



AARHUS  
UNIVERSITET

# Optimization and Data Analytics

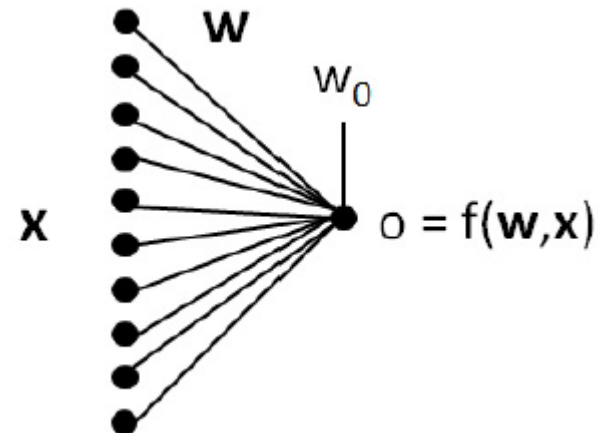
Alexandros Iosifidis

@

Aarhus University, Department of Engineering

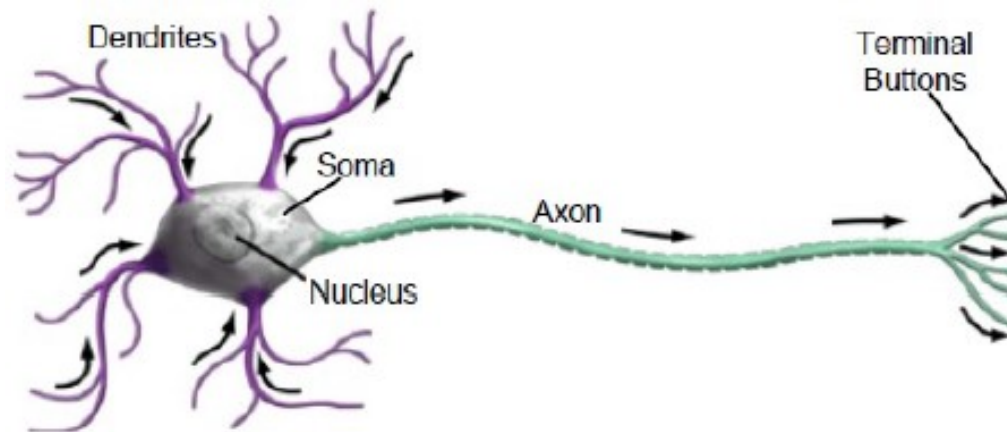
# Two-layer neural network

$$f(\mathbf{w}, \mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$$



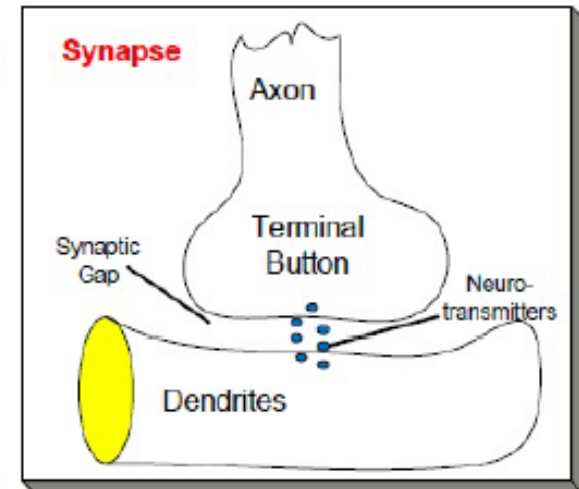
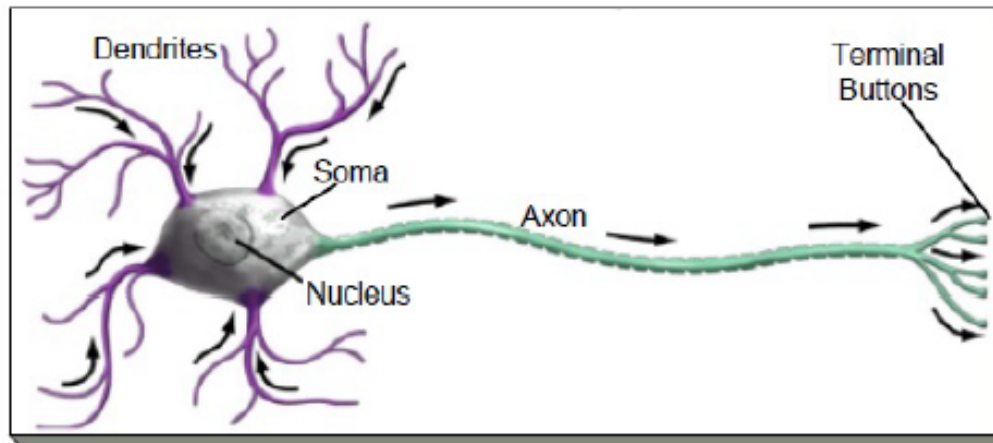
# The neuron

## Biological neuron



# The neuron

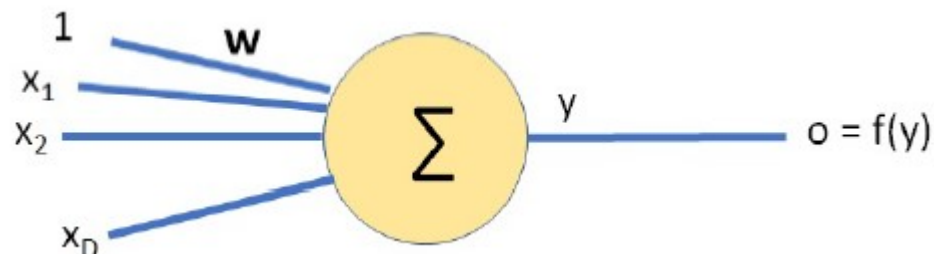
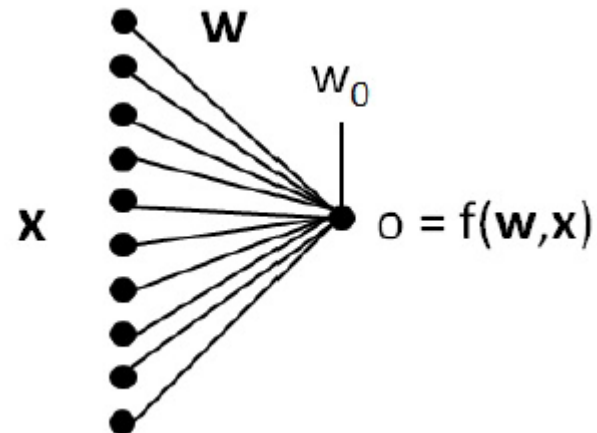
## Biological neuron



# The neuron

Lets take a look again to the artificial neuron

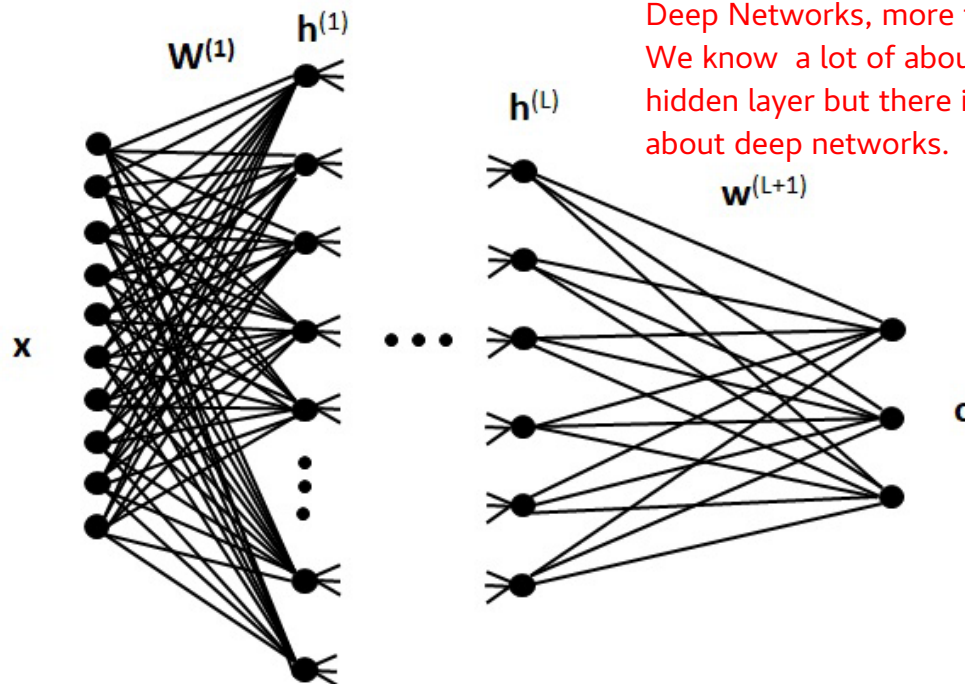
$$f(\mathbf{w}, \mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0$$



# Multilayer neural network

We can 'stack' many neurons on top of the other in order to create a 'layer'.

We can stack one layer on top of the other in order to create a 'deeper neural network'



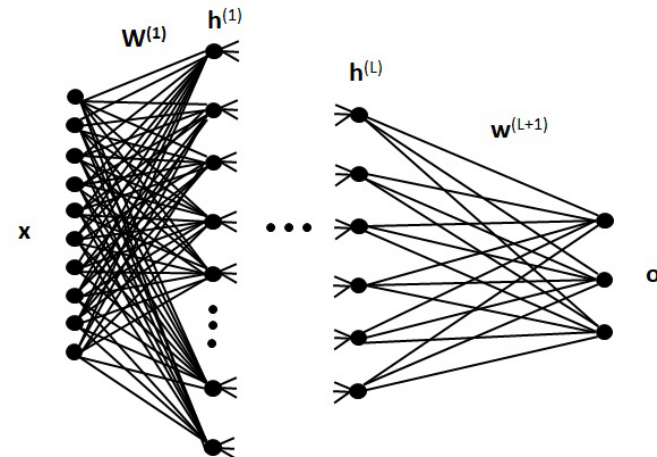
Deep Networks, more than 2 hidden layers.  
We know a lot of about MLP with one single hidden layer but there is almost no research about deep networks.

# Multilayer neural network

We can 'stack' many neurons on top of the other in order to create a 'layer'.

We can stack one layer on top of the other in order to create a 'deeper neural network'.

Because the actual operation of the network transfers the information from the input to the output, these networks are usually called feedforward neural networks.



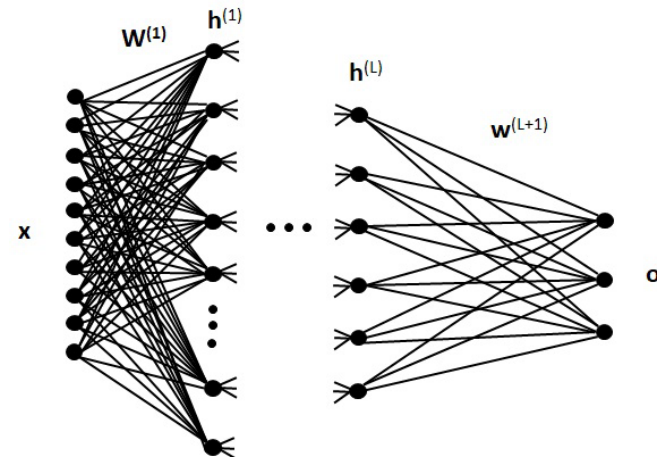
# Multilayer neural network

We can 'stack' many neurons on top of the other in order to create a 'layer'.

We can stack one layer on top of the other in order to create a 'deeper neural network'.

Because the actual operation of the network transfers the information from the input to the output, these networks are usually called feedforward neural networks.

**Which are the parameters of the network?**

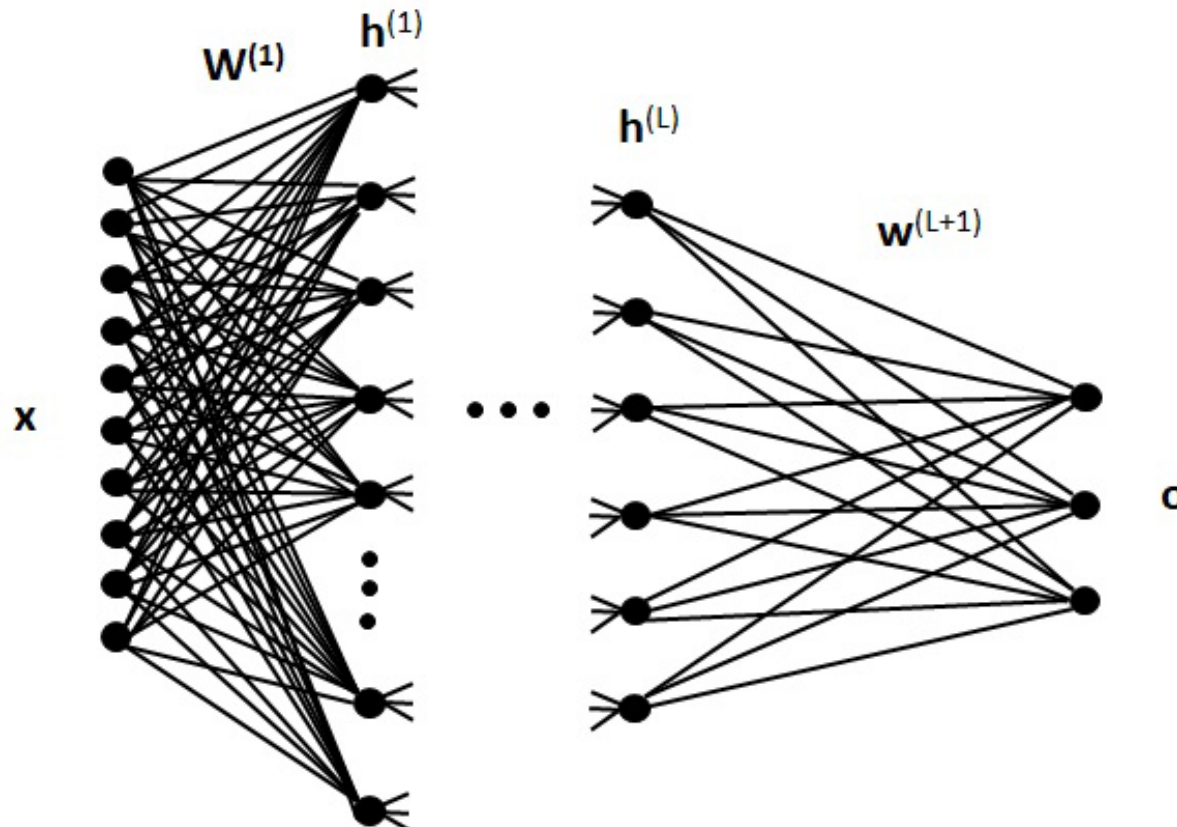




# Multilayer neural network

**Which are the parameters of the network?**

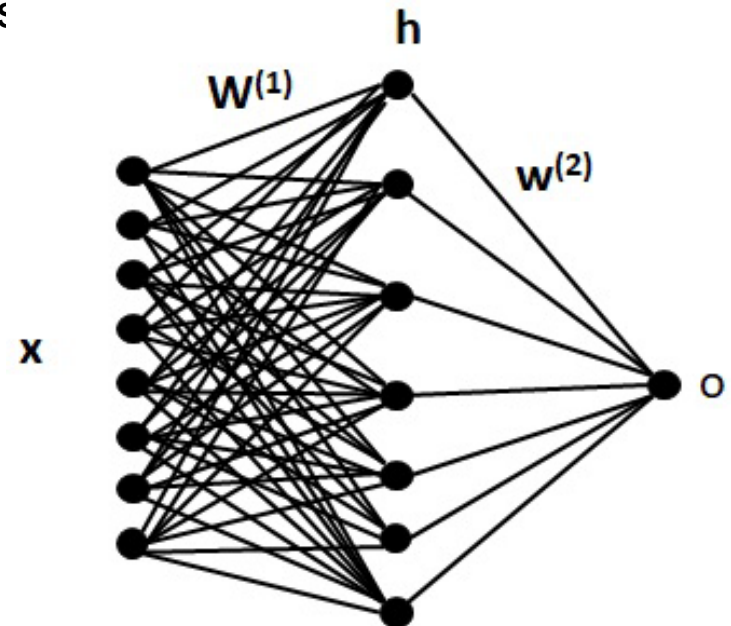
The weight matrices  $W$ s and the topology of the network:  
- number of layers,  
- what activation functions to use



# Three-layer neural network

Let us focus on the three-layer topology.

Since this neural networks having this topology (usually called architecture) have one input, one output and one hidden layer, they are usually called Single-hidden Layer Feedforward Neural networks

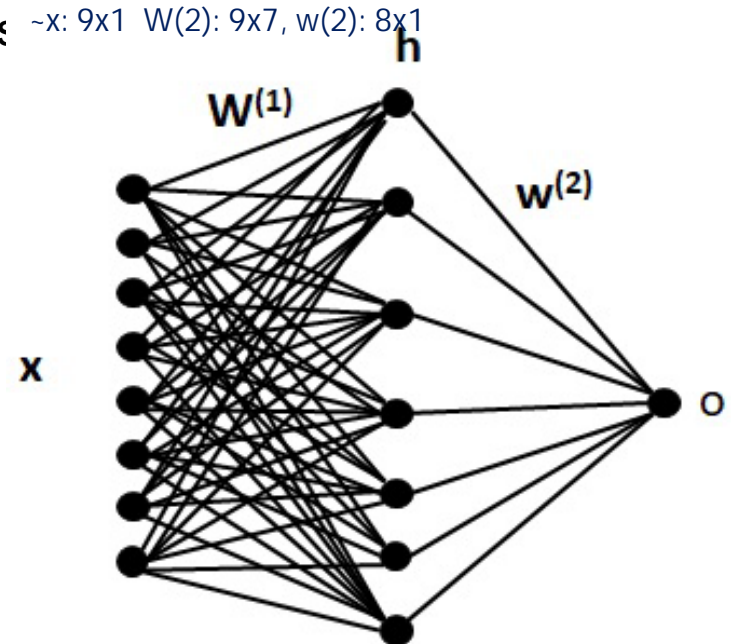


# Three-layer neural network

Let us focus on the three-layer topology.

Since this neural networks having this topology (usually called architecture) have one input, one output and one hidden layer, they are usually called Single-hidden Layer Feedforward Neural networks

**Which are the parameters of the network?**



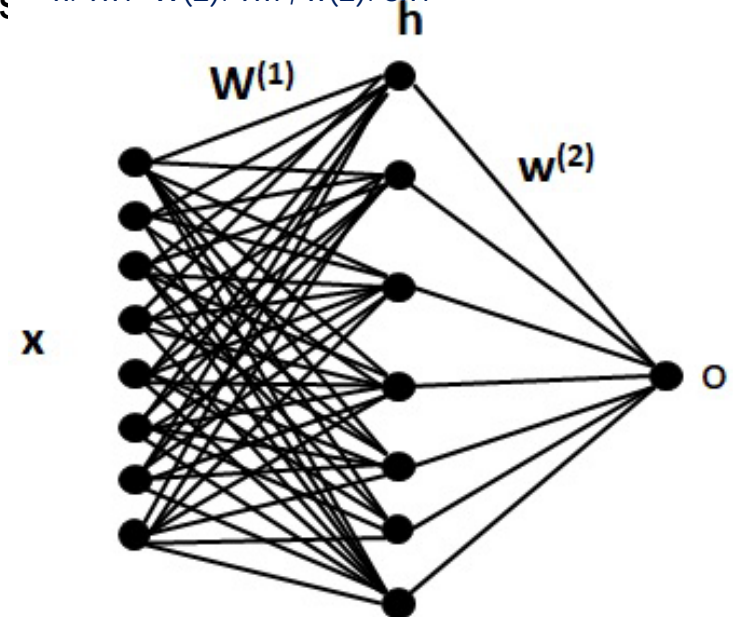
# Three-layer neural network

Let us focus on the three-layer topology.

Since this neural networks having this topology (usually called architecture) have one input, one output and one hidden layer, they are usually called Single-hidden Layer Feedforward Neural networks

$x: 9 \times 1$   $W(2): 9 \times 7$ ,  $w(2): 8 \times 1$

**Can you describe the operations involved in the feedforward phase?**



If we have a huge network with many hidden layers  
but if the activation function is linear then  
it is similar to MLP with a single hidden layer  
because all the  $W$ 's are the same as a single  $W$

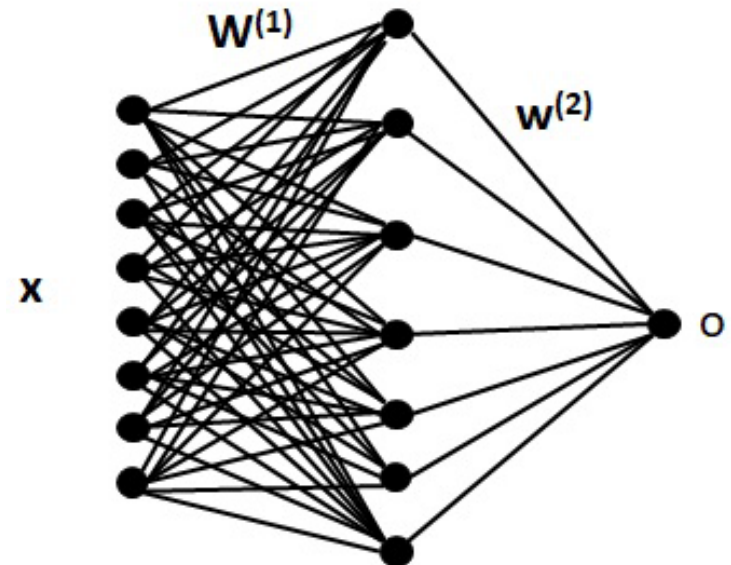
# Three-layer neural network

Let us focus on the three-layer topology.

Each of the hidden neurons has an output equal to

$$h_k = f(\mathbf{W}_k^{(1)T} \mathbf{x}), \quad k = 1, \dots, K = 7$$

$K$  is the number of neurons in the hidden layer  $\mathbf{h}$



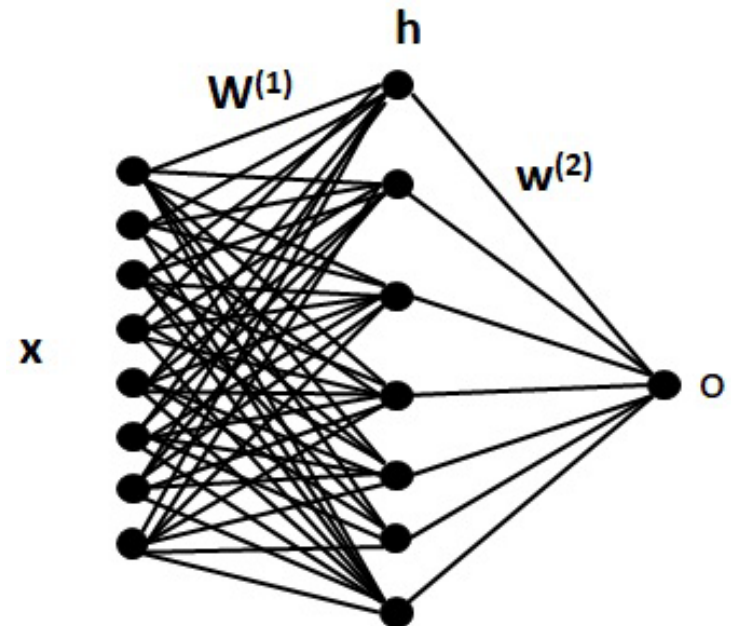
# Three-layer neural network

Let us focus on the three-layer topology.

Each of the hidden neurons has an output equal to

$$h_k = f\left(\mathbf{W}_k^{(1)T} \mathbf{x}\right), \quad k = 1, \dots, K = 7$$

All  $h_k$  outputs are used to form the vector  $\mathbf{h}$



# Three-layer neural network

Let us focus on the three-layer topology.

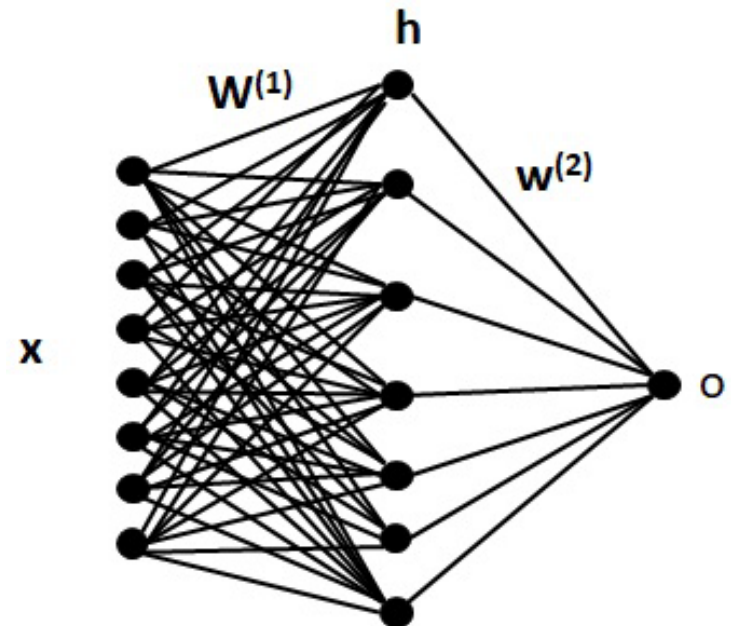
Each of the hidden neurons has an output equal to

$$h_k = f(\mathbf{W}_k^{(1)T} \mathbf{x}), k = 1, \dots, K = 7$$

All  $h_k$  outputs are used to form the vector  $\mathbf{h}$

The output neuron has an output equal to

$$\begin{aligned} o &= f(\mathbf{w}^{(2)T} \mathbf{h}) = f\left(\sum_{k=1}^K w_k^{(2)} h_k\right) \\ &= f\left(\sum_{k=1}^K w_k^{(2)} f(\mathbf{W}_k^{(1)T} \mathbf{x})\right). \end{aligned}$$



# Four-layer neural network

**Can you describe the calculations involved in a four-layer feedforward network?**



# Three-layer neural network

One case of particular interest is when the activation function of the output layer is linear, i.e. when  $f(\mathbf{o}) = \mathbf{o}$ . Then, we have:

$$o = \mathbf{w}^{(2)T} \mathbf{h} = \sum_{k=1}^K w_k^{(2)} h_k = \sum_{k=1}^K w_k^{(2)} f(\mathbf{W}_k^{(1)T} \mathbf{x})$$

Why is this case interesting? We can prove that if the number of neurons is the same as the number of samples, then we can approximate everything. This is a proof that shows that all neural networks are universal approximators.

# Three-layer neural network

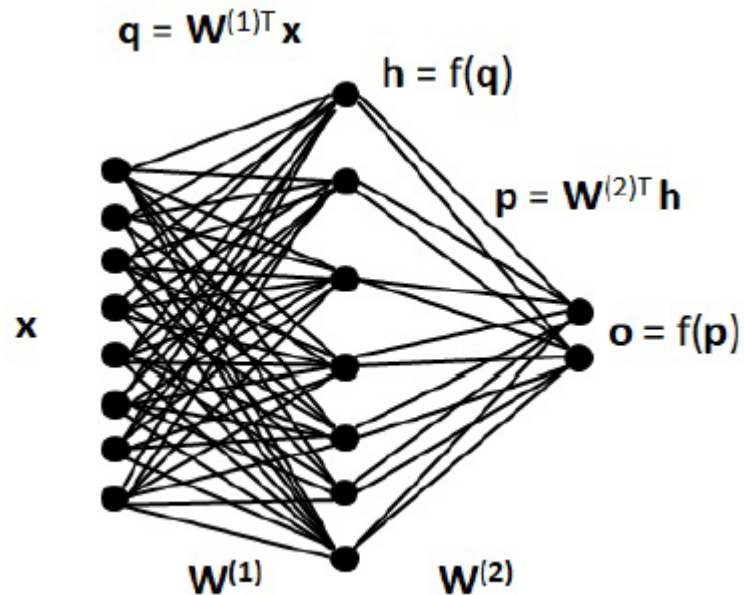
One case of particular interest is when the activation function of the output layer is linear, i.e. when  $f(\mathbf{o}) = \mathbf{o}$ . Then, we have:

$$o = \mathbf{w}^{(2)T} \mathbf{h} = \sum_{k=1}^K w_k^{(2)} h_k = \sum_{k=1}^K w_k^{(2)} f(\mathbf{W}_k^{(1)T} \mathbf{x})$$

How can we create a three-layer network solving a multi-class classification problem?

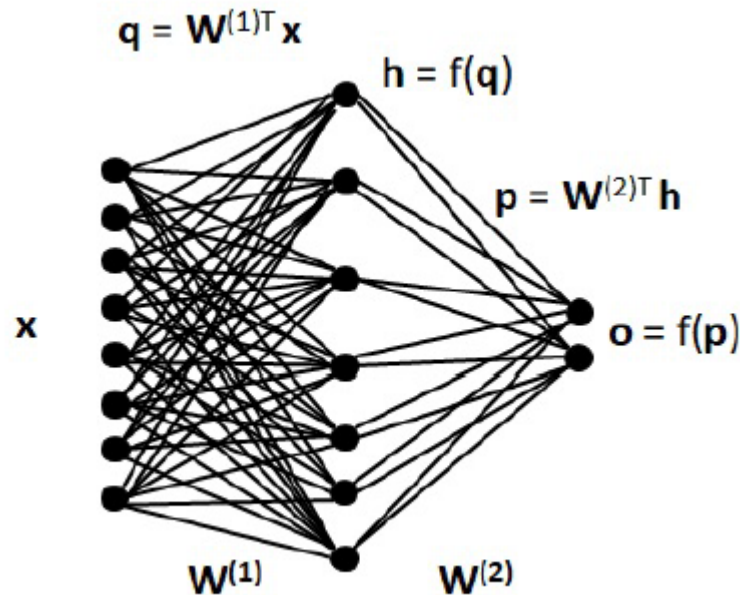
# Three-layer neural network

How can we create a three-layer network solving a multi-class classification problem?



# Three-layer neural network

How can we create a three-layer network solving a multi-class classification problem?

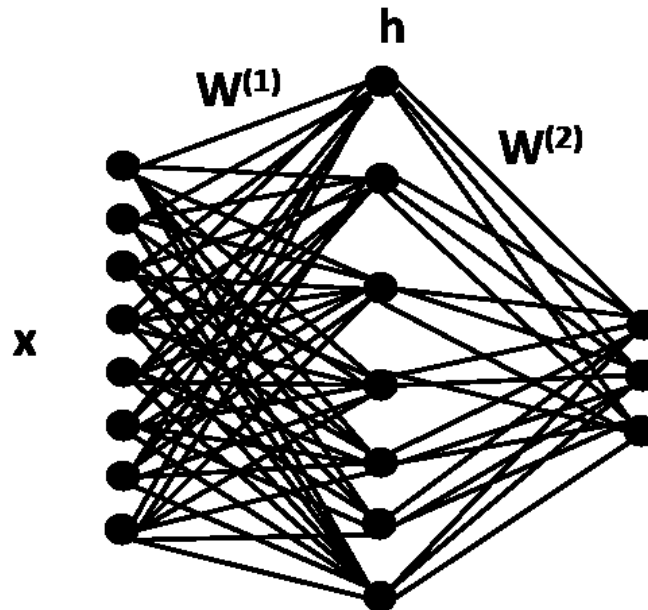


Is the above network different than a network using one output neuron?

No, it is the same

# Three-layer neural network

In order to solve a K-class classification problem, the output layer is formed by K neurons



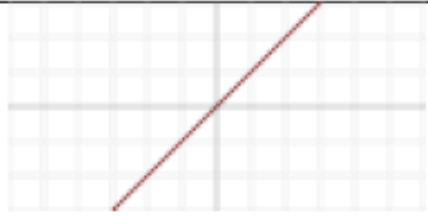
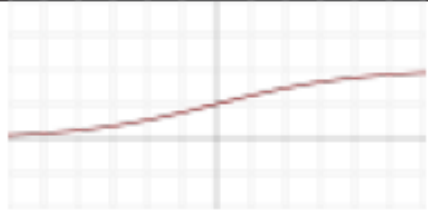
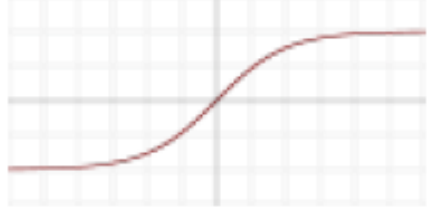
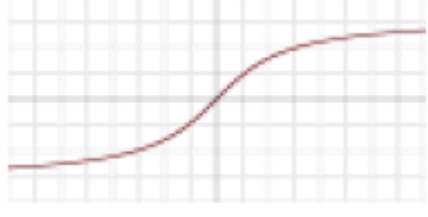
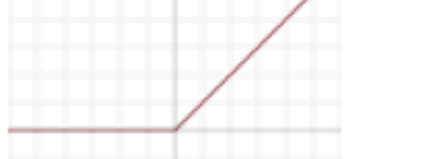
# Activation functions

Any function that is differentiable can be used as the activation function.  
This is because we need to differentiate the activation function when computing the update rule for the gradient descent (during backpropagation).

Name	Equation	Derivative
Identity	$f(x) = x$	$f'(x) = 1$
Logistic	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
HyperbTan	$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan	$f(x) = \tanh^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
ReLU	$f(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ x, & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} 0, & \text{if } x \leq 0 \\ 1, & \text{if } x > 0 \end{cases}$
Softmax	$f_i(\mathbf{x}) = \frac{e^{x_k}}{\sum_{l=1}^K e^{x_l}}, \quad k = 1, \dots, K$	$\frac{\partial f_i(\mathbf{x})}{\partial x_j} = f_i(\mathbf{x})(\delta_{ij} - f_j(\mathbf{x}))$

The softmax activation function is commonly used for the neurons of the last layer of a neural network,

# Activation functions

Name	Plot
Identity	
Logistic	
ArcTan	
ArcTan	
ReLU	

LeakyReLU: allows to leak some of the information for negative values.

# Training multi-layer neural networks

After defining the parameters of a feedforward neural network (including the architecture and the activation functions), we need to determine good parameters for a specific classification task. **We have defined the topology of the network.**



# Training multi-layer neural networks

After defining the parameters of a feedforward neural network (including the architecture and the activation functions), we need to determine good parameters for a specific classification task.

We usually do not make assumptions regarding the distributions of the classes involved in the classification problem. Instead, we define the goodness of the network (and its parameters) based on the performance of the network on a set of data.

The maths behind the Neural network make no assumption of the data

# Training multi-layer neural networks

After defining the parameters of a feedforward neural network (including the architecture and the activation functions), we need to determine good parameters for a specific classification task.

We usually do not make assumptions regarding the distributions of the classes involved in the classification problem. Instead, we define the goodness of the network (and its parameters) based on the performance of the network on a set of data.

After initializing the network's parameters, we apply an iterative optimization process, which refines the network's parameters based on its output (usually called response) to a given input vector.

# Training multi-layer neural networks

After defining the parameters of a feedforward neural network (including the architecture and the activation functions), we need to determine good parameters for a specific classification task.

We usually do not make assumptions regarding the distributions of the classes involved in the classification problem. Instead, we define the goodness of the network (and its parameters) based on the performance of the network on a set of data.

After initializing the network's parameters, we apply an iterative optimization process, which refines the network's parameters based on its output (usually called response) to a given input vector.

This is done by calculating the error for the specific vector and propagating (the changes required to reduce) the error from the output layer to the input layer.

# The (error) Backpropagation algorithm

Let us consider as an error function the following criterion

$$\mathcal{J}(\mathbf{w}) = \frac{1}{2} \sum_{k=1}^K (t_k - o_k)^2 = \frac{1}{2} \|\mathbf{t} - \mathbf{o}\|_2^2$$

When the training error is measured using the MSE criterion

where  $\mathbf{t} \in \mathbb{R}^K$  is the target vector for a K-class classification problem.

# The (error) Backpropagation algorithm

Let us consider as an error function the following criterion

$$\mathcal{J}(\mathbf{w}) = \frac{1}{2} \sum_{k=1}^K (t_k - o_k)^2 = \frac{1}{2} \|\mathbf{t} - \mathbf{o}\|_2^2$$

We do the square because we want to penalise much when the error is really high. Our reaction of the large error is to run very fast towards downhill and when the error is low then we want to go slower

where  $\mathbf{t} \in \mathbb{R}^K$  is the target vector for a K-class classification problem.

We usually set the values of  $\mathbf{t}$  equal to:

- $t_k = 1$ , if  $\mathbf{x}$  belongs to class  $c_k$
- $t_k = 0$ , if  $\mathbf{x}$  belongs to class  $c_l \neq c_k$

Can we use target values equal to  $\{-1, 1\}$ ?

# The (error) Backpropagation algorithm

Let us consider as an error function the following criterion

$$\mathcal{J}(\mathbf{w}) = \frac{1}{2} \sum_{k=1}^K (t_k - o_k)^2 = \frac{1}{2} \|\mathbf{t} - \mathbf{o}\|_2^2$$

where  $\mathbf{t} \in \mathbb{R}^K$  is the target vector for a K-class classification problem.

We usually set the values of  $\mathbf{t}$  equal to:

- $t_k = 1$ , if  $x$  belongs to class  $c_k$
- $t_k = 0$ , if  $x$  belongs to class  $c_l \neq c_k$

Can we use target values equal to  $\{-1, 1\}$ ?

In principle, the answer is yes.

It depends on the activation function of the output layer!

So for ReLU and Softmax  
this would not work because they  
will never generate negative values.

# The (error) Backpropagation algorithm

Let us consider as an error function the following criterion

$$\mathcal{J}(\mathbf{w}) = \frac{1}{2} \sum_{k=1}^K (t_k - o_k)^2 = \frac{1}{2} \|\mathbf{t} - \mathbf{o}\|_2^2$$

where  $\mathbf{t} \in \mathbb{R}^K$  is the target vector for a K-class classification problem.

The backpropagation update rule is based on gradient descent

$$\Delta \mathbf{w} = -\eta \frac{\partial \mathcal{J}}{\partial \mathbf{w}}$$

Is there a closed form solution?

or in an element-wise form

$$\Delta w_{ij} = -\eta \frac{\partial \mathcal{J}}{\partial w_{ij}}$$

# The (error) Backpropagation algorithm

The backpropagation update rule is based on gradient descent

$$\Delta \mathbf{w} = -\eta \frac{\partial \mathcal{J}}{\partial \mathbf{w}}$$

or in an element-wise form

$$\Delta w_{ij} = -\eta \frac{\partial \mathcal{J}}{\partial w_{ij}}$$

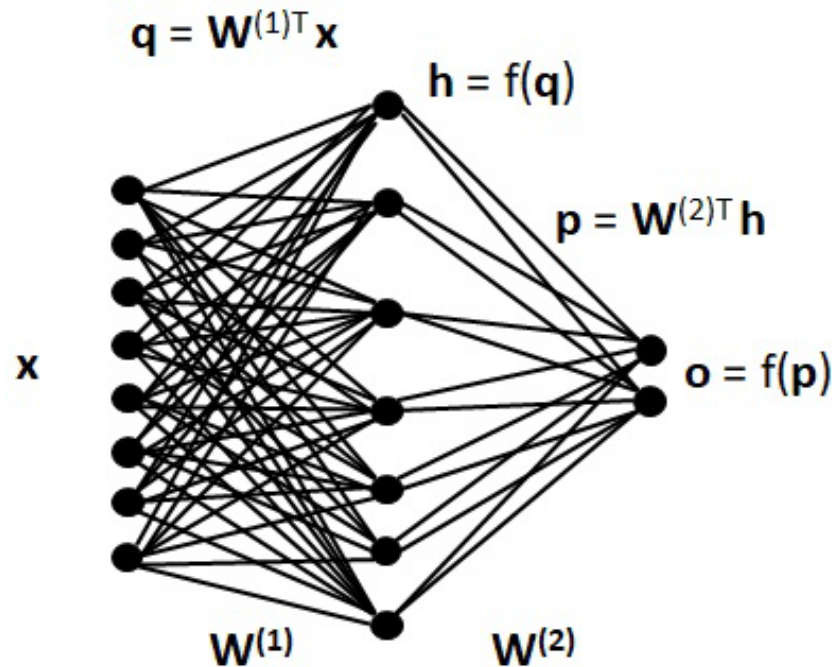
After calculating the gradient, the weights are updated as follows

$$\mathbf{w}(t+1) = \mathbf{w}(t) + \Delta \mathbf{w}.$$



# The (error) Backpropagation algorithm

Let's take as an example the following three-layer neural network



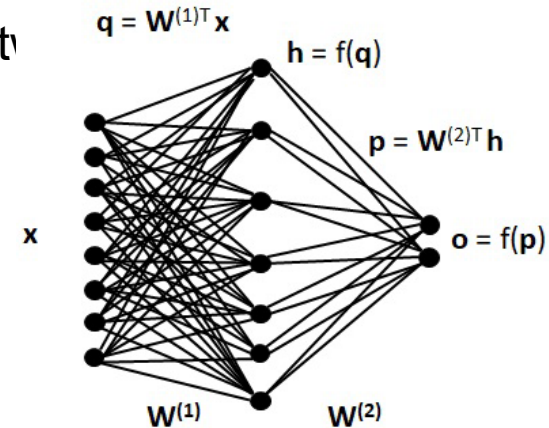
# The (error) Backpropagation algorithm

Let's take as an example the following three-layer neural net

First we focus on the weights  $\mathbf{W}^{(2)}$ .

For the weights  $\mathbf{W}_{jk}^{(2)}$  connecting the hidden neuron  $j$  with the output neuron  $k$ , we have *because of the chain rule*

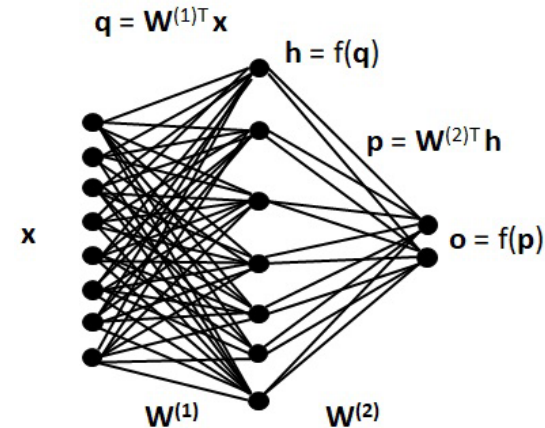
$$\frac{\partial \mathcal{J}}{\partial W_{jk}^{(2)}} = \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} \frac{\partial p_k}{\partial W_{jk}^{(2)}}$$



# The (error) Backpropagation algorithm

For the weights  $\mathbf{W}_{jk}^{(2)}$  connecting the hidden neuron  $j$  with the output neuron  $k$ , we have

$$\frac{\partial \mathcal{J}}{\partial W_{jk}^{(2)}} = \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} \frac{\partial p_k}{\partial W_{jk}^{(2)}}$$



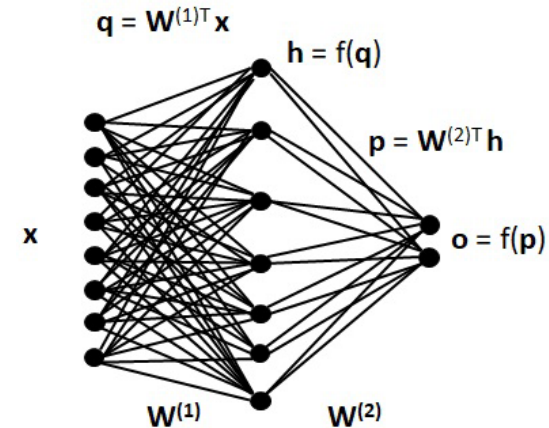
The last term is equal to

$$\frac{\partial p_k}{\partial W_{jk}^{(2)}} = \frac{\partial}{\partial W_{jk}^{(2)}} \left( \sum_{l=1}^{|\mathbf{h}|} W_{lk}^{(2)} h_l \right) = \frac{\partial}{\partial W_{jk}^{(2)}} (W_{jk}^{(2)} h_j) = h_j$$

# The (error) Backpropagation algorithm

For the weights  $\mathbf{W}_{jk}^{(2)}$  connecting the hidden neuron  $j$  with the output neuron  $k$ , we have

$$\frac{\partial \mathcal{J}}{\partial W_{jk}^{(2)}} = \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} \frac{\partial p_k}{\partial W_{jk}^{(2)}}$$



The second term is equal to

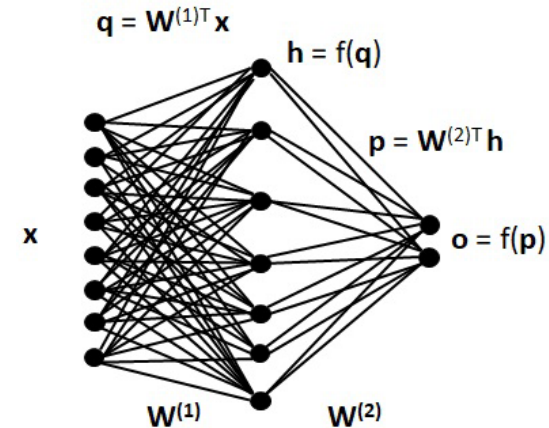
$$\frac{\partial o_k}{\partial p_k} = \frac{\partial}{\partial p_k} f(p_k) = f'(p_k)$$

This is why we need the activation function  $f(\cdot)$  to be differentiable.

# The (error) Backpropagation algorithm

For the weights  $\mathbf{W}_{jk}^{(2)}$  connecting the hidden neuron  $j$  with the output neuron  $k$ , we have

$$\frac{\partial \mathcal{J}}{\partial W_{jk}^{(2)}} = \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} \frac{\partial p_k}{\partial W_{jk}^{(2)}}$$



The first term is equal to

$$\frac{\partial \mathcal{J}}{\partial o_k} = \frac{\partial}{\partial o_k} \left( \frac{1}{2} (t_k - o_k)^2 \right) = o_k - t_k$$

# The (error) Backpropagation algorithm

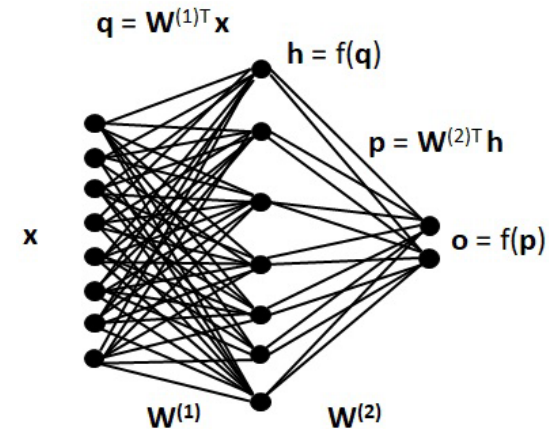
For the weights  $\mathbf{W}_{jk}^{(2)}$  connecting the hidden layer  $j$  with  
The output neuron  $k$ , we have

$$\frac{\partial \mathcal{J}}{\partial W_{jk}^{(2)}} = \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} \frac{\partial p_k}{\partial W_{jk}^{(2)}}$$

Thus, we have

$$\frac{\partial \mathcal{J}}{\partial W_{jk}^{(2)}} = (o_k - t_k) f'(p_k) h_k$$

Is this easy to be calculated?

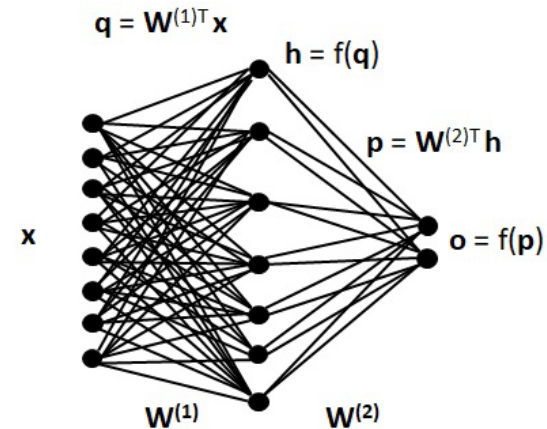


# The (error) Backpropagation algorithm

Let's now focus on the weights  $\mathbf{W}^{(1)}$ .

For the weights  $\mathbf{W}_{ij}^{(1)}$  connecting the input neuron  $i$  with the hidden neuron  $j$ , we have

$$\frac{\partial \mathcal{J}}{\partial W_{ij}^{(1)}} = \frac{\partial \mathcal{J}}{\partial h_j} \frac{\partial h_j}{\partial q_j} \frac{\partial q_j}{\partial W_{ij}^{(1)}}$$



# The (error) Backpropagation algorithm

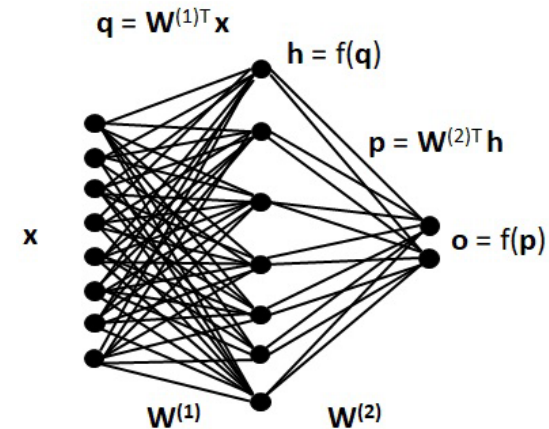
Let's now focus on the weights  $\mathbf{W}^{(1)}$ .

For the weights  $\mathbf{W}_{ij}^{(1)}$  connecting the input neuron  $i$  with the hidden neuron  $j$ , we have

$$\frac{\partial \mathcal{J}}{\partial W_{ij}^{(1)}} = \frac{\partial \mathcal{J}}{\partial h_j} \frac{\partial h_j}{\partial q_j} \frac{\partial q_j}{\partial W_{ij}^{(1)}}$$

The last term is equal to

$$\frac{\partial q_j}{\partial W_{ij}^{(1)}} = \frac{\partial}{\partial W_{ij}^{(1)}} \left( \sum_{l=1}^D W_{lj}^{(1)} x_l \right) = \frac{\partial}{\partial W_{ij}^{(1)}} (W_{ij}^{(1)} x_i) = x_i$$





# The (error) Backpropagation algorithm

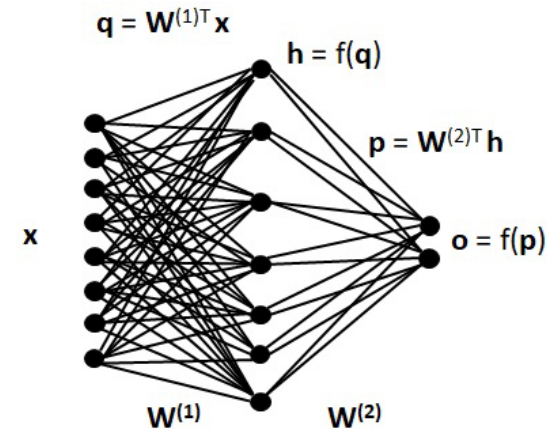
Let's now focus on the weights  $\mathbf{W}^{(1)}$ .

For the weights  $\mathbf{W}_{ij}^{(1)}$  connecting the input neuron  $i$  with the hidden neuron  $j$ , we have

$$\frac{\partial \mathcal{J}}{\partial W_{ij}^{(1)}} = \frac{\partial \mathcal{J}}{\partial h_j} \frac{\partial h_j}{\partial q_j} \frac{\partial q_j}{\partial W_{ij}^{(1)}}$$

The second term is equal to

$$\frac{\partial h_j}{\partial q_j} = \frac{\partial}{\partial q_j} f(q_j) = f'(q_j)$$

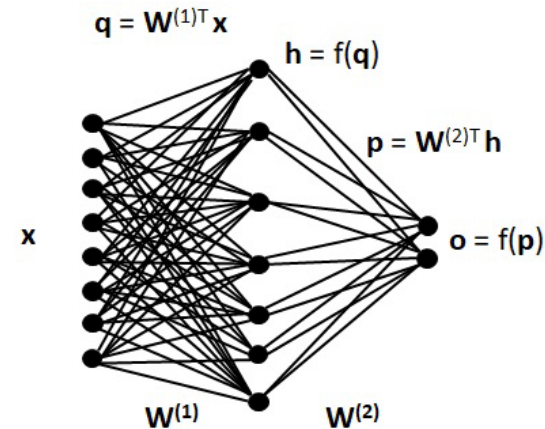


# The (error) Backpropagation algorithm

Let's now focus on the weights  $\mathbf{W}^{(1)}$ .

For the weights  $\mathbf{W}_{ij}^{(1)}$  connecting the input neuron  $i$  with the hidden neuron  $j$ , we have

$$\frac{\partial \mathcal{J}}{\partial W_{ij}^{(1)}} = \frac{\partial \mathcal{J}}{\partial h_j} \frac{\partial h_j}{\partial q_j} \frac{\partial q_j}{\partial W_{ij}^{(1)}}$$



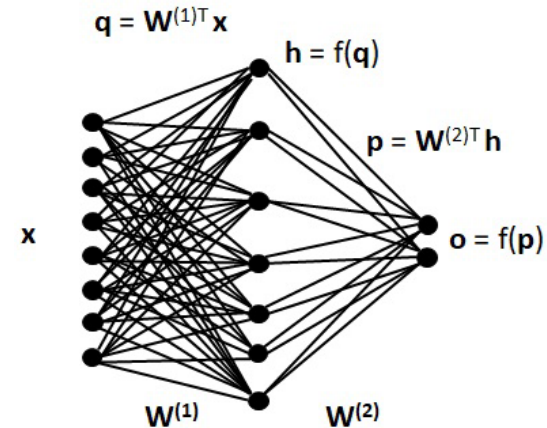
The calculation of the first term seems complicated, because neuron  $j$  belongs to the hidden layer.

# The (error) Backpropagation algorithm

Let's now focus on the weights  $\mathbf{W}^{(1)}$ .

For the weights  $\mathbf{W}_{ij}^{(1)}$  connecting the input neuron  $i$  with the hidden neuron  $j$ , we have

$$\frac{\partial \mathcal{J}}{\partial W_{ij}^{(1)}} = \frac{\partial \mathcal{J}}{\partial h_j} \frac{\partial h_j}{\partial q_j} \frac{\partial q_j}{\partial W_{ij}^{(1)}}$$



The calculation of the first term seems complicated, because neuron  $j$  belongs to the hidden layer.

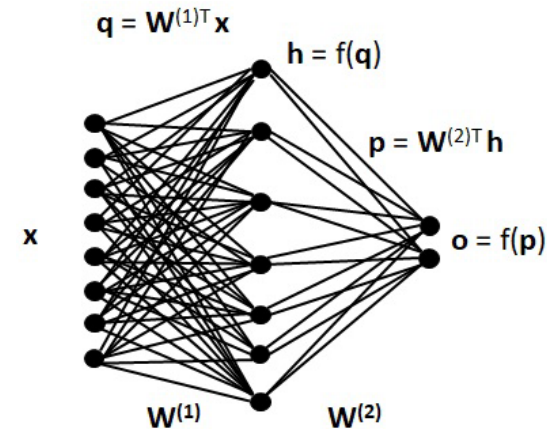
This means that neuron  $j$  affects the output (and the criterion  $\mathcal{J}$ ) through all values in  $\mathbf{W}^{(2)}$ .

# The (error) Backpropagation algorithm

Let's now focus on the weights  $\mathbf{W}^{(1)}$ .

For the weights  $\mathbf{W}_{ij}^{(1)}$  connecting the input neuron  $i$  with the hidden neuron  $j$ , we have

$$\frac{\partial \mathcal{J}}{\partial W_{ij}^{(1)}} = \frac{\partial \mathcal{J}}{\partial h_j} \frac{\partial h_j}{\partial q_j} \frac{\partial q_j}{\partial W_{ij}^{(1)}}$$



The first term is equal to

$$\frac{\partial \mathcal{J}}{\partial h_j} = \sum_{k=1}^K \left( \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} \frac{\partial p_k}{\partial h_j} \right) = \sum_{k=1}^K \left( \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} W_{jk}^{(2)} \right)$$

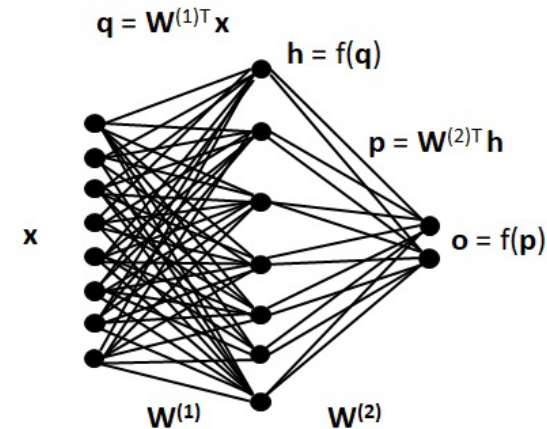
# The (error) Backpropagation algorithm

Let's now focus on the weights  $\mathbf{W}^{(1)}$ .

The first term is equal to

$$\frac{\partial \mathcal{J}}{\partial h_j} = \sum_{k=1}^K \left( \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} \frac{\partial p_k}{\partial h_j} \right) = \sum_{k=1}^K \left( \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} W_{jk}^{(2)} \right)$$

$$\frac{\partial o_k}{\partial p_k} = \frac{\partial}{\partial p_k} f(p_k) = f'(p_k)$$

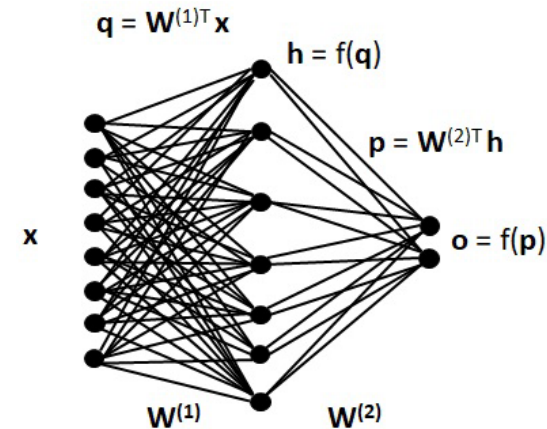


# The (error) Backpropagation algorithm

Let's now focus on the weights  $\mathbf{W}^{(1)}$ .

The first term is equal to

$$\frac{\partial \mathcal{J}}{\partial h_j} = \sum_{k=1}^K \left( \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} \frac{\partial p_k}{\partial h_j} \right) = \sum_{k=1}^K \left( \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} W_{jk}^{(2)} \right)$$



$$\frac{\partial \mathcal{J}}{\partial o_k} = \frac{\partial}{\partial o_k} \left( \frac{1}{2} (t_k - o_k)^2 \right) = o_k - t_k$$

# The (error) Backpropagation algorithm

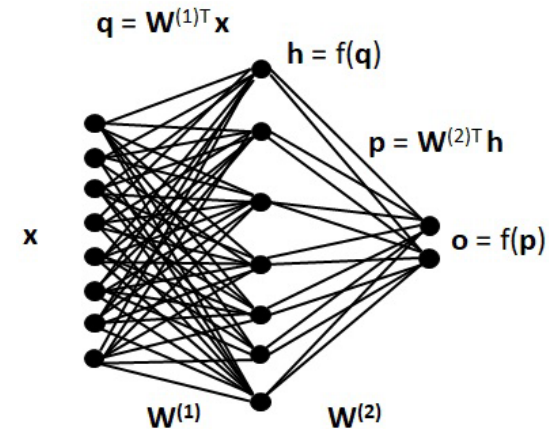
Let's now focus on the weights  $\mathbf{W}^{(1)}$ .

Thus the derivative

$$\frac{\partial \mathcal{J}}{\partial W_{ij}^{(1)}} = \frac{\partial \mathcal{J}}{\partial h_j} \frac{\partial h_j}{\partial q_j} \frac{\partial q_j}{\partial W_{ij}^{(1)}}$$

becomes

$$\frac{\partial \mathcal{J}}{\partial W_{ij}^{(1)}} = \sum_{k=1}^K \left( \frac{\partial \mathcal{J}}{\partial o_k} \frac{\partial o_k}{\partial p_k} W_{jk}^{(2)} \right) f'(q_j) x_i$$



# The (error) Backpropagation algorithm

In the general case where the neural network is formed by more than three layers, the weight connecting the  $i$ -th neuron in layer  $l$  with the  $j$ -th neuron in layer  $l+q$  is updated using the gradient

$$\frac{\partial \mathcal{J}}{\partial W_{ij}^{(l)}} = \delta_j^l h_i^l$$

Delta: the output of the derivative for the forward neuron. This value is stored so we can use it when we go back to compute the derivative for the previous layers. This can be seen as the derivative of the error.

Delta<sub>j</sub> backpropagated error!

where

$$\delta_j^l = \left( \sum_{q \in \mathcal{L}_{l+1}} \delta_q^{l+1} W_{jq}^{l+1} \right) f'(h_j^{l+1})$$



# The (error) Backpropagation algorithm

In the general case where the neural network is formed by more than three layers, the weight connecting the  $i$ -th neuron in layer  $l$  with the  $j$ -th neuron in layer  $l+q$  is updated using the gradient

$$\frac{\partial \mathcal{J}}{\partial W_{ij}^{(l)}} = \delta_j^l h_i^l$$

where

$$\delta_j^l = \left( \sum_{q \in \mathcal{L}_{l+1}} \delta_q^{l+1} W_{jq}^{l+1} \right) f'(h_j^{l+1})$$

and the gradient is used to update the weight using  $W_{ij}^{(l)} = W_{ij}^{(l)} - \eta \frac{\partial \mathcal{J}}{\partial W_{ij}^{(l)}}$

# The (error) Backpropagation algorithm

---

**Algorithm 12:** Stochastic Backpropagation

---

- 1: Initialize the network and its weights  $\mathbf{w}$ ,  $\eta(\cdot)$ ,  $\theta$ ,  $t = 0$
  - 2: **Do**  $t \leftarrow t + 1$
  - 3:   Select (randomly) a vector  $\mathbf{x}_i$
  - 4:   Calculate the network's output for  $\mathbf{x}_i$
  - 5:   Update all network's weights  $\mathbf{W}^{(l)}$ ,  $l = 1, \dots, L$
  - 6:    $W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \eta(t) \delta_j^l h_i^l$
  - 7: **until**  $\nabla \mathcal{J}(\mathbf{W}) < \theta$    Until the objective function doesn't change much
-

# The (error) Backpropagation algorithm

---

**Algorithm 13:** Batch Backpropagation

---

- 1: Initialize the network and its weights  $\mathbf{w}$ ,  $\eta(\cdot)$ ,  $\theta$ ,  $t = 0$ ,  $M$
  - 2: **Do**  $t \leftarrow t + 1$
  - 3:   Set:  $m = 0$ ,  $\Delta W_{ij}^l$ ,  $l = 1, \dots, L$
  - 4:   **do**  $m = m + 1$
  - 5:     Select (randomly) a vector  $\mathbf{x}_i$  (not selected before)
  - 6:     Calculate the network's output for  $\mathbf{x}_i$
  - 7:      $\Delta W_{ij}^l \leftarrow W_{ij}^{(l)} + \eta(t) \delta_j^l h_i^l$
  - 8:   **until**  $m = M$
  - 9:   Update all network's weights  $\mathbf{W}^{(l)}$ ,  $l = 1, \dots, L$
  - 10:    $W_{ij}^{(l)} \leftarrow W_{ij}^{(l)} - \eta(t) \Delta W_{ij}^l$
  - 11: **until**  $\nabla \mathcal{J}(\mathbf{W}) < \theta$
-

# Practical matters

There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?

Trail and error, apply transfer learning e.g. AlexNet

1. Start with one hidden layer.
2. Try different network topologies

# Practical matters

There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?
- Data pre-processing?

Pre-processing helps avoid the weights of the network to be trapped in regions of the activation function making their output saturated?

Centering: subtract the mean from the samples

Standardisation: center the data and scale each dimension by its standard deviation.

Normalisation: scale each sample independently using L2 norm or L1 norm

Decorrelation: apply PCA without dimensionality reduction to rotate data using eigenvectors of the covariance matrix.

Whitening: decorrelate and scale each dimension by the eigenvalues, transforming the covariance matrix to a identity matrix

It is important to pre-process the input data. Otherwise the weights may increase

# Practical matters

There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?
- Data pre-processing?
- Weights initialization?

Usually random initialization within a uniform distribution  $[-\gamma, \gamma]$ . where  $\gamma = \frac{1}{\sqrt{D} \cdot c}$ , where  $\alpha$  is a small value and  $D$  is the number of dimensions.

If our pre-processing step, we performed  $\square$  data standardization, then we have positive and negative values equally, on average, so we want positive and negative weights as well. Therefore, we choose weights from a uniform distribution.

If the  $\square$  value of  $\gamma$  is too small, depending on the chosen activation function,  $\square$  the response of that layer might be too small making the learning process of the  $\square$  entire network difficult.

If the value of  $\gamma$  is selected to be too big, then the neurons  $\square$  in that layer might saturate even before the actual training is done.

# Practical matters

There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?
- Data pre-processing?
- Weights initialization?
- What learning rate?

Usually a small number  $10^q$  where  $q$  is between -1 to -5.

Adaptive learning rate means that we decrease the learning rate in each iteration:

- start with e.g. 0.1 and each iteration  $i$  compute  $LR = \eta_0 * e^{-i * \text{decaying\_rate}}$

# Practical matters

There are several issues that need to be appropriately handled when training a multilayer neural network:

- What type of architecture, how many layers, how many neurons?
- Data pre-processing?
- Weights initialization?
- What learning rate?

Why Learn with Momentum? When training neural networks using a sample based or mini-batch based process, the direction of the gradients for the successive updates might be inconsistent. This might result in oscillations and slow convergence. Learning with momentum allows a smoother update of the network's parameters

Learning with momentum:

$$W_{ij}^{(l)}(t+1) = W_{ij}^{(l)}(t) - \eta \left( \frac{\partial \mathcal{J}}{\partial W_{ij}^{(l)}(t)} + \alpha \frac{\partial \mathcal{J}}{\partial W_{ij}^{(l)}(t-1)} \right)$$

Gradient at iteration t      Gradient computed in the previous iteration

Momentum can be seen as a process averaging the stochastic variations in the weight updates during stochastic (sample-based) learning.

Learning with momentum is motivated from a physical perspective of the optimization problem. The optimization problem is interpreted as climbing down a hill. In this case, the force pushing the object downward is affected by its momentum and the local variance of the forces at a particular position

Alpha -- a value between 0 and 1 -- determine how much weight we put on the gradient from the previous iteration. If alpha is 0 then we don't have any momentum. Typically, alpha=0.9



# Other types of feedforward neural networks

There are several types of neural networks. Here we will discuss for:

- Radial Basis Function (RBF) network
- Auto-Encoder (AE) networks
- Self-Organizing Maps (SOMs)
- Convolutional Neural Networks (CNNs)

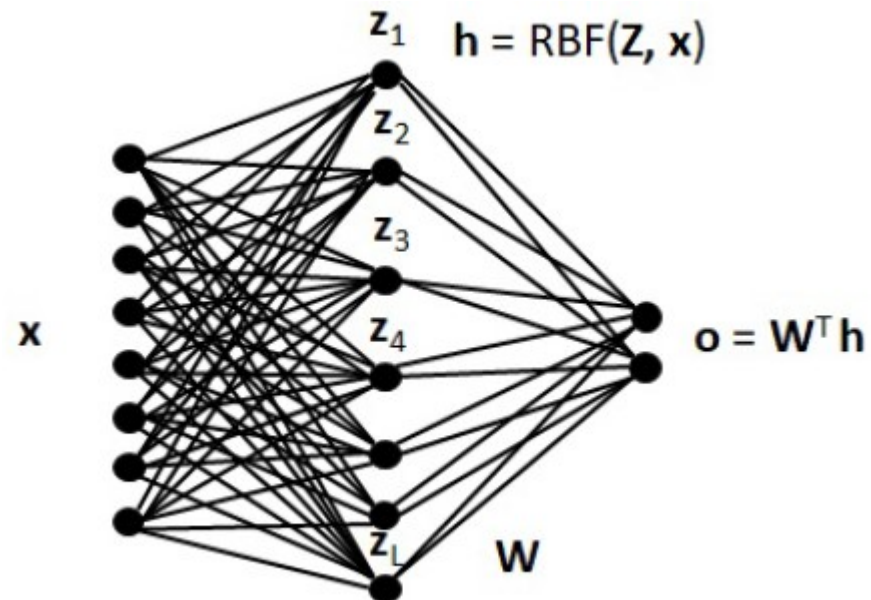
# Radial Basis Function networks

RBF network is a three-layer network. It is formed by:

- An input layer
- A hidden layer formed by RBF neurons
- An output layer with linear neurons

Radial means that a neuron will react the same way depending how "near" it is to the target value.  
Reacting = the output of the neuron

Why are we doing this? Function approximation that has multiple modes or multiple peaks.



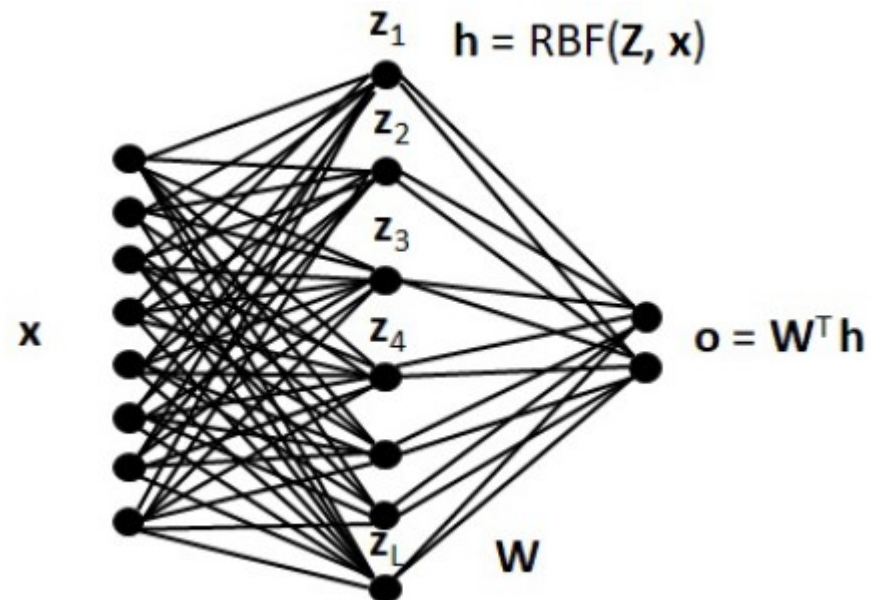
# Radial Basis Function networks

RBF network is a three-layer network. It is formed by:

- An input layer
- A hidden layer formed by RBF neurons
- An output layer with linear neurons

An RBF neuron has as parameters

- a vector  $\mathbf{z}_i \in \mathbb{R}^D$  and
- a value  $\sigma$



# Radial Basis Function networks

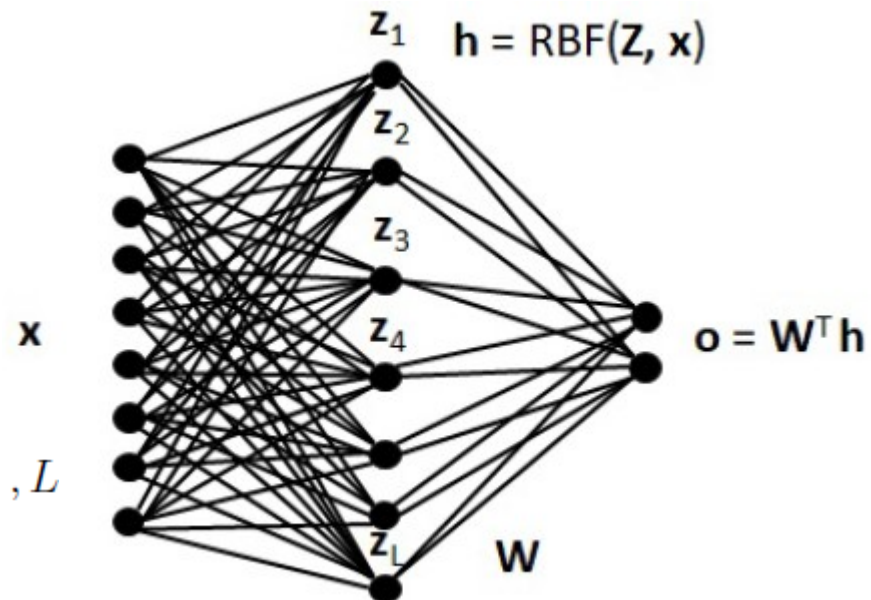
RBF network is a three-layer network. It is formed by:

- An input layer
- A hidden layer formed by RBF neurons
- An output layer with linear neurons

An RBF neuron has as parameters

- a vector  $\mathbf{z}_l \in \mathbb{R}^D$  and
- a value  $\sigma$

$$h_l = RBF(\mathbf{z}_l, \mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{z}_l\|_2^2}{\sigma^2}\right), \quad l = 1, \dots, L$$



# Radial Basis Function networks

Another way is to take a set of points in the training set

RBF network is a three-layer network. It is formed by:

- An input layer
- A hidden layer formed by RBF neurons
- An output layer with linear neurons

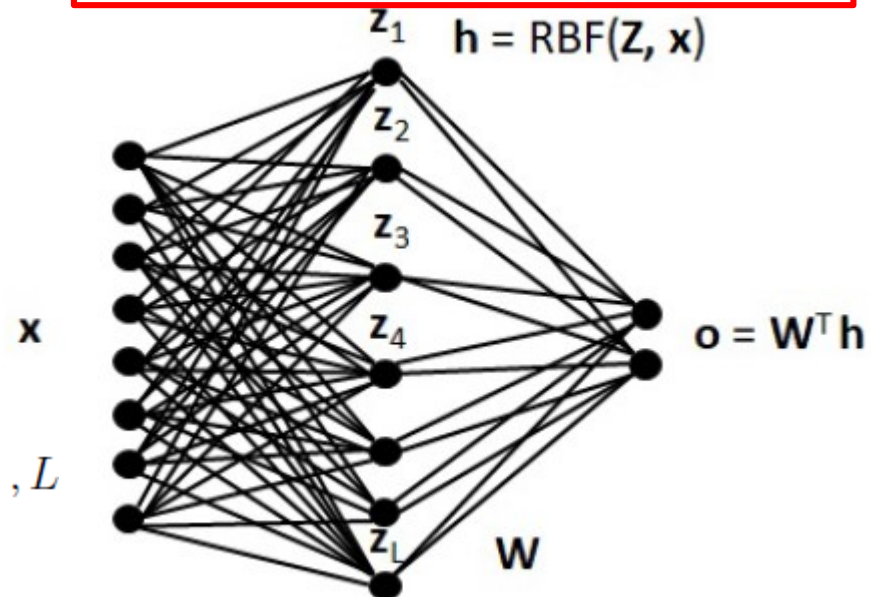
$\mathbf{z}_l \in \mathbb{R}^D$ ,  $l=1, \dots, K$  are calculated by applying a clustering algorithm on the training data  $\mathbf{x}$ .

An RBF neuron has as parameters

- a vector  $\mathbf{z}_l \in \mathbb{R}^D$  and
- a value  $\sigma$

$$h_l = RBF(\mathbf{z}_l, \mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{z}_l\|_2^2}{\sigma^2}\right), \quad l = 1, \dots, L$$

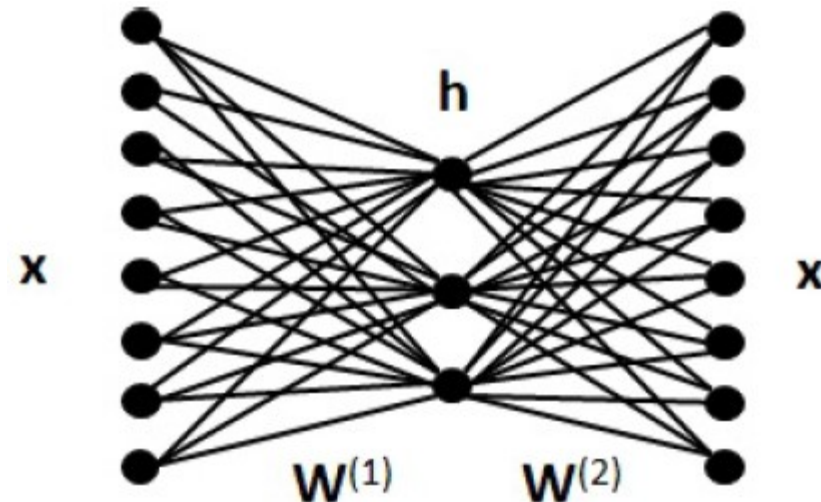
You can train the sigma



# Auto-Encoder network

An Auto-Encoder network (or simply Auto-Encoder) is a three layer neural network formed by:

- An input layer
- A hidden layer which usually corresponds to a bottleneck
- An output layer with number of neurons equal to the input

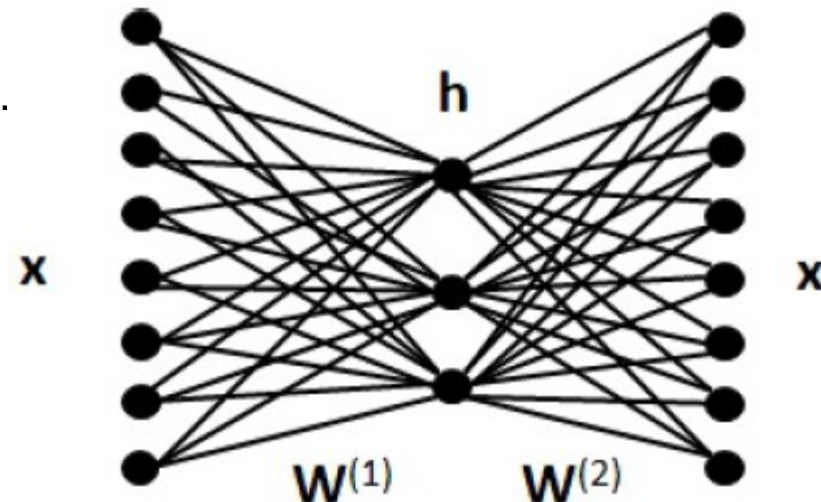


# Auto-Encoder network

An Auto-Encoder network (or simply Auto-Encoder) is a three layer neural network formed by:

- An input layer
- A hidden layer which usually corresponds to a bottleneck
- An output layer with number of neurons equal to the input

The network is trained by applying standard Backpropagation and using as output the input. This corresponds to unsupervised training!



# Auto-Encoder network

An Auto-Encoder network (or simply Auto-Encoder) is a three layer neural network formed by:

- An input layer
- A hidden layer which usually corresponds to a bottleneck
- An output layer with number of neurons equal to the input

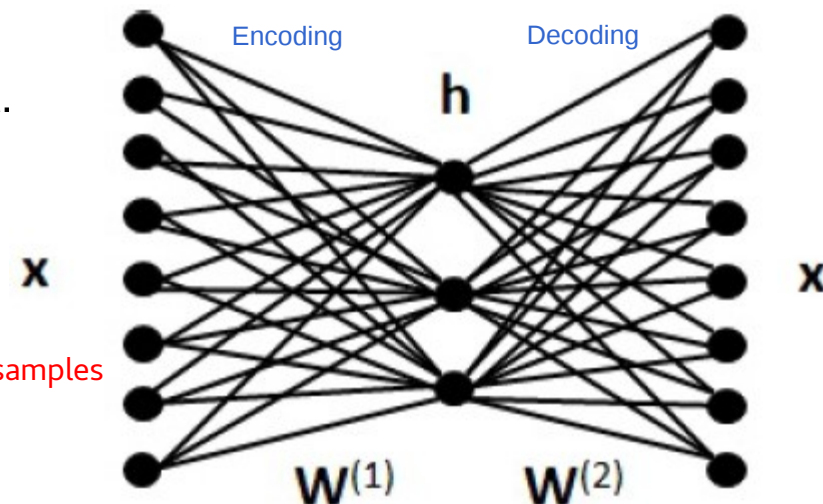
Create sparse representation of the input data.  
Once the hidden layer can reconstruct the input data, then it has learned the projection matrix in a lower dimension.

Bottleneck layer forces  
the next to find a compressed representation with a much smaller set of neurons

The network is trained by applying standard Backpropagation and using as output the input. This corresponds to unsupervised training!

## Why to do this?

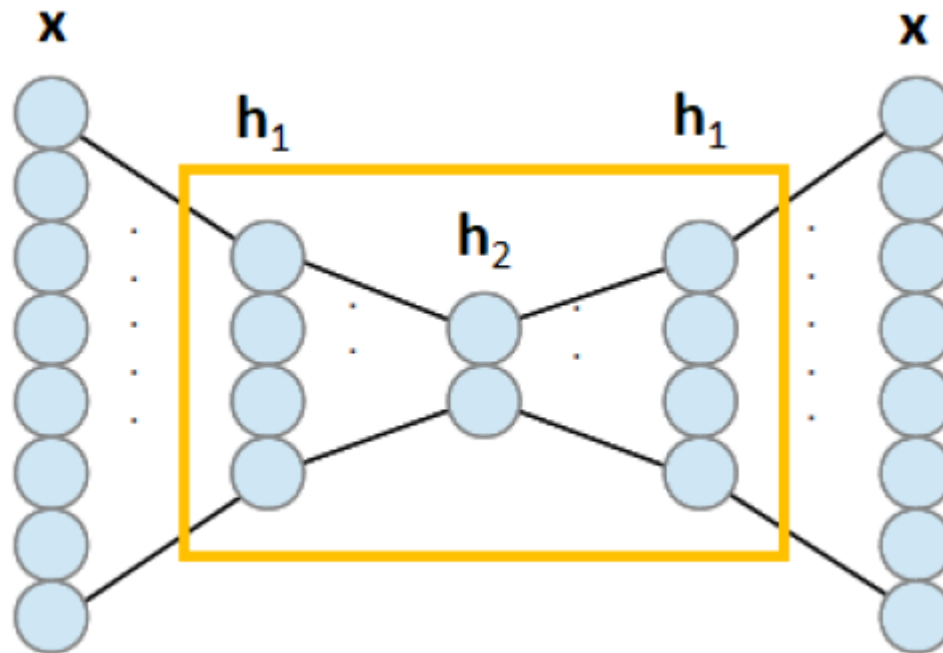
To remove noise. The bottleneck may throw away noise given enough samples because the noise will be randomness. It is basically, automatic dimensionality reduction. Also to feature selection





# Auto-Encoder network

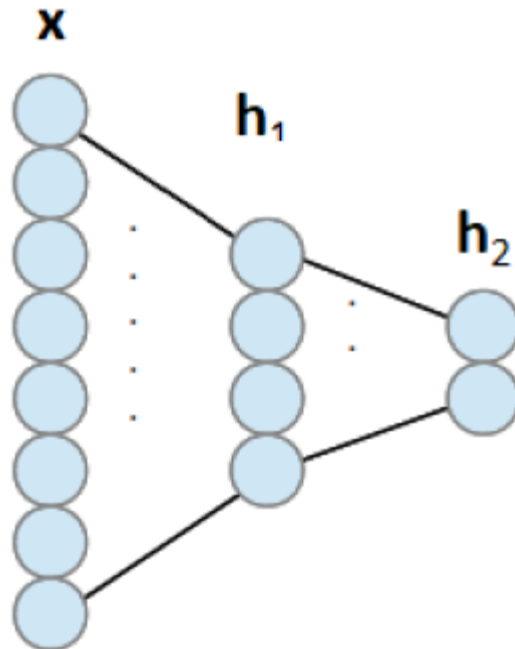
Auto-Encoders have been used for pre-training the weights of deep networks



# Auto-Encoder network

Auto-Encoders have been used for pre-training the weights of deep networks

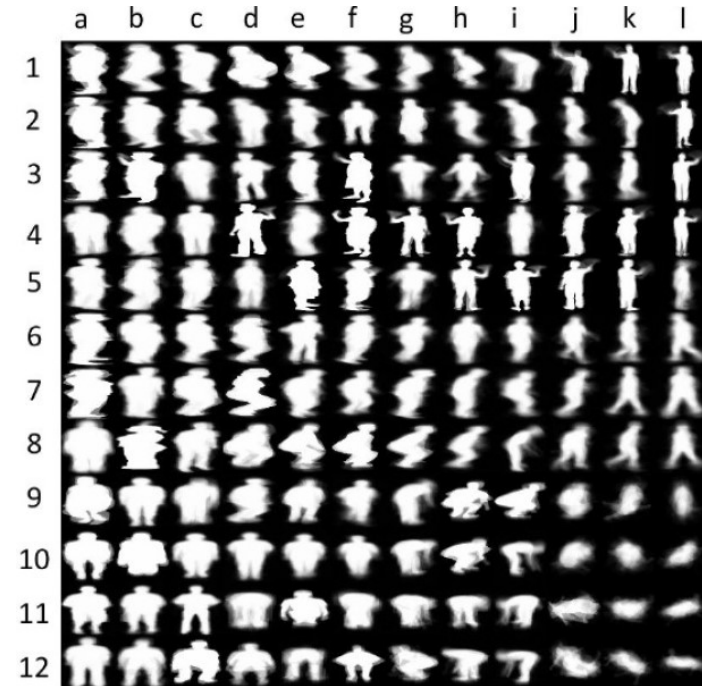
After initializing the three-layer architecture, an output (classification) layer is stacked on top of the third layer and the (now) four-layer neural network can be trained using (error) Backpropagation.



# Self-Organizing Map

A SOM is a neural network trained in an unsupervised way to define prototypes of data.

It defines a topographic map, in which each neuron corresponds to a prototype.



# Self-Organizing Map

It's training process is based on three procedures:

- Competition
- Cooperation
- Adaptation

The neurons of the SOM are randomly initialized and the training samples are introduced in a random sequence to update the neurons.

# Self-Organizing Map

## Competition

For each of the training vectors  $\mathbf{x}_i \in \mathbb{R}^D$ , its Euclidean distance from every SOM weight,  $\mathbf{w}_j \in \mathbb{R}^D$ ,  $j = 1, \dots, K$  is calculated. The winning neuron is the one that gives the smallest distance:

$$j^* = \arg \min_j \| \mathbf{x}_i - \mathbf{w}_j \|_2.$$

# Self-Organizing Map

## Cooperation

The winning neuron  $j^*$  indicates the center of a topological neighborhood  $h_{j^*}$ . Neurons are excited depending on their lateral distance,  $r_{j^*}$ , from this neuron. A typical choice of  $h_{j^*}$  is the Gaussian function:

$$h_{j^*k}(n) = \exp\left(-\frac{r_{j^*k}^2}{2\sigma^2(n)}\right),$$

where  $k$  corresponds to the neuron at hand,  $n$  is the iteration of the algorithm,  $r_{j^*k}$  is the Euclidean distance between neurons  $j^*$  and  $k$  in the lattice space and  $\sigma$  is the “effective width” of the topological neighborhood.  $\sigma$  is a function of  $n$ :  $\sigma(n) = \sigma_0 \exp(-\frac{n}{N_0})$ , where  $N_0$  is the total number of training iterations.  $\sigma(0) = \frac{l_w + l_h}{4}$  in our experiments.  $l_w$  and  $l_h$  are the lattice width and height, respectively.

# Self-Organizing Map

## Adaptation

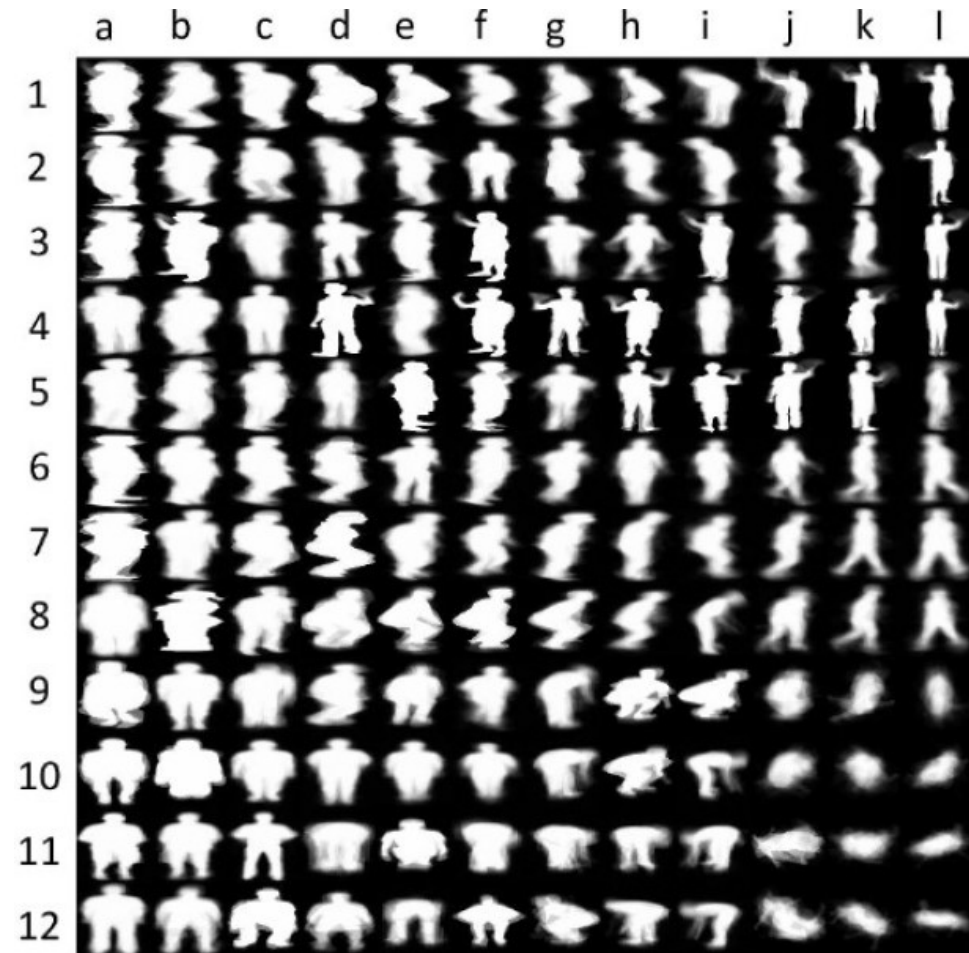
At this step, each neuron is adapted with respect to its lateral distance from the winning neuron as follows:

$$\mathbf{w}_k(n+1) = \mathbf{w}_k(n) + \eta(n)h_{j^*k}(n)(\mathbf{x}_i - \mathbf{w}_k(n)), \quad (6.29)$$

where  $\eta(n)$  is the learning-rate parameter:  $\eta(n) = \eta(0) \exp(-\frac{n}{N_0})$ .  $\eta(0) = 0.1$  in our experiments.

# Self-Organizing Map

Properties of the (trained) SOM





# Convolutional Neural Networks

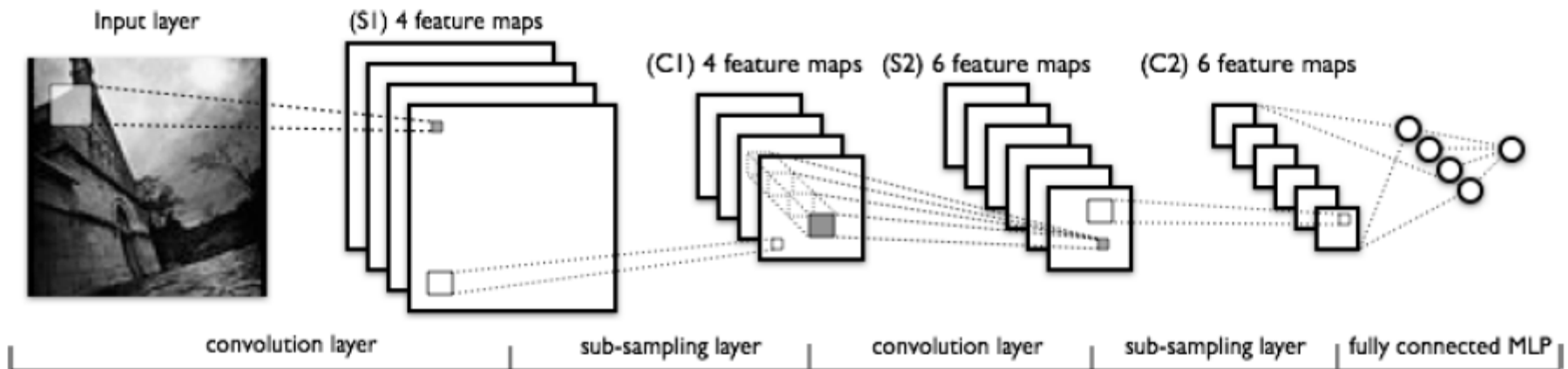
All neural networks we have seen until now take vectors as inputs.

That is, in order to use them in a classification problem, we need to find a way to convert the input data (e.g. an image) to a vector.

Such data representations are usually referred to as hand-crafted data representations.

CNNs can (usually) lead to higher performances since they take as input the raw (image) data and optimize both the representation and classification parameters in a combined manner.

# Convolutional Neural Networks



# Convolutional Neural Networks

A CNN is formed by three types of layers:

- Convolutional layers
- Sub-sampling layers
- Fully-connected layers

# Convolutional Neural Networks

## Convolutional layer

It is formed by multiple neurons, each neuron equipped with:

- a filter with  $(h \times w)$  elements
- a nonlinear activation function

The filter of each neuron 'scans' the image/tensor  $I_{m-1}$  and at each position it calculates

$$I_m(y, x) = f \left( \sum_{d=1}^D \sum_{i=0}^1 \sum_{j=0}^1 w_{ij}^{0d} I_{m-1}^d(y + i, x + j) \right)$$

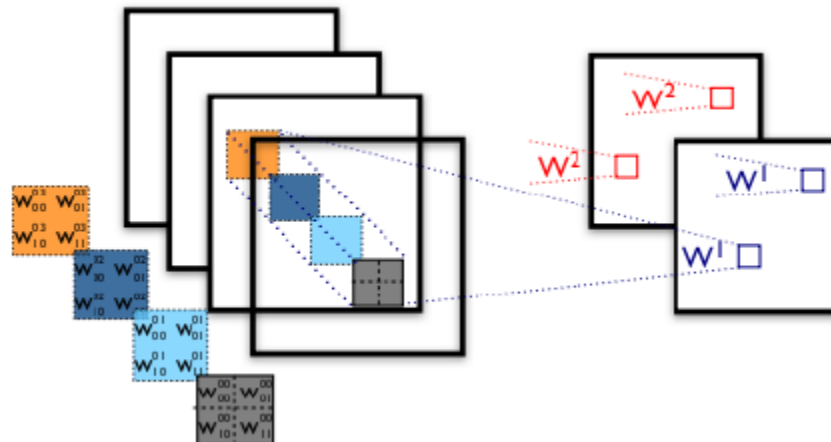
Doing the above for all positions in the image  $I_{m-1}$  leads to the creation of a new image  $I_m$ .

# Convolutional Neural Networks

## Convolutional layer

The filter of each neuron 'scans' the image/tensor  $I_{m-1}$  and at each position it calculates

$$I_m(y, x) = f \left( \sum_{d=1}^D \sum_{i=0}^1 \sum_{j=0}^1 w_{ij}^{0d} I_{m-1}^d(y + i, x + j) \right)$$



# Convolutional Neural Networks

## Convolutional layer

The filter of each neuron 'scans' the image/tensor  $I_{m-1}$  and at each position it calculates

$$I_m(y, x) = f \left( \sum_{d=1}^D \sum_{i=0}^1 \sum_{j=0}^1 w_{ij}^{0d} I_{m-1}^d(y + i, x + j) \right)$$

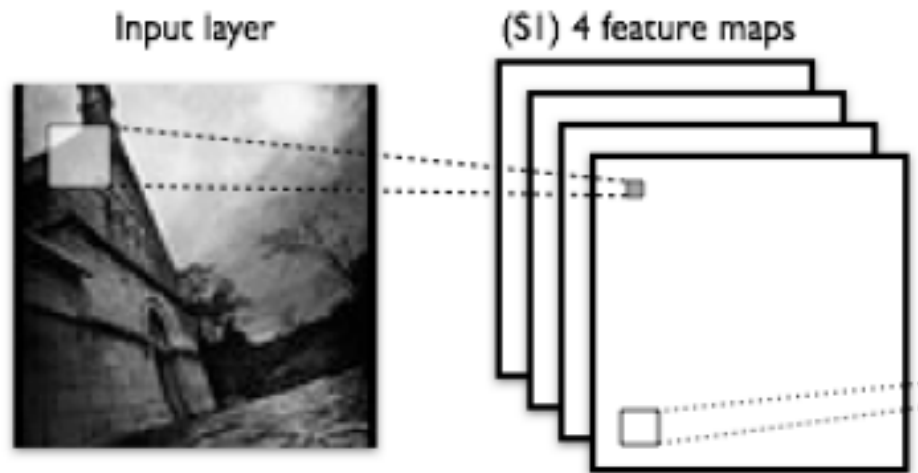
Doing the above for all positions in the image  $I_{m-1}$  leads to the creation of a new image  $I_m$ .

The new image  $I_m$  is called feature map.

Multiple neurons in a layer create a new feature map each.

# Convolutional Neural Networks

## Convolutional layer

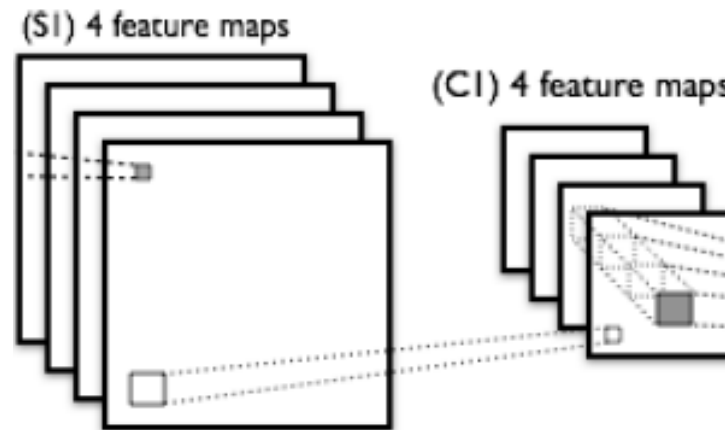


# Convolutional Neural Networks

## Subsampling layer

It resizes the feature maps of layer  $l$  to create a new layer  $l+1$ :

- average pooling
- max pooling
- rand pooling

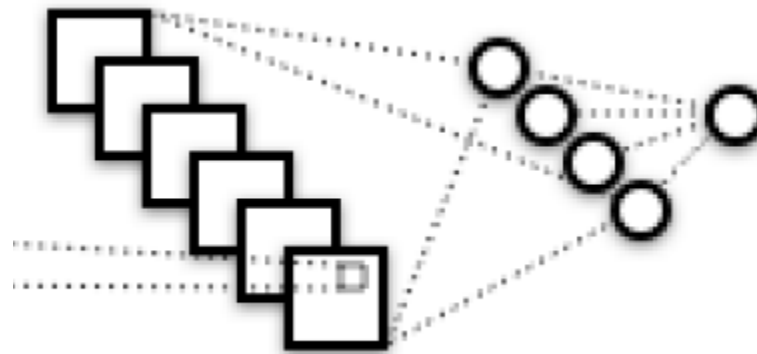




# Convolutional Neural Networks

## Fully-connected layer

Standard vector-based neural network layer



# Training protocols

When determining some parameters of an algorithm that need to be selected by the user (these are called hyper-parameters), we need to find a way to compare the goodness of models.

# Training protocols

When determining some parameters of an algorithm that need to be selected by the user (these are called hyper-parameters), we need to find a way to compare the goodness of models.

Comparing the goodness of different models using the performance on the training data might lead to overfitting.

How to select the hyper-parameters?

# Training protocols

There are two ways to select the hyper-parameters:

- Evaluate the goodness on a separate set for which we know the truth (e.g. labels). This set is called validation set. For example, we can split the training set in 70%-30% splits

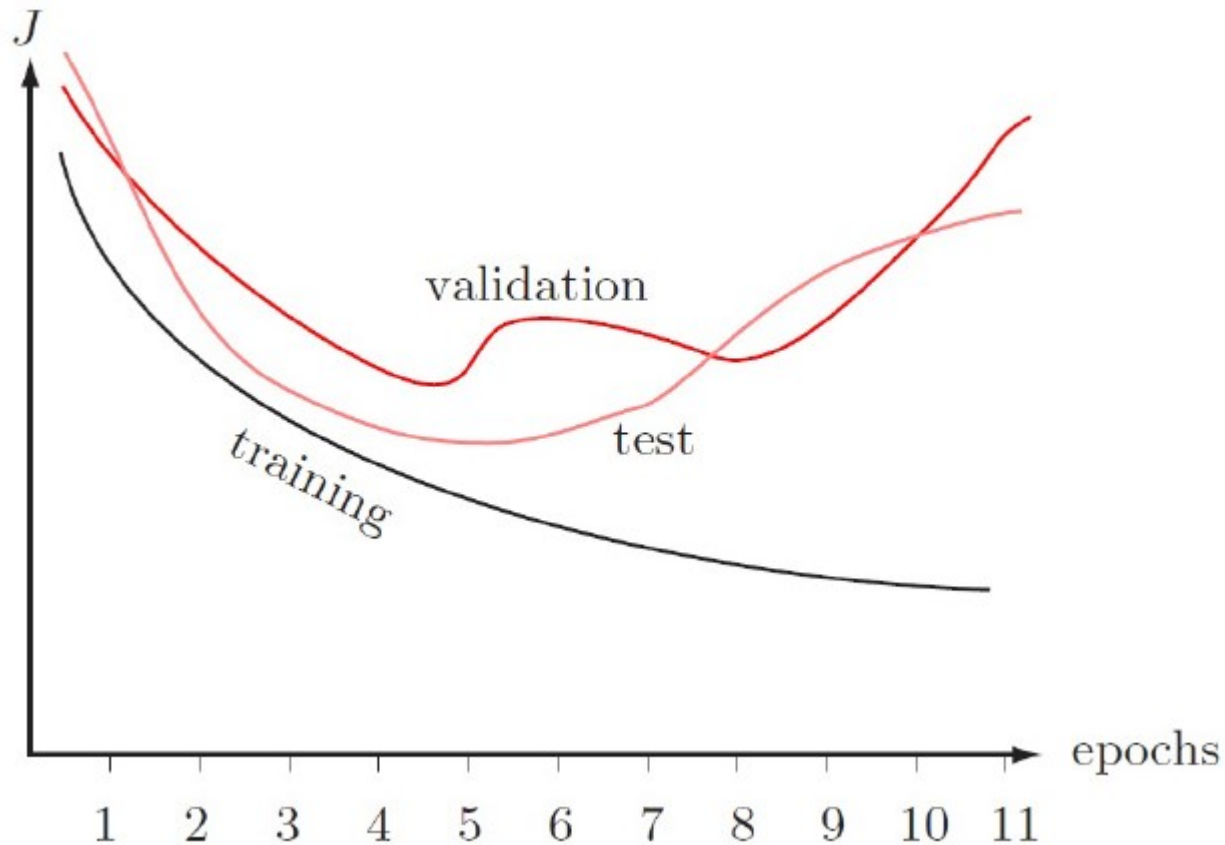
# Training protocols

There are two ways to select the hyper-parameters:

- Evaluate the goodness on a separate set for which we know the truth (e.g. labels). This set is called validation set. For example, we can split the training set in 70%-30% splits
- Apply cross-validation:
  - We split the training set in  $K$  (mutually exclusive) subsets ( $K$ -fold cross validation).
  - For a set of hyper-parameter values, we train the model  $K$  times using  $K-1$  sets and we test it with the remaining set. Then, we calculate the average performance over all  $K$  experiments and the corresponding standard deviation.
  - We select the hyper-parameter values providing the best overall performance
- Apply Leave-One-Out cross validation

# Training protocols

Validation error as criterion for stopping training.



# Demos

---

SOM:

<http://www.cis.hut.fi/research/javasomdemo/>

Convolutional Neural Network:

<http://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html>

AutoEncoder:

<http://cs.stanford.edu/people/karpathy/convnetjs/demo/autoencoder.html>

General:

<http://deeplearning.net/demos/>

# Demos

---

Feedforward network:

[link](#)