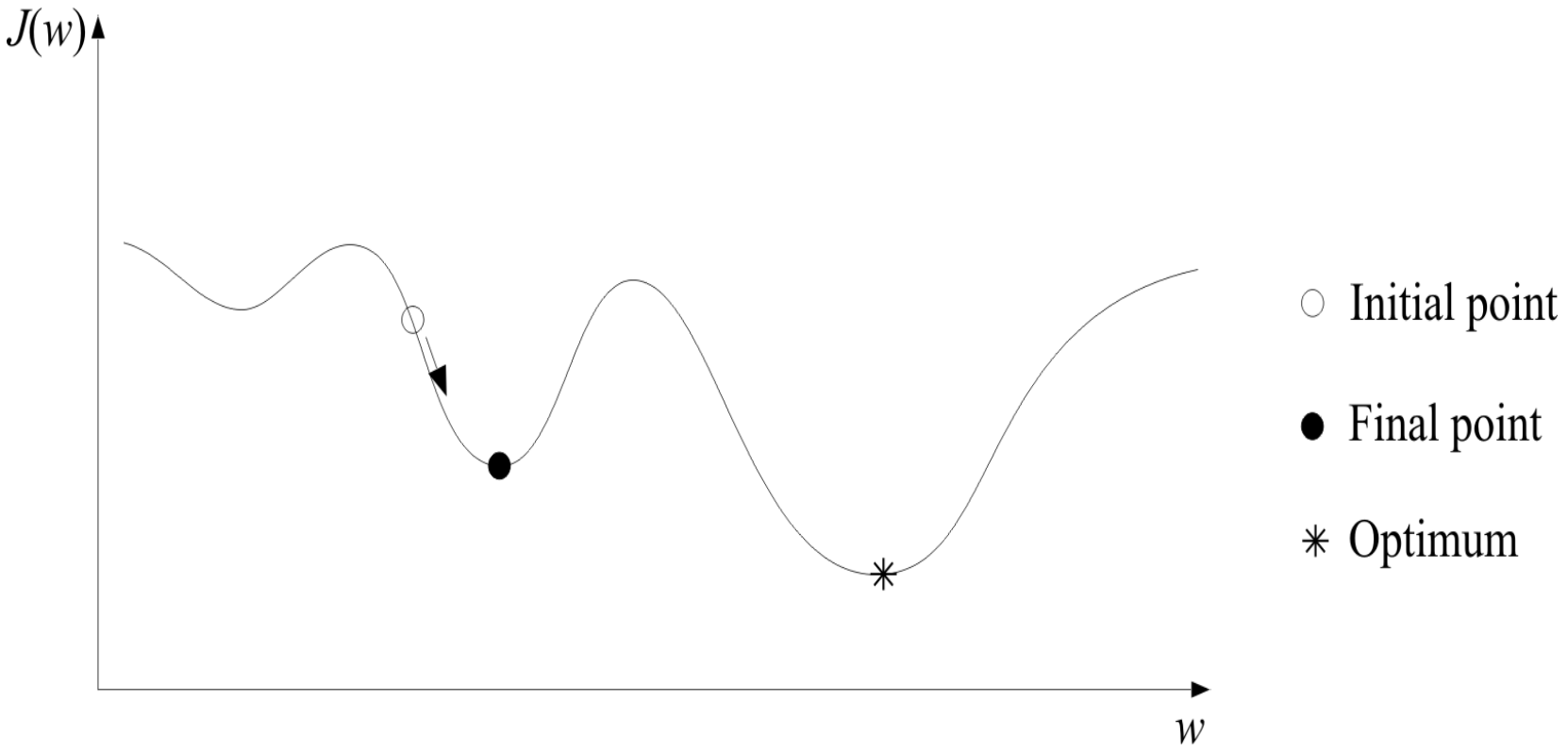
$$w'_{46} = w_{46} + \eta\,\delta\,\frac{df_6(e)}{de}\,y_4$$

$$w'_{56} = w_{56} + \eta\,\delta\,\frac{df_6(e)}{de}\,y_5$$




Batch
Computational flow
1st
2nd
3rd
The Algorithm
Pattern
Input
Desired output

⇨ A major problem: The algorithm may converge to a local minimum



⇨ The Cost function choice

Examples:

→**The Least Squares**

$$J = \sum_{i=1}^{N} E(i)$$

$$E(i) = \sum_{m=1}^{k} e_m^2(i) = \sum_{m=1}^{k} (y_m(i) - \hat{y}_m(i))^2$$

$$i = 1,2,...,N$$

$y_m(i) \rightarrow$ Desired response of the m^{th} output neuron (1 or 0) for $\underline{x}(i)$

$\hat{y}_m(i) \rightarrow$ Actual response of the m^{th} output neuron, in the interval [0, 1], for input $\underline{x}(i)$

⇨ The cross-entropy

$$J = \sum_{i=1}^{N} E(i)$$

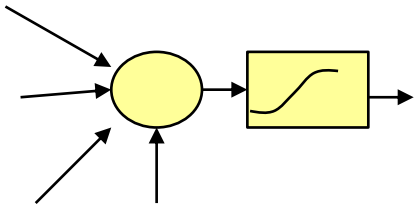
$$E(i) = \sum_{m=1}^{k} \left\{ y_m(i) \ln \hat{y}_m(i) + (1 - y_m(i)) \ln(1 - \hat{y}_m(i)) \right\}$$

This presupposes an interpretation of y and $\hat{y}$ as **probabilities**

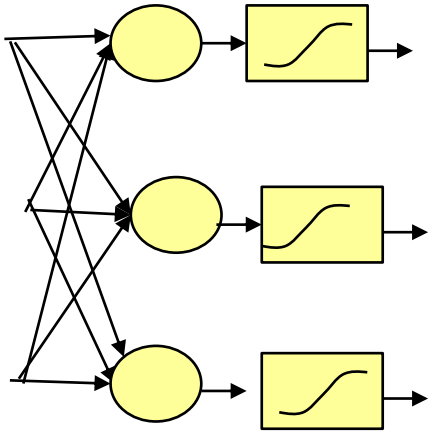
⇨ Classification error rate. This is also known as discriminative learning. Most of these techniques use a smoothed version of the classification error.

A simple MLP in Python

➔**A neural network can be built by combining multiple neuron layers, which in turns combine neurons**



Neuron



Neuron Layer

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/

Neuron Class

```
def __init__(self, bias):
        self.bias = bias
        self.weights = []

def calculate_output(self, inputs):
         self.inputs = inputs
         self.output = self.squash(self.calculate_total_net_input())
         return self.output

def calculate_total_net_input(self):
        total = 0
        for i in range(len(self.inputs)):
             total += self.inputs[i] * self.weights[i]
        return total + self.bias

def squash(self, total_net_input):
         return 1 / (1 + math.exp(-total_net_input))
```

Neuron Class: Tuning functions

```
# Determine how much the neuron's total input has to change to move closer to the expected output

def calculate_pd_error_wrt_total_net_input(self, target_output):
        return self.calculate_pd_error_wrt_output(target_output) * self.calculate_pd_total_net_input_wrt_input();

# The error for each neuron is calculated by the Mean Square Error method
def calculate_error(self, target_output):
        return 0.5 * (target_output - self.output) ** 2

# The partial derivate of the error with respect to actual output then is calculated by:
# = 2 * 0.5 * (target output - actual output) ^ (2 - 1) * -1

def calculate_pd_error_wrt_output(self, target_output):
        return -(target_output - self.output)

# The partial derivative of the squash function (total net input) with respect to input:
def calculate_pd_total_net_input_wrt_input(self):
        return self.output * (1 - self.output)

def calculate_pd_total_net_input_wrt_weight(self, index):
        return self.inputs[index]
```

Neuron Layer

```
def __init__(self, num_neurons, bias):
# Every neuron in a layer shares the same bias
        self.bias = bias if bias else random.random()
        self.neurons = []
        for i in range(num_neurons):
            self.neurons.append(Neuron(self.bias))

def inspect(self):
        print('Neurons:', len(self.neurons))
        for n in range(len(self.neurons)):
            print(' Neuron', n)
            for w in range(len(self.neurons[n].weights)):
                print('  Weight:', self.neurons[n].weights[w])
            print('  Bias:', self.bias)

def feed_forward(self, inputs):
        outputs = []
        for neuron in self.neurons:
            outputs.append(neuron.calculate_output(inputs))
        return outputs

def get_outputs(self):
        outputs = []
        for neuron in self.neurons:
            outputs.append(neuron.output)
        return outputs
```

Neural Network

```
class NeuralNetwork:
    LEARNING_RATE = 0.5
def __init__(self, num_inputs, num_hidden, num_outputs, hidden_layer_weights = None,
hidden_layer_bias = None, output_layer_weights = None, output_layer_bias = None):
        self.num_inputs = num_inputs
        self.hidden_layer = NeuronLayer(num_hidden, hidden_layer_bias)
        self.output_layer = NeuronLayer(num_outputs, output_layer_bias)
        self.init_weights_from_inputs_to_hidden_layer_neurons(hidden_layer_weights)
        self.init_weights_from_hidden_layer_neurons_to_output_layer_neurons(output_layer_weights)

def init_weights_from_inputs_to_hidden_layer_neurons(self, hidden_layer_weights):
       weight_num = 0
        for h in range(len(self.hidden_layer.neurons)):
            for i in range(self.num_inputs):
                if not hidden_layer_weights:
                    self.hidden_layer.neurons[h].weights.append(random.random())
                else:
                    self.hidden_layer.neurons[h].weights.append(hidden_layer_weights[weight_num])
                weight_num += 1

def init_weights_from_hidden_layer_neurons_to_output_layer_neurons(self, output_layer_weights):
        weight_num = 0
 [...]
def feed_forward(self, inputs):
        hidden_layer_outputs = self.hidden_layer.feed_forward(inputs)
        return self.output_layer.feed_forward(hidden_layer_outputs)
```

Neural Network

```
def train(self, training_inputs, training_outputs):
        self.feed_forward(training_inputs)

        # 1. Output neuron deltas
        pd_errors_wrt_output_neuron_total_net_input = [0] * len(self.output_layer.neurons)
        for o in range(len(self.output_layer.neurons)):
            pd_errors_wrt_output_neuron_total_net_input[o] =
self.output_layer.neurons[o].calculate_pd_error_wrt_total_net_input(training_outputs[o])

       # 2. Hidden neuron deltas
       […]

        # 3. Update output neuron weights
        for o in range(len(self.output_layer.neurons)):
            for w_ho in range(len(self.output_layer.neurons[o].weights)):
                pd_error_wrt_weight = pd_errors_wrt_output_neuron_total_net_input[o] *
self.output_layer.neurons[o].calculate_pd_total_net_input_wrt_weight(w_ho)
self.output_layer.neurons[o].weights[w_ho] -= self.LEARNING_RATE * pd_error_wrt_weight

        # 4. Update hidden neuron weights
        […]
```

Python Exercises

- **Try the simple NN trainer for approximating a bidimensional function defined in 4 points (rather than a single point)**
  - Does it work? Which actions can we take for improving the regressor?
  - Change the training points with binary outputs. Is it working better?

In sklearn:
MLP Classifier and MLP Regressor

➔ **MLPClassifier vs. Regressor**

- ⇨ Discrete output or continous output
- ⇨ Usual methods fit/predict

➔ **Constructor specifying:**

- ⇨ number of neurons for each layer
- ⇨ squash function
- ⇨ Solver (different gradient variants)
- ⇨ Learning rate, max iterations
- ⇨ Regularization factor

Python exercise

➔ **Use a two layer NN to approximate the function y(x)=sin(2*pi*x) with x in [0,1] and y(x)=x^2 with x in [-4,4]. To this end, generate a sufficient number of data points for the training and use the backpropagation algorithm.**

- ⇨ What happens if we change the parameters of the MLPRegressor?
  - → Try relu, tanh, logistic.
- ⇨ What happens if we reduce the training points?
  - → Try 10, 100, and 1000 training points
- ⇨ What happens if we use noisy training points?
  - → Try the effect of the alpha parameter for different numbers of hidden neurons

Python Exercises

- **Implement the XOR function by using the MLPClassifier;**

- **Implement a digit classificator on the digits data set provided by sklearn by using the MLPClassifier.**
  - Plot the average error achieved for different sizes of the hidden layer.
  - What is the best MLP structure?

⇨ Remark 1: A common feature of all the above is the danger of local minimum convergence. "**Well formed**" cost functions guarantee convergence to a "good" solution, that is one that classifies correctly ALL training patterns, provided such a solution exists. The cross-entropy cost function **is** a well formed one. The Least Squares **is not**.

⇨ Remark 2: Both, the Least Squares and the cross entropy lead to output values $\hat{y}_m(i)$ that approximate optimally class a-posteriori probabilities!!!

$$\hat{y}_m(i) \cong P(\omega_m | \underline{x}(i))$$

That is, the probability of class ω_m given $\underline{x}(i)$.

This is a very interesting result. It **does not** depend on the underlying distributions. It is a characteristic of **certain** cost functions. How good or bad is the approximation, depends on the underlying model. Furthermore, it is only valid at the global minimum.

⇨ Choice of the network size.

How big a network can be. How many layers and how many neurons per layer?? There are two major directions

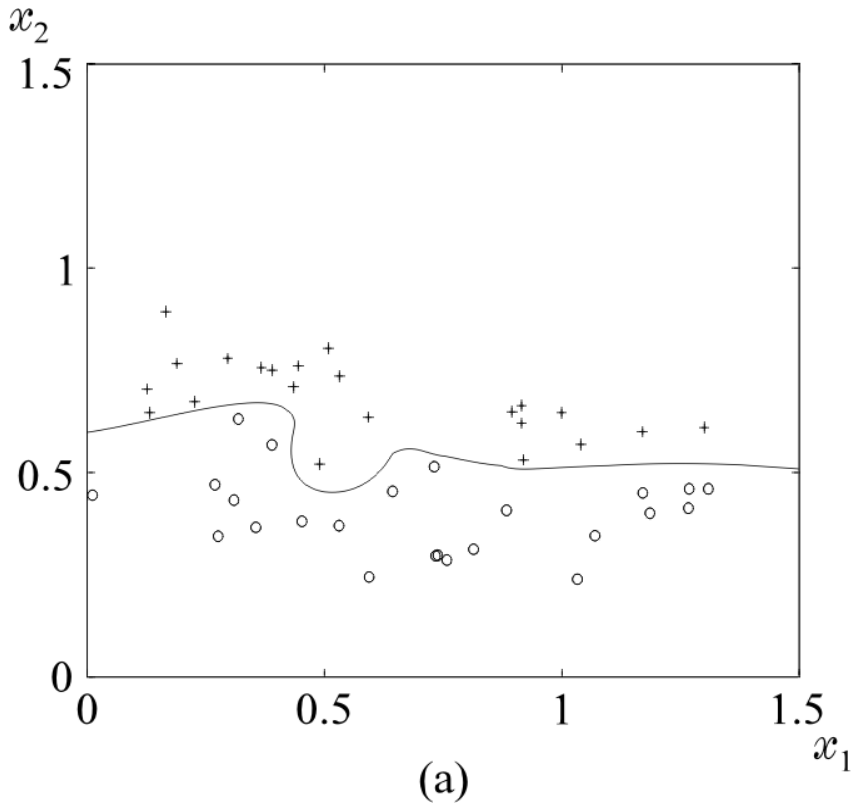
→**Pruning Techniques:** These techniques start from a large network and then weights and/or neurons are removed iteratively, according to a criterion.

⇨ Remark: Why not start with a large network and leave the algorithm to decide which weights are small?? This approach is just naïve. It overlooks that classifiers must have good **generalization** properties. A large network can result in small errors for the training set, since it can learn the particular details of the training set. On the other hand, it will not be able to perform well when presented with data unknown to it. The size of the network must be:

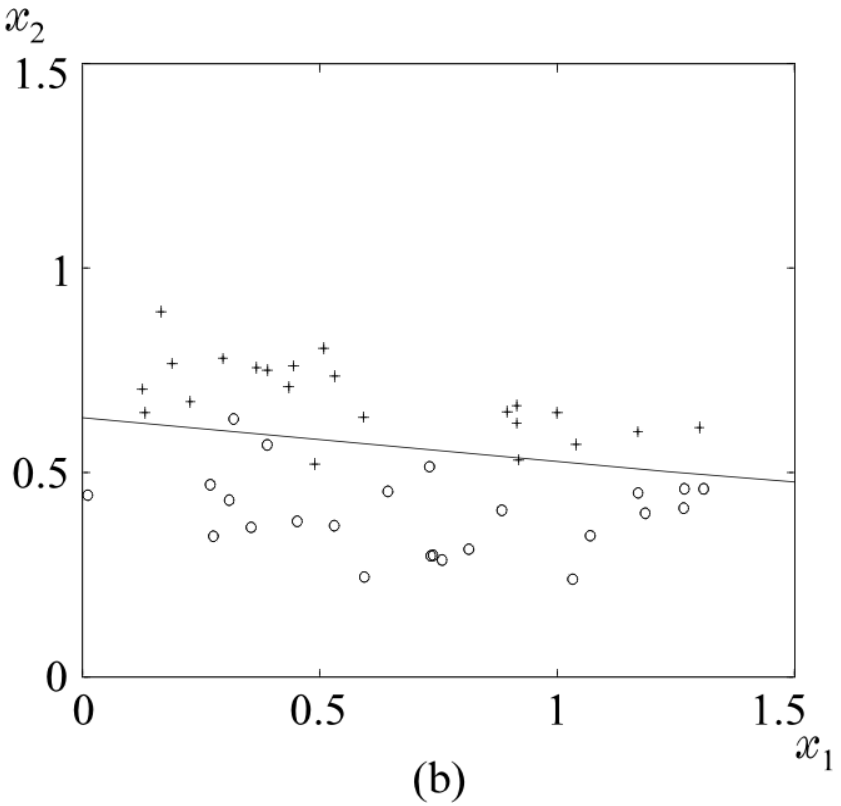
→**Large enough** to learn what makes data of the same class similar and data from different classes dissimilar

→**Small enough** not to be able to learn underlying differences between data of the same class. This leads to the so called overfitting.

Example:

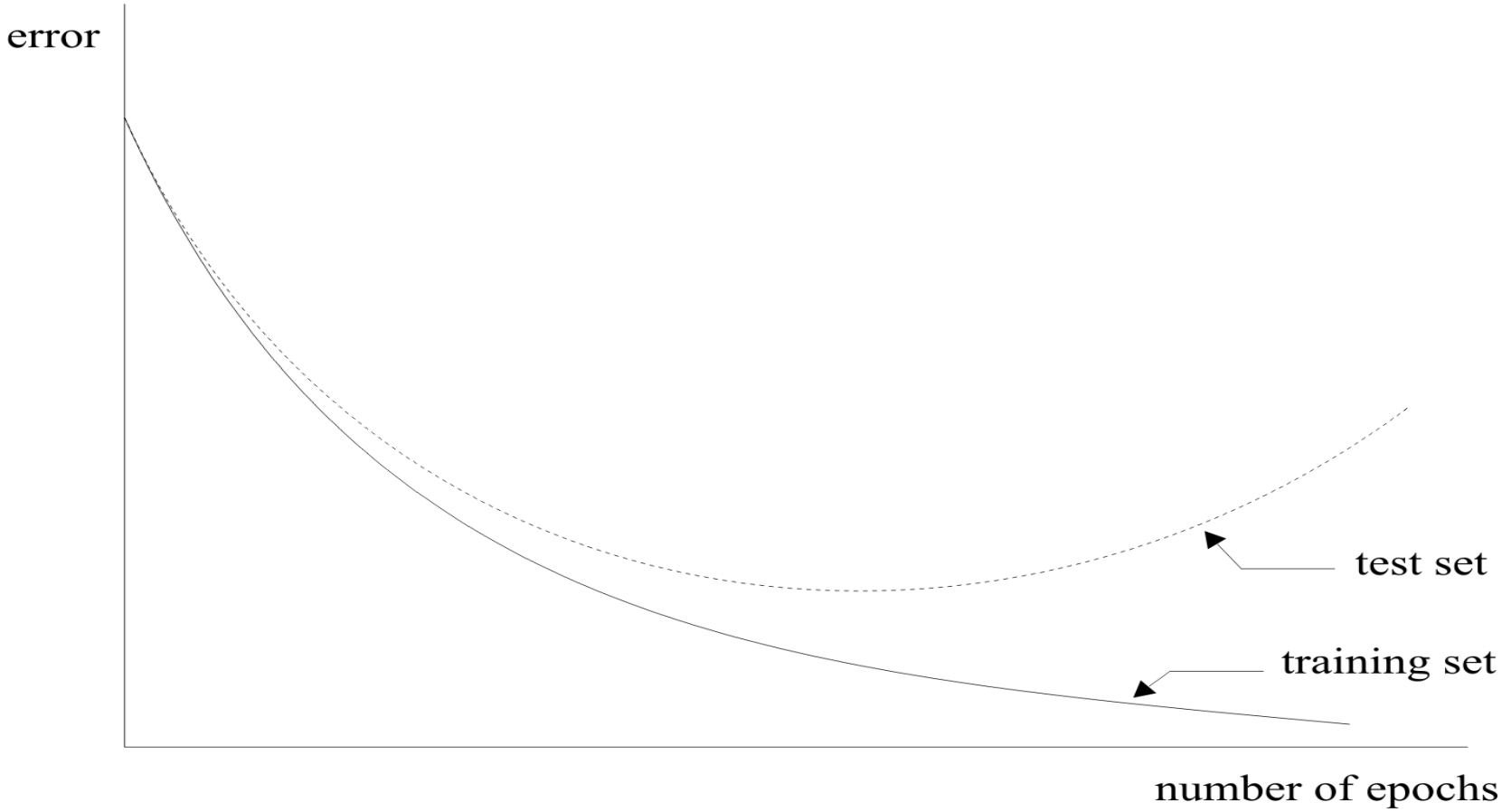


(a)



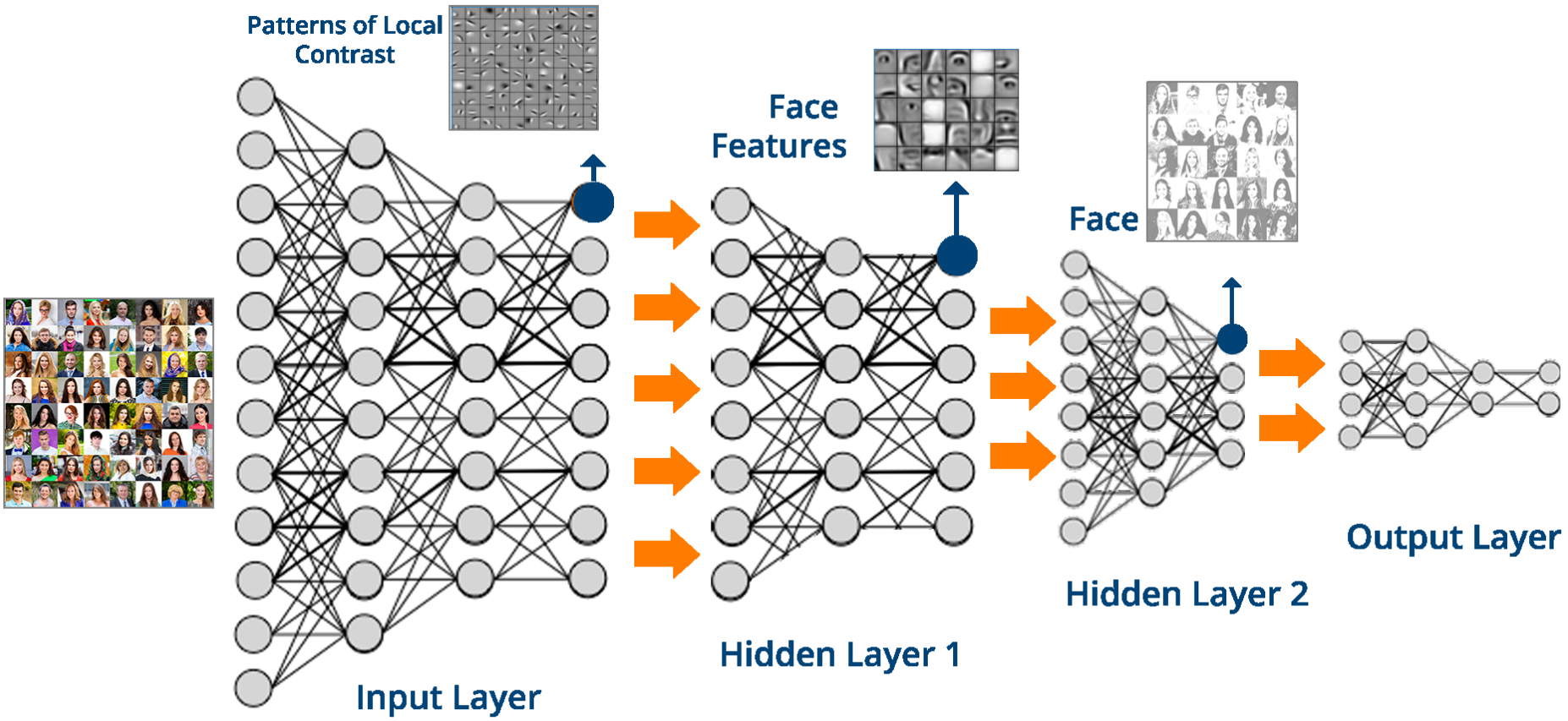
(b)

⇨ Overtraining is another side of the same coin, i.e., the network adapts to the peculiarities of the training set.



When NNs become deep?

more than one hidden layer!



With deep learning, we can make automatic the process of feature extraction, provided enough input parameters and complexity!

Generalized Linear Classifiers

⇨ Remember the XOR problem. The mapping

$$\underline{x} \rightarrow \underline{y} = \begin{bmatrix} f(g_1(\underline{x})) \\ f(g_2(\underline{x})) \end{bmatrix}$$

$f(.) \rightarrow$ The activation function transforms the nonlinear task into a linear one.

⇨ In the more general case:

→Let $\underline{x} \in R^l$ and a nonlinear classification task.

$$f_i(.), i = 1,2,...,k$$

→Are there any functions and an appropriate k, so that the mapping

$$\underline{x} \rightarrow \underline{y} = \begin{bmatrix} f_1(\underline{x}) \\ \dots \\ f_k(\underline{x}) \end{bmatrix}$$

transforms the task into a linear one, in the space? $\underline{y} \in R^k$

→If this is true, then there exists a hyperplane $\underline{w} \in R^k$ so that

$$\text{If } \; w_0 + \underline{w}^T \underline{y} > 0, \quad \underline{x} \in \omega_1$$

$$w_0 + \underline{w}^T \underline{y} < 0, \quad \underline{x} \in \omega_2$$

⇨ In such a case this is equivalent with approximating the nonlinear discriminant function $g(\underline{x})$, in terms of $f_i(\underline{x})$, i.e.,

$$g(\underline{x}) \cong w_0 + \sum_{i=1}^{k} w_i f_i(\underline{x}) \quad (><) \quad 0$$

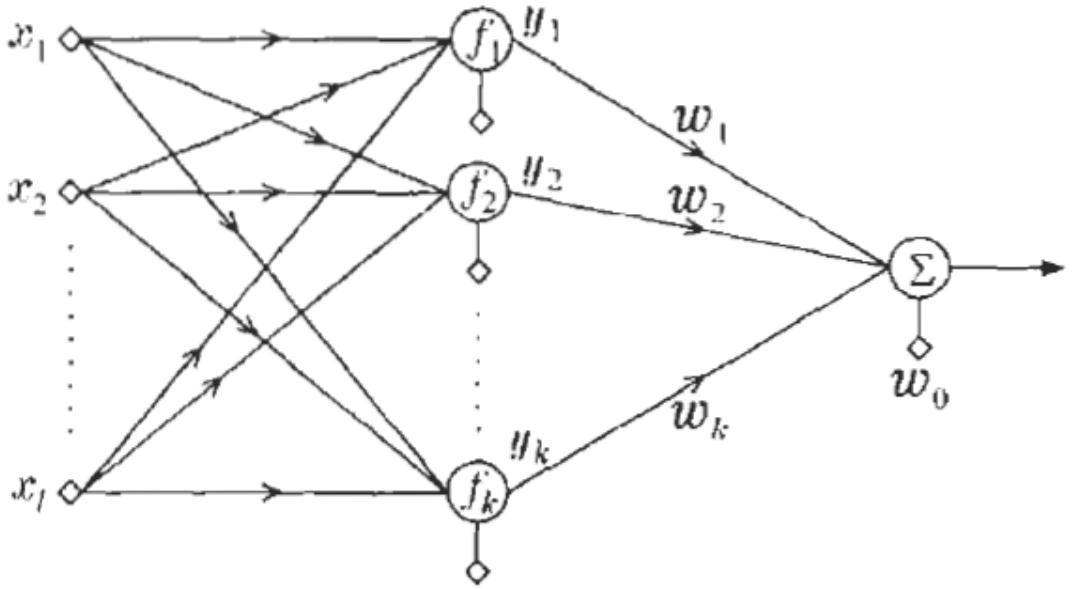
⇨ Given $f_i(\underline{x})$, the task of computing the weights is a linear one.

⇨ How sensible is this??

→From the numerical analysis point of view, this is justified if $f_i(\underline{x})$ are interpolation functions.

→From the Pattern Recognition point of view, this is justified by Cover's theorem

Generalized classifier structure



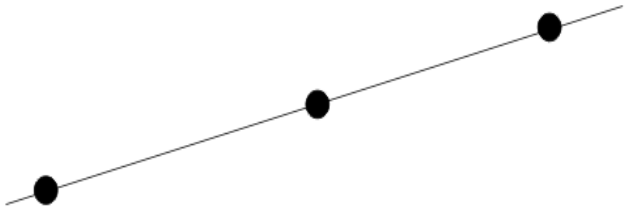
l inputs

k interpolation functions

Typical interpolation functions:
1) polinomial functions
2) radial basis functions (distance from centers)

Linear Dichotomies in l-dimensional space

⇨ Assume N points in R^l assumed to be in general position, that is: no subset of l+1 points among this N lie on a l-1 dimensional space



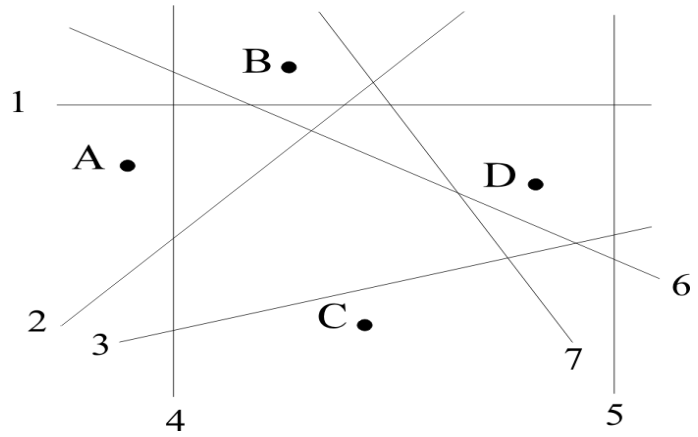
not in general position

general position

⇨ Cover's theorem states: The number of groupings that can be formed by (l-1)-dimensional hyperplanes to separate N points in two classes is lower than 2^N and equal to:

$$O(N,l) = 2\sum_{i=0}^{l}\binom{N-1}{i}, \qquad \binom{N-1}{i} = \frac{(N-1)!}{(N-1-i)!i!}$$

Example: N=4, l=2, $O(4,2)$=**14**

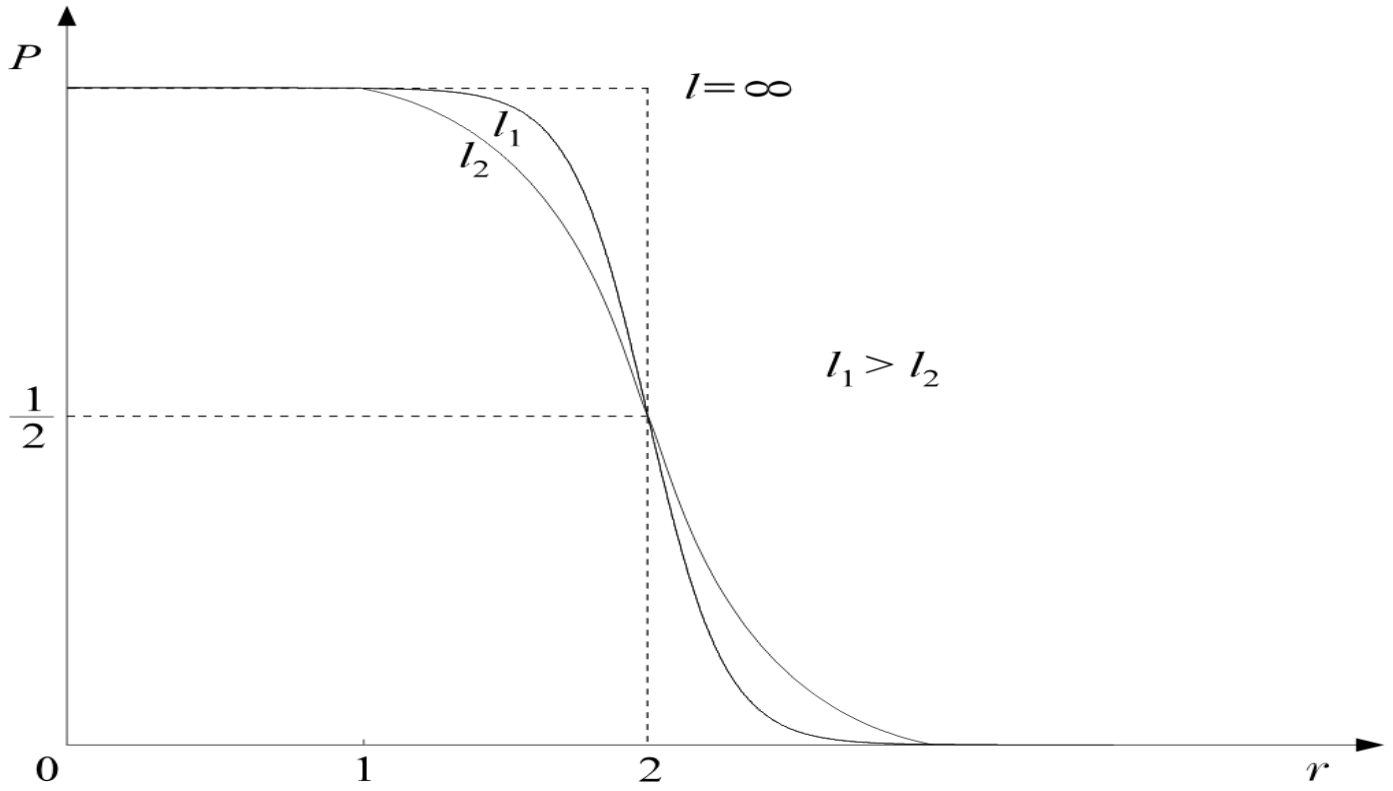


Linear-separable groups:
ABCD;
A, BCD; B, ACD; D, ABC; C, ABD
AB, CD; AC, BD
2 linear dichotomy possibilities:
(w1, w2) or (w2, w1)

Notice: The total number of possible groupings is 2^4=16, but such a number includes non-linear combinations!!

⇨ It follows that probability of grouping N points in two linearly separable classes is

$$\frac{O(N,l)}{2^N} = P_N^l$$



$$N = r(l+1)$$

Sketch of demonstration

- **Assume to have N points linearly independent in l space and to know the number of linear separable partitions O(N, l); now add a new point**
  - ⇨ Previous hyperplanes can pass through the new points and therefore lead to two partitions moving infinitesimally the point
  - ⇨ Previous hyperplanes cannot pass through the new point and therefore the point can be univocally assing to a partition
- **It follows:**
  - ⇨ O(N+1, l)=O(N, l) + O(N, l-1)
- **Iterating:**
  - ⇨ $O(N+1,l) = \binom{N}{0}O(1,l) + \binom{N}{1}O(1,l-1) + \cdots \binom{N}{N}O(1,l-N)$

Thus, the probability of having N points in linearly separable classes tends to 1, for large l, provided $N<2(l+1)$

Hence, by mapping to a higher dimensional space, we increase the probability of **linear separability,** provided the space is not too densely populated.

Decision Trees

This is a family of non-linear classifiers. They are multistage decision systems, in which classes are **sequentially rejected**, until a finally accepted class is reached. To this end:

⇨ The feature space is split into **unique** regions in a sequential manner.

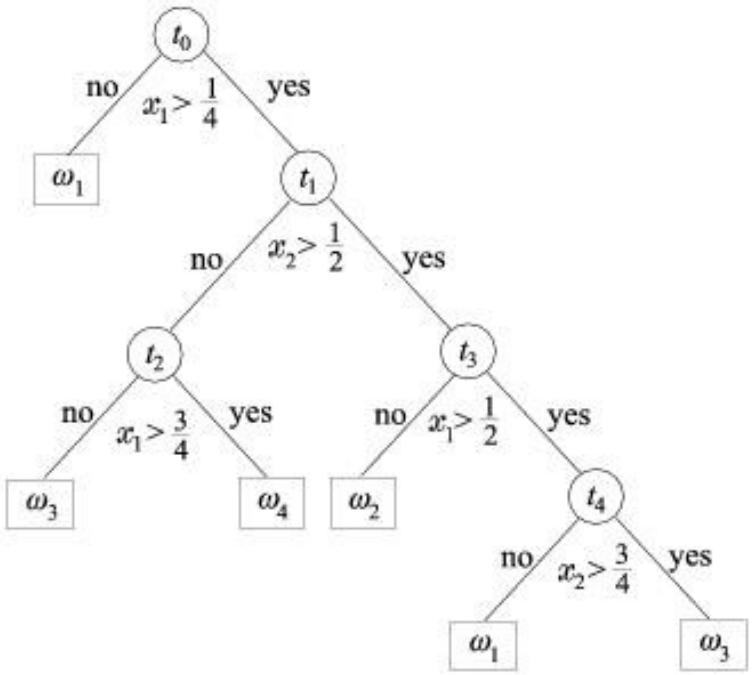
⇨ Upon the arrival of a feature vector, sequential decisions, assigning features to specific regions, are performed along a path of **nodes** of an appropriately constructed **tree**.

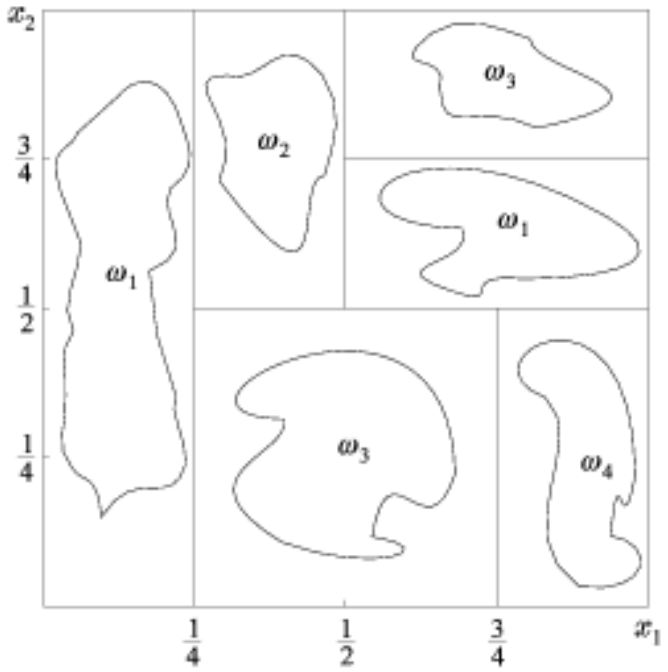
⇨ The sequence of decisions is applied to individual features, and the queries performed in each node are of the type:

$x_i \leq a$ is feature

where a is a pre-chosen (during training) threshold.

⇨ The figures below are such examples. This type of trees is known as **Ordinary Binary Classification Trees (OBCT).** The decision hyperplanes, splitting the space into regions, are parallel to the axis of the spaces. Other types of partition are also possible, yet less popular.





⇨ Design Elements that define a decision tree.

→Each node, *t,* is associated with a subset $X_t \subseteq X$, where X is the training set. At each node, X_t is split into **two** (binary splits) disjoint descendant subsets $X_{t,Y}$ and $X_{t,N}$, where

$$X_{t,Y} \cap X_{t,N} = \emptyset$$
$$X_{t,Y} \cup X_{t,N} = X_t$$

$X_{t,Y}$ is the subset of X_t for which the answer to the query at node t is YES. $X_{t,N}$ is the subset corresponding to NO. The split is decided according to an adopted question (query).

→A splitting criterion must be adopted for the best split of X_t into $X_{t,Y}$ and $X_{t,N}$.

→A stop-splitting criterion must be adopted that controls the growth of the tree and a node is declared as terminal (leaf).

→A rule is required that assigns each (terminal) leaf to a class.

⇨ **Set of Questions:** In OBCT trees the set of questions is of the type

$$\text{is} \quad x_i \leq a \quad ?$$

The choice of the specific x_i and the value of the threshold a, for each node t, are the results of searching, during training, among the features and a set of possible threshold values. The final combination is the one that results to the best value of a criterion.

⇨ **Splitting Criterion:** The main idea behind splitting at each node is the resulting descendant subsets $X_{t,Y}$ and $X_{t,N}$ to be more class homogeneous compared to X_t. What does it mean class homogeneous?

→The ratio of points belonging to the majority class increases

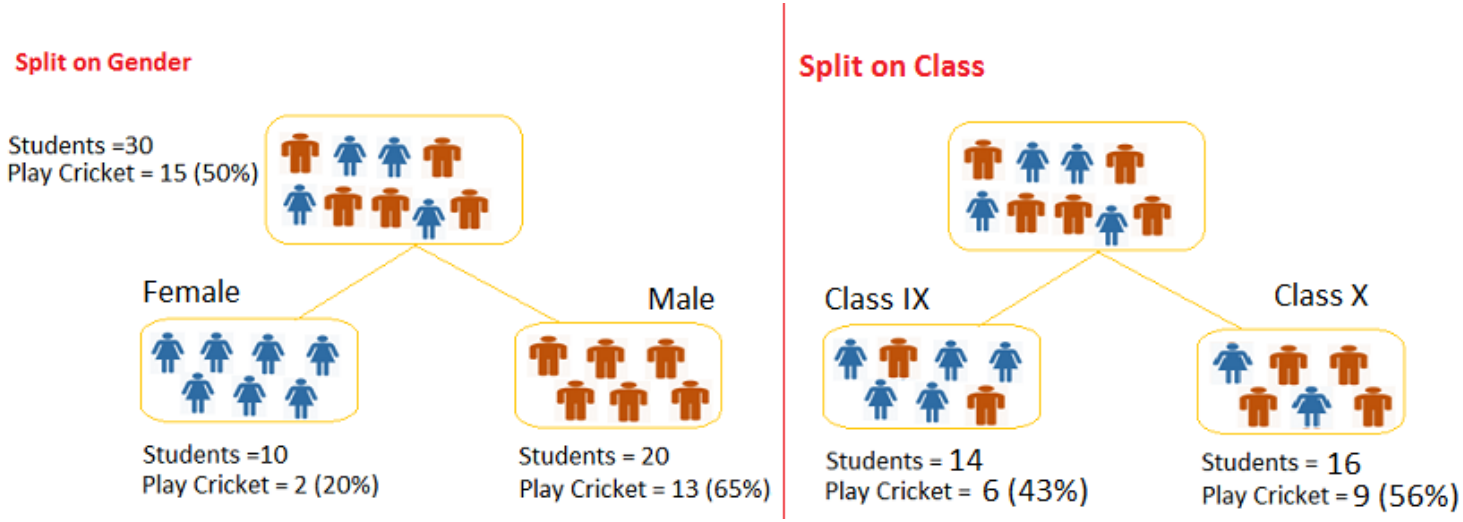
C0: 5
C1: 5

Non-homogeneous,
High degree of impurity

C0: 9
C1: 1

Homogeneous,
Low degree of impurity

→How to measure impurity and splitting classes?



⇨ **Measure impurity:** Being N_t^i the number of points among the total number of X_t belonging to a given class i, a classical metric of homogeneity is the so called impurity of a class, i.e. the entropy:

$$I(t) = -\sum_{i=1}^{M} P(\omega_i \mid t) \log_2 P(\omega_t \mid t)$$

where

$$P(\omega_i \mid t) \approx \frac{N_t^i}{N_t}$$

The decrease in node impurity is defined as:

$$\Delta I(t) = I(t) - \frac{N_{t,\mathrm{Y}}}{N_t} I(t_{\mathrm{Y}}) - \frac{N_{t,N}}{N_t} I(t_N)$$

Impurity computation example

C1	**0**
C2	**6**

P(C1) = 0/6 = 0 P(C2) = 6/6 = 1

Entropy = – 0 log 0 – 1 log 1 = – 0 – 0 = 0

C1	**1**
C2	**5**

P(C1) = 1/6 P(C2) = 5/6

Entropy = – (1/6) $\log_2$ (1/6) – (5/6) $\log_2$ (1/6) = 0.65

C1	**2**
C2	**4**

P(C1) = 2/6 P(C2) = 4/6

Entropy = – (2/6) $\log_2$ (2/6) – (4/6) $\log_2$ (4/6) = 0.92

maximum entropy equal to 1 when set X_t include the same number of points belonging to each class (e.g. C1: 5 points; C2: 5 points)

→The goal is to choose the parameters in each node (feature and threshold) that result in a split with the highest decrease in impurity.

→Why highest decrease? Observe that the highest value of $I(t)$ is achieved if all classes are equiprobable, i.e., X_t is the least homogenous.

⇨ Stop - splitting rule. Adopt a threshold T and stop splitting a node (i.e., assign it as a leaf), if the impurity decrease is less than T. That is, node t is "pure enough".

⇨ Class Assignment Rule: Assign a leaf to a class ω_j , where:

$$j = \arg\max_i P(\omega_i \mid t)$$

⇨ Summary of an OBCT algorithmic scheme:

- Begin with the root node, i.e., $X_t = X$
- For each new node t
  - For every feature $x_k, k = 1, 2, \ldots, l$
    - For every value $\alpha_{kn}, n = 1, 2, \ldots, N_{tk}$
      - Generate X_{tY} and X_{tN} according to the answer in the question: is $x_k(i) \leq \alpha_{kn},\ i = 1, 2, \ldots, N_t$
      - Compute the impurity decrease
    - End
    - Choose α_{kn_0} leading to the maximum decrease w.r. to x_k
  - End
  - Choose x_{k_0} and associated $\alpha_{k_0 n_0}$ leading to the overall maximum decrease of impurity
  - If stop-splitting rule is met declare node t as a leaf and designate it with a class label
  - If not, generate two descendant nodes t_Y and t_N with associated subsets X_{tY} and X_{tN}, depending on the answer to the question: is $x_{k_0} \leq \alpha_{k_0 n_0}$
- End

Combining Classifiers

The basic philosophy behind the combination of different classifiers lies in the fact that even the "best" classifier fails in some patterns that other classifiers may classify correctly. Combining classifiers aims at exploiting this complementary information residing in the various classifiers**.**

Thus, one designs different optimal classifiers and then combines the results with a specific rule.

⇨ Assume that each of the, say, L designed classifiers provides at its output the posterior probabilities:

$$P(\omega_i \mid \underline{x}), i = 1,\ 2,\ ...,\ M$$

→**Product Rule:** Assign $\underline{x}$ to the class ω_i:

$$i = \arg\max_k \prod_{j=1}^{L} P_j(\omega_k \mid \underline{x})$$

Joint event that all classifier produces the same output

where $P_j(\omega_k \mid \underline{x})$ is the respective posterior probability of the j^{th} classifier.

→**Sum Rule:** Assign $\underline{x}$ to the class ω_i :

$$i = \arg\max_k \sum_{j=1}^{L} P_j(\omega_k \mid \underline{x})$$

Union of events in which classifiers produce a given output

→**Majority Voting Rule:** Assign $\underline{x}$ to the class for which there is a consensus or when at least of the classifiers agree on the class label of where: λ_c

$$\lambda_c = \begin{cases} \frac{L}{2} + 1, \text{L even} \\ \frac{L+1}{2}, \text{L odd} \end{cases}$$

otherwise the decision is rejection, that is no decision is taken.

Thus, correct decision is made if the majority of the classifiers agree on the correct label, and wrong decision if the majority agrees in the wrong label.

⇨ Dependent or not Dependent classifiers?

→Although there are not general theoretical results, experimental evidence has shown that the more independent in their decision the classifiers are, the higher the expectation should be for obtaining improved results after combination. However, there is no guarantee that combining classifiers results in better performance compared to the "best" one among the classifiers.

⇨ Towards Independence: A number of Scenarios.

→Train the individual classifiers using different training data points. To this end, choose among a number of possibilities:

→Bootstrapping: This is a popular technique to combine unstable classifiers such as decision trees (Bagging belongs to this category of combination).

» **Stacking:** Train the combiner with data points that have been excluded from the set used to train the individual classifiers.

» **Use different subspaces to train individual classifiers**: According to the method, each individual classifier operates in a different feature subspace. That is, use different features for each classifier.

⇨ Remarks:

→The majority voting and the summation schemes rank among the most popular combination schemes.

→Training individual classifiers in different subspaces seems to lead to substantially better improvements compared to classifiers operating in the same subspace.

→Besides the above three rules, other alternatives are also possible, such as to use the median value of the outputs of individual classifiers.

Python exercise

- Give a look to tree_2.py for retrieving the tree structure found by scikit-learn for the Iris Data Set.
- Now compare the structure found by using gini index and entropy as impurity measures. Which configuration performs the best?
- Now make a prediction on a test case; what is the different between predict and predict_proba?

Python exercise

➔**Decision trees tend to be sensitive to small changes in the data set. Remove the *widest* petal from the training set (length of 4.8cm and width of 1.8cm) and train a classifier working on the first two features only.**

⇨Plot the decision boundary using the function implemented in tree.py

Python exercise

- **Open the fruit data set and compare a tree-based classifier with other approaches**
  - Give a look to the data for evaluating the most suitable approach.
  - Find by yourself the first rule of the decision tree.