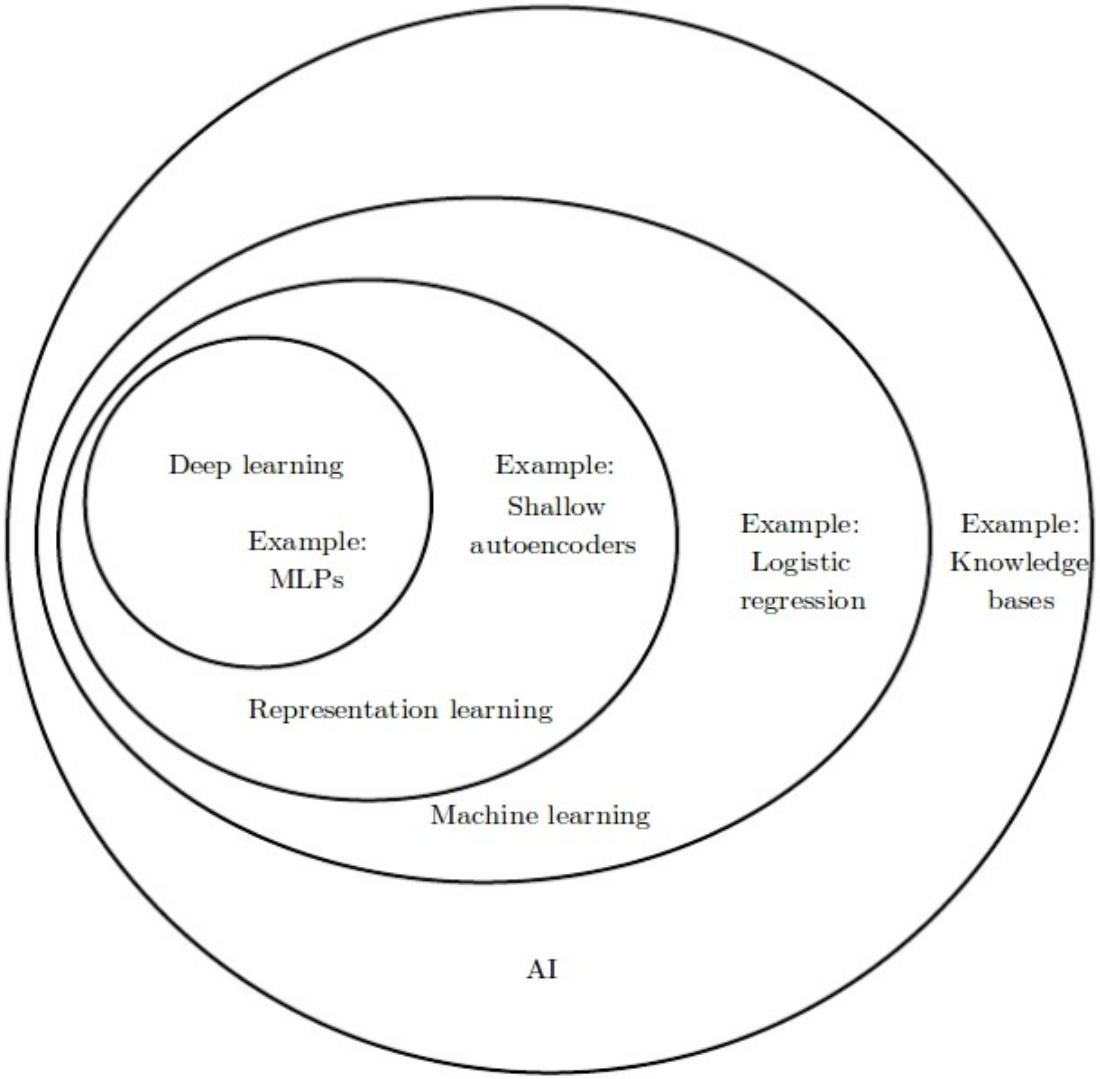
Neural Networks and Deep Learning

InnoMech

Céline Teulière
Celine.teuliere@uca.fr

Introduction to Deep Learning



Deep Learning, Goodfellow, Bengio, Courville, MIT Press 2016

Neural networks and supervised learning

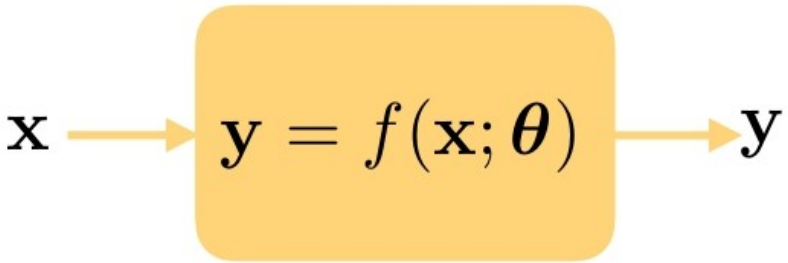
Supervised learning goal: estimate a function

$$f^* \ : \ y^* = f^*(\mathbf{x})$$

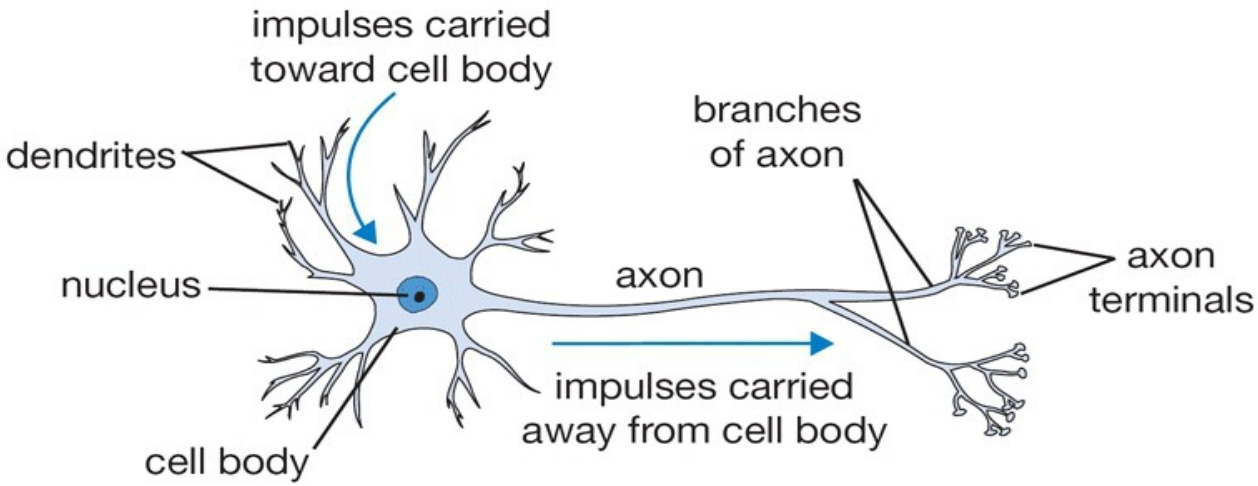
from a set of (potentially noisy) samples

$$\{\mathbf{x}_i, y_i\}_{i=1..m} \qquad y_i \approx f^*(\mathbf{x}_i)$$

Neural networks are a specific family of parametric functions $f(\mathbf{x}; \theta)$ trained to best fit the function f^*



From biological to artificial neuron



From biological to artificial neuron

BULLETIN OF
MATHEMATICAL BIOPHYSICS
VOLUME 5, 1943

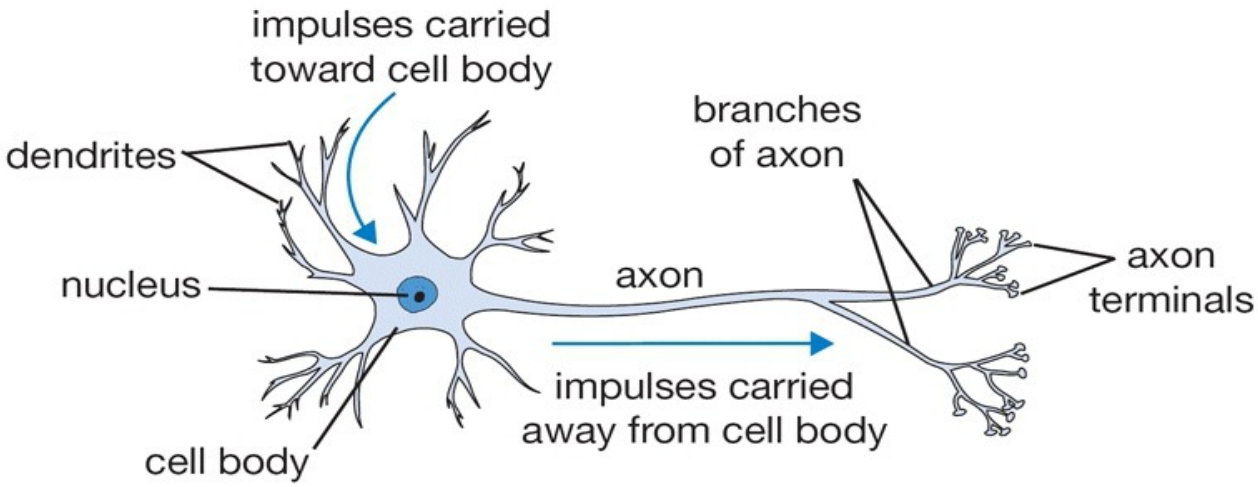
A LOGICAL CALCULUS OF THE IDEAS IMMANENT IN NERVOUS ACTIVITY

WARREN S. MCCULLOCH AND WALTER PITTS

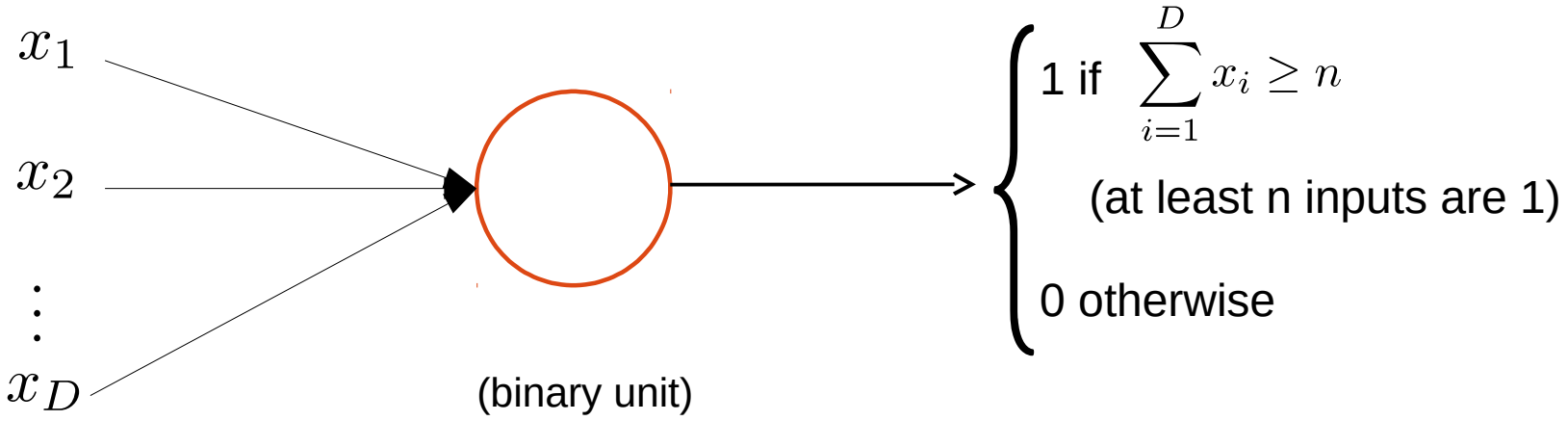
FROM THE UNIVERSITY OF ILLINOIS, COLLEGE OF MEDICINE,
DEPARTMENT OF PSYCHIATRY AT THE ILLINOIS NEUROPSYCHIATRIC INSTITUTE,
AND THE UNIVERSITY OF CHICAGO

[McCulloch et Pitts 1943]

From biological to artificial neuron



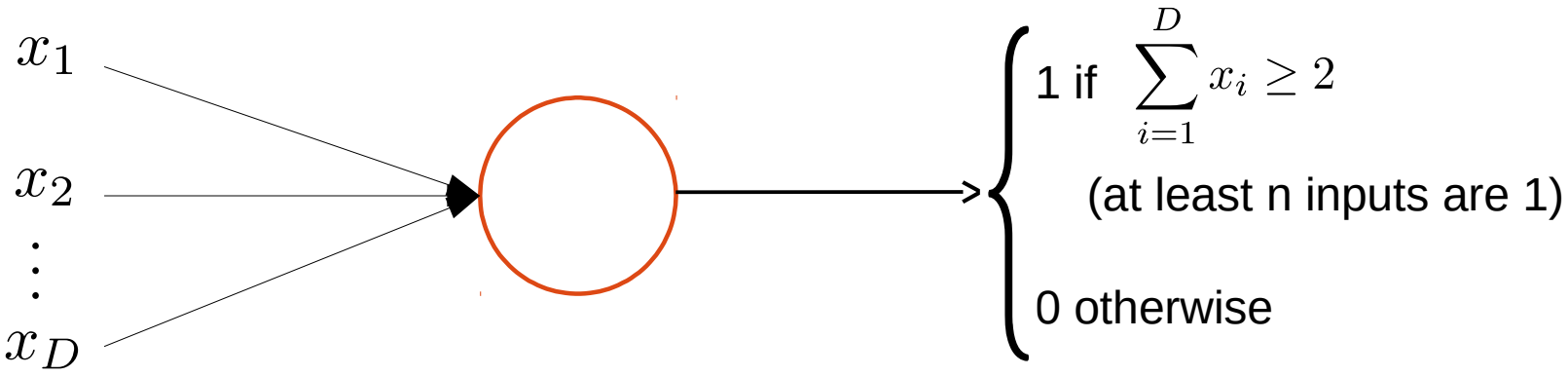
Artificial Neuron model

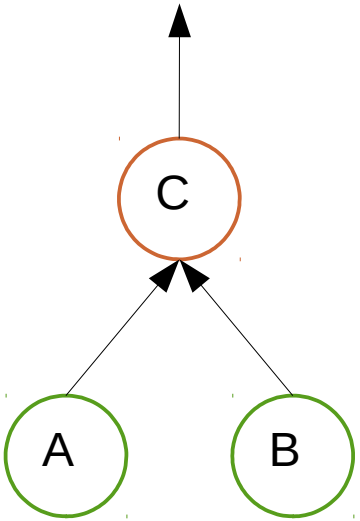


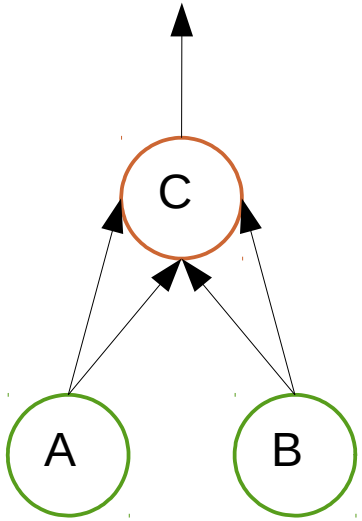
[McCulloch et Pitts 1943]

Artificial neuron

Example of operation with n = 2 (output activated if at least 2 inputs are activated)







Perceptron

Psychological Review
Vol. 65, No. 6, 1958

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

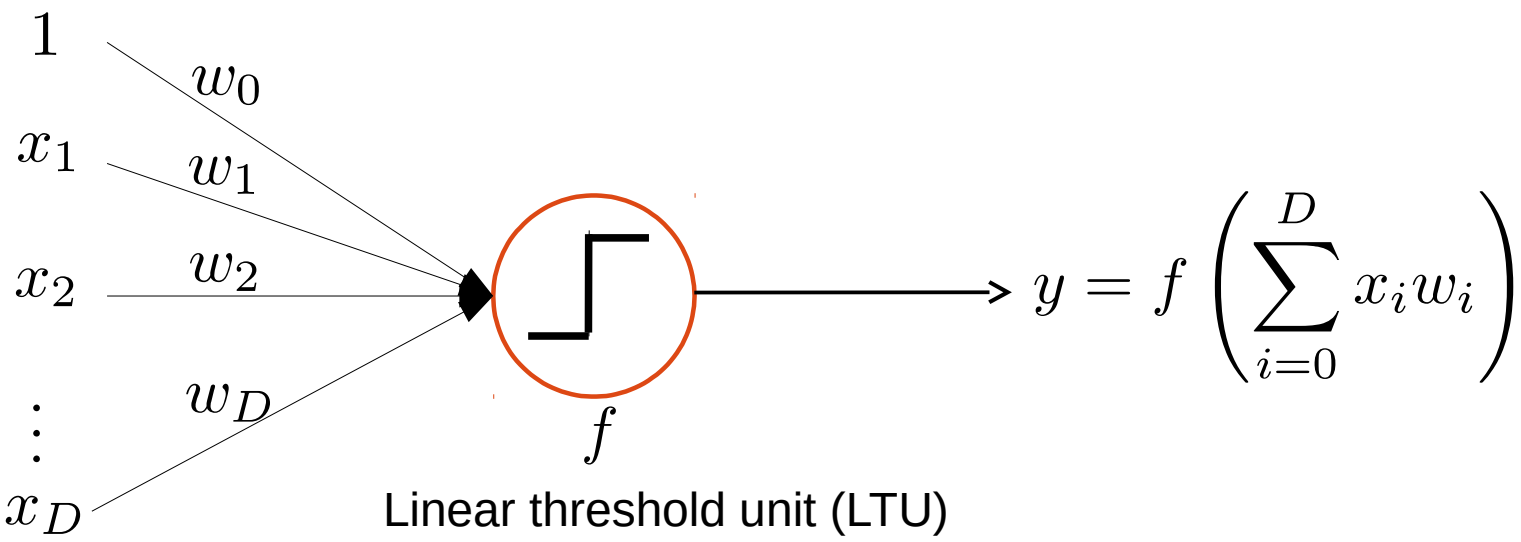
If we are eventually to understand the capability of higher organisms for perceptual recognition, generalization, recall, and thinking, we must first have answers to three fundamental questions:

1. How is information about the physical world sensed, or detected, by the biological system?
2. In what form is information stored, or remembered?
3. How does information contained in storage, or in memory, influence recognition and behavior?

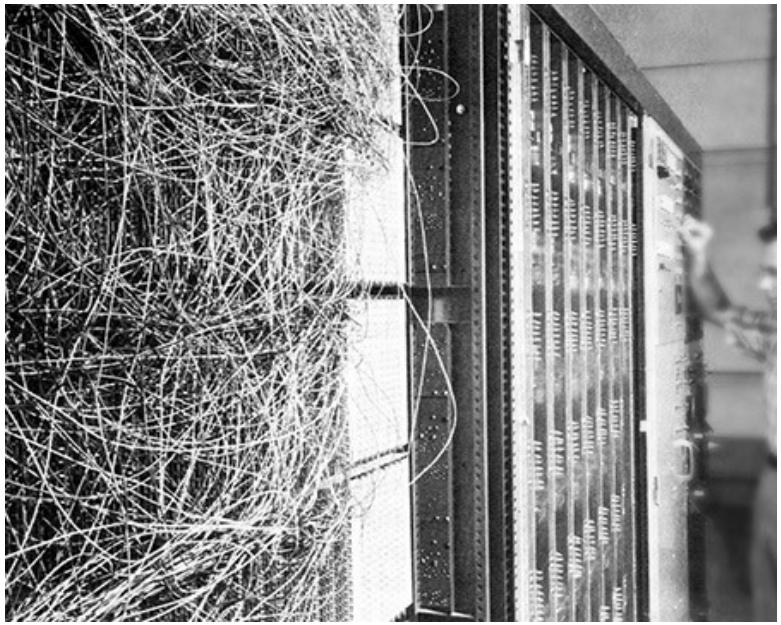
and the stored pattern. According to this hypothesis, if one understood the code or "wiring diagram" of the nervous system, one should, in principle, be able to discover exactly what an organism remembers by reconstructing the original sensory patterns from the "memory traces" which they have left, much as we might develop a photographic negative, or translate the pattern of electrical charges in the "memory" of a digital computer. This hypothesis is appealing in its simplicity and ready intelligibility, and a large family of theoretical brain

[Rosenblatt 1958]

Perceptron



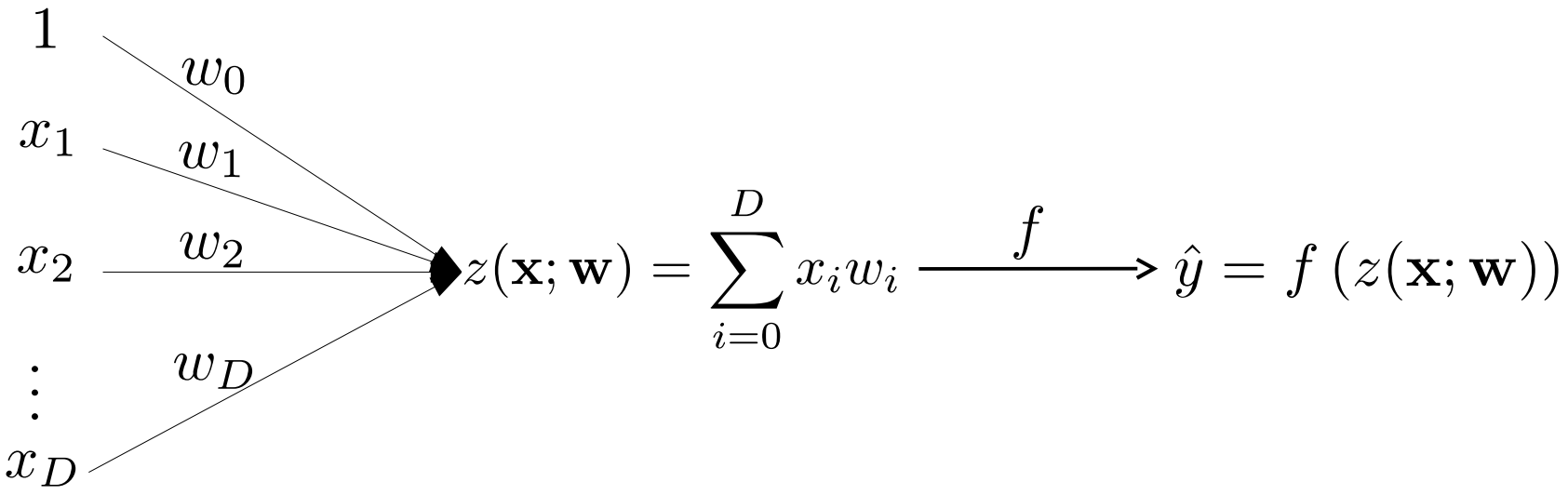
Linear threshold unit (LTU)



[Rosenblatt 1958]

Artificial neuron

Model used in this class



z : Linear pre-activation function

f : non linear activation function

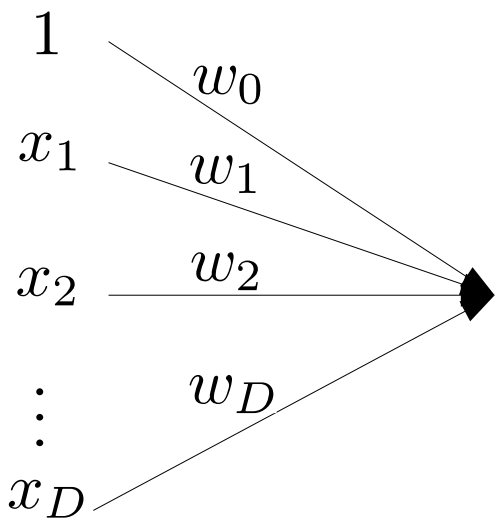
$\mathbf{w} = (w_0, .., w_n)^\top$: network weights

w_0 : biais

$(x_1, .., x_n)^\top$: data vector

$$z(\mathbf{x};\mathbf{w}) = \mathbf{w}^\top \mathbf{x} = \sum_{i=0}^{D} x_i w_i$$

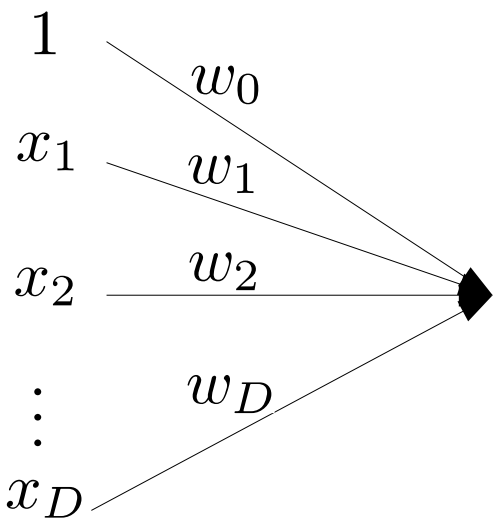
Linear separator



$$z(\mathbf{x};\mathbf{w}) = \mathbf{w}^\top \mathbf{x} = \sum_{i=0}^{D} x_i w_i$$

Example 2 dimensions: $z(\mathbf{x};\mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2$

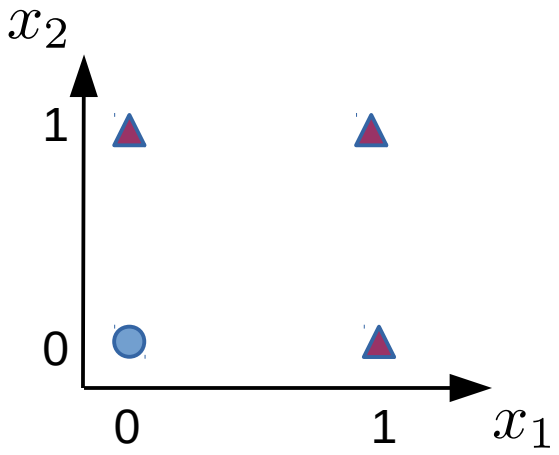
Linear separator



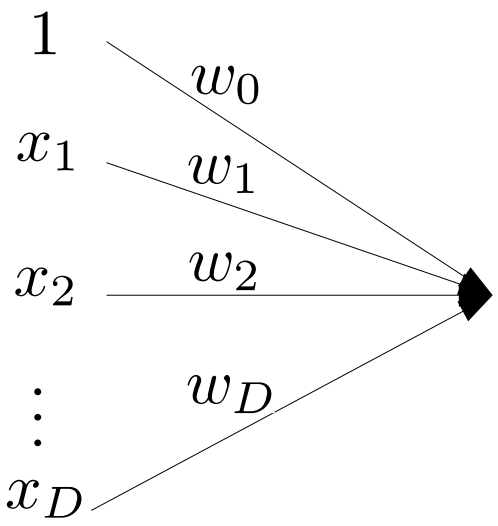
$$z(\mathbf{x};\mathbf{w}) = \mathbf{w}^\top \mathbf{x} = \sum_{i=0}^{D} x_i w_i$$

Example 2 dimensions: $z(\mathbf{x};\mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2$

OR
(0,0) → 0
(0,1) → 1
(1,0) → 1
(1,1) → 1



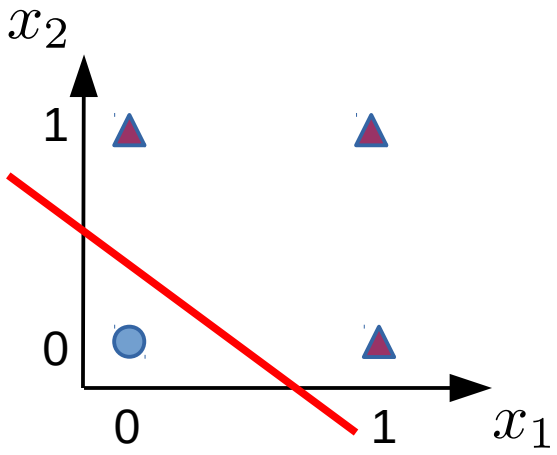
Linear separator



$$z(\mathbf{x};\mathbf{w}) = \mathbf{w}^\top \mathbf{x} = \sum_{i=0}^{D} x_i w_i$$

Example 2 dimensions: $z(\mathbf{x};\mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2$

OR
(0,0) → 0
(0,1) → 1
(1,0) → 1
(1,1) → 1

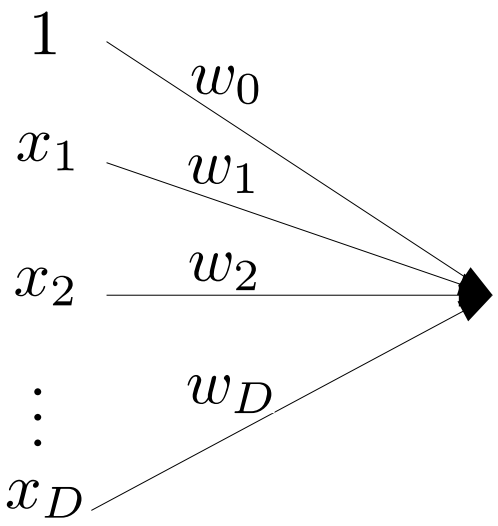


Example of learning the OR function

étape	w_0	w_1	w_2	Entrée	$\sum_{i=0}^{2} w_i x_i$	o	c	w_0	w_1	w_2
Init								0	1	-1
1	0	1	-1	100						
2				101						
3				110						
4				111						
5				100						
6				101						
7				110						
8				111						
9				100						
10				101						

$$w_i \leftarrow w_i + (c_n - o_n) x_n^i$$

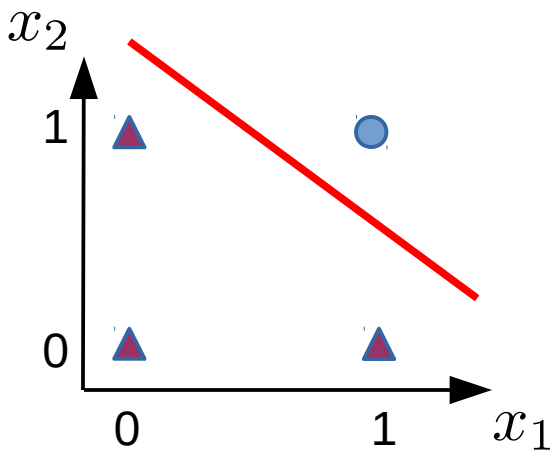
Linear separator



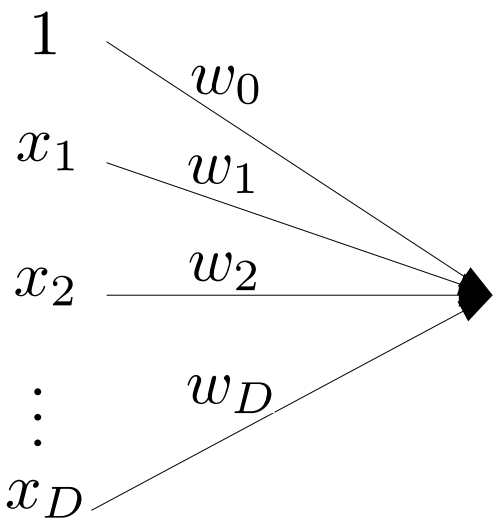
$$z(\mathbf{x};\mathbf{w}) = \mathbf{w}^\top \mathbf{x} = \sum_{i=0}^{D} x_i w_i$$

Example 2 dimensions: $z(\mathbf{x};\mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2$

AND
(0,0) → 0
(0,1) → 0
(1,0) → 0
(1,1) → 1



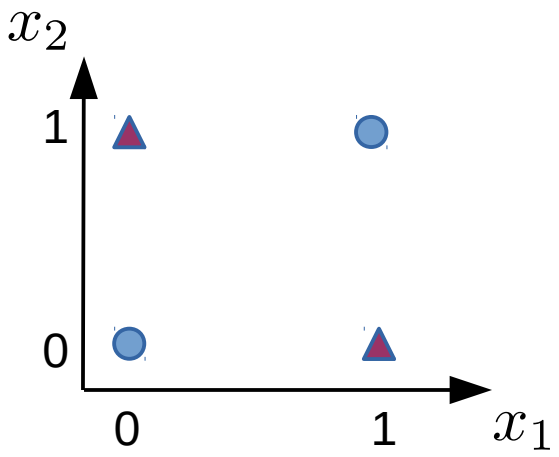
Linear separator



$$z(\mathbf{x};\mathbf{w}) = \mathbf{w}^\top \mathbf{x} = \sum_{i=0}^{D} x_i w_i$$

Example 2 dimensions: $z(\mathbf{x};\mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2$

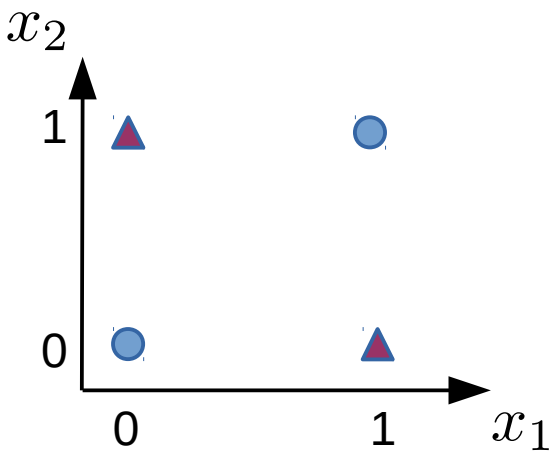
XOR
(0,0) → 0
(0,1) → 1
(1,0) → 1
(1,1) → 0



Linear separator

Example in 2 dimensions: $z(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2$

XOR
(0,0) → 0
(0,1) → 1
(1,0) → 1
(1,1) → 0



Non linearly separable problem!

A single neuron cannot solve this problem
If we apply a **non linear transform** to the data, the problem can become linearly separable:

$AND(\overline{x_1}, x_2)$

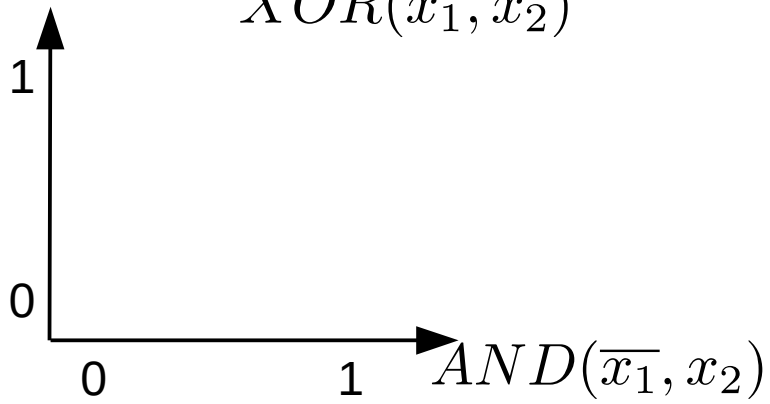
(0,0) →
(0,1) →
(1,0) →
(1,1) →

$AND(x_1, \overline{x_2})$

(0,0) →
(0,1) →
(1,0) →
(1,1) →

$AND(x_1, \overline{x_2})$

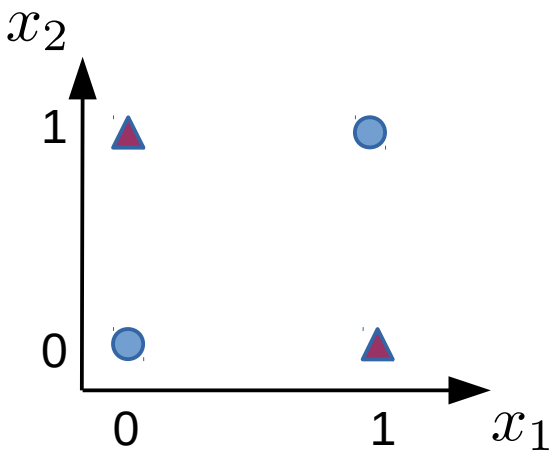
$XOR(x_1, x_2)$



Linear separator

Example in 2 dimensions: $z(\mathbf{x}; \mathbf{w}) = w_0 + w_1 x_1 + w_2 x_2$

XOR
(0,0) → 0
(0,1) → 1
(1,0) → 1
(1,1) → 0



Non linearly separable problem!

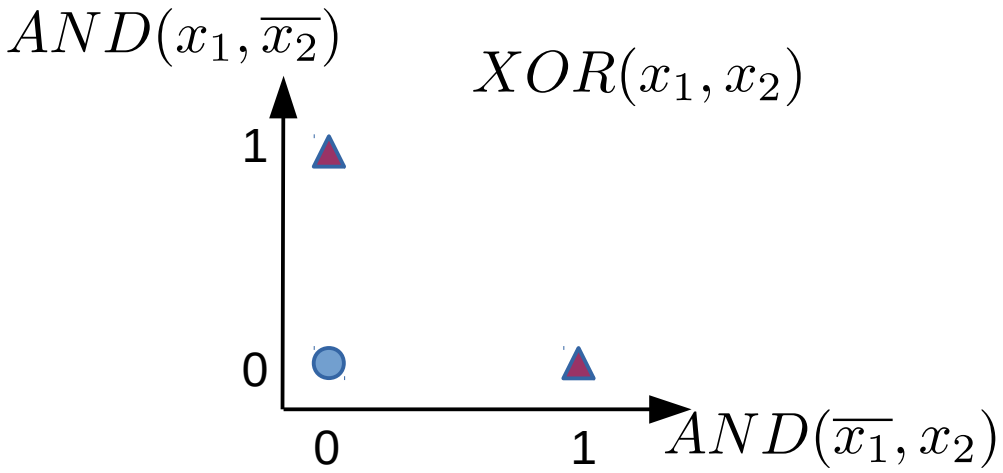
A single neuron cannot solve this problem
If we apply a **non linear transform** to the data, the problem can become linearly separable:

$AND(\overline{x_1}, x_2)$

(0,0) → 0
(0,1) → 1
(1,0) → 0
(1,1) → 0

$AND(x_1, \overline{x_2})$

(0,0) → 0
(0,1) → 0
(1,0) → 1
(1,1) → 0



Linear separator

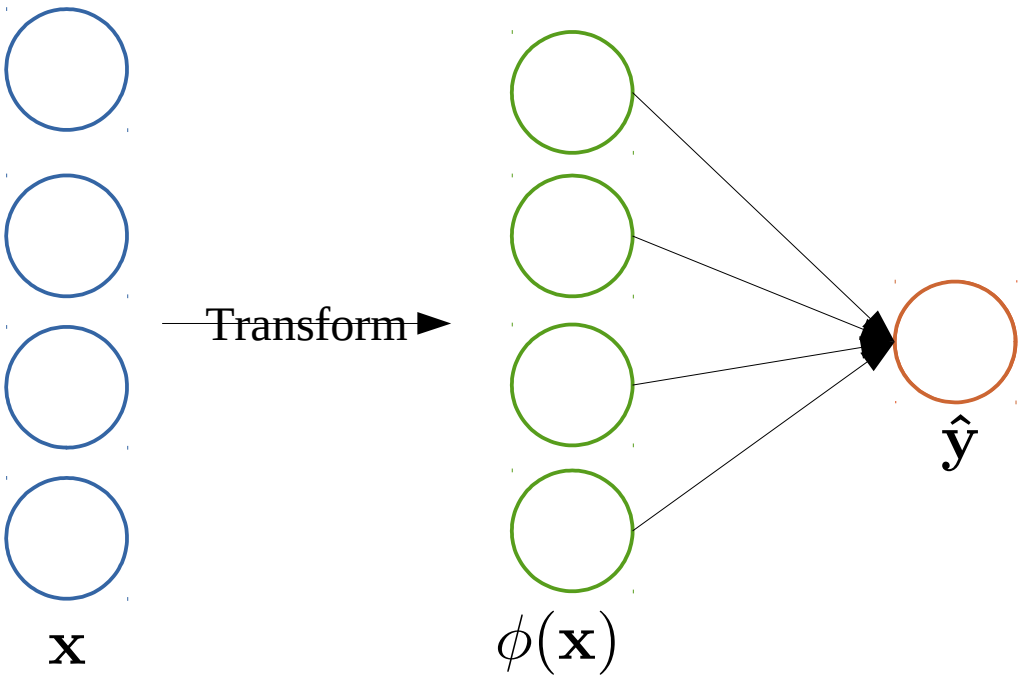
Limitations: the neural network capacity of linear models is limited, they cannot modelise interaction between several inputs.

To extend these models we can apply them to a non-linear transform of the data instead of applying them to the data directly.

How to determine this non linear transform?

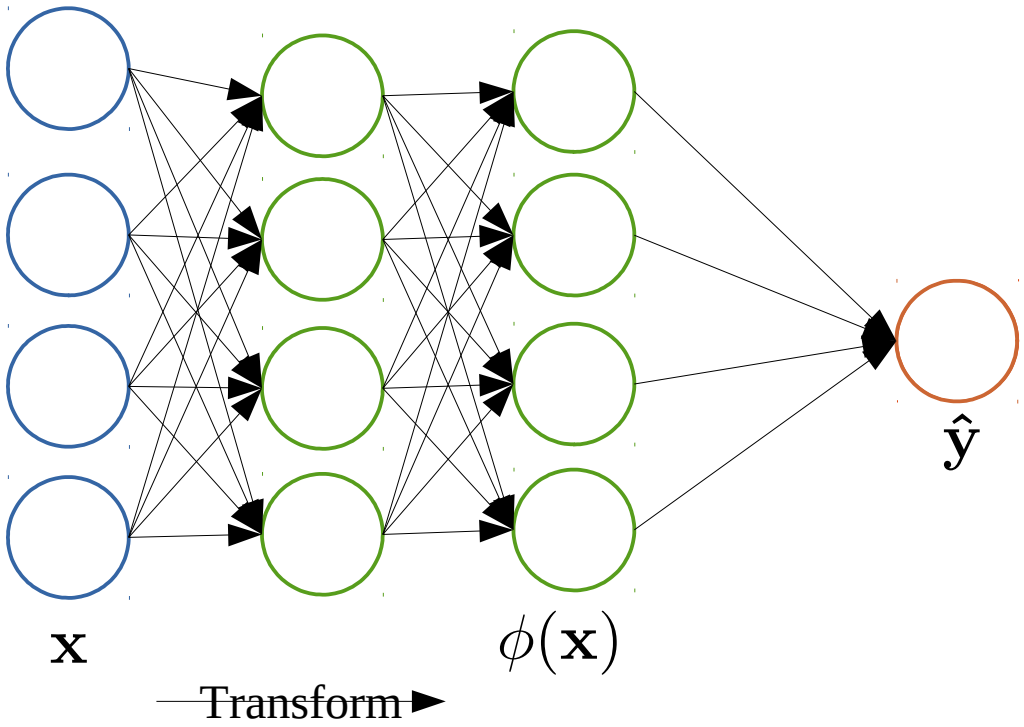
- Use a set of generic transforms
- Hand-specify them
- Learn them → strategy used in deep learning

Features transform



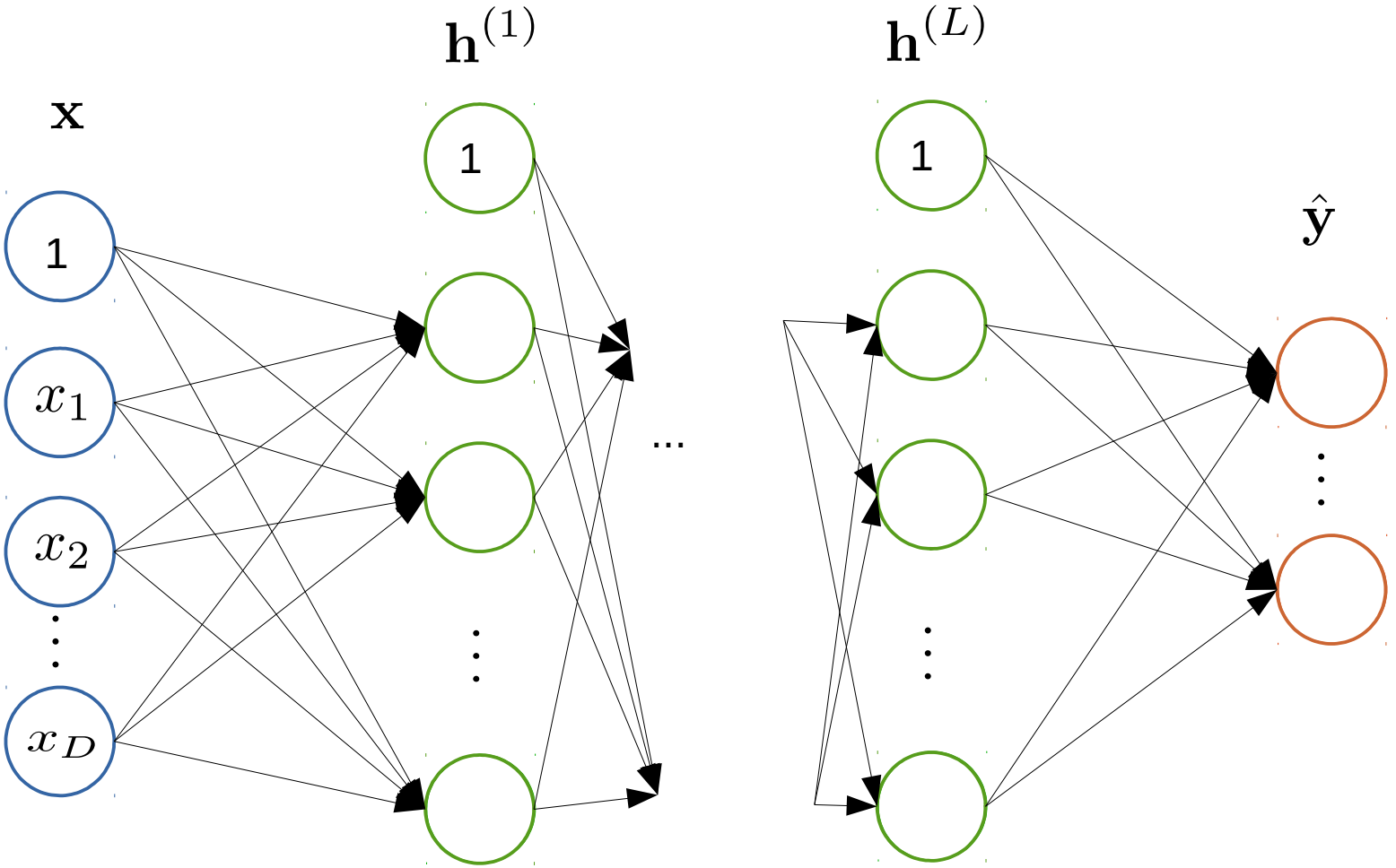
Idea: Non linearly transform the data into another vector of data for which the separation would be linear

Features transform



This transform can be modelised or learned using neural networks.

Multi-layer feedforward neural networks



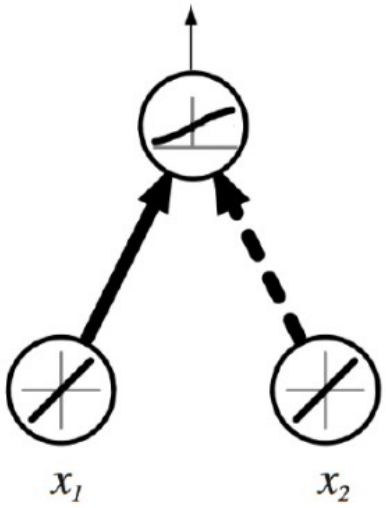
Input layer

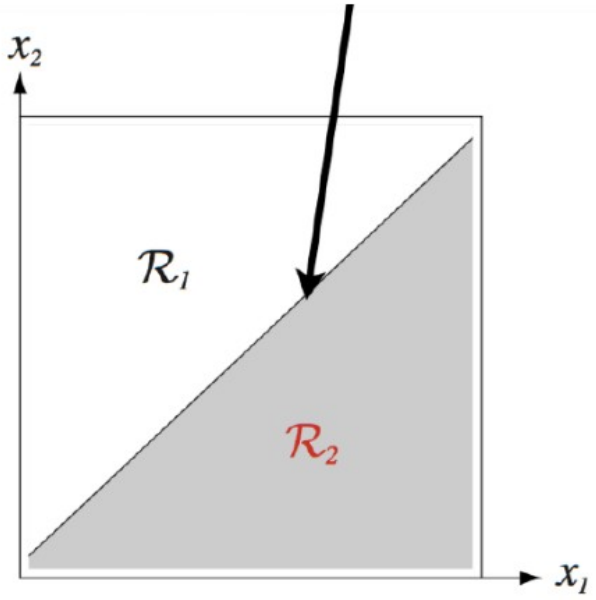
Hidden layers

Ouput layer

Single neuron

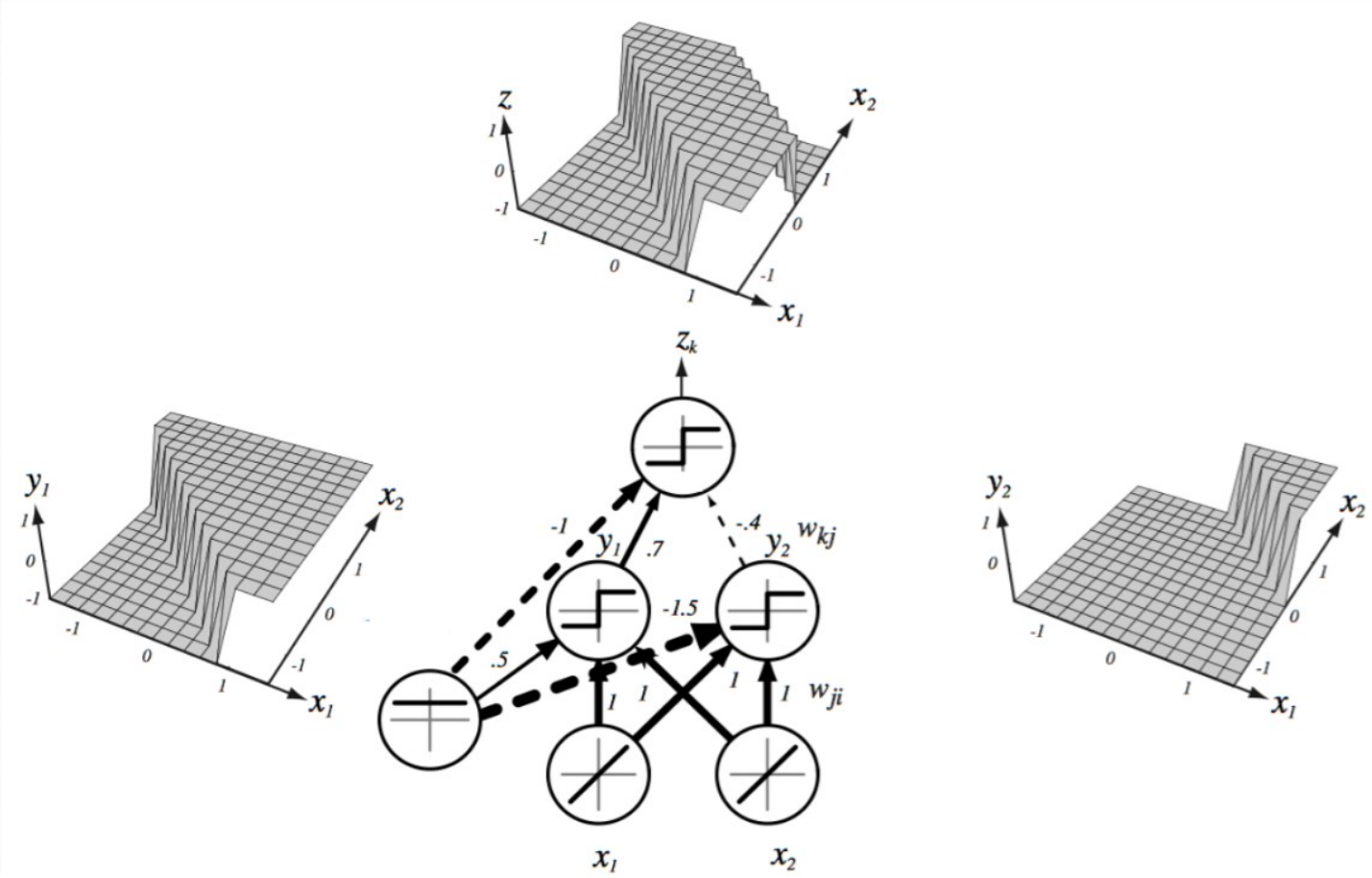
Linear separation





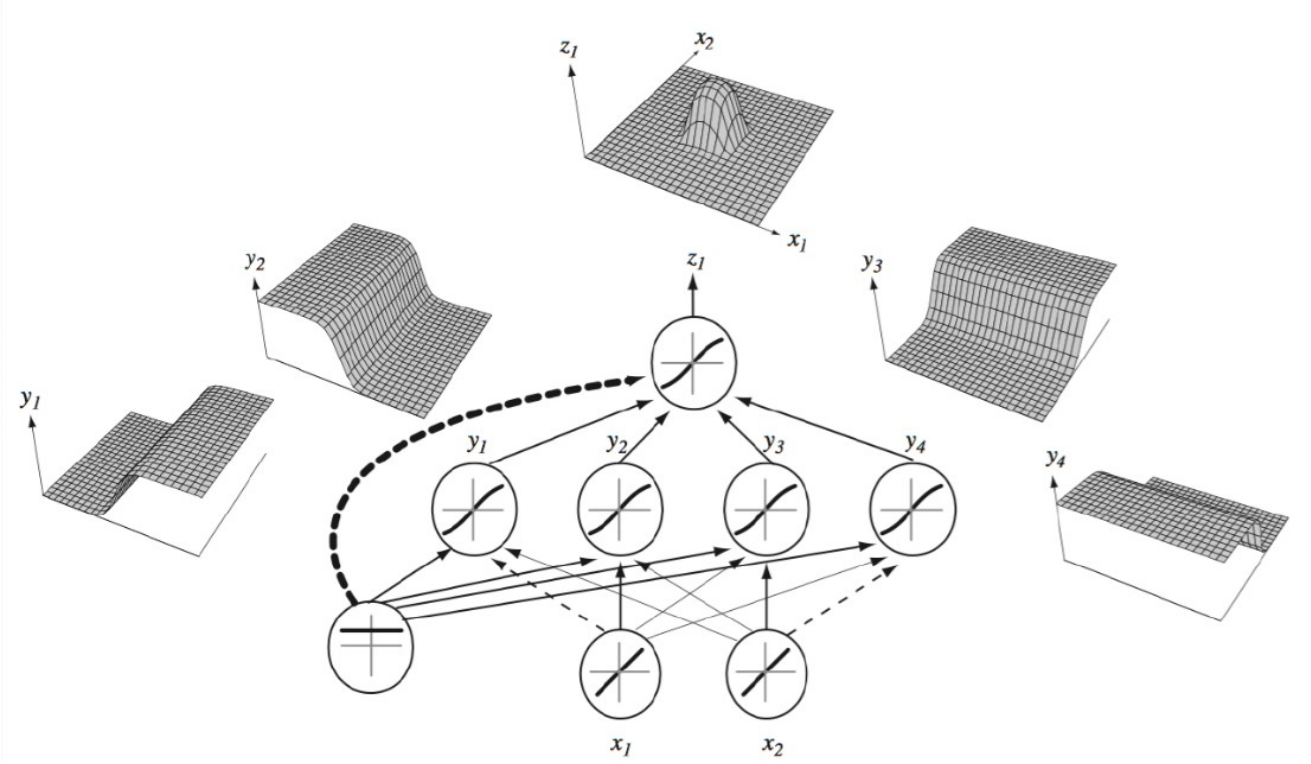
Slide Credit : Vincent Barra

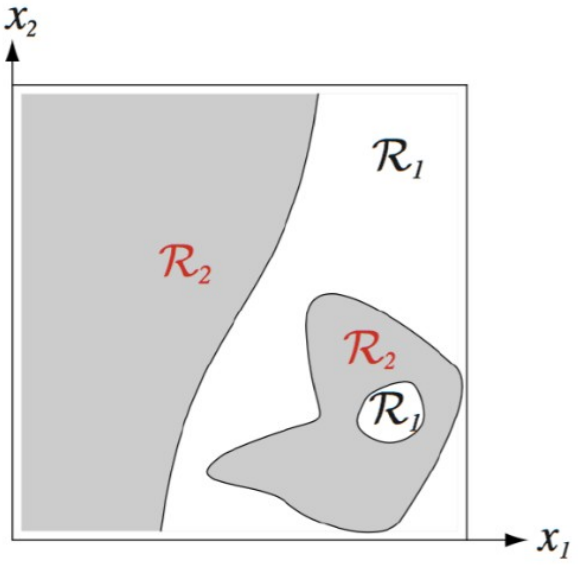
2 neurons hidden layer



Slide Credit : Vincent Barra

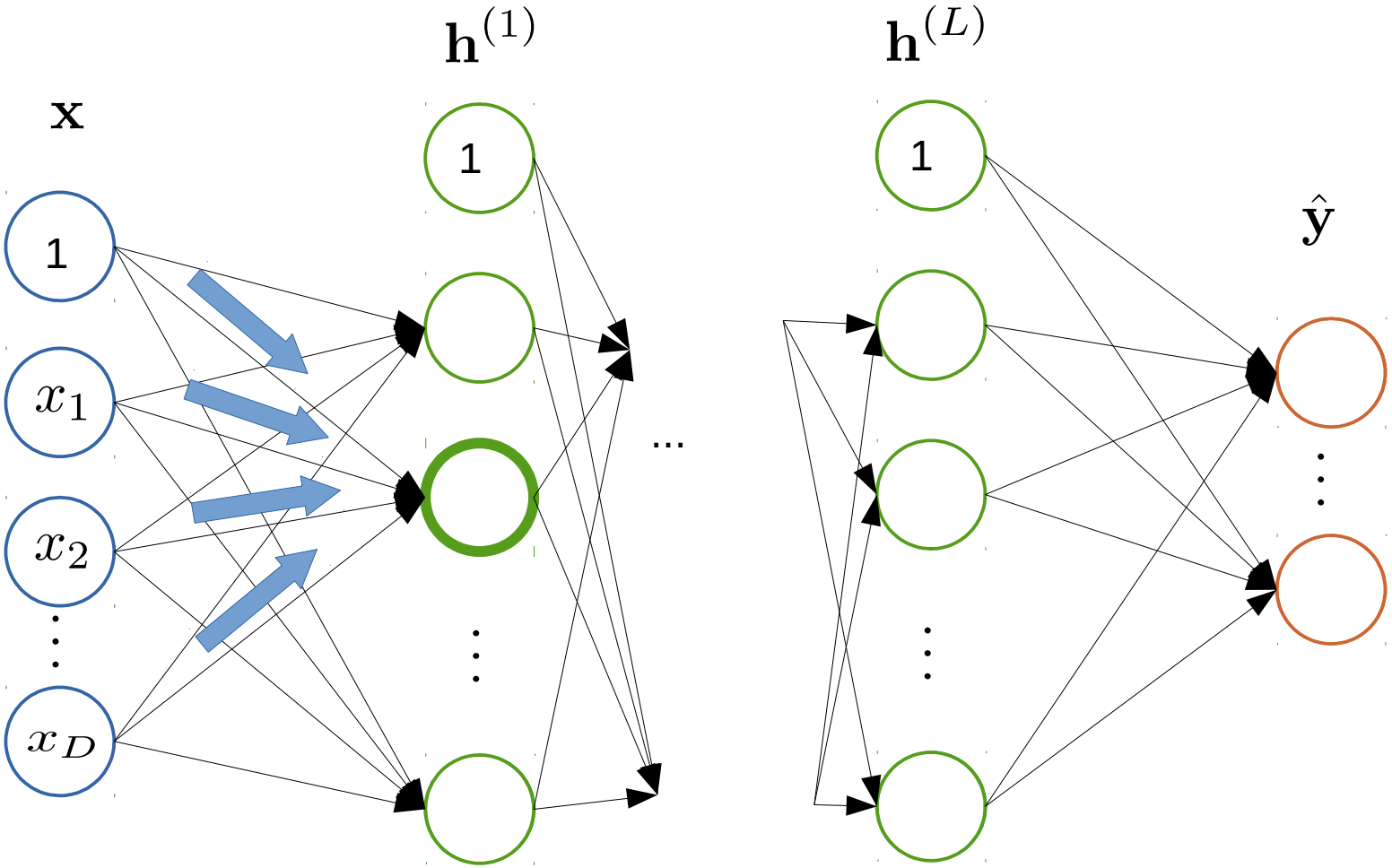
4 neurons hidden layer





Slide Credit : Vincent Barra

Multi-layer feedforward neural networks



$$h_2^{(1)} = f\left(\mathbf{w}_2^{(1)\top}\mathbf{x}\right)$$

Multi-layer feedforward neural networks

Universal approximation theorem (Hornik 1989, Cybenko 1989):

For every continuous function, there is a neural network with one hidden layer which can approximate this function as accurately as needed.

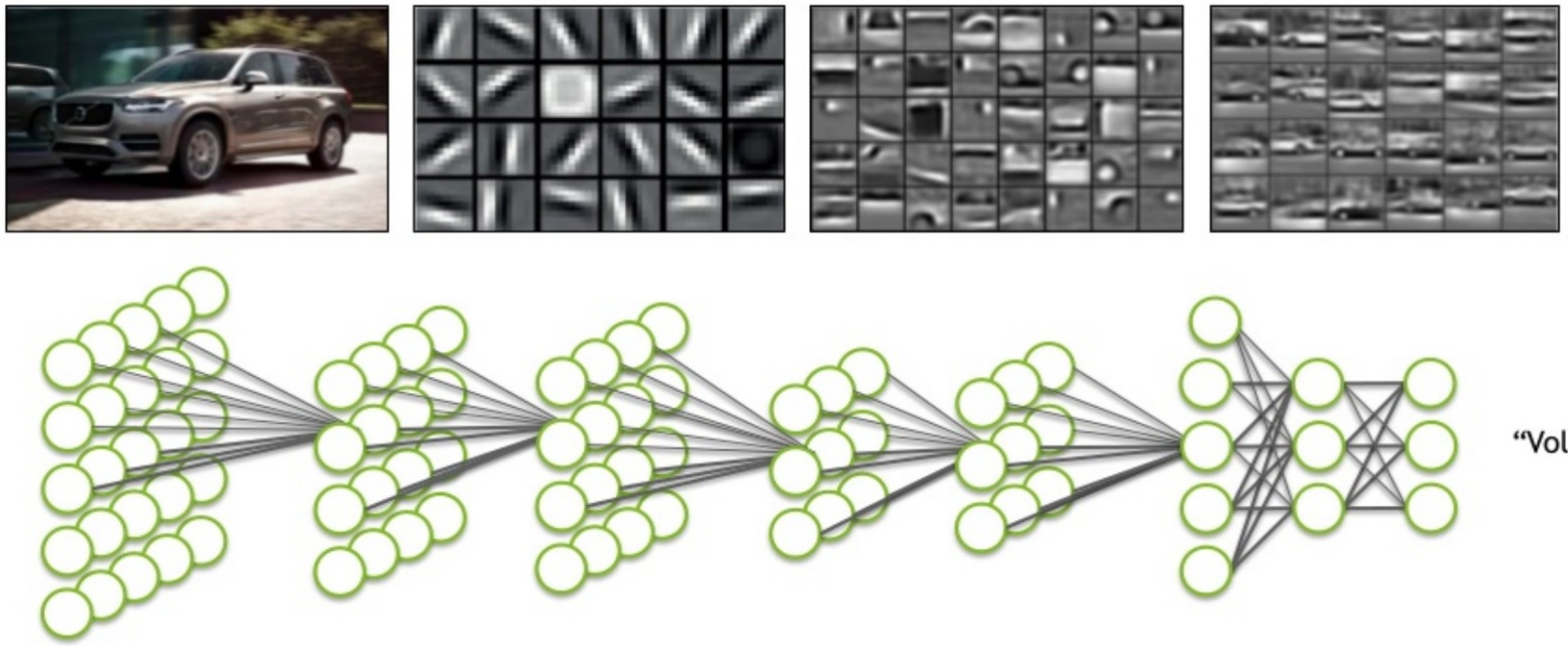
BUT

It does not say anything about the number of neurons of the hidden layer nor on the possibility to train it

We say a neural network is deep when it has at least several hidden layers.

Multi-layer feedforward neural networks

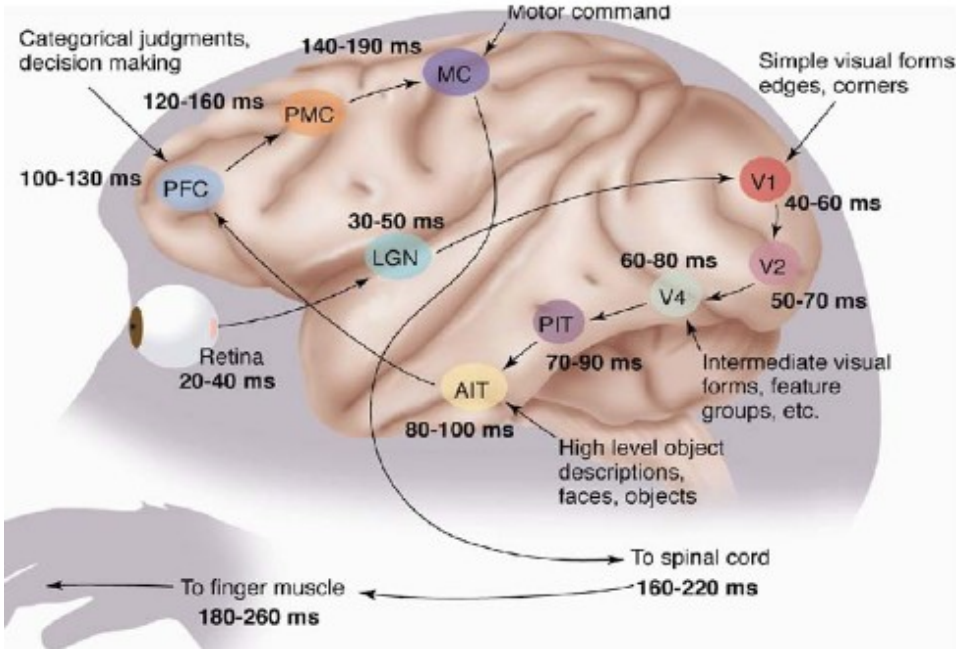
The use of several layers allows to extract a hierarchical representation of information



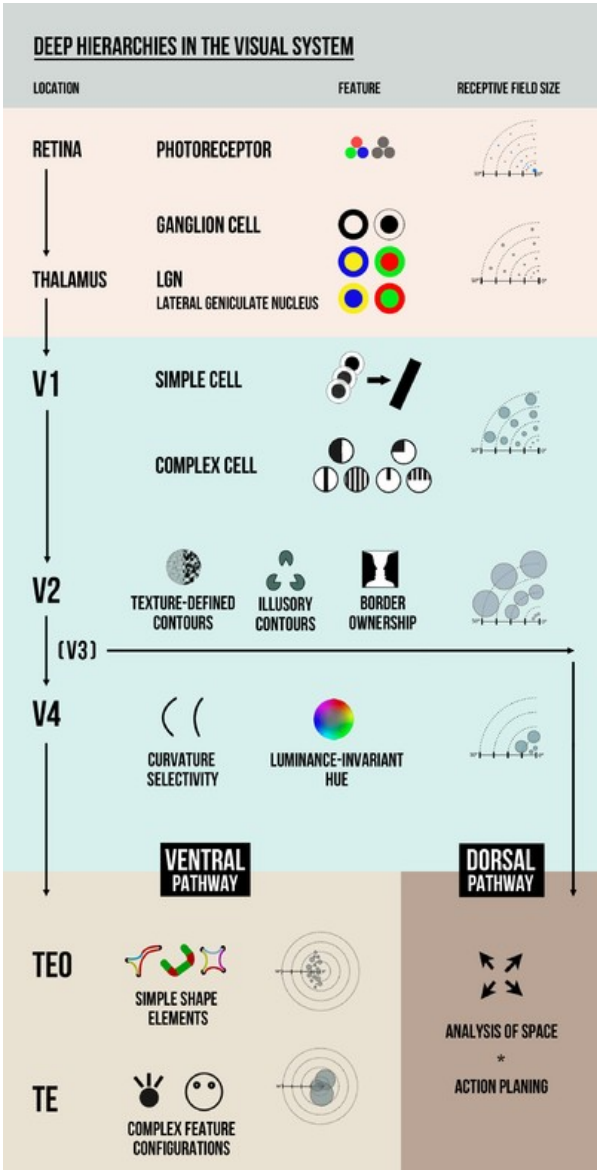
[Lee, ICML 2009]

Deep learning: hierarchical data representation

Deep learning allows to learn a hierarchical representation of data.



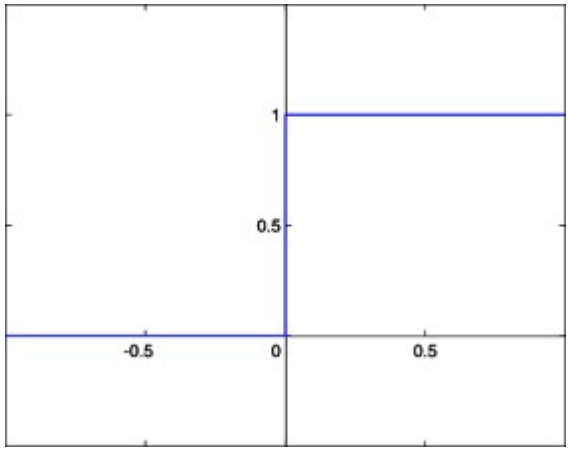
10^{11} neurones



Kruger et al, 2012

Activation function

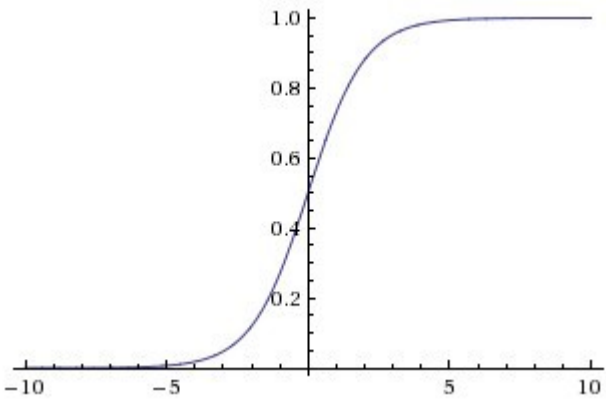
In the brain, neurons are activated (high spiking frequency) when the signal overpasses a certain threshold. The activation function used in the original perceptron is a simple step function:



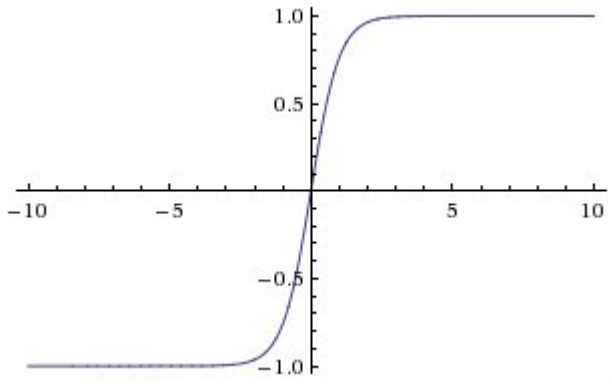
Problem: the gradient is zero almost everywhere

Activation functions

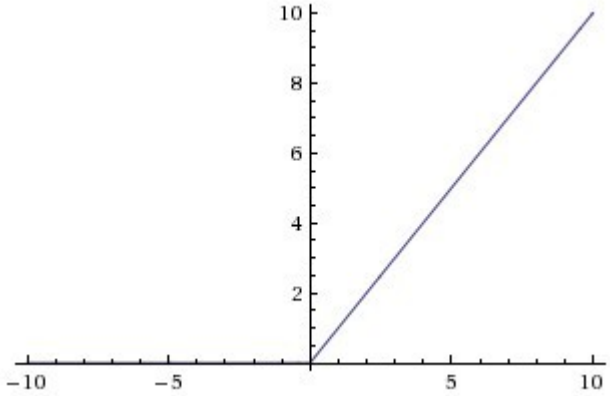
Examples of activation functions



Sigmoïd



Tanh



ReLU

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

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

$$f(z) = max(0, z)$$

Note: If activation functions are linear, then an equivalent network with one single hidden layer can be found...

How to learn network weights?

We look for the parameters that minimize a cost function expressed on all the data samples of the database

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} l\left(f(\mathbf{x}^{(i)}, \theta), \mathbf{y}^{(i)}\right)$$

l: « loss function »
n: number of samples

Optimization problem

With regularization : $$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} l\left(f(\mathbf{x}^{(i)}, \theta), \mathbf{y}^{(i)}\right) + \lambda R(\theta)$$

Stochastic gradient descent

Optimization problem remarks: $J(\theta) = \frac{1}{n}\sum_{i=1}^{n} l\left(f(\mathbf{x}^{(i)}, \theta), \mathbf{y}^{(i)}\right) + \lambda R(\theta)$

- The training set has to be very big to have generalization properties
- The cost function is expressed on all the dataset whoch makes it costly to compute when the dataset is big.
- The idea of SGD is to compute an approximation of the gradient from a single example or a subset of examples (mini-batch) and make a descent step in the direction of this approximate gradient. We then iterate has in standard gradient descent.

For each example $(\mathbf{x}^{(t)}, \mathbf{y}^{(t)})$

$$\Delta = \nabla_\theta l\left(f(\mathbf{x}^{(t)}, \theta), \mathbf{y}^{(t)}\right) + \lambda \nabla_\theta R(\theta)$$

$$\theta := \theta - \alpha\Delta$$

Stochastic gradient descent

To apply SGD, we then need :

- A model (number of neurons, choice of activation function)
- A cost function (with regularization)
- A weight initialization
- The computation of gradients

Training strategies

To apply SGD, we then need :

- A model (number of neurons, choice of **activation function**)
- **A cost function** (with regularization)
- A weight initialization
- The computation of gradients

Loss functions

Sample error for sample $(\mathbf{x}^{(t)}, \mathbf{y}^{(t)})$

$$l\left(f(\mathbf{x}^{(t)}, \theta), \mathbf{y}^{(t)}\right)$$

Examples of loss functions:

$$l\left(f(\mathbf{x}^{(t)}, \theta), \mathbf{y}^{(t)}\right) = |f(\mathbf{x}^{(t)}, \theta) - \mathbf{y}^{(t)}|$$

$$l\left(f(\mathbf{x}^{(t)}, \theta), \mathbf{y}^{(t)}\right) = \|f(\mathbf{x}^{(t)}, \theta) - \mathbf{y}^{(t)}\|_2^2$$

Case of classification applications where the network outputs the probability to belong to each class: we want to maximize the probability P(y=k|x)
Practically we minimize the negative log-likelihood -log(P(y|x))

$$l\left(f(\mathbf{x}^{(t)}, \theta), \mathbf{y}^{(t)}\right) = -log\left(f(\mathbf{x}^{(t)}, \theta)_y\right)$$

Output layer activation

- Linear neuron: outputs an unbounded real value (ex : regression case)
- Sigmoid: The output is set between 0 and 1 as a probability of belonging to a particular class

In the case of multi-class classification, we want to estimate a probability that an input belongs to each of the classes

We use softmax function:

$$\sigma(\mathbf{z})_k = \hat{y}_k = \frac{e^{z_k}}{\sum_j e^{z_j}}$$

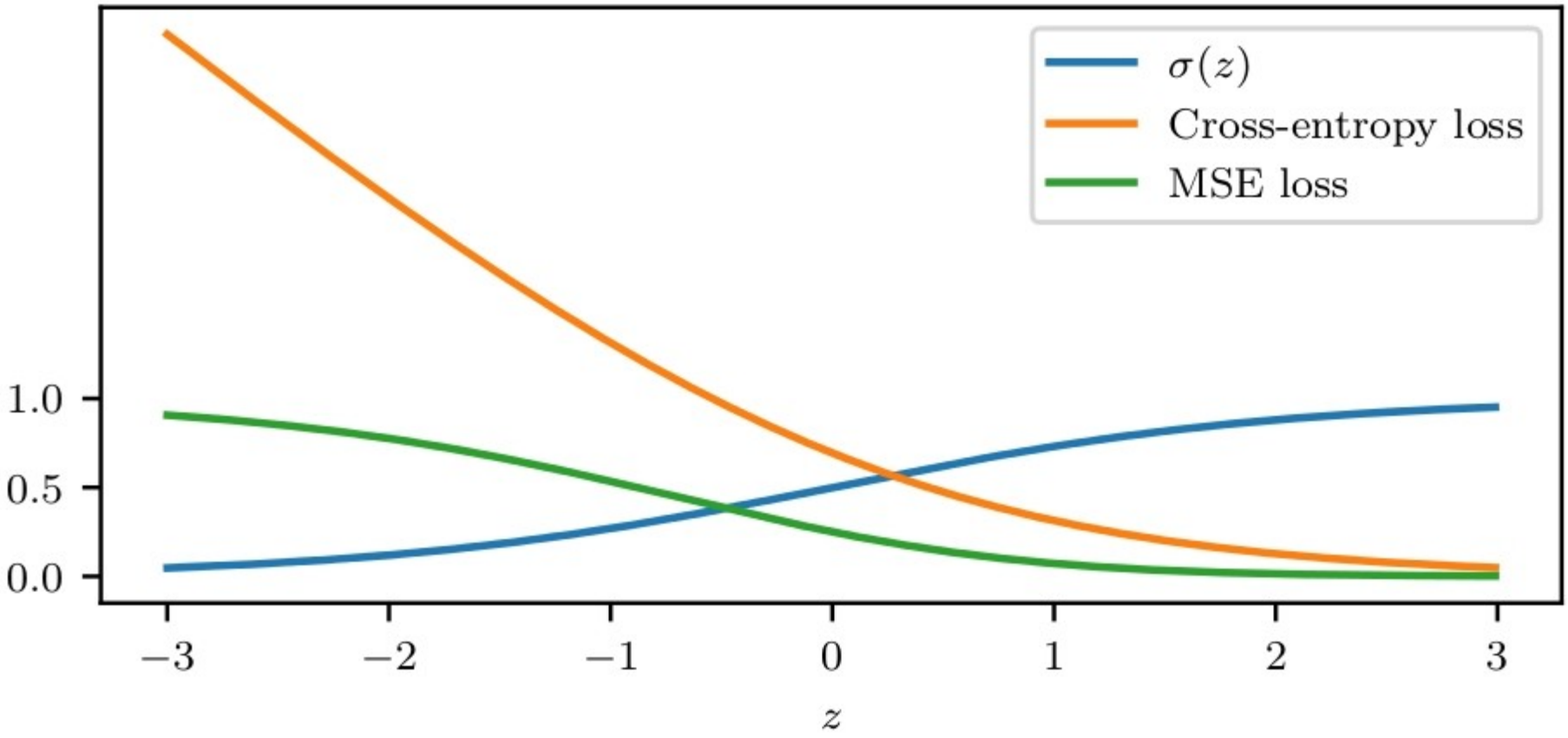
Choice of output activations and cost functions

The cost is often formalized as the maximisation of the log-likelihood between a target distribution and the likelihood estimated by the network, hence the cost function:

$$J(\theta) = -\mathbb{E}\, log(p(\mathbf{y}|\mathbf{x}))$$

Problem	**Output**	**Output activation**	**Loss function**
Regression	Real	Linear	MSE : $\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
Classification	Binary	Sigmoid : $\sigma(z) = \frac{1}{1+e^{-z}}$	Binary cross entropy : $-(y\ log(\hat{y} + (1-y)log(1-\hat{y}))$
Classification	One label, several classes	Softmax : $f(z) = \frac{exp(z)}{\sum_i exp(z_i)}$	cross entropy : $-\sum_i^M y_i\ log(\hat{y}_i)$
Classification	Several labels, several classes	Sigmoid : $\sigma(z) = \frac{1}{1+e^{-z}}$	Binary cross entropy : $-\sum_i^M (y_i\ log(\hat{y}_i + (1-y_i)log(1-\hat{y}_i))$

Choice of output activations and cost functions



Example with a dessired output of 1

Goodfellow 2017

Training strategies

To apply SGD, we then need :

- A model (number of neurons, choice of activation function)
- A cost function (with regularization)
- **A weight initialization**
- The computation of gradients

Weight initialization

Weights cannot all be equal, nor too big or too small.

Xavier method:

Normal or uniform law with zero mean and the standard deviation of a layer being:

$$\sigma = \sqrt{\frac{2}{n_{input} + n_{output}}}$$

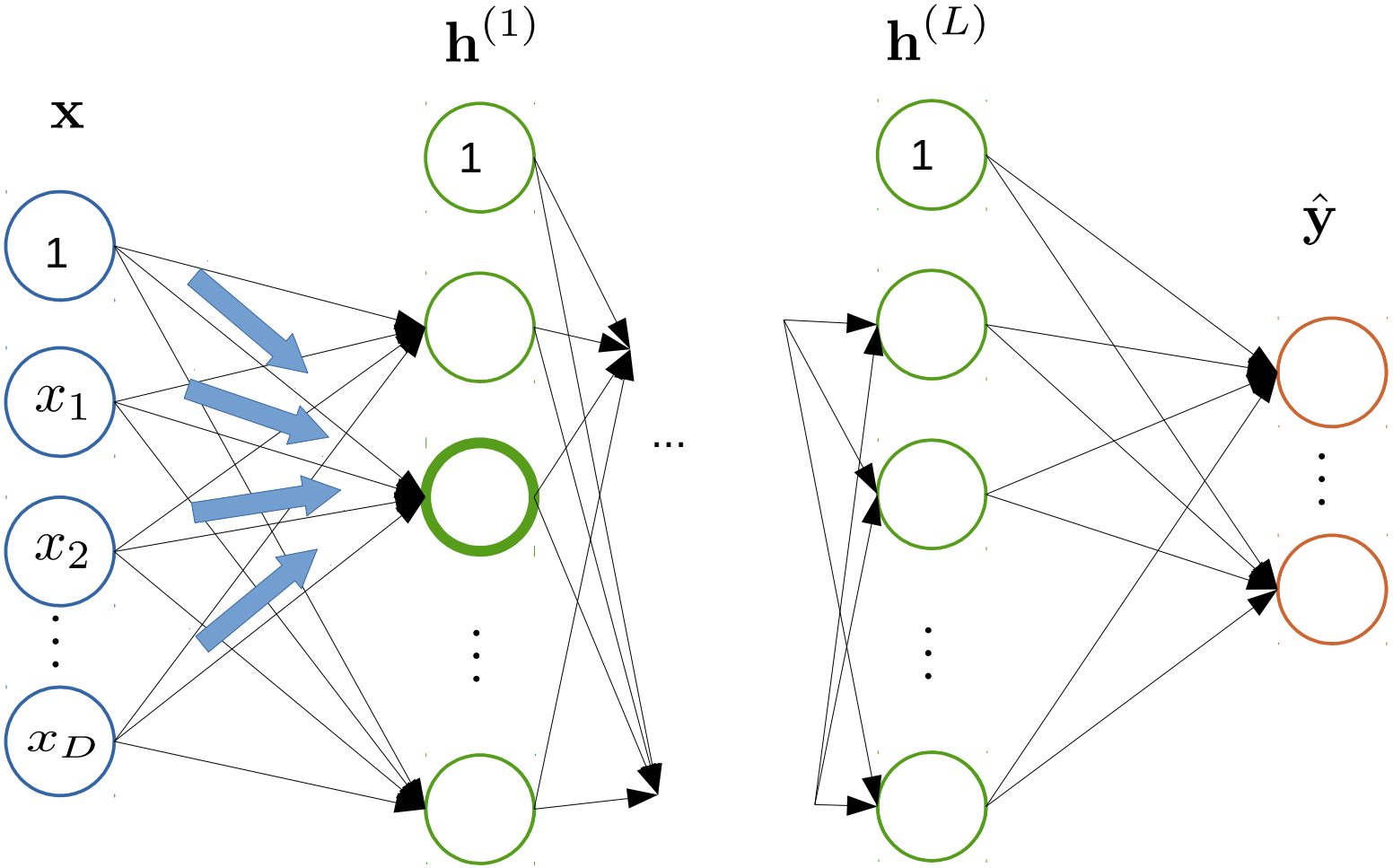
Understanding the difficulty of training deep feedforward neural networks, Gloriot et Bengio, 2010

Training strategies

To apply SGD, we then need :

- A model (number of neurons, choice of activation function)
- A cost function (with regularization)
- A weight initialization
- **The computation of gradients**

Multi-layer feedforward neural networks



$$h_2^{(1)} = f\left(\mathbf{w}_2^{(1)\top}\mathbf{x}\right)$$

Multi-layer feedforward neural networks

Example: multi-layer classification

D input neurons (+ bias)
K output neurons
$(\mathbf{x}^{(t)}, \mathbf{y}^{(t)})$ A learning example with the associated label

$\mathbf{x}^{(t)} = (x_1^t, x_2^t, ..., x_D^t)^\top$

$\mathbf{y}^{(t)} = (y_1^t, y_2^t, ..., y_K^t)^\top$

$y_k^t = 1$ if $\mathbf{x}^{(t)}$ belongs to class k

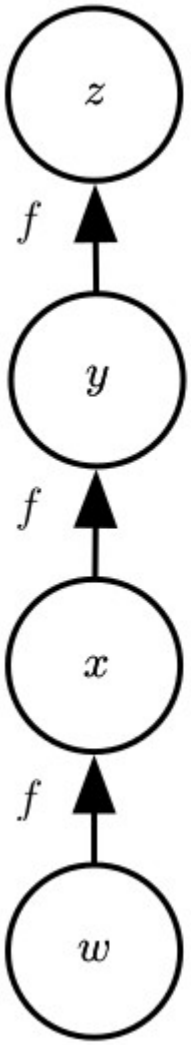
$y_k^t = 0$ otherwise

For a given layer $w_{j,i}$ corresponds to the weight of the connection between the neuron j and the neuron i of the previous layer

The output of neuron j is: $f\left(\sum_{i=1}^{R} h_i w_{j,i} + w_{j,0}\right)$

with R = number of neurons of the previous layer

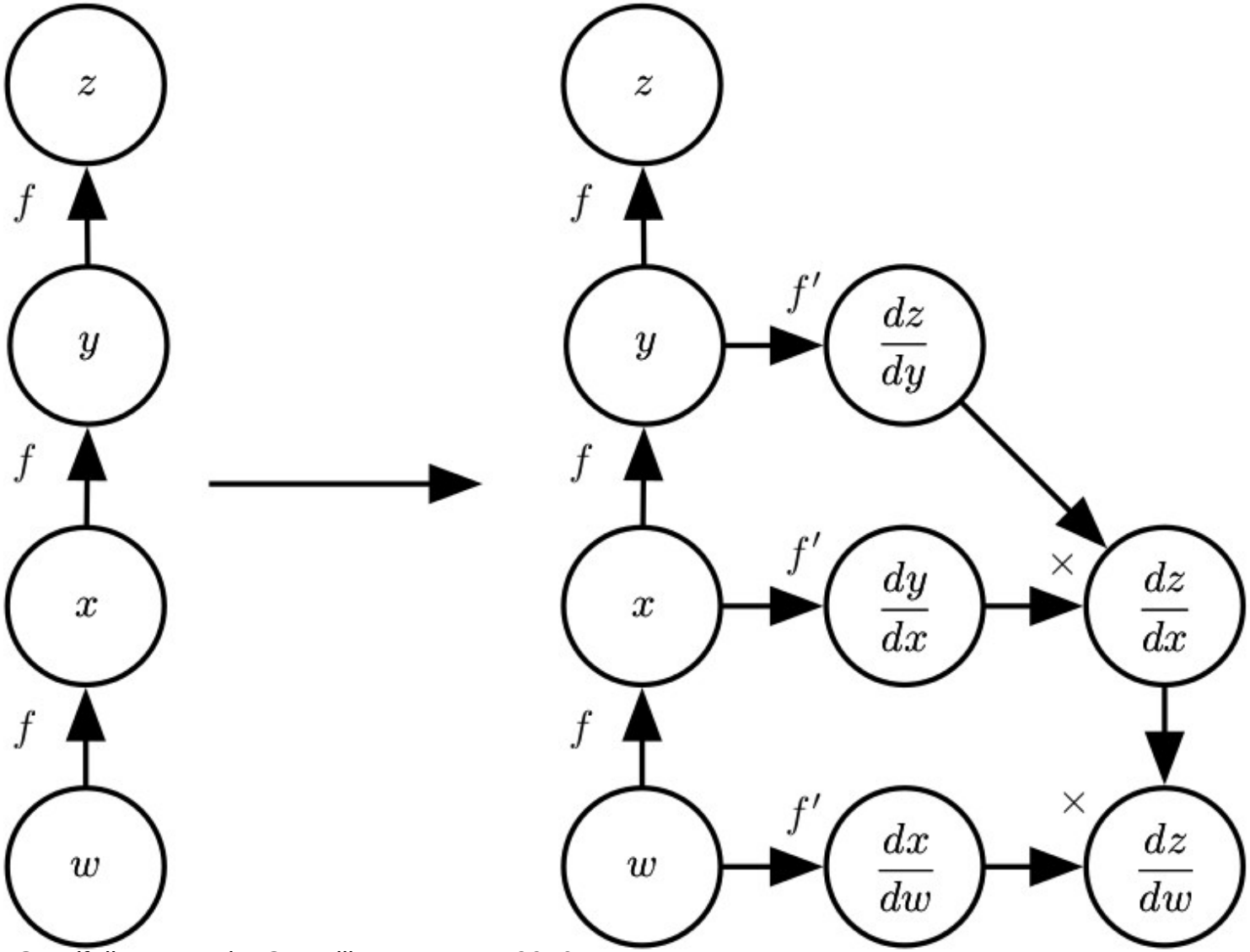
Chain rule for gradient



$$\frac{\partial z}{\partial w}$$
$$= \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$
$$= f'(y)f'(x)f'(w)$$
$$= f'(f(f(w)))f'(f(w))f'(w)$$

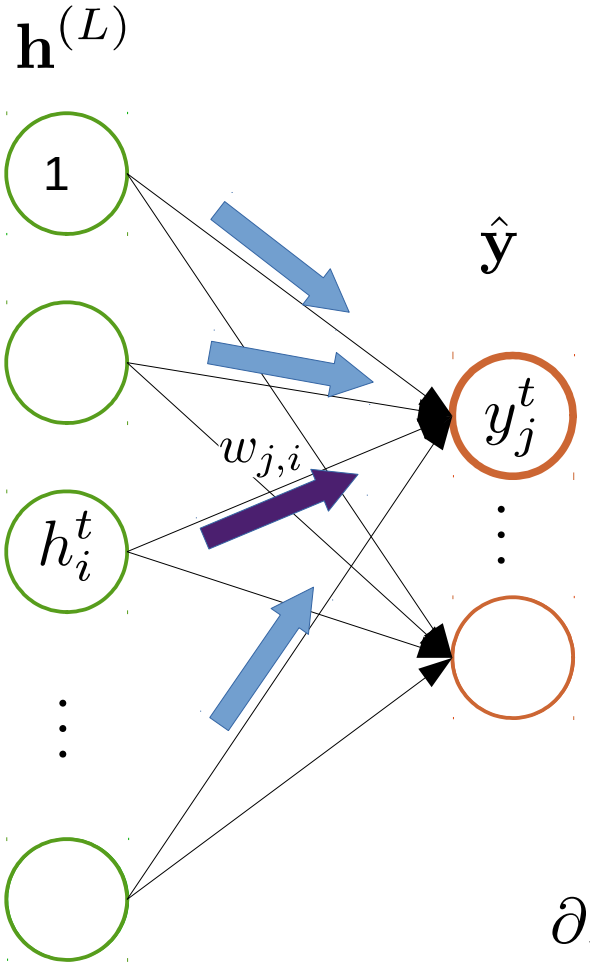
Deep Learning, Goodfellow, Bengio, Courville, MIT Press 2016

Chain rule for gradient



Deep Learning, Goodfellow, Bengio, Courville, MIT Press 2016

Gradient computation



Example:

- Sigmoid activation function
- Quadratic cost function

$$E^t = \frac{1}{2}\|\mathbf{y}^{(t)} - \mathbf{y}^{*(t)}\|_2^2 = \frac{1}{2}\sum_k \left(e_k^t\right)^2$$

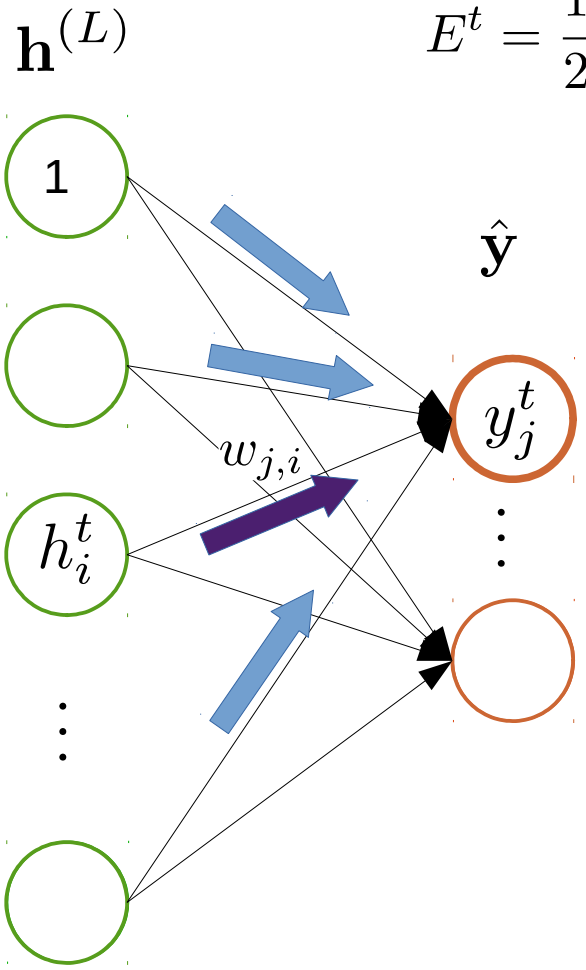
$$e_j^t = y^{*t}_j - y_j^t$$

$$y_j^t = \sigma(z_j^t)$$

$$z_j^t = \sum_{i=1}^{R} h_i^t w_{j,i} + w_{j,0}$$

$$\frac{\partial E^t}{\partial w_{j,i}} = \frac{\partial E^t}{\partial e_j^t}\frac{\partial e_j^t}{\partial y_j^t}\frac{\partial y_j^t}{\partial z_j^t}\frac{\partial z_j^t}{\partial w_{j,i}}$$

Gradient computation



$$E^t = \frac{1}{2}\|\mathbf{y}^{(t)} - \mathbf{y}^{*(t)}\|_2^2 = \frac{1}{2}\sum_k \left(e_k^t\right)^2 \quad \Rightarrow \quad \frac{\partial E^t}{\partial e_j^t} = ?$$

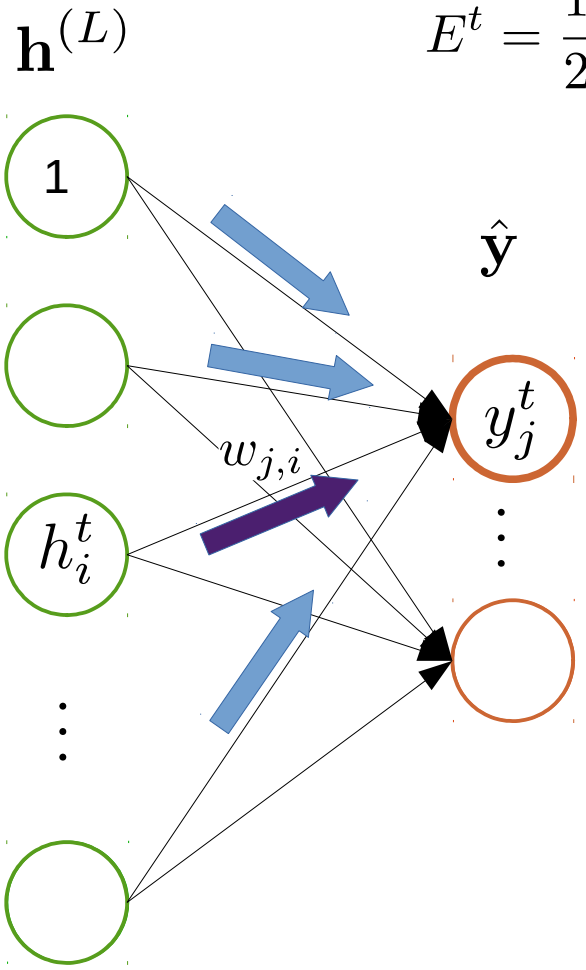
$$e_j^t = y_j^{*t} - y_j^t \quad \Rightarrow \quad \frac{\partial e_j^t}{\partial y_j^t} = ?$$

$$y_j^t = \sigma(z_j^t) = \frac{1}{1 + e^{-z_j^t}}$$

$$\Rightarrow \quad \frac{\partial y_j^t}{\partial z_j^t} = ?$$

$$z_j^t = \sum_{l=1}^{R} h_l^t w_{j,l} + w_{j,0} \quad \Rightarrow \quad \frac{\partial z^t}{\partial w_{j,i}} = ?$$

Gradient computation



$$E^t = \frac{1}{2}\|\mathbf{y}^{(t)} - \mathbf{y}^{*(t)}\|_2^2 = \frac{1}{2}\sum_j \left(e_j^t\right)^2 \quad \Rightarrow \quad \frac{\partial E^t}{\partial e_j^t} = e_j^t$$

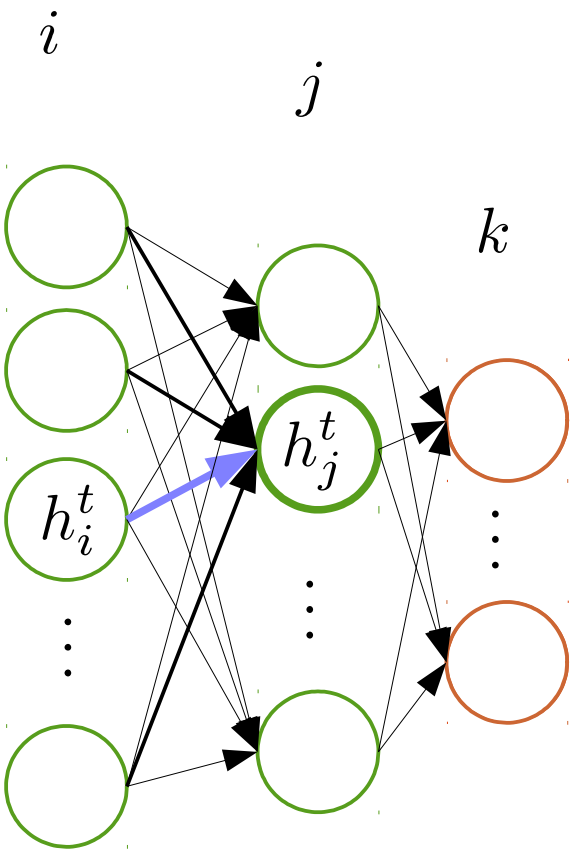
$$e_j^t = y_j^{*t} - y_j^t \quad \Rightarrow \quad \frac{\partial e_j^t}{\partial y_j^t} = -1$$

$$y_j^t = \sigma(z_j^t) = \frac{1}{1+e^{-z_j^t}}$$

$$\Rightarrow \quad \frac{\partial y_j^t}{\partial z_j^t} = y_j^t(1-y_j^t)$$

$$z_j^t = \sum_{l=1}^{R} h_l^t w_{j,l} + w_{j,0} \quad \Rightarrow \quad \frac{\partial z^t}{\partial w_{j,i}} = h_i^t$$

Hidden layer backpropagation



$$\frac{\partial E^t}{\partial w_{j,i}} = \frac{\partial E^t}{\partial h_j^t} \frac{\partial h_j^t}{\partial z_j^t} \frac{\partial z_j^t}{\partial w_{j,i}}$$

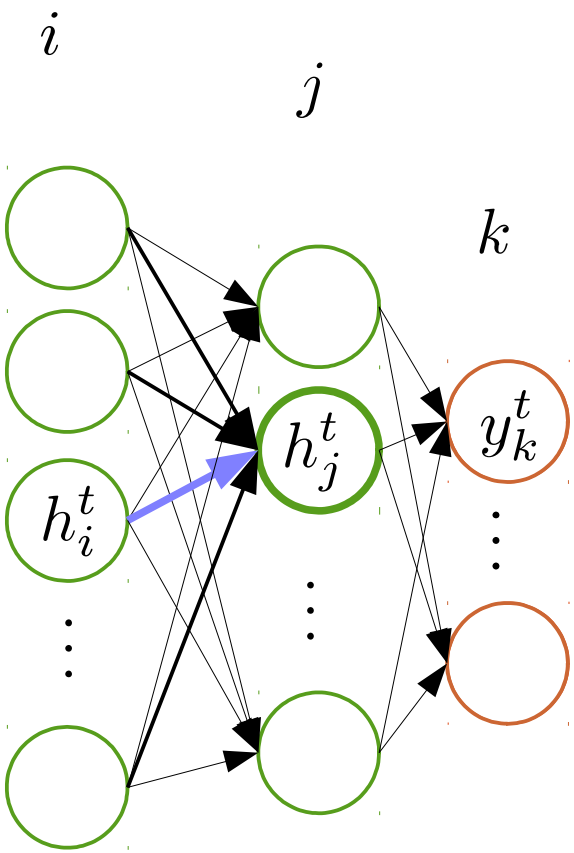
$$\frac{\partial E^t}{\partial h_j^t} = \frac{\partial}{\partial h_j^t} \frac{1}{2} \sum_k \left(e_k^t\right)^2 = \sum_k e_k^t \frac{\partial e_k^t}{\partial h_j^t}$$

$$\frac{\partial E^t}{\partial h_j^t} = \sum_k e_k^t \frac{\partial e_k^t}{\partial z_k^t} \frac{\partial z_k^t}{\partial h_j^t}$$

$$\frac{\partial E^t}{\partial h_j^t} = \sum_k e_k^t \frac{\partial (y_k^{*t} - y_k^t)}{\partial z_k^t} \frac{\partial (\sum_l w_{k,l} h_l^t + w_{l,0})}{\partial h_j^t}$$

$$\frac{\partial E^t}{\partial h_j^t} = \sum_k e_k^t \left(-y_k^t (1 - y_k^t)\right) w_{k,j}$$

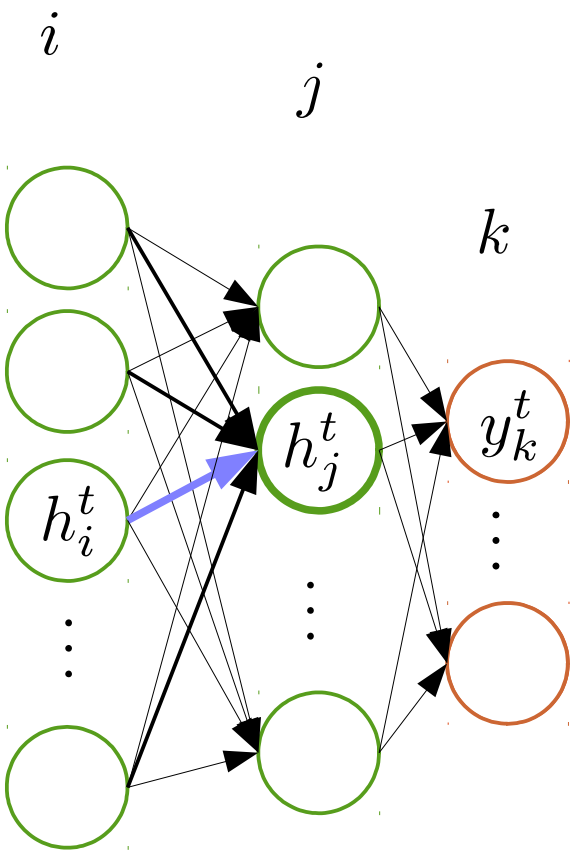
Hidden layer backpropagation



$$\frac{\partial E^t}{\partial w_{j,i}} = \frac{\partial E^t}{\partial h_j^t} \frac{\partial h_j^t}{\partial z_j^t} \frac{\partial z_j^t}{\partial w_{j,i}}$$

$$\begin{cases} \dfrac{\partial E^t}{\partial h_j^t} & = -\sum_k e_k^t \left(y_k^t(1-y_k^t)\right) w_{k,j} \\ \dfrac{\partial h_j^t}{\partial z_j^t} & = h_j^t(1-h_j^t) \\ \dfrac{\partial z_j^t}{\partial w_{j,i}} & = h_i^t \end{cases}$$

Hidden layer backpropagation



$$\frac{\partial E^t}{\partial w_{j,i}} = \frac{\partial E^t}{\partial h_j^t}\frac{\partial h_j^t}{\partial z_j^t}\frac{\partial z_j^t}{\partial w_{j,i}}$$

$$\begin{cases} \frac{\partial E^t}{\partial h_j^t} & = -\sum_k e_k^t \left(y_k^t(1-y_k^t)\right) w_{k,j} \\ \frac{\partial h_j^t}{\partial z_j^t} & = h_j^t(1-h_j^t) \\ \frac{\partial z_j^t}{\partial w_{j,i}} & = h_i^t \end{cases}$$

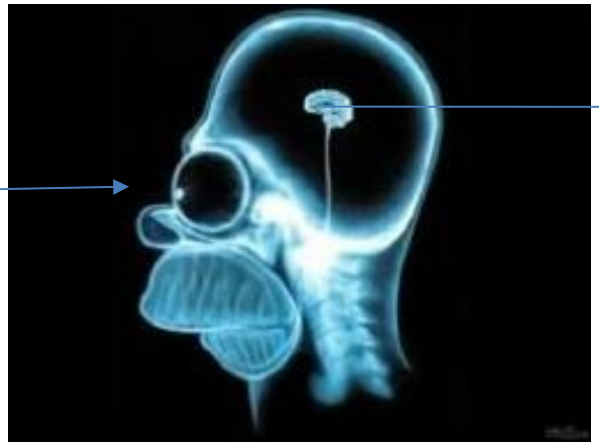
$$-\Delta w_{j,i} = \delta_j^t h_i^t$$

$$\delta_j^t = h_j^{(t)}(1-h_j^{(t)}) \sum_k \delta_k w_{k,j}$$

The gradient of a neuron is computed from the gradients of the next layer!

Weight learning

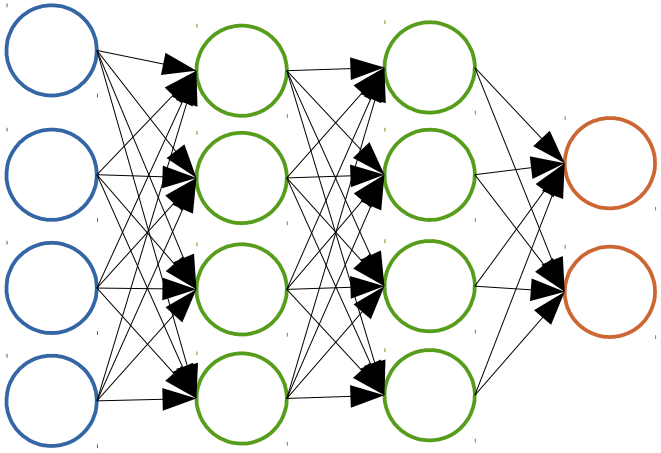
Input $\mathbf{x}$



Output $\hat{y}$

5

Class y



Weight correction through backpropagation

SGD convergence

SGD Convergence if

$$\sum_{i=1}^{\infty} \alpha_i = \infty$$

$$\sum_{i=1}^{\infty} \alpha_i^2 < \infty$$

With α_i the learning rate at step i
A constant learning rate does not guaranty convergence.
To ensure convergence on can use a decreasing learnign rate, for example:

$$\alpha_i = \frac{\alpha_i}{1 + \delta_i}$$

In practice we do not want to go all the way to convergence to avoid overfitting on the training set.
A constant rate is not necessarily a bad choice

Important remarks

- In machine learning we try to learn parameters that minimize a cost function: why is it different from an optimization problem?
  - What distinguishes machine learning from other optimization problems is that the cost that we minimize is expressed with respect to a training test but we really want to minimize the **generalization** error, on **new** data (error on the test set for example)

- How to improve the performances on a test set when we only observe the training set?
  - If the training set and the test set follow completely different distributions, there is not much to expect
  - Otherwise cross-validation can be used

Model selection

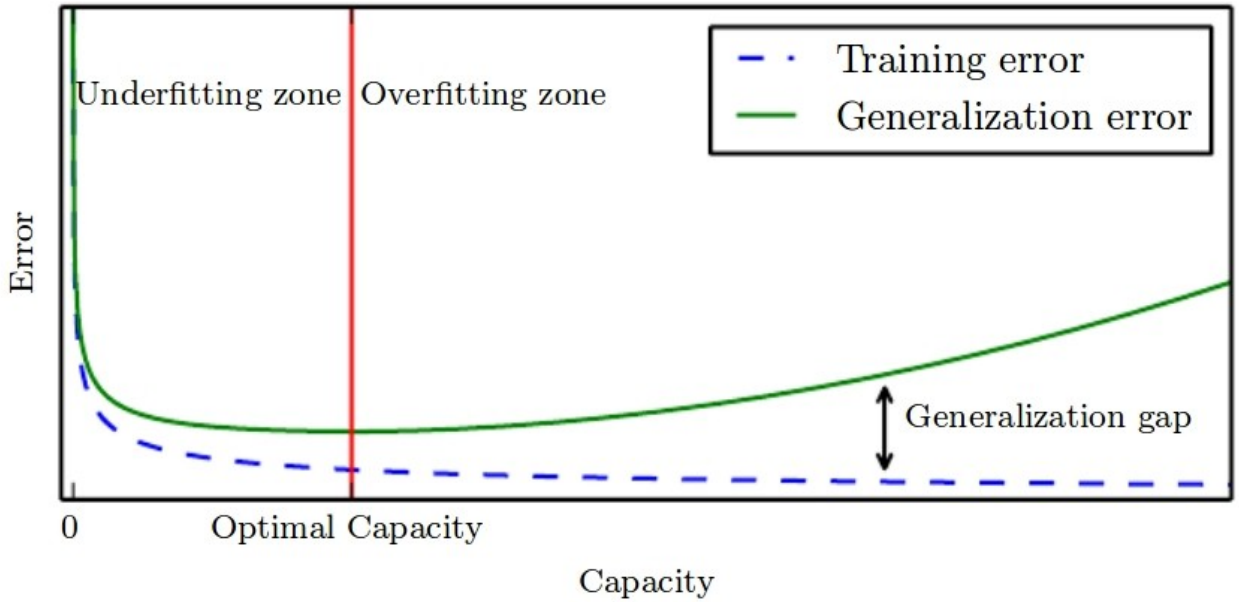
Learning strategy:

- Training set to learn a model
- Validation set to select hyperparameters
- Test set which estimates the generalization error

End of training: when the error on the validation test increases

Important remarks

- How to avoid overfitting?
  - Limit the capacity of the model (decrease the number of parameters, add constraints such as regularization,...)
  - Increase the number of samples in the dataset



Deep Learning, Goodfellow, Bengio, Courville, MIT Press 2016

Training problems

- If there are a lot of parameters, we need lots of data or we face overfitting
- Optimization of a non convex function
- « Vanishing gradient »: In the backprogagation process, less and less gradient information is transmitted to the deeper layers: weights of first layers evolve slowly.

Choice of cost function regularization (L2, L1, dropout...),and optimization method (adptative learning rate, RMSProp, Adagrad, ADAM,…) are important

Regularization

Modification of a learning algorithm so as to reduce the generalization error without changing the learning error

- Restrictions on the parameters
- Additional terms on the cost function (soft constraint on parameter values)

The real distribution of data is often extremely complex. In this context, it is very difficult to find the minimal number of parameters for a given problem. It is often simpler to take a bigger model and regularize it.

Regularization

Penalty on the norm of the parameters
In practice we regularize the weights of the network not the biases

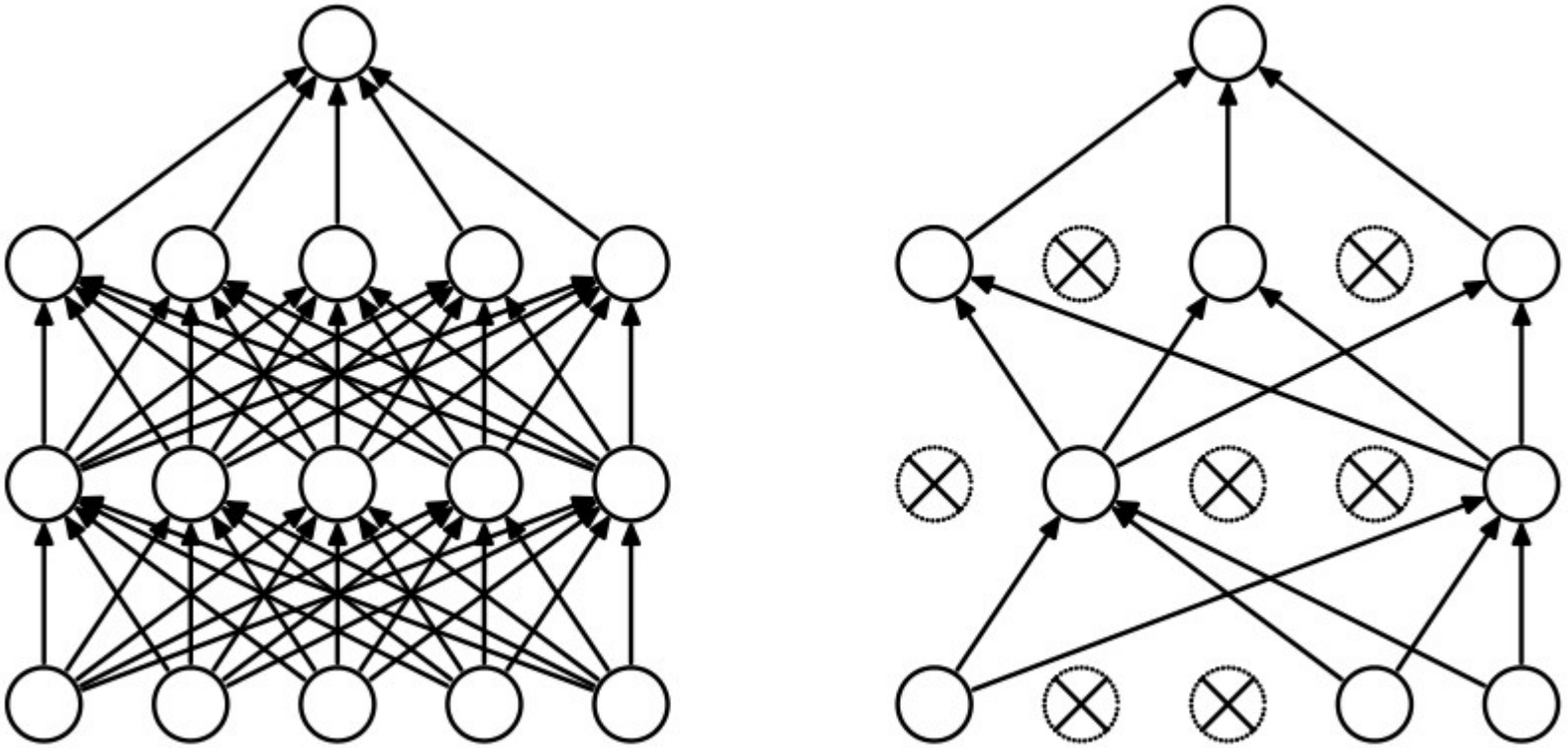
$$J(\theta) = J_{mle}(\theta) + \lambda R(\theta)$$

L2 norm: « Weight decay » : $R(\theta) = \frac{1}{2}\mathbf{w}^\top \mathbf{w}$

L1 norm: « Weight decay » : $R(\theta) = \sum_i |w_i|$

Other regularization technique

- Dropout



SGD variations

- Use of mini-batch
- « Momentum »: gradient computed using the previous gradient:

$$\Delta w = -\alpha \nabla_\theta J^{(t)}_{mle}(\theta) + \beta \Delta w$$

- AdaGrad (adaptive gradient)

$$\theta_j = \theta_j - \frac{\alpha}{\sqrt{\sum_{i=0}^{t} (\nabla_{i\theta_j} J(\theta))^2}} \nabla_\theta J(\theta)$$

  Idea: Higher learning rate for the parameters scarsely updated

- RMS Prop
  Like AdaGrad but with an exponential mean of the previous gradient values
- Adam (takes into account order 2 moments)
- Hessian computation?