

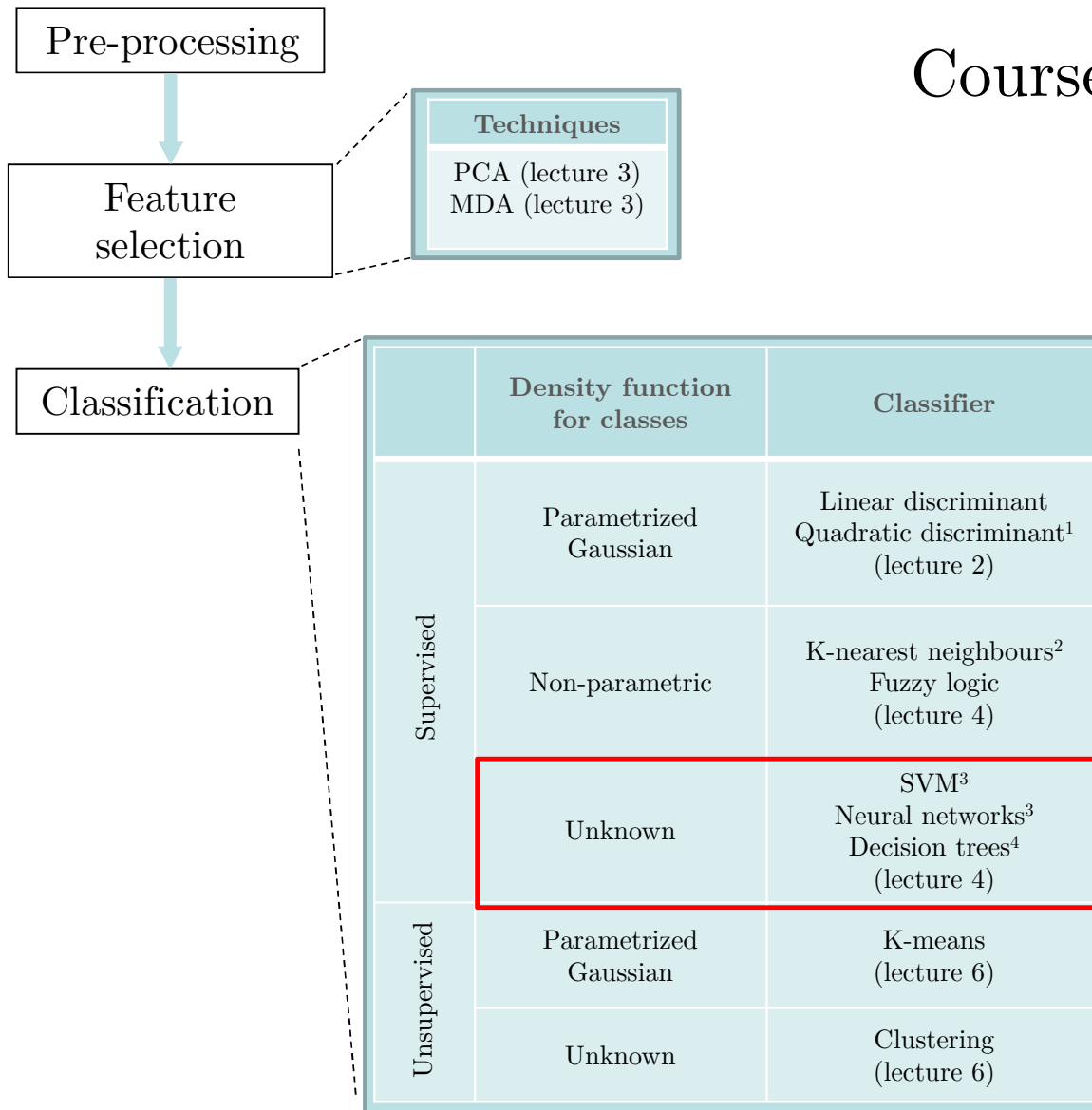
Chapter 4.3

Multilayer Neural Networks

Recommended bibliography: *I. Goodfellow et al, "Deep Learning", The MIT Press, Cambridge, MA, 2016*

Credits: Some figures are taken from *Pattern Classification (2nd ed)* by *R. O. Duda, P. E. Hart and D. G. Stork, John Wiley & Sons, 2000* with the permission of the authors

Course overview



1. Useful only if covariance matrices are not rank deficient.

2. Useful with the number of features is very large, even larger than the number of training vectors.

3. Imposes a structure to the classifier irrespective of the training data base.

4. Useful when non-numeric features are present.

INDEX

4.3 Multilayer neural networks

- 4.3.1 Logistic regression
- 4.3.2 Biological neurons
- 4.3.3 Multilayer feed-forward networks
- 4.3.4 Selection of the activation function
- 4.3.5 Train with the backpropagation algorithm
- 4.3.6 Avoiding overfitting
- 4.3.7 Accelerating convergence in gradient methods
- 4.3.8 Rules for the improvement of convergence
- 4.3.9 The power of neural networks
- 4.3.10 Other NN architectures

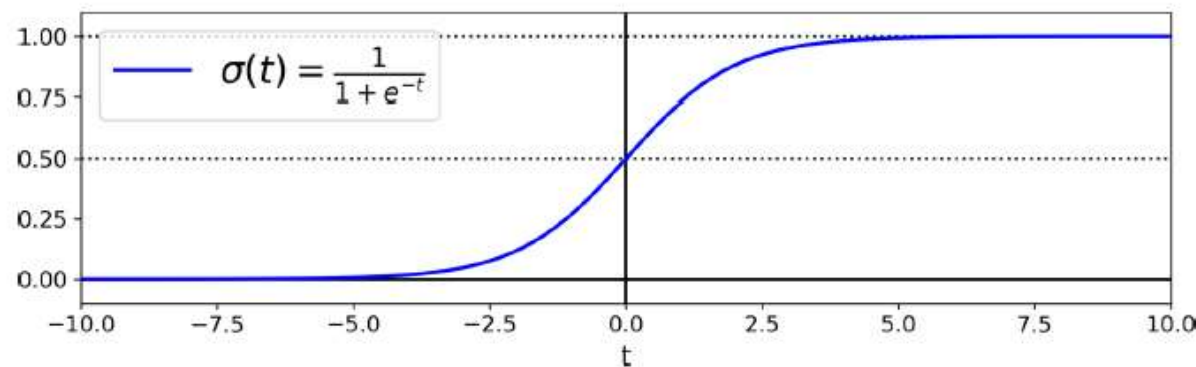
1. Logistic regression

Let us start with two classes, $c = 2$, and estimate the probability that \mathbf{x}_i belongs to a particular class $\Pr(y_i|\mathbf{x}_i)$ with $y_i \in \{0,1\}$.

Take a logistic function for the model:

$$h_{\theta}(\mathbf{x}_i) = \sigma(\mathbf{x}_i^T \boldsymbol{\theta})$$

We usually include a 1 in vector \mathbf{x}
(see why in the sequel)



Then, the classification rule is $\hat{y}_i = \begin{cases} 1 & \text{if } h_{\theta}(\mathbf{x}_i) > 0.5 \\ 0 & \text{if } h_{\theta}(\mathbf{x}_i) < 0.5 \end{cases}$

Cost functions

Training the model implies selecting θ such that $h_\theta(\cdot)$ is large for $y = 1$ and small for $y = 0$. A suitable function is the [logloss](#):

$$L(y_1, \dots, y_N, \mathbf{x}_1, \dots, \mathbf{x}_N; \theta) = -\frac{1}{N} \sum_{i=1}^N [y_i \log h_\theta(\mathbf{x}_i) + (1 - y_i) \log (1 - h_\theta(\mathbf{x}_i))]$$

which is a [convex function](#). The single minimum has no closed expression but can be obtained using gradient descent. Noticing that for the logistic function:

$$\log(1 - h_\theta(\mathbf{x})) = \mathbf{x}^T \theta + \log h_\theta(\mathbf{x})$$

$$L = -\frac{1}{N} \sum_{i=1}^N [\log h_\theta(\mathbf{x}_i) + (1 - y_i) \mathbf{x}_i^T \theta]$$

and the gradient follows easily:

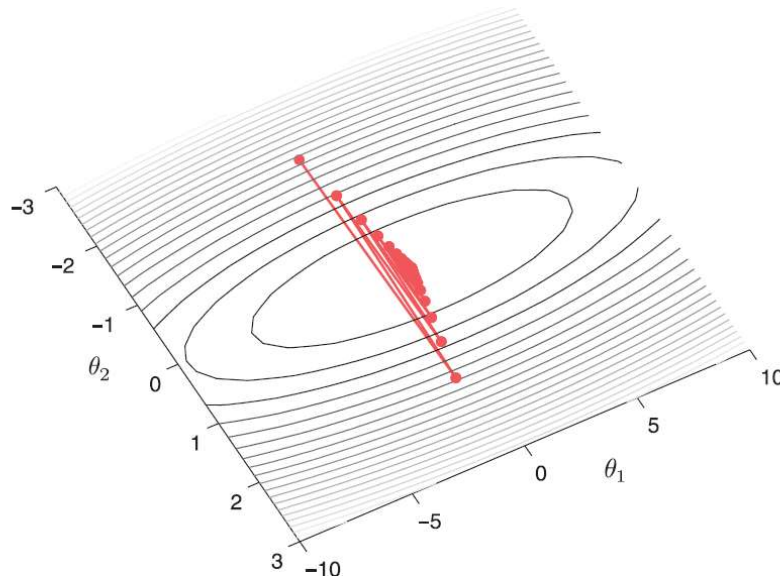
$$\nabla_\theta L = \frac{1}{N} \sum_{i=1}^N (h_\theta(\mathbf{x}_i) - y_i) \mathbf{x}_i$$

Gradient descent

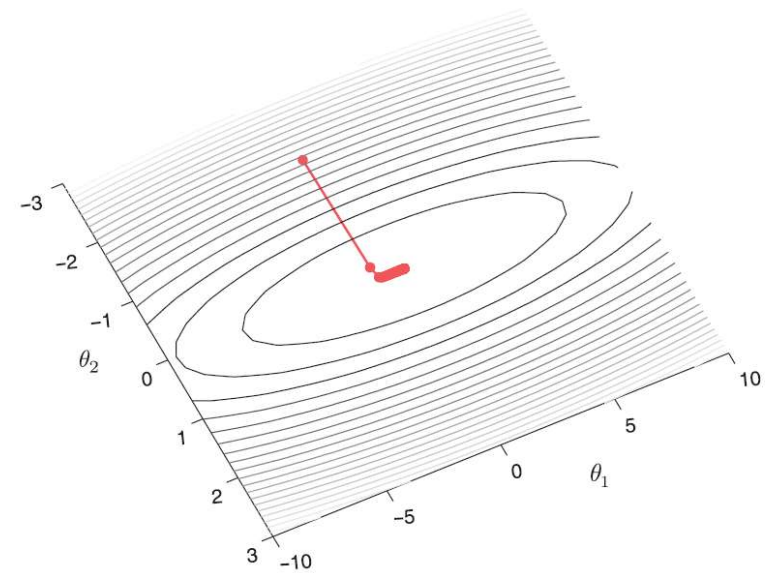
Iterative solutions by following the direction of the negative gradient of L :

$$\theta_{k+1} = \theta_k - \eta \nabla_{\theta} L|_{\theta=\theta_k}$$

Using an arbitrary value for θ_0 . The speed of convergence depends on the value of the learning rate η .



Large η



Small η

Decision boundary

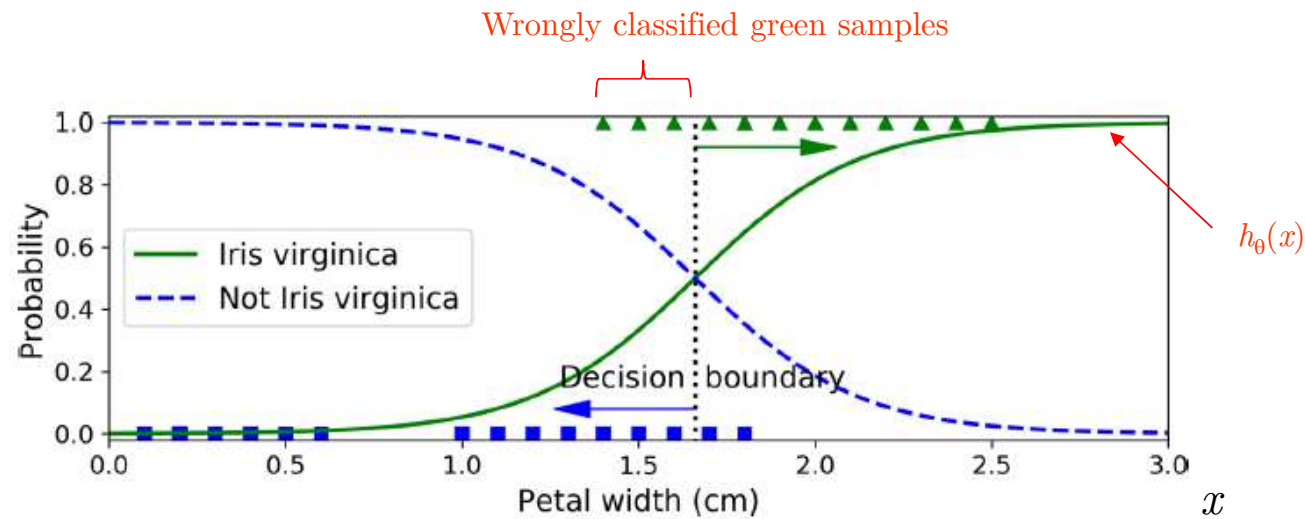
Decisions are made depending on the position of vectors $\mathbf{x} \in R^d$ on a d -dimensional space. The decision boundary is the indecisive set of \mathbf{x} , let us illustrate it.

Example. The iris dataset, where $c = 3$ and $d = 2$



Decision boundary

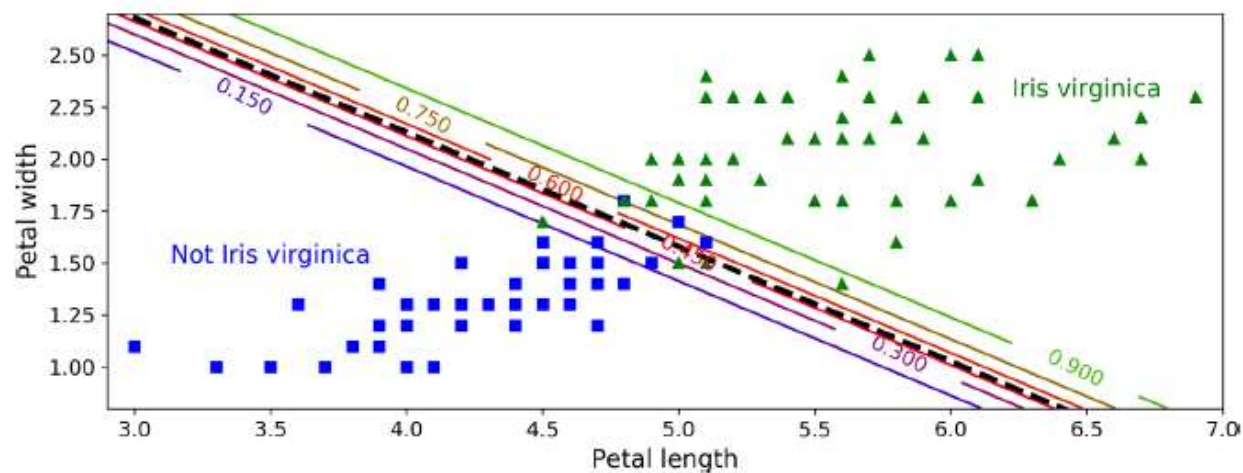
Let us take two classes {Iris Virginica, Not-Iris Virginica} and one feature, the petal width.



Decision boundary

Let us take two classes {Iris Virginica, Not-Iris Virginica} and two features, the petal width and petal length.

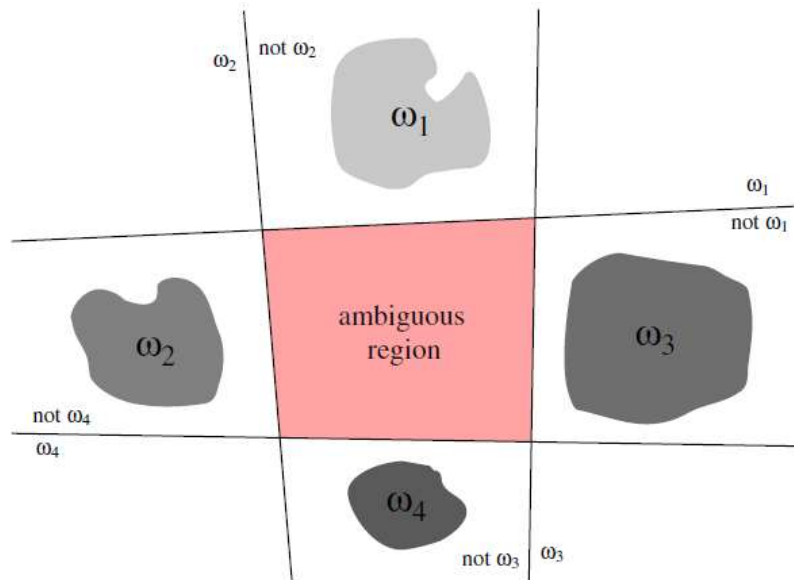
The decision boundary is linear (just solve the equation $h_{\theta}(\mathbf{x}) = 1/2$).



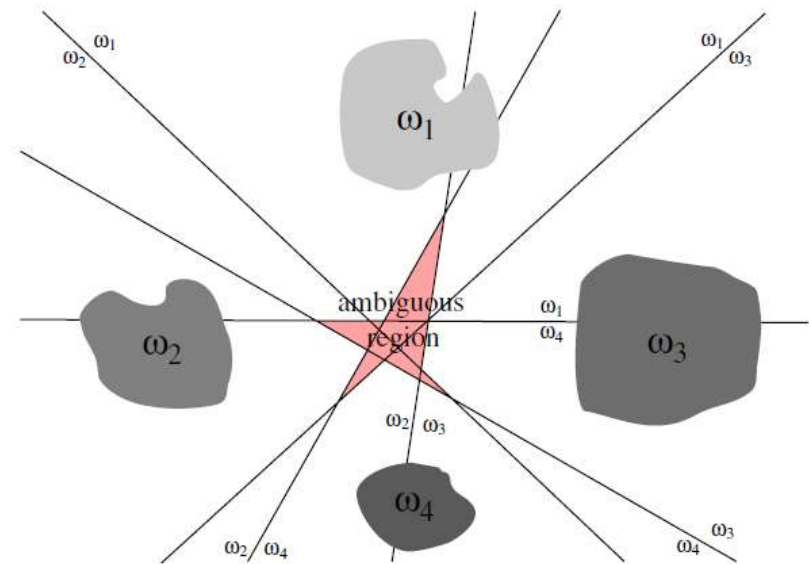
Multi-class classification

Is it possible to use logistic regression for a c -class problem?

Two straightforward solutions:



c classifiers, *one-versus-the-rest*



One classifier per each pair, *one-versus-one*

Multi-class classification

More rigourously, can we generalize logistic regression to solve a multi-class problem?

Softmax regression estimates $\Pr(y|\mathbf{x})$ with $y \in \{0, 1, \dots, c-1\}$:

$$h_k(\mathbf{x}) = \frac{\exp(\mathbf{x}^T \boldsymbol{\theta}_k)}{\sum_{j=0}^{c-1} \exp(\mathbf{x}^T \boldsymbol{\theta}_j)} \quad k = 0, \dots, c-1$$

The predicted class is obtained as

$$\hat{y} = \arg \max_k h_k(\mathbf{x}) = \arg \max_k \mathbf{x}^T \boldsymbol{\theta}_k$$

Gradient training for softmax regression

Let us generalize the **logloss** objective function and define the **cross-entropy**:

$$L(\boldsymbol{\theta}_0, \dots, \boldsymbol{\theta}_{c-1}) = -\frac{1}{N} \sum_{i=1}^N \sum_{k=0}^{c-1} y_{k,i} \log h_k(\mathbf{x}_i)$$

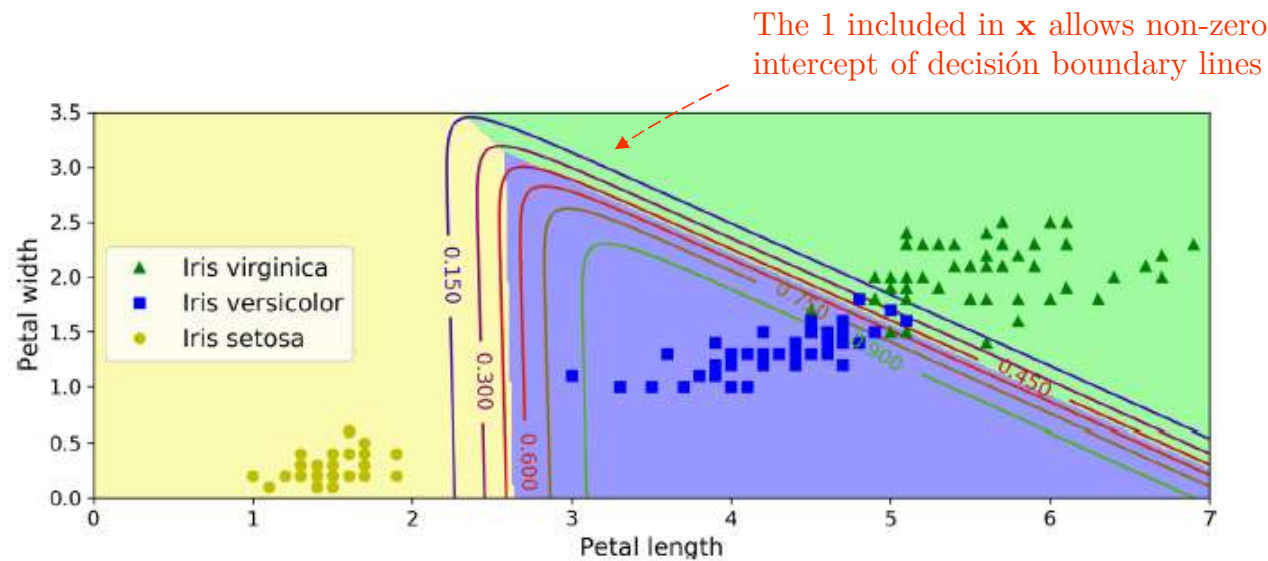
where $y_{k,i}$ is the target probability that vector \mathbf{x}_i belongs to class k . It takes values $\{0,1\}$.

The logloss is the Kullback-Leibler divergence between distributions $y_{k,i}/N$ and $h_k(\mathbf{x}_i)$, as long as $y_{k,i}$ takes $\{0,1\}$ values. It is a convex function and its gradient is:

$$\nabla_{\boldsymbol{\theta}_k} L(\boldsymbol{\theta}_0, \dots, \boldsymbol{\theta}_{c-1}) = \frac{1}{N} \sum_{i=1}^N (h_k(\mathbf{x}_i) - y_{k,i}) \mathbf{x}_i$$

Multi-class decision boundaries

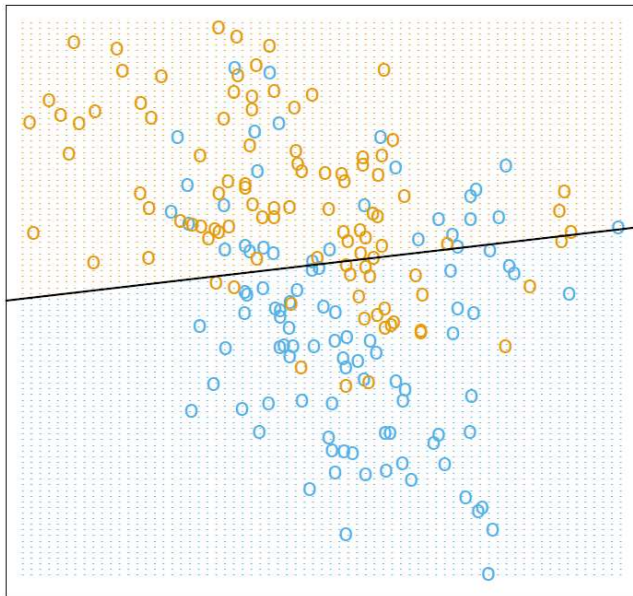
We now involve the three classes and two features. Decision boundaries are linear, we just have to solve the equation $h_k(\mathbf{x}) = h_j(\mathbf{x})$.



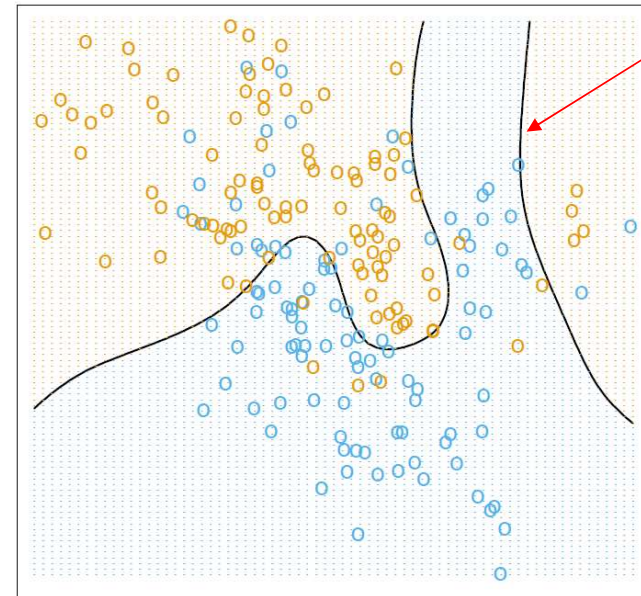
The logistic regression approach provides only piece-wise linear decision boundaries, but many classification problems require more complex solutions.

Example. A 2-classes problem, with synthetically generated samples:

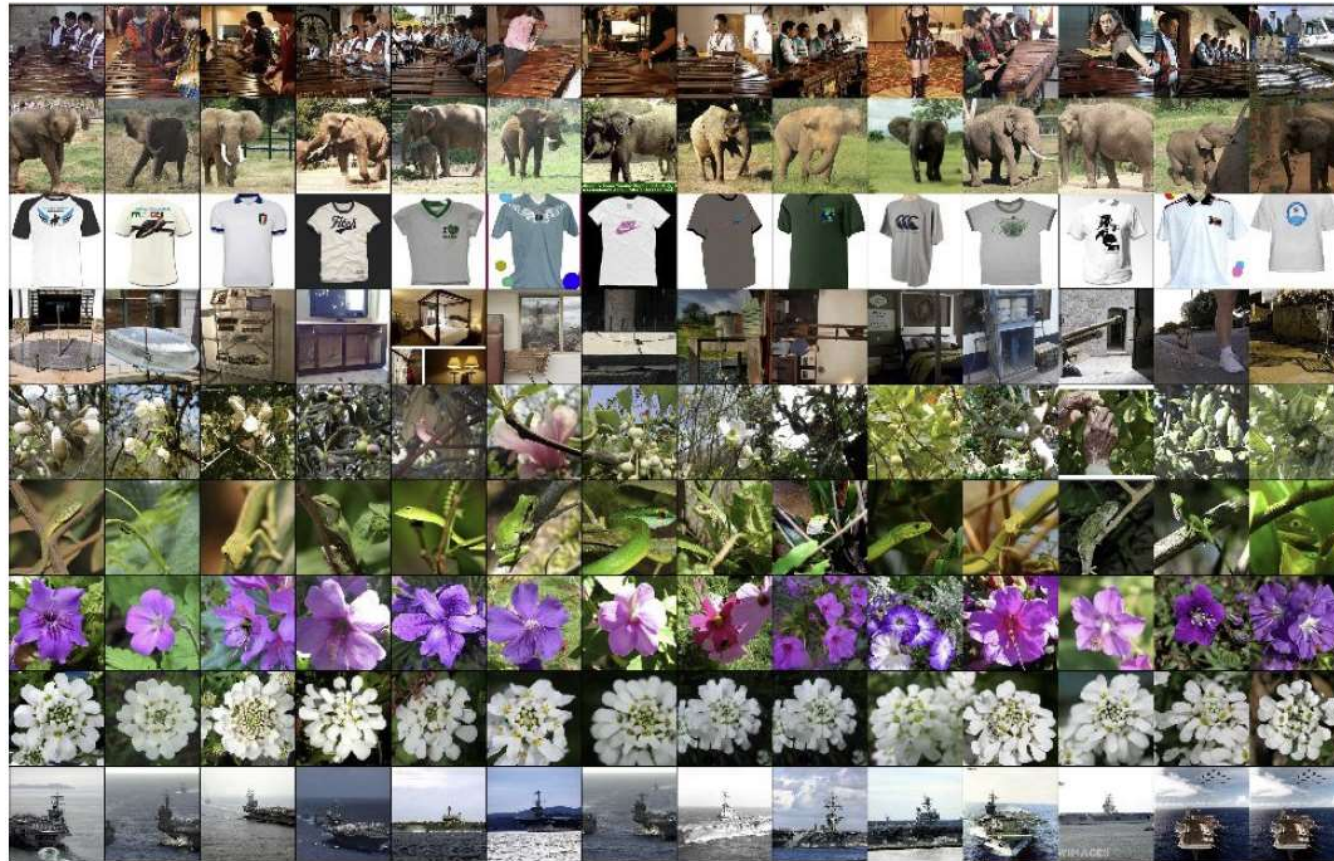
Decision boundary of
the logistic regressor



Decision boundary of
the MAP classifier



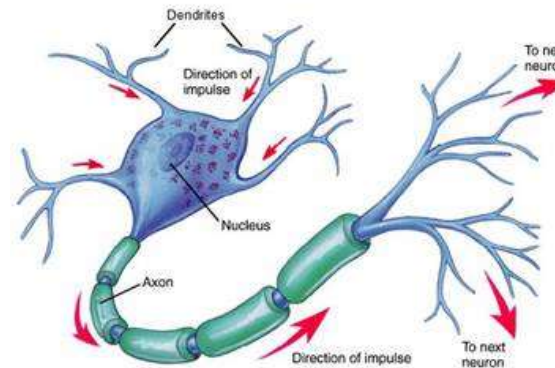
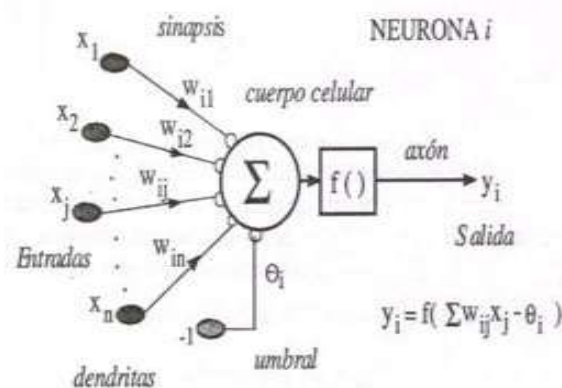
How to face complex problems like “what is in this image”?



Let us stack several logistic regressors.

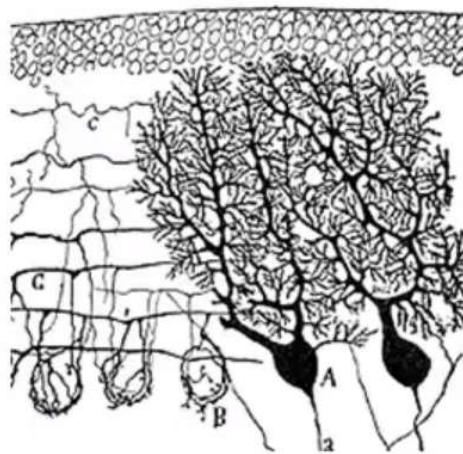
2. Biological neurons

- Artificial NN originated from the idea to **model mathematically human intellectual abilities** by biologically plausible engineering designs.
- They are meant to be **parallel computational schemes** resembling a real brain.
- **Small changes in the training data** might lead to a large change in the classifier, both in its structure and parameters.
- Modeling of the human brain, at either morphological or functional level, and trying to understand NN's cognitive capacity are also important research topics.

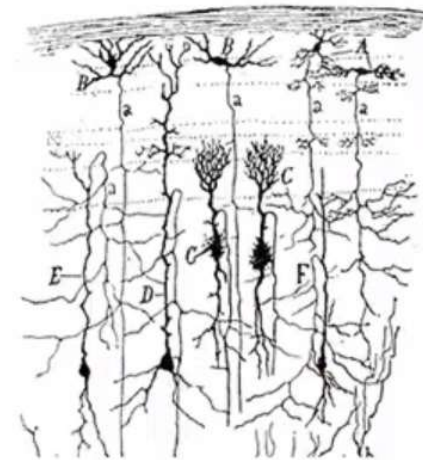




Visual Cortex

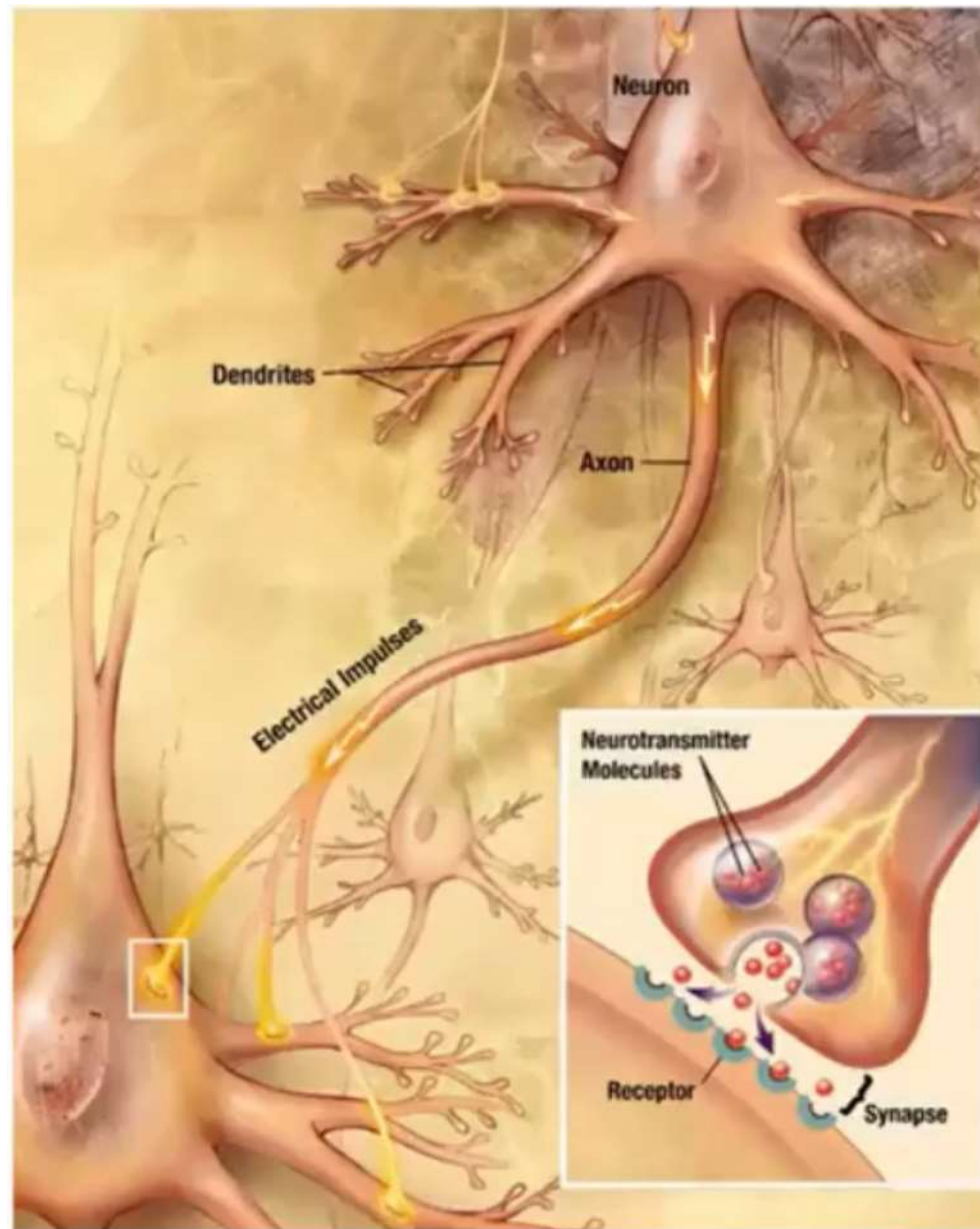


Cerebellum



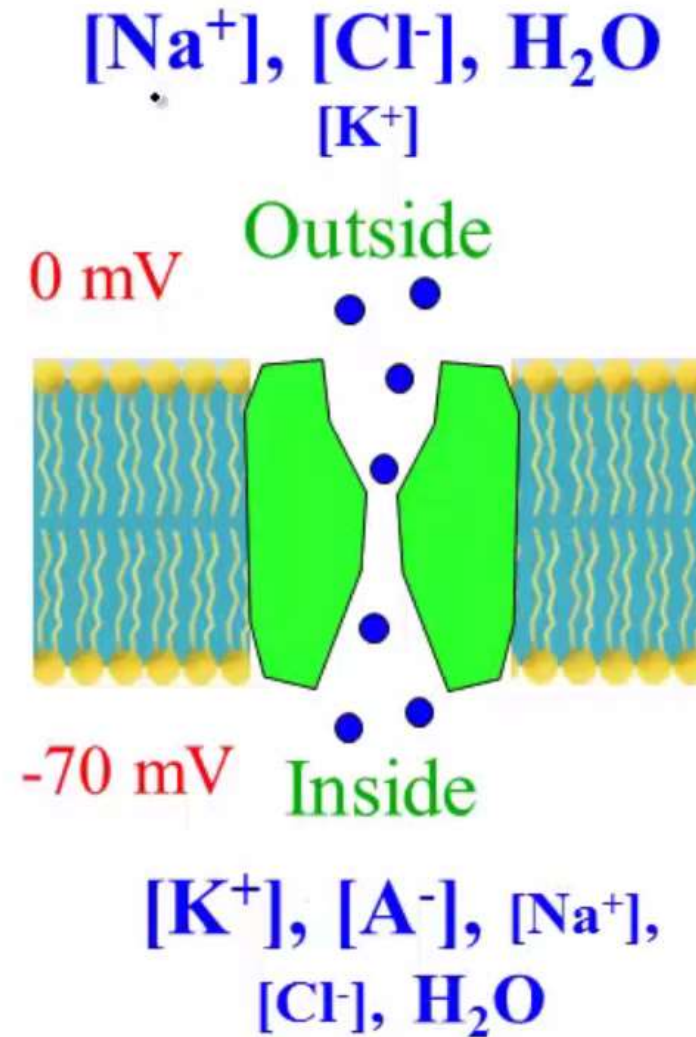
Optic Tectum

(Drawings by Ramón y Cajal, c. 1900)



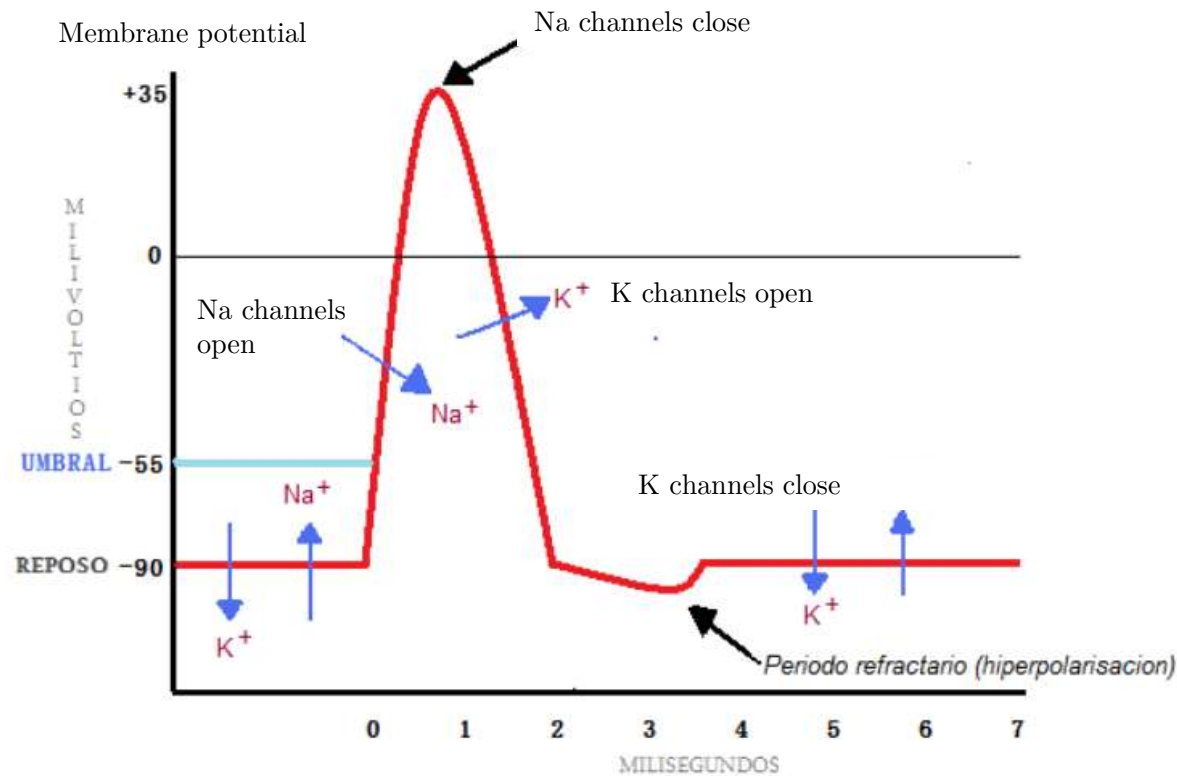
Potential difference across the membrane

Why more water and sodium outside? Maybe because life was generated in the sea, where water, clorus and sodium are very much present! Just an speculation...



Voltage-gated channels cause action potentials (spikes)

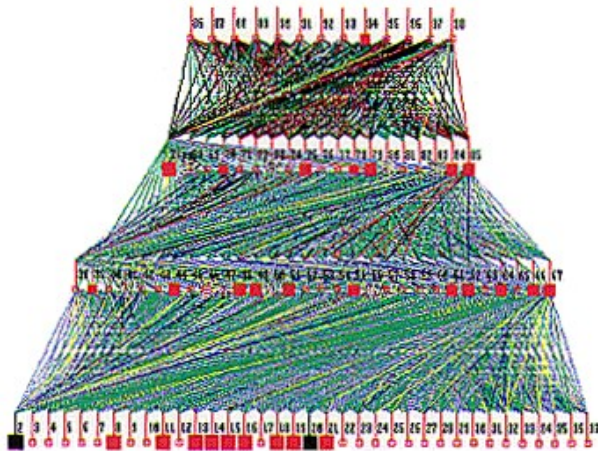
1. Strong depolarization opens Na^+ channels, causing rapid Na^+ influx and more channels to open until they inactivate
2. K^+ outflux restores membrane potential



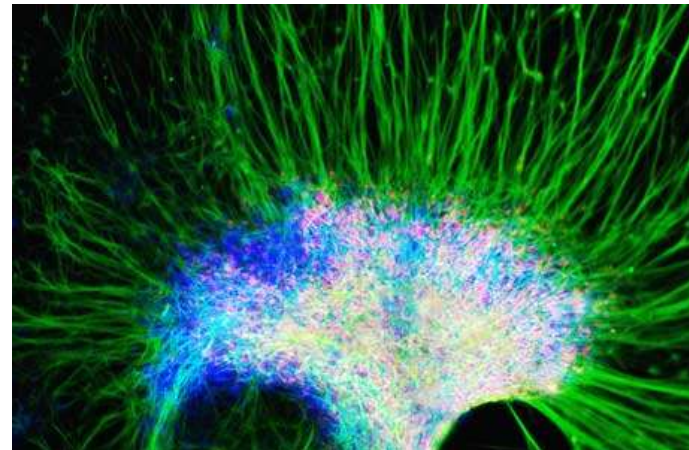
More details on action potentials in neurons: https://www.youtube.com/watch?v=iBDXOt_uHTQ

Differences between natural and artificial NN are in terms of behaviour but also in terms of complexity...

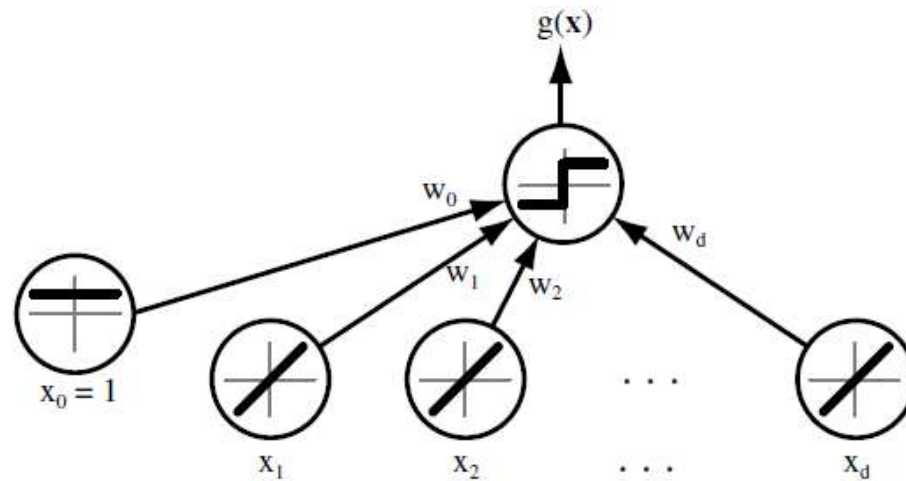
Artificial NN: 37 synapses



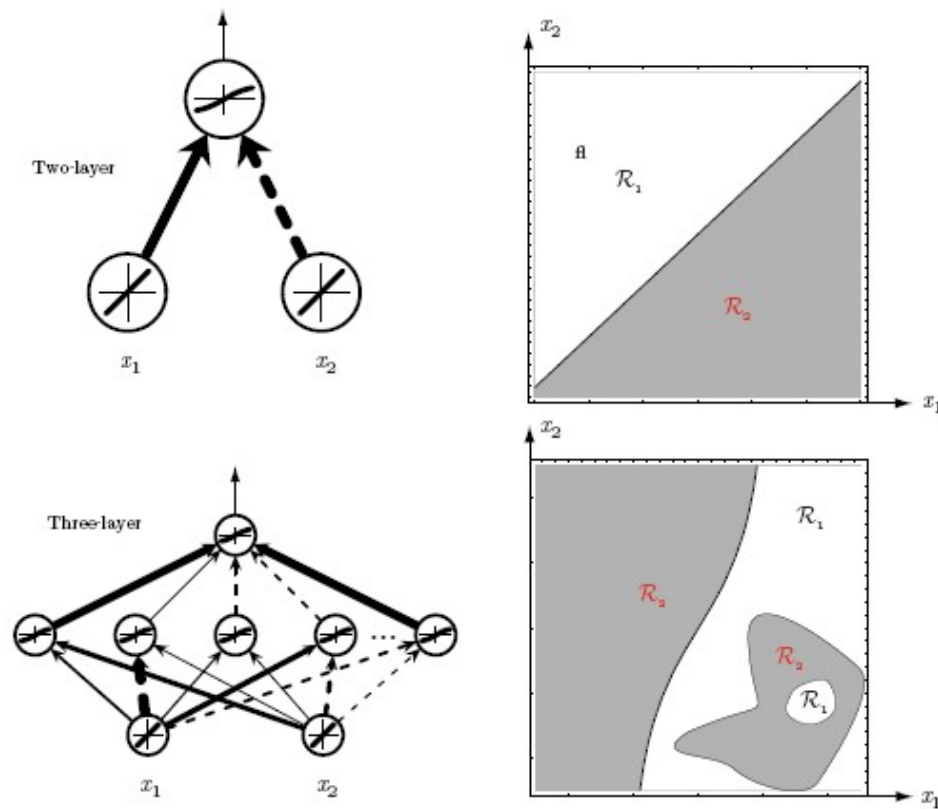
Natural NN: 10^{11} neurones with 10^4 synapses each
6 hidden layers in the cortex



A linear classifier e.g. the perceptron of linear SVM, is able to implement hyperplane decision boundaries and simple logic functions.



- A **multilayer neural network** is a non linear function of the inputs, where the parameters are learnt from the training database.
- Non linear functions are used at each neuron to obtain arbitrary shape decision regions
- **Optimal network topology** depends upon the problem at hand. A complexity adjustment is necessary to decide how many free parameters we need.

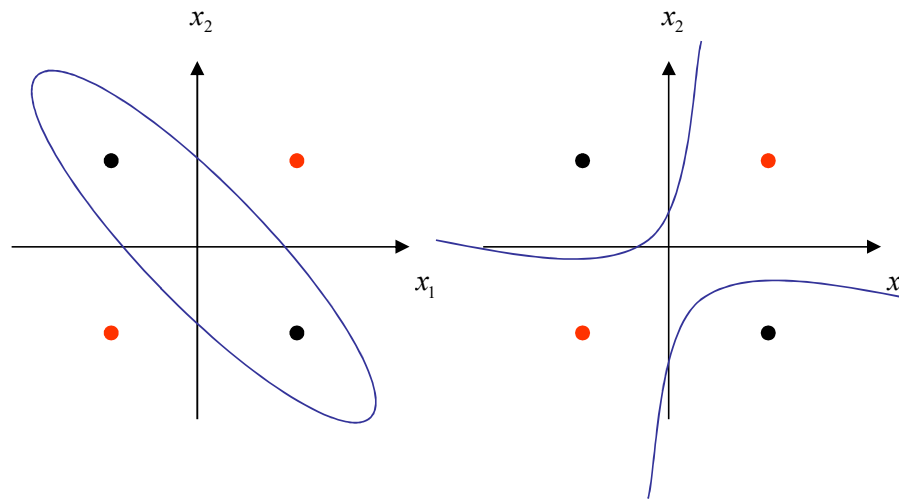


Example 1: the OR-exclusive problem

AND and OR can be implemented by the perceptron, but the simple XOR operation is a non-linearly separable problem.

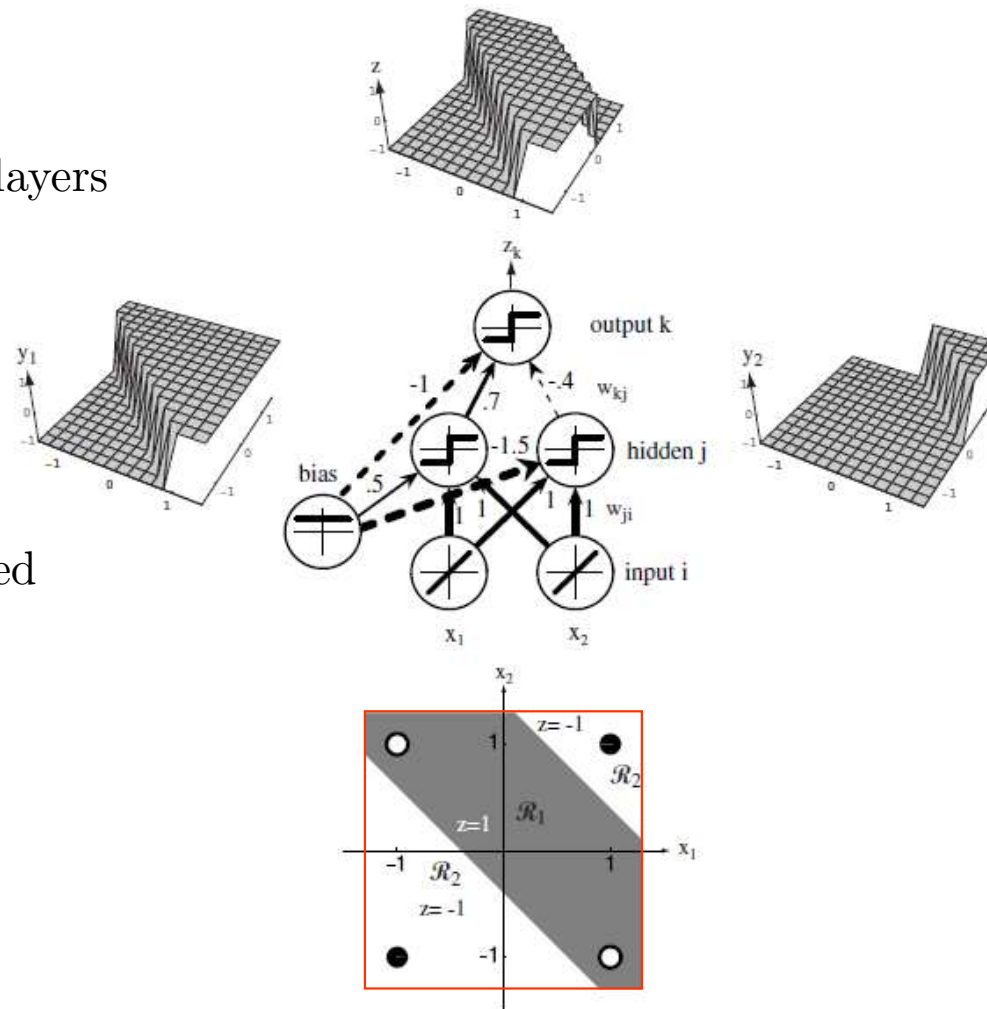
$$(x_1 \ x_2)^T$$

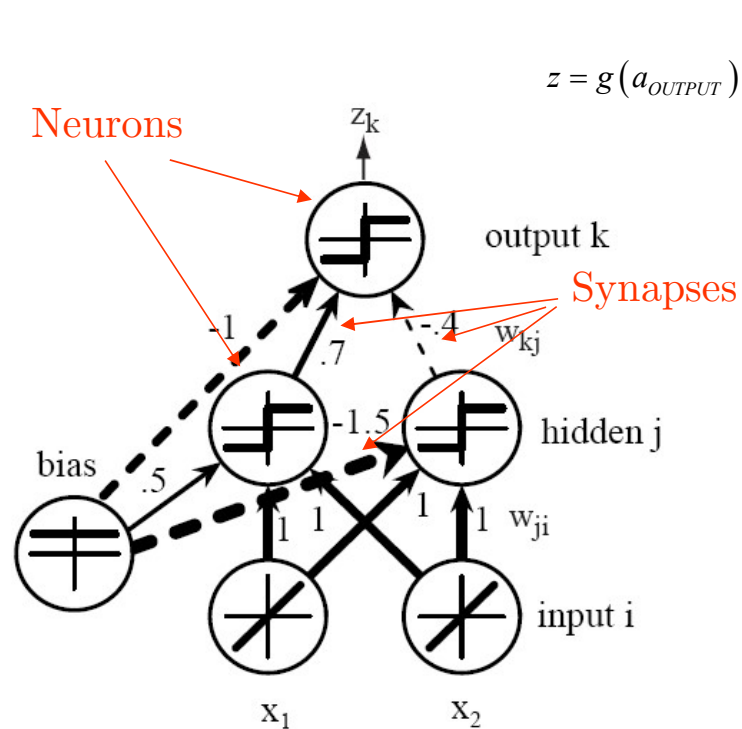
Input vector	Class
$(-1,-1)$	2
$(-1,+1)$	1
$(+1,-1)$	1
$(+1,+1)$	2



The XOR as a three-layers neural network:

- An input layer
- A hidden layer
- An output layer
- 2-2-1 fully connected topology





$$z = g(a_{OUTPUT}) = \text{sign} \left(\begin{pmatrix} -1 & +0,7 & -0,4 \end{pmatrix} \begin{pmatrix} bias \\ y_1 \\ y_2 \end{pmatrix} \right)$$

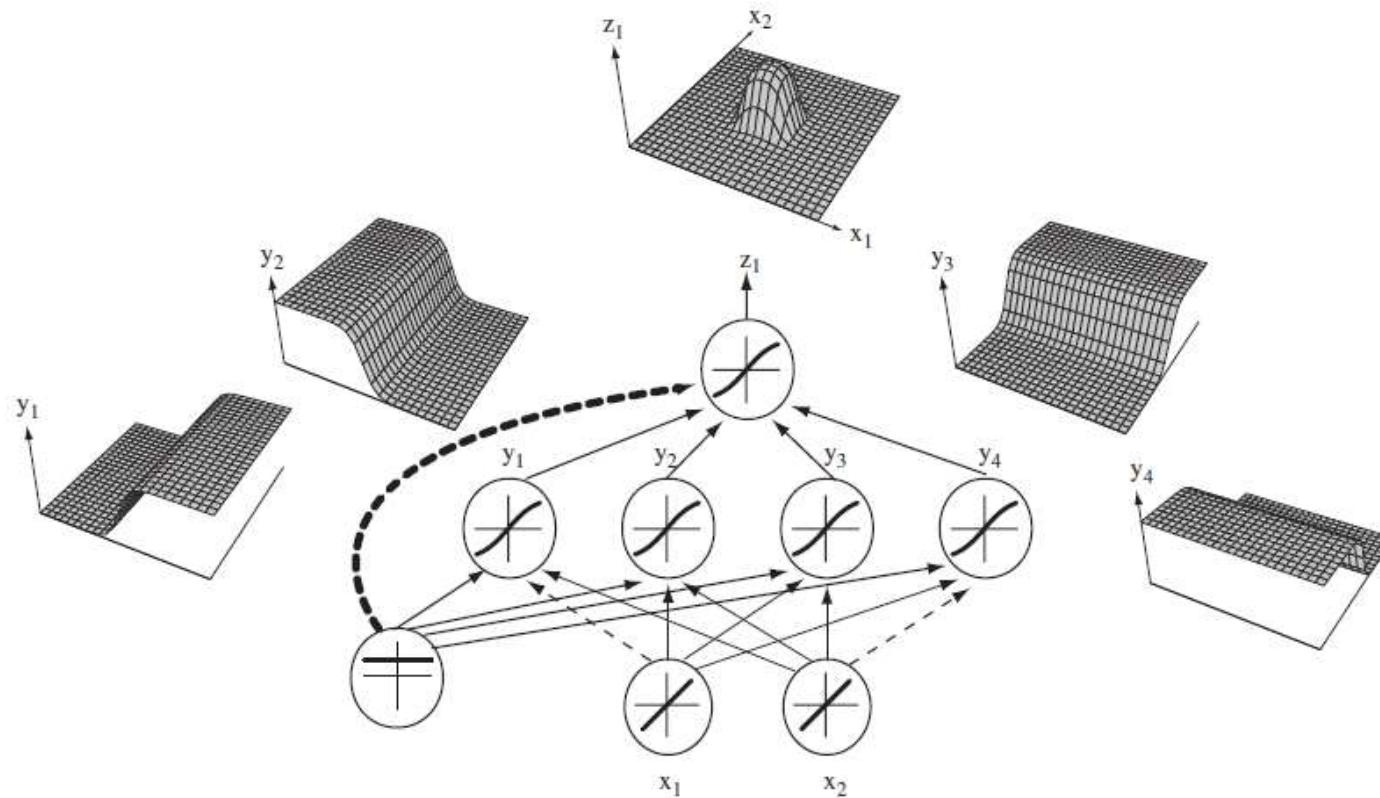
$$y_1 = f(a_1) = \text{sign} \left(\begin{pmatrix} +0,5 & +1 & +1 \end{pmatrix} \begin{pmatrix} bias \\ x_1 \\ x_2 \end{pmatrix} \right)$$

$$y_2 = f(a_2) = \text{sign} \left(\begin{pmatrix} -1,5 & +1 & +1 \end{pmatrix} \begin{pmatrix} bias \\ x_1 \\ x_2 \end{pmatrix} \right)$$

Net activation (a): weighted sum of inputs to the neuron

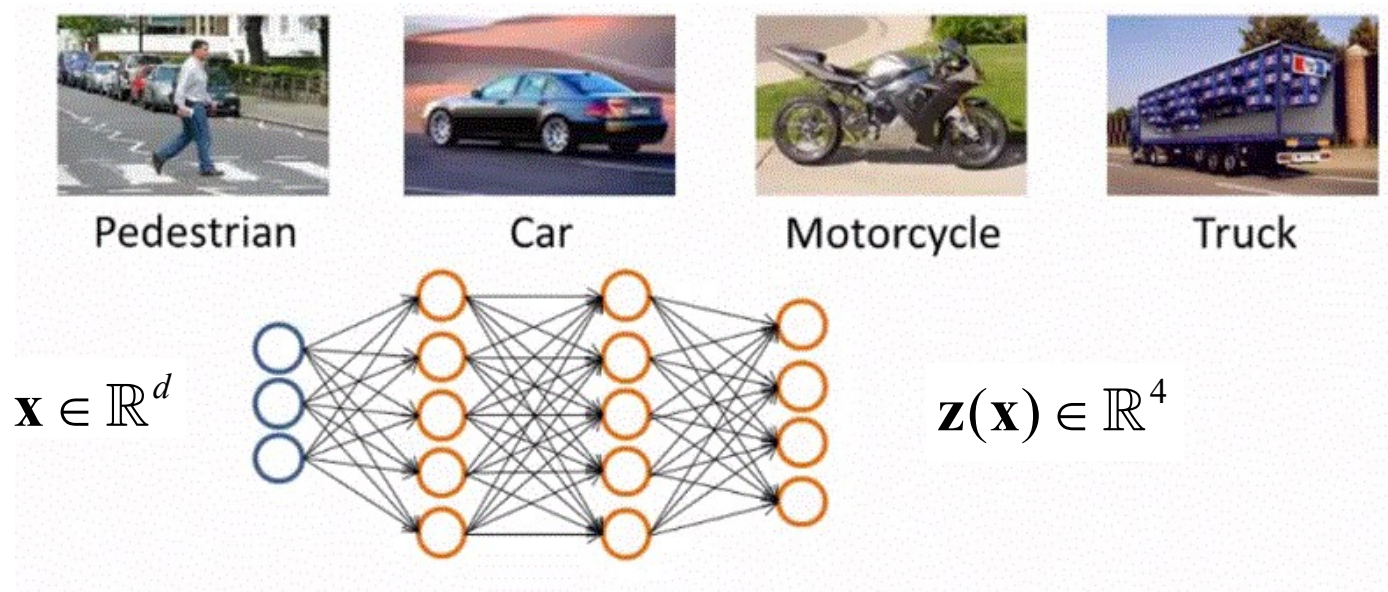
Activation function ($f(.)$): non-linearity applied on the net activation

Example 2: the flexibility in the structure and the set of weights in the NN allows defining different decision functions.



The non-linear functions of the hidden layer have the flexibility to engineer new features and feed them to a sort of linear decisor.

But we do not care much about the shape of the final decision function, only about taking good decisions:



we want







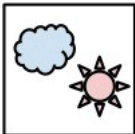


$$\mathbf{z}(\mathbf{x}) \simeq \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \text{if } \mathbf{x} \text{ is a pedestrian}$$

$$\mathbf{z}(\mathbf{x}) \simeq \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \text{if } \mathbf{x} \text{ is a car}$$

$$\mathbf{z}(\mathbf{x}) \simeq \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \text{if } \mathbf{x} \text{ is a motorcycle}$$

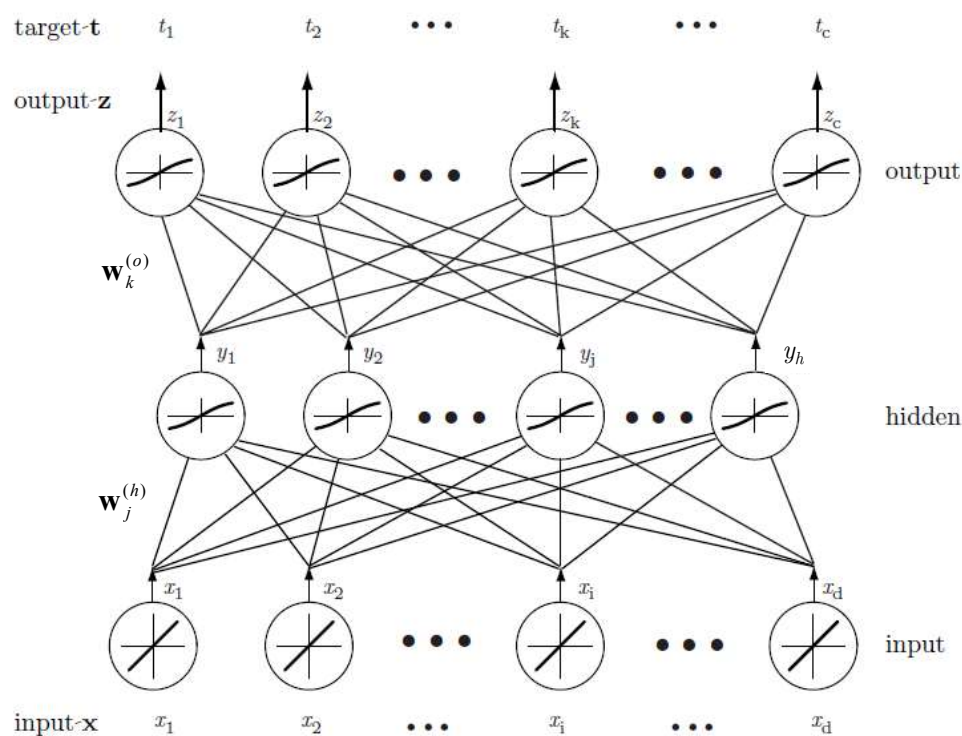
$$\mathbf{z}(\mathbf{x}) \simeq \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \quad \text{if } \mathbf{x} \text{ is a truck}$$

We might also have multi-label classification problem...

	Multi-Class	Multi-Label
$C = 3$	Samples (\mathbf{x})	Samples (\mathbf{x})
  	  	  
	Labels (\mathbf{t})	Labels (\mathbf{t})
	$[0\ 0\ 1]$ $[1\ 0\ 0]$ $[0\ 1\ 0]$	$[1\ 0\ 1]$ $[0\ 1\ 0]$ $[1\ 1\ 1]$

3. Multilayer feed-forward networks

- A three-layer network with d-h-c nodes



$$t_k \in \{0,1\} \quad k = 1, \dots, c$$

At the testing phase, the classe is decided from the maximum of the output values

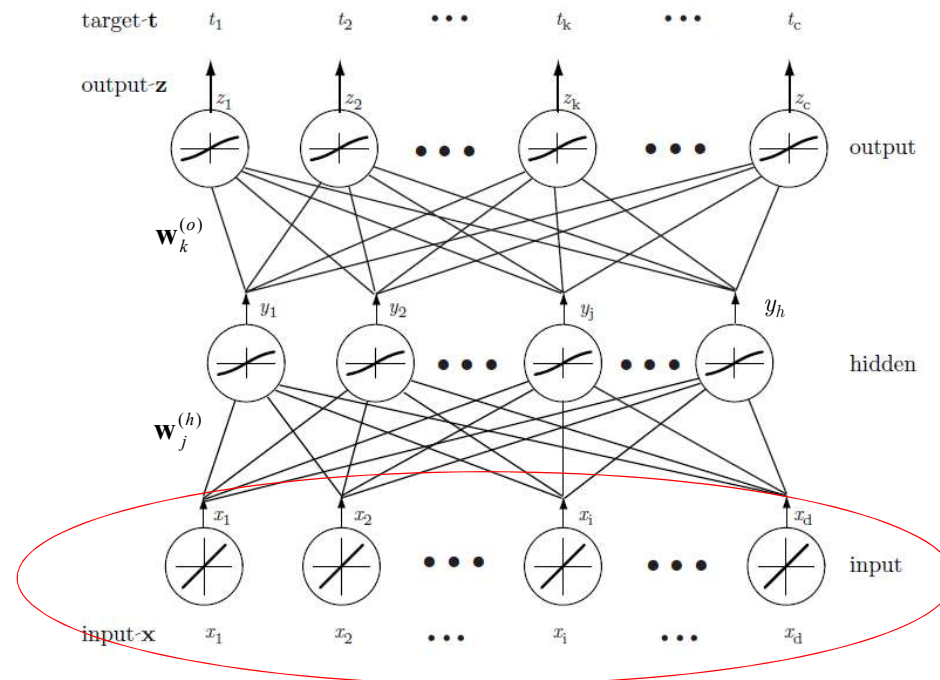
$$\hat{c} = \arg \max_i (z_i)$$

When training, weights between layers have to be computed by comparing NN outputs to the label of the training vector \mathbf{x} in the database

At the input layer there is one node per feature

For a three-layer network considered...

- **Input vector** of **dimension d** $\mathbf{x} = (x_1 \ x_2 \ \dots \ x_d)$

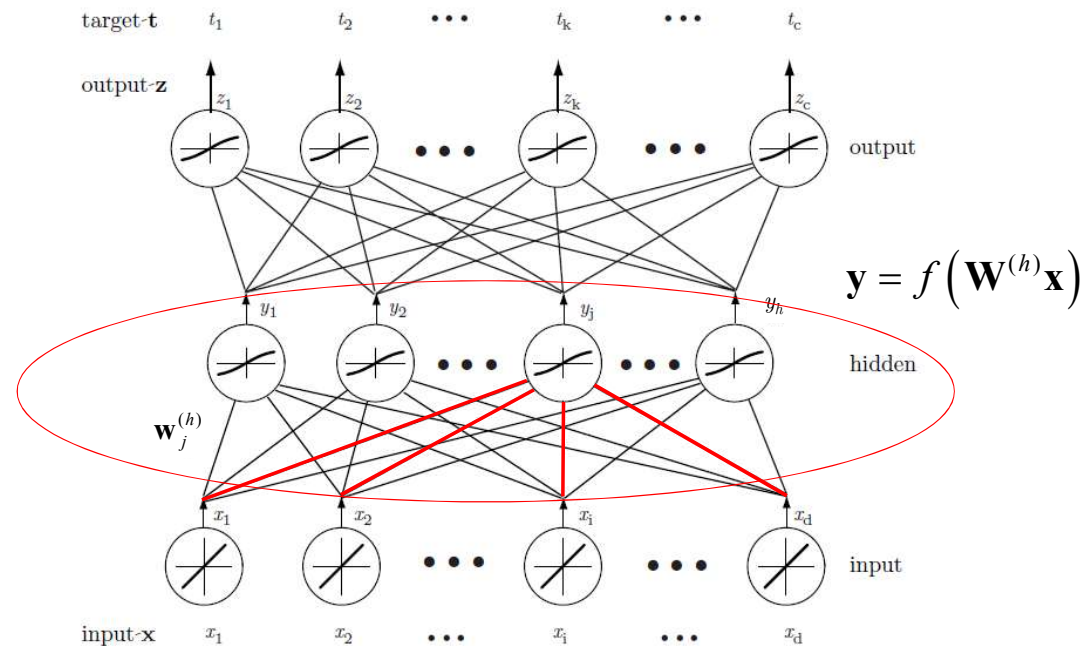


- Hidden layer: *h*-neurons

$$y_j = f\left(\sum_{i=1}^d w_{ji}^{(h)} x_i + w_{j0}^{(h)}\right) = f\left(\sum_{i=0}^d w_{ji}^{(h)} x_i\right) = f\left(\mathbf{w}_j^{(h)T} \mathbf{x}\right)$$

$$j = 1, 2, \dots, h$$

$w_{j0}^{(o)}$: bias terms add one degree of freedom to all weight vectors.

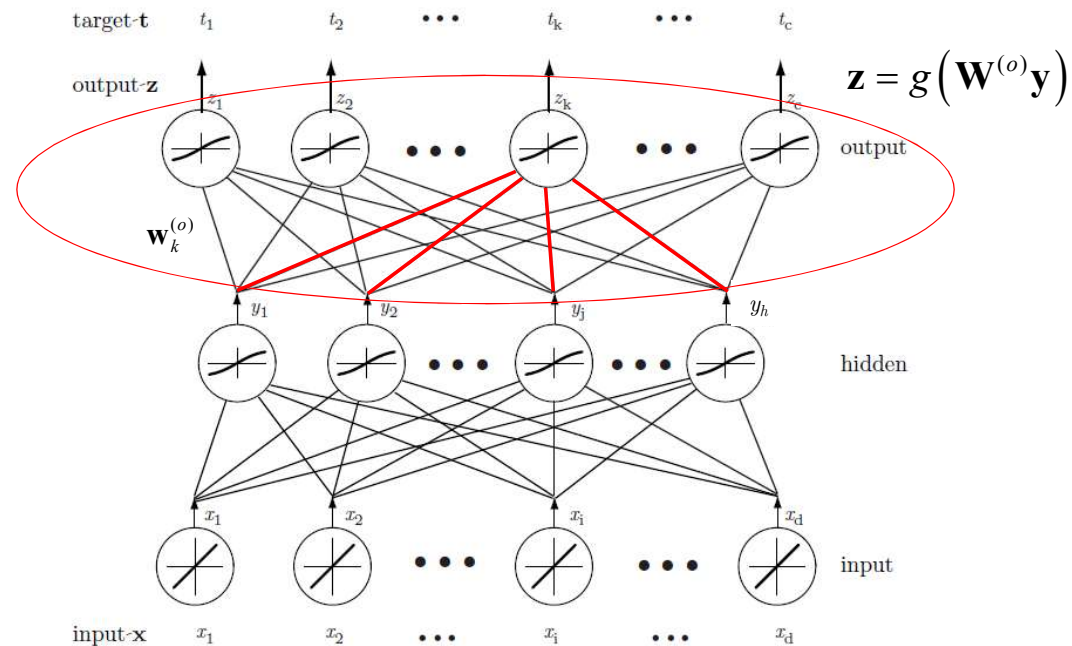


- **Output layer:** c scalar output discriminants are implemented.

$$z_k(\mathbf{x}) = g\left(\sum_{j=1}^h w_{kj}^{(o)} y_j + w_{k0}^{(o)}\right) = g\left(\sum_{j=0}^h w_{kj}^{(o)} y_j\right) = g(\mathbf{w}_k^{(o)T} \mathbf{y})$$

$$k = 1, 2, \dots, c$$

$w_{k0}^{(o)}$: bias terms add a new coordinate to all the weight vectors.



- Challenges in the definition of the NN

Activation functions:

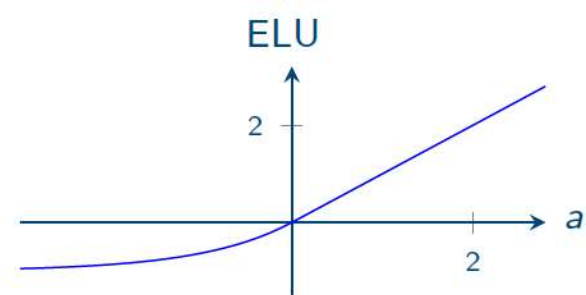
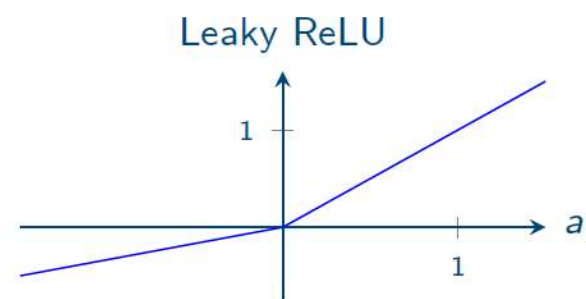
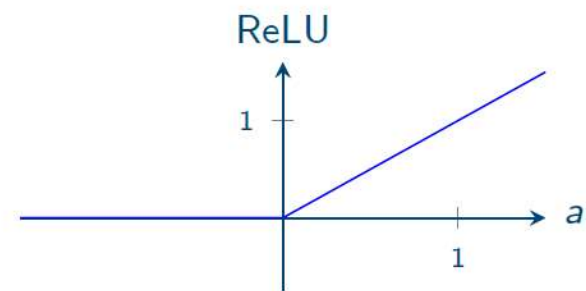
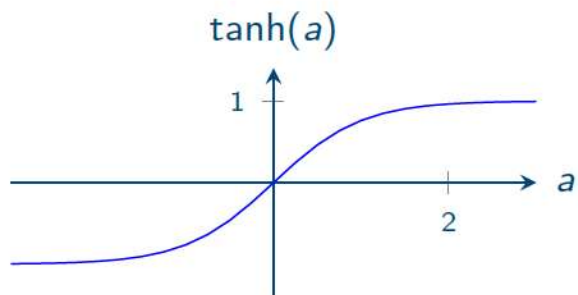
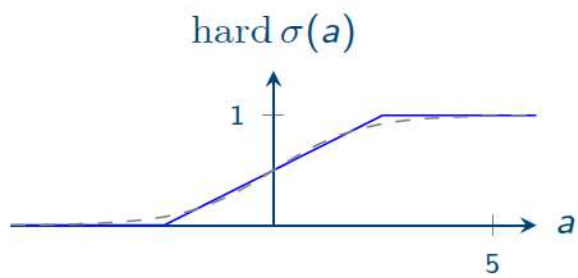
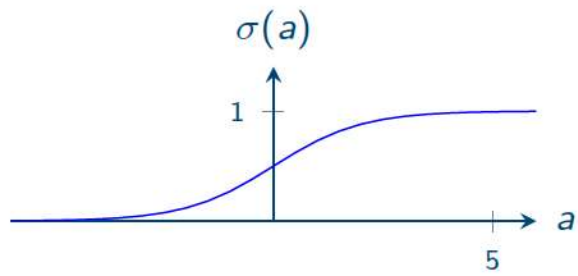
- Non-linear scalar functions $f(\cdot)$, $g(\cdot)$ smooth enough to learn from gradient descent techniques

Computation of weights:

- $(d+1)h + (h+1)c$ scalar parameters to be designed
- The *Back Propagation Algorithm* is one of the simplest and most general methods for supervised training of NN.
- It is a gradient method to compute the weights at each layer.
- The algorithm is developed in one step per hidden layer plus one step for the output layer.

3. Selection of the activation function $f(.)$

- The activation functions $f(.)$ must be
 - Continuous so that the decision boundaries are smoothly connected
 - Non linear, otherwise three layer network results two layer network
 - Linear for some values, allows implementing linear classifier if this is the optimum solution
 - Saturate at large input values (have some minimum and maximum output values)
 - Monotonic, helps to avoid undesirable local extreme points



- For classification, at the output layer, $g(\cdot)$ is chosen as softmax function:

$$g_k(\mathbf{x}) = \frac{e^{x_k}}{\sum_{i=1}^c e^{x_i}}$$

so that network outputs can be considered as a probability distribution.

- For regression we have a linear single output:

$$g(\mathbf{x}) = \mathbf{w}^T \mathbf{x}$$

4. Train with the backpropagation algorithm

The computation of the weights departs from the knowledge of a labeled database....

- For a supervised database a **Training Target Function $J(\mathbf{w})$** (scalar function) is defined.
- The error measures the difference between the desired output t_k (the actual classes) and the network outputs z_k .
- The back-propagation learning rule **is based on gradient descent**.
- The **weights are initialized with random values**, and then they are changed in a direction that will reduce the error.
- The **learning rate η** indicates the relative size of the change in weights.
- The final performance **depends on the starting point for the weights**.

- Objective functions $J(\mathbf{w})$ to be minimized:
 - Non-differentiable ones: *recall, accuracy, error rate, etc.*
 - Differentiable ones: *mean squared error, absolute error, logarithmic loss (cross-entropy), Kullback-Leibler divergence, cosine distance,...*

- Objective function $J(\mathbf{w})$ to be minimized:

Mean squared error

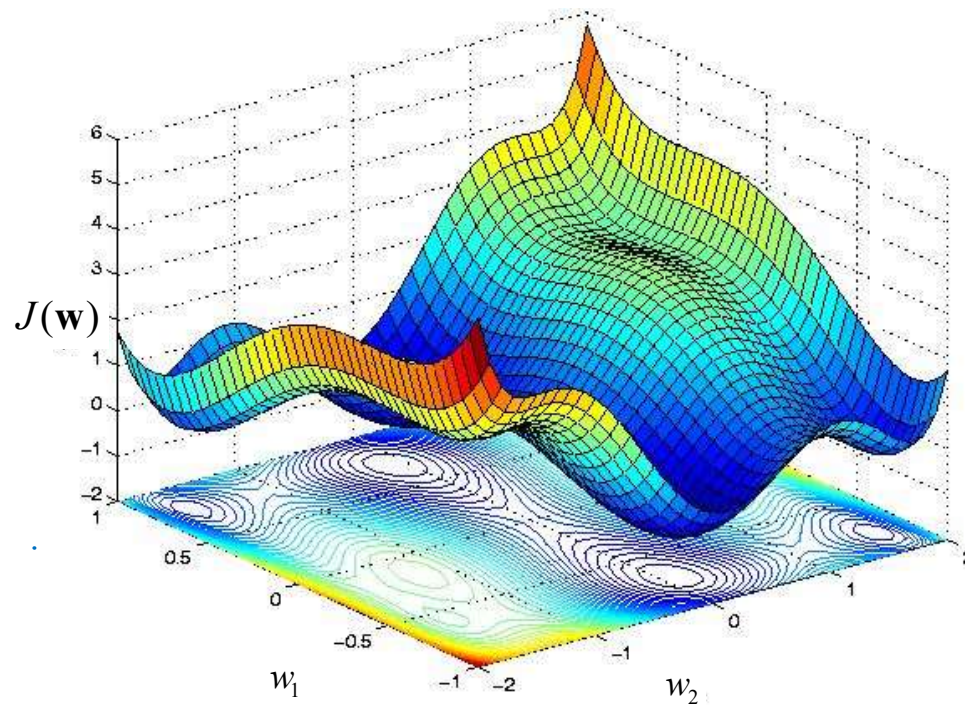
$$\underset{\mathbf{w}}{\text{minimize}} \frac{1}{2} \sum_{k=1}^c \underbrace{(t_k - z_k(\mathbf{x}))^2}_{e_k(\mathbf{x})}$$

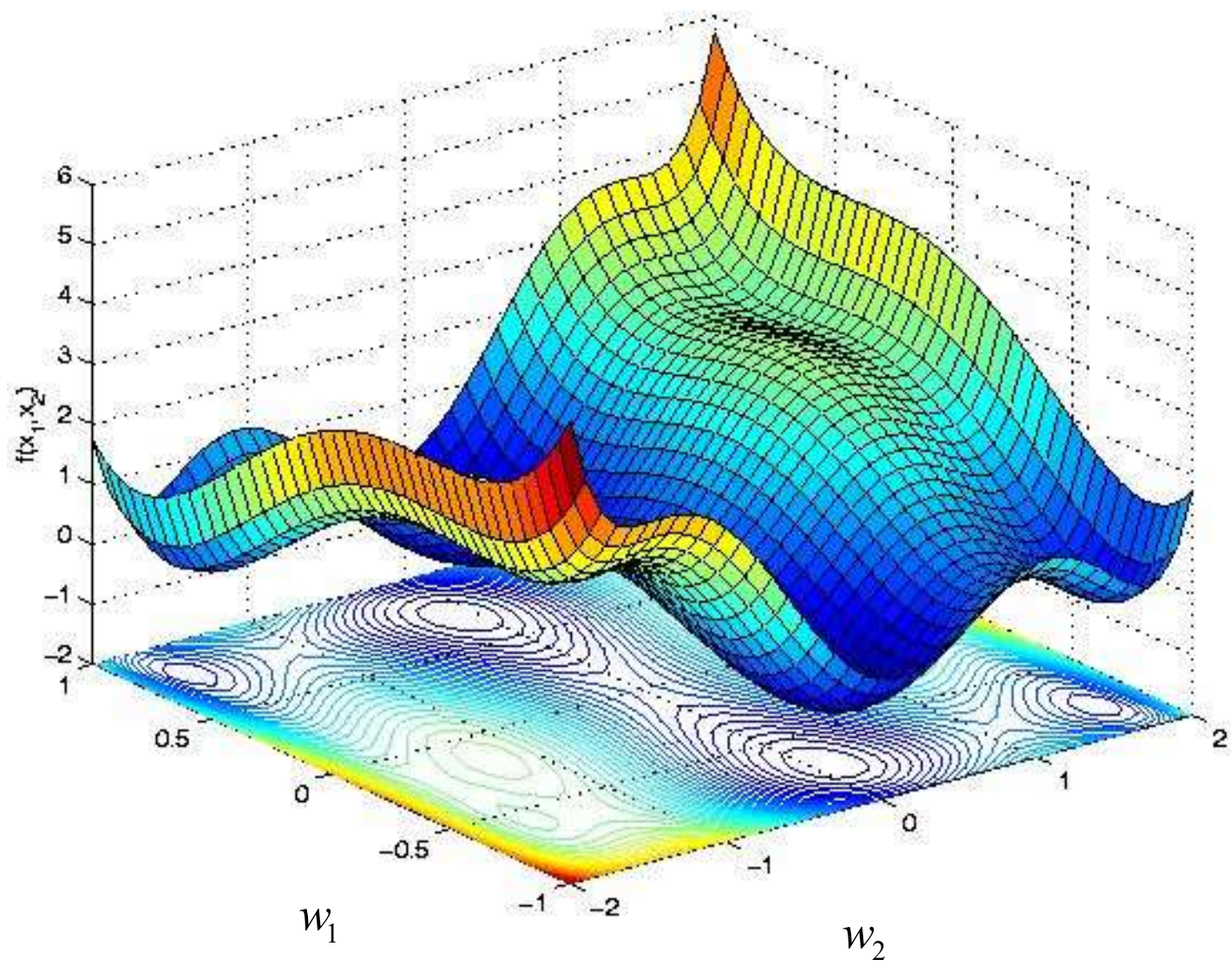
All weights \mathbf{w} are here!

$$t_k = \begin{cases} +1 & \mathbf{x} \in \text{class } k \\ 0 & \mathbf{x} \notin \text{class } k \end{cases}$$

Logarithmic loss

$$\underset{\mathbf{w}}{\text{minimize}} - \sum_{k=1}^c t_k \log z_k(\mathbf{x})$$





- Objective function $J(\mathbf{w})$ to be minimized:

Mean squared error minimize $\frac{1}{2} \sum_{k=1}^c \underbrace{(t_k - z_k(\mathbf{x}))}_{e_k(\mathbf{x})}^2$ $t_k = \begin{cases} +1 & \mathbf{x} \in \text{class } k \\ 0 & \mathbf{x} \notin \text{class } k \end{cases}$

Logarithmic loss minimize $-\sum_{k=1}^c t_k \log z_k(\mathbf{x})$

All weights \mathbf{w} are here!

- Gradient-based weight adaptation rule

$$\Delta \mathbf{w}[n] = -\eta \left. \frac{\partial J}{\partial \mathbf{w}} \right|_{\mathbf{w}=\mathbf{w}[n]}$$

$$\mathbf{w}[n+1] = \mathbf{w}[n] + \Delta \mathbf{w}[n]$$

- Output assigned class

$$\hat{c} = \arg \max_i z_i(\mathbf{x})$$

Denotes iteration in the gradient algorithm

Computation of gradients (for MSE criterion)



$$z_k \equiv g_k(\mathbf{x}) = g\left(\underbrace{\mathbf{w}_k^{(o)T}}_{a_k^{(o)}} \underbrace{\mathbf{y}}_{a_j^{(h)}}\right) = g\left(\sum_{j=1}^h w_{kj}^{(o)} \underbrace{f\left(\underbrace{\sum_{i=1}^d w_{ji}^{(h)} x_i + w_{j0}}_{a_j^{(h)}}\right)}_{y_j} + w_{k0}\right) \quad \begin{matrix} y_0 = 1 \\ x_0 = 1 \end{matrix}$$

$k = 1, \dots, c$

Update of the **hidden-to-output** layer weights

$$\frac{\partial J}{\partial \mathbf{w}_k^{(o)}} = \frac{\partial J}{\partial a_k^{(o)}} \frac{\partial a_k^{(o)}}{\partial \mathbf{w}_k^{(o)}} = \frac{\partial J}{\partial z_k} \frac{\partial z_k}{\partial a_k^{(o)}} \frac{\partial a_k^{(o)}}{\partial \mathbf{w}_k^{(o)}} = - \underbrace{(t_k - z_k) g'(a_k^{(o)})}_{\delta_k} \mathbf{y}$$

Hidden-to-output weights!

$$\mathbf{w}_k^{(o)}[n+1] = \mathbf{w}_k^{(o)}[n] - \eta \nabla_{\mathbf{w}_k^{(o)}} J(\mathbf{w}) \Big|_{\mathbf{w}=\mathbf{w}[n]} \in \mathbb{R}^{h \times 1}$$

$$\nabla_{\mathbf{w}_k^{(o)}} J(\mathbf{w}) \Big|_{\mathbf{w}=\mathbf{w}[n]} = -(t_k[n] - z_k[n]) g'(a_k^{(o)}[n]) \mathbf{y}[n] = -\delta_k[n] \mathbf{y}[n] \quad k = 1, \dots, c$$



Update of the **input-to-hidden** layer weights

$$\begin{aligned}
 \frac{\partial J}{\partial \mathbf{w}_j^{(h)}} &= \frac{\partial J}{\partial y_j} \frac{\partial y_j}{\partial a_j^{(h)}} \frac{\partial a_j^{(h)}}{\partial \mathbf{w}_j^{(h)}} = \left[-\sum_{l=1}^c (t_l - z_l) \frac{\partial z_l}{\partial y_j} \right] \frac{\partial y_j}{\partial a_j^{(h)}} \frac{\partial a_j^{(h)}}{\partial \mathbf{w}_j^{(h)}} = \\
 &= \left[-\sum_{l=1}^c (t_l - z_l) g'(a_l^{(o)}) w_{lj}^{(o)} \right] \frac{\partial y_j}{\partial a_j^{(h)}} \frac{\partial a_j^{(h)}}{\partial \mathbf{w}_j^{(h)}} = \\
 &= -\sum_{l=1}^c \underbrace{(t_l - z_l) g'(a_l^{(o)}) w_{lj}^{(o)}}_{\delta_l} f'(a_j^{(h)}) \mathbf{x} = -(\boldsymbol{\delta}^T \tilde{\mathbf{w}}_j^{(o)}) f'(a_j^{(h)}) \mathbf{x}
 \end{aligned}$$

Input-to-hidden
weights!

$$\mathbf{w}_j^{(h)}[n+1] = \mathbf{w}_j^{(h)}[n] - \eta \nabla_{\mathbf{w}_j^{(h)}} J(\mathbf{w}) \Big|_{\mathbf{w}=\mathbf{w}[n]} \in \mathbb{R}^{d \times 1}$$

Hidden-to-output weights **reorganised!!!**
(notice difference with previous slide)

$$\nabla_{\mathbf{w}_j^{(h)}} J(\mathbf{w}) \Big|_{\mathbf{w}=\mathbf{w}[n]} = -(\boldsymbol{\delta}^T[n] \tilde{\mathbf{w}}_j^{(o)}[n]) f'(a_j^{(h)}[n]) \mathbf{x}[n] \quad j = 1, \dots, h$$

Back propagation: the c terms in $\boldsymbol{\delta}$ and the coefficients $\tilde{\mathbf{w}}_j^{(o)}[n]$ from the output layer are needed to update the weights at the input-to-hidden layer

Algorithm 1 (Stochastic backpropagation)

```
1 begin initialize network topology (#hidden units),  $\mathbf{w}_k^{(o)}$ ,  $\mathbf{w}_j^{(h)}$ ,  $\eta$ ,  $n \leftarrow 0$ 
2   do  $n \leftarrow n+1$ 
3      $\mathbf{x}[n] \leftarrow$  randomly chosen feature vector
4      $\Delta \mathbf{w}_k^{(o)} = \delta_k[n] \mathbf{y}[n]$  % Compute gradient hidden-to-output
5      $\mathbf{w}_k^{(o)}[n+1] = \mathbf{w}_k^{(o)}[n] + \eta \Delta \mathbf{w}_k^{(o)}$   $k = 1, \dots, c$  % Update hidden-to-output coefficients
6      $\Delta \mathbf{w}_j^{(h)} = (\boldsymbol{\delta}^T[n] \tilde{\mathbf{w}}_j^{(o)}[n]) f'(a_j^{(h)}[n]) \mathbf{x}[n]$  % Compute gradient input-to-hidden
7      $\mathbf{w}_j^{(h)}[n+1] = \mathbf{w}_j^{(h)}[n] + \eta \Delta \mathbf{w}_j^{(h)}$   $j = 1, \dots, h$  % Update input-to-hidden coefficients
8   until  $\Delta J(\mathbf{w}) < \theta$ 
9   return  $\mathbf{w}_k^{(o)}$ ,  $\mathbf{w}_j^{(h)}$   $k = 1, \dots, c$   $j = 1, \dots, h$ 
10 end
```

- Alternate batch-type training criteria: now all the vectors in the data base are in $J(\mathbf{w})$

Mean squared error

$$\underset{\mathbf{w}}{\text{minimize}} \quad \frac{1}{2} \sum_{n=1}^N \sum_{k=1}^c \underbrace{\left(t_k[n] - z_k(\mathbf{x}[n]) \right)^2}_{e_k[n]}$$

Logarithmic loss

$$\underset{\mathbf{w}}{\text{minimize}} \quad - \sum_{n=1}^N \sum_{k=1}^c t_k[n] \log z_k(\mathbf{x}[n])$$

$$t_k[n] = \begin{cases} +1 & \mathbf{x}[n] \in \text{class } k \\ 0 & \mathbf{x}[n] \notin \text{class } k \end{cases}$$

n denotes index in the database

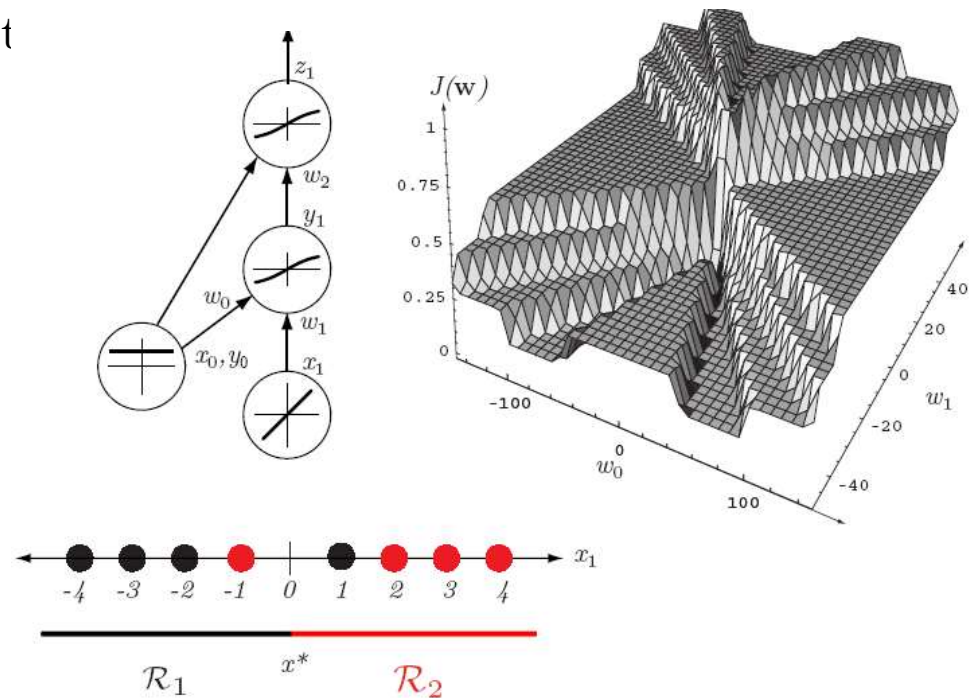
Algorithm 2 (Batch backpropagation)

```
1 begin initialize network topology (#hidden units),  $\mathbf{w}_k^{(o)}$ ,  $\mathbf{w}_j^{(h)}$ ,  $\eta$ ,  $r \leftarrow 0$ ,  $n \leftarrow 0$ 
2   do  $r \leftarrow r+1$ 
3      $n \leftarrow 0$ ;  $\Delta \mathbf{w}_k^{(o)} \leftarrow \mathbf{0}$ ;  $\Delta \mathbf{w}_j^{(h)} \leftarrow \mathbf{0}$ 
4     do  $n \leftarrow n+1$ 
5        $\mathbf{x}[n] \leftarrow$  randomly chosen feature vector
6        $\Delta \mathbf{w}_k^{(o)} \leftarrow \Delta \mathbf{w}_k^{(o)} + \delta_k[n] \mathbf{y}[n]$  % Compute gradients
7        $\Delta \mathbf{w}_j^{(h)} \leftarrow \Delta \mathbf{w}_j^{(h)} + (\boldsymbol{\delta}^T[n] \tilde{\mathbf{w}}_j^{(o)}[n]) f'(a_j^{(h)}[n]) \mathbf{x}[n]$ 
8     until  $n = N$ 
9      $\mathbf{w}_k^{(o)}[r+1] = \mathbf{w}_k^{(o)}[r] + \eta \Delta \mathbf{w}_k^{(o)}$   $k = 1, \dots, c$  % Update hidden-to-output coefficients
10     $\mathbf{w}_j^{(h)}[r+1] = \mathbf{w}_j^{(h)}[r] + \eta \Delta \mathbf{w}_j^{(h)}$   $j = 1, \dots, h$  % Update input-to-hidden coefficients
11  until  $\Delta J(\mathbf{w}) < \theta$ 
12 return  $\mathbf{w}_k^{(o)}, \mathbf{w}_j^{(h)}$   $k = 1, \dots, c$   $j = 1, \dots, h$ 
13 end
```

The computed gradient takes into account all N vectors in the training data base. It shows better convergence properties than the stochastic version. In practice a mini-batch approach is preferred.

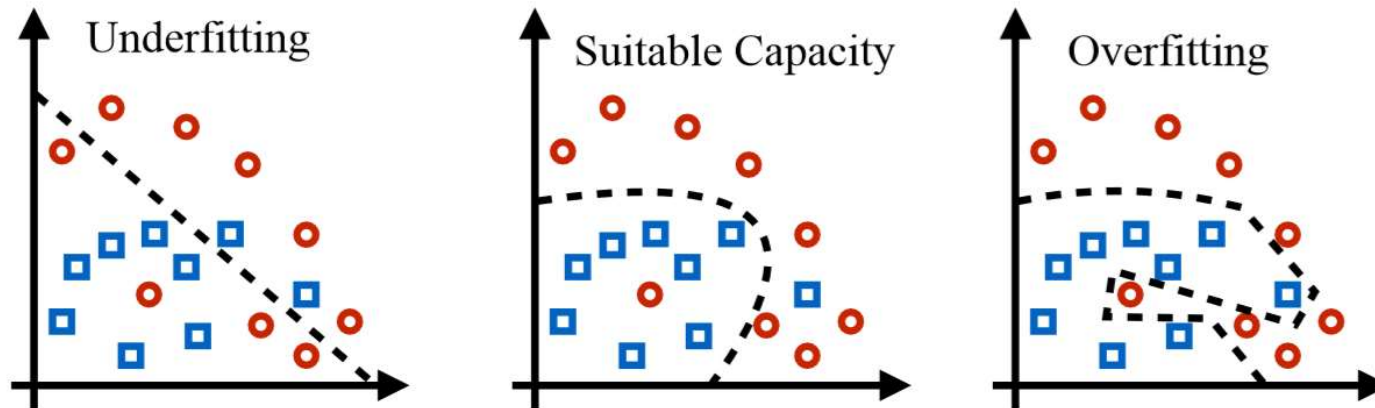
The error function can have local minima.
How to find the global one?

- **Local minima:** Difficult analytic solution. Gradient descent techniques with multiple initialization points are preferred.



6. How to avoid over/underfitting?

Neural networks have a large capacity of defining complex decision borders, so we have to prevent too much adaptation to the training set.



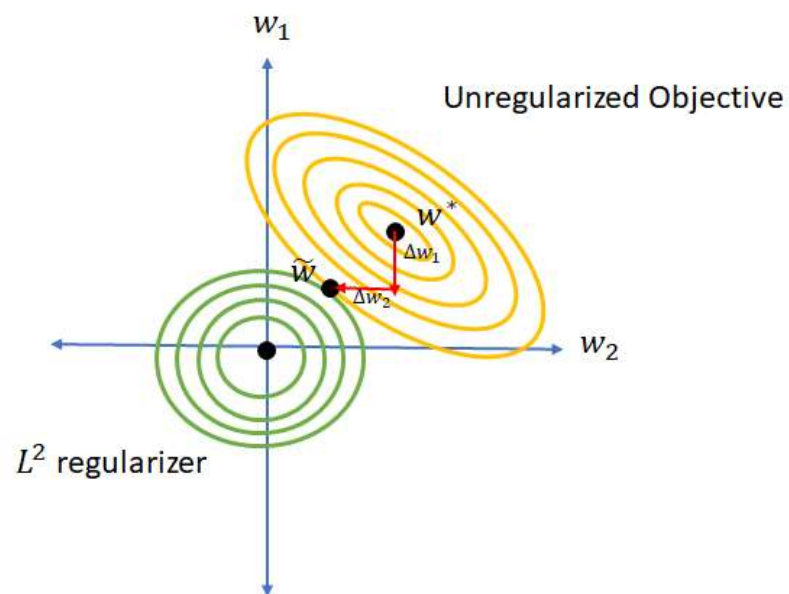
How to select the right NN model?

If overfit is due to a **large number of parameters...**

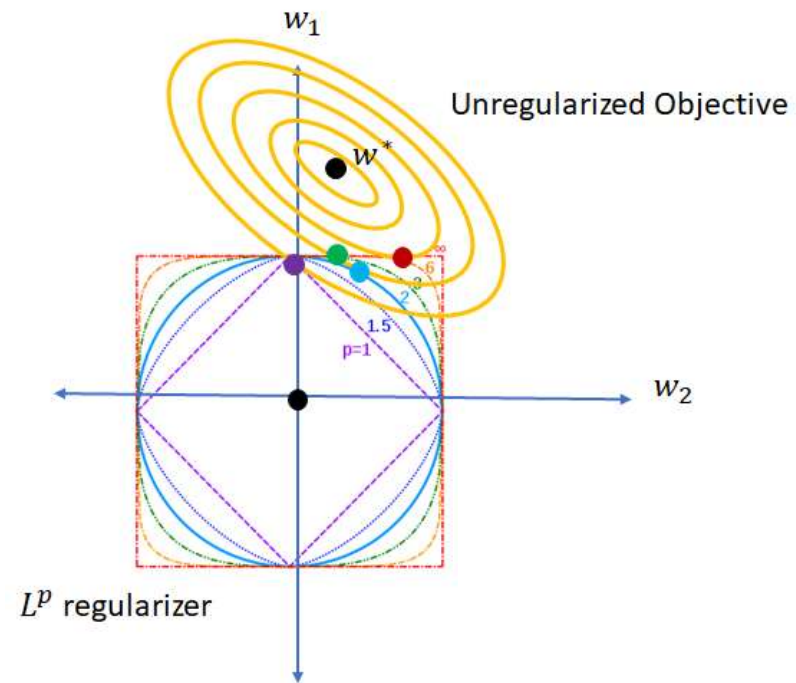
Regularization: add a penalization on the cost function, with parameter $\lambda > 0$ tuned from the validation set:

$$\hat{J}(\mathbf{w}, \lambda) = J(\mathbf{w}) + \lambda \Omega(\mathbf{w})$$

Usually, functions for Ω are the p -norm (norm-1 is for $p = 1$, the Euclidean norm is for $p = 2$).

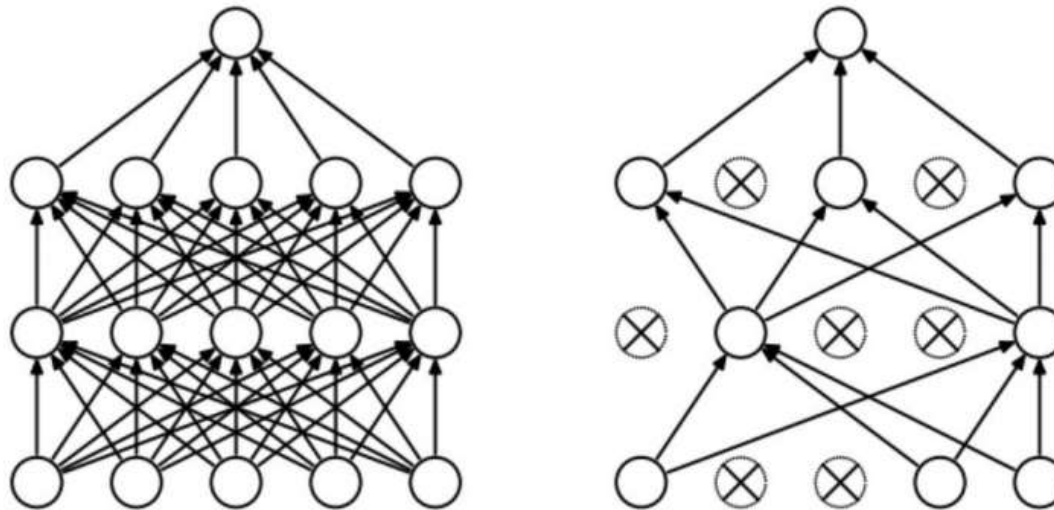


The 1-norm tends to null some of the weights, but it is non-differentiable:



Weight sharing: make the model simpler by forcing some weights to take the same values.

Dropout: randomly nulling some coefficients in every epoch.

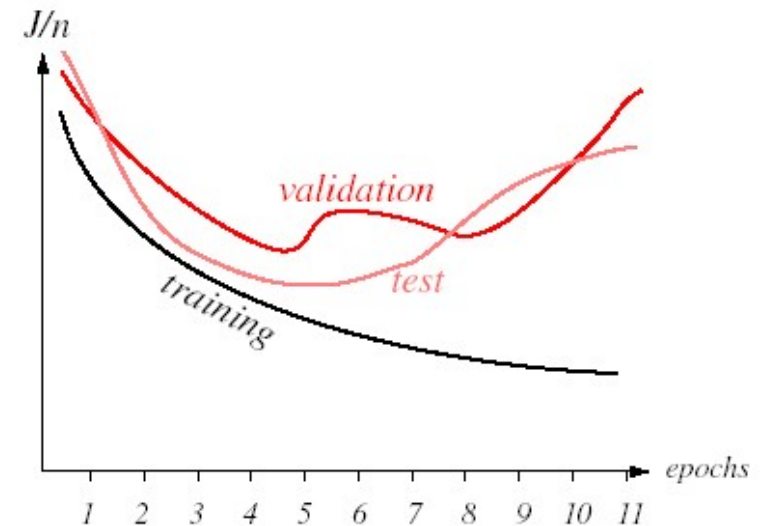


Hyperparameters in each of these procedures (value of λ , fraction of equal weights, fraction of null weights) have to be validated.

If overfit is due to **too much training**...

Early stop training: Error evolution with the number of epochs (decreases for the training set and has a minimum with the test set)

The epoch is each step in which all the elements in the training set are processed.

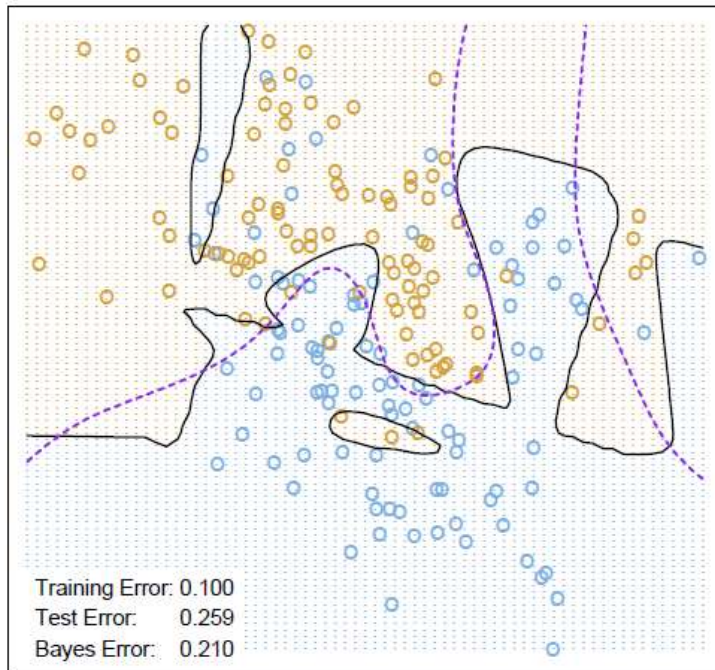


The stopping training epoch stopping has to be validated.

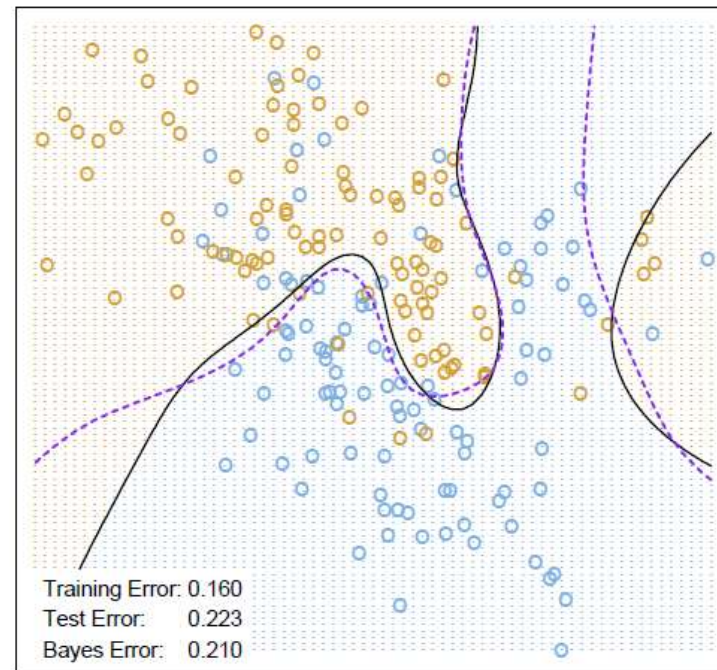
Model averaging: Train several networks with different initial values and combine (soft or hard) outputs.

Example 3: Regularization

Neural Network - 10 Units, $\lambda=0$



Neural Network - 10 Units, $\lambda=0.02$



Example 4: Stop training

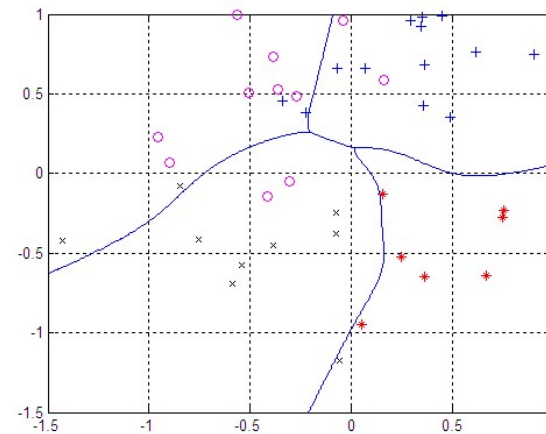
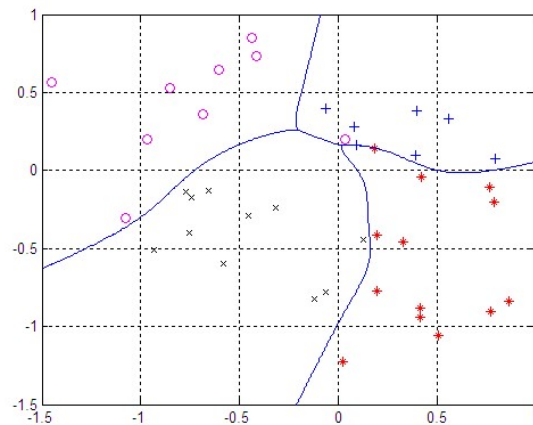
$c = 4$ classes (QPSK, SNR=3 dB)

Backpropagation algorithm

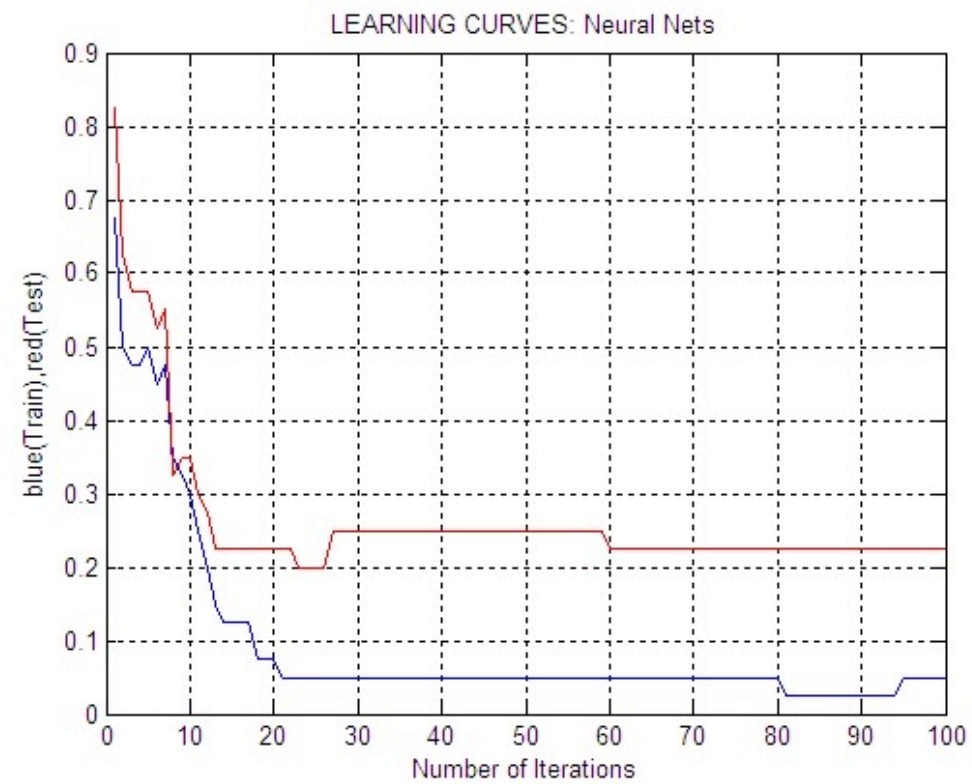
Let us apply a NN with the following structure: 2-5-5-4 NN

$$f(x) = \text{sigmoid}(\mathbf{w}^T \mathbf{x}) = \frac{2}{1 + \exp(-\mathbf{w}^T \mathbf{x})} - 1$$

$$a = 1 \quad b = 1$$



72



7. Accelerating convergence in gradient methods

Gradient-based approach is inherently slow due to:

1. Non-convexity of J .
2. In general, the gradient does not point to the minimum of J
3. Slow when reaching a plateau in J .
4. Gradients are inexactly estimated from data, so they have some variance.

Some modifications of the gradient-descent method that address the second, third and fourth problems...

<https://ruder.io/optimizing-gradient-descent/>

- Second order methods

Let us improve the convergence speed in batch-learning (slides 35, 36), based on Gauss-Newton method. The method changes the direction of the gradient in a smart way.

Let us approximate the objective function J at $\mathbf{w}[k+1]$ up to second order by the value of J around $\mathbf{w}[k]$:

$$J[k+1] \simeq J[k] + \Delta \mathbf{w}[k]^T \nabla_{\mathbf{w}} J[k] + \frac{1}{2} \Delta \mathbf{w}[k]^T \mathbf{H}[k] \Delta \mathbf{w}[k]$$

$\Delta \mathbf{w}[k] = \mathbf{w}[k+1] - \mathbf{w}[k]$

\mathbf{H} is the Hessian matrix

RECALL: n denotes index of vectors in the database
 k denotes iteration of the algorithm

- We want the gradient to go to zero in one step:

$$\nabla_{\mathbf{w}} J[k+1] = \mathbf{0} \Rightarrow \nabla J_{\mathbf{w}}[k] + \mathbf{H}[k] \Delta \mathbf{w}[k] = \mathbf{0}$$

$$\Delta \mathbf{w}[k] = -\mathbf{H}[k]^{-1} \nabla_{\mathbf{w}} J[k] \simeq - \left(\sum_{n=1}^N \mathbf{Y}^T[n] \mathbf{Y}[n] \right)^{-1} \bigg|_{\mathbf{w}=\mathbf{w}[k]} \cdot \sum_{n=1}^N \mathbf{Y}^T[n] \mathbf{e}[n] \bigg|_{\mathbf{w}=\mathbf{w}[k]}$$

See next slide...

- The new updating equation in backpropagation is...

$$\mathbf{w}[k+1] = \mathbf{w}[k] - \eta \mathbf{H}[k]^{-1} \nabla_{\mathbf{w}} J[k]$$

$\mathbf{Y}[n]$ is the Jacobian Matrix

$$\mathbf{e}[n] = \mathbf{t}[n] - \mathbf{z}[n]$$



Computation of the gradient and Hessian:

$$J = \frac{1}{2} \sum_{n=1}^N \mathbf{e}^T[n] \mathbf{e}[n]$$

$$\nabla_{\mathbf{w}} J[k] = \sum_{n=1}^N \frac{\partial \mathbf{e}^T[n]}{\partial \mathbf{w}} \mathbf{e}[n] \Big|_{\mathbf{w}=\mathbf{w}[k]} = \sum_{n=1}^N \begin{pmatrix} \frac{\partial e_1[n]}{\partial w_1} & \frac{\partial e_2[n]}{\partial w_1} & \dots & \frac{\partial e_c[n]}{\partial w_1} \\ \frac{\partial e_1[n]}{\partial w_2} & \frac{\partial e_2[n]}{\partial w_2} & \dots & \frac{\partial e_c[n]}{\partial w_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial e_1[n]}{\partial w_L} & \frac{\partial e_2[n]}{\partial w_L} & \dots & \frac{\partial e_c[n]}{\partial w_L} \end{pmatrix} \begin{pmatrix} e_1[n] \\ e_2[n] \\ \vdots \\ e_c[n] \end{pmatrix} = \sum_{n=1}^N \mathbf{Y}^T[n] \mathbf{e}[n] \Big|_{\mathbf{w}=\mathbf{w}[k]}$$

$$\mathbf{H}[k] = \nabla_{\mathbf{w}} \sum_{n=1}^N \mathbf{e}^T[n] \mathbf{Y}[n] \Big|_{\mathbf{w}=\mathbf{w}[k]} = \sum_{n=1}^N \left(\mathbf{Y}^T[n] \mathbf{Y}[n] + [\mathbf{A}_1[n] \mathbf{e}[n] \quad \mathbf{A}_2[n] \mathbf{e}[n] \quad \dots \quad \mathbf{A}_L[n] \mathbf{e}[n]] \right)$$

$$\cong \sum_{n=1}^N \mathbf{Y}^T[n] \mathbf{Y}[n] \Big|_{\mathbf{w}=\mathbf{w}[k]}$$

Derivatives computed from backpropagation equations (check slides 32 and 33)

Second derivatives of the error w.r.t. \mathbf{w} appear on each $\mathbf{A}_i[n]$

... assuming that errors are zero-mean and independent of the data $\mathbf{x}[n]$

$$\sum_{n=1}^N \mathbf{A}_i[n] \mathbf{e}[n] \cong N \cdot E \{ \mathbf{A}_i[n] \mathbf{e}[n] \} = N \cdot E \{ \mathbf{A}_i[n] \} E \{ \mathbf{e}[n] \} = \mathbf{0}$$

As Gauss-Newton may get trapped in saddle points or in maxima, Levenberg-Marquardt (LM) algorithm is defined, as a blend of Gradient Descent and Gauss-Newton methods:

- **Gradient descent** (safe and slow):

$$\mathbf{w}[k+1] = \mathbf{w}[k] - \eta \nabla_{\mathbf{w}} J[k]$$

- **Gauss-Newton** (quick for nearly quadratic surfaces)

$$\mathbf{w}[k+1] = \mathbf{w}[k] - \eta \mathbf{H}[k]^{-1} \nabla_{\mathbf{w}} J[k]$$

- **LM** (augmented μ , better general convergence)

$$\mathbf{w}[k+1] = \mathbf{w}[k] - \eta (\mathbf{H}[k] + \mu \mathbf{I})^{-1} \nabla_{\mathbf{w}} J[k]$$

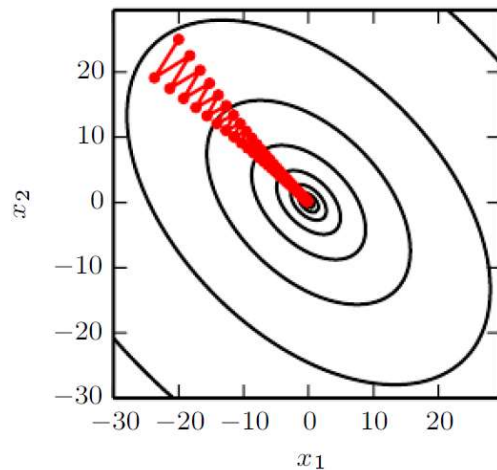
Rule of thumb: If at a given step k the error function J increases, increase μ by a factor of 10 (move in the gradient direction). If J decreases, do the opposite: decrease μ by a factor of 10 (move in the Gauss-Newton direction)

- **Momentum method**

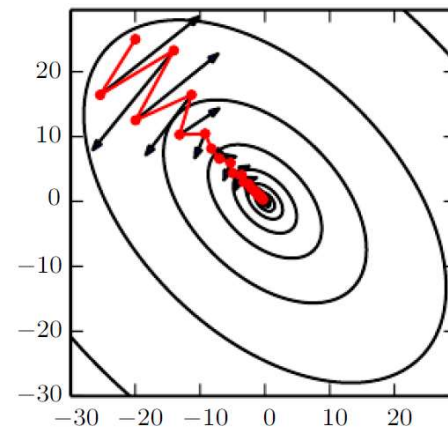
This method pushes the weights towards the minimum using some memory on the past gradient values:

$$\mathbf{v}[n+1] = \alpha \mathbf{v}[n] - \eta \left. \frac{\partial J}{\partial \mathbf{w}} \right|_{\mathbf{w}=\mathbf{w}[n]}$$
$$\mathbf{w}[n+1] = \mathbf{w}[n] + \mathbf{v}[n+1]$$

It is able to accelerate learning in high curvature zones, small gradients or noisy gradients. Typical values for α are 0.5, 0.9 and 0.99.



Without momentum



With momentum

- **Nesterov momentum**

A way to accelerate the gradient is by using the update rule:

$$\begin{aligned}\mathbf{v}[n+1] &= \alpha \mathbf{v}[n] - \eta \frac{\partial J}{\partial \mathbf{W}} \bigg|_{\mathbf{w}=\mathbf{w}[n]+\alpha \mathbf{v}[n]} \\ \mathbf{w}[n+1] &= \mathbf{w}[n] + \mathbf{v}[n+1]\end{aligned}$$

The gradient is evaluated after the current update is applied on the weights. It further improves the rate of convergence of momentum methods but only for batch or mini-batch gradient algorithms.

These algorithms adapt the learning rate individually for each parameter...

AdaGrad (Duchi, 2011)

The learning rate is taken as inversely proportional to the accumulated magnitude of the gradient...

$$\mathbf{g}_k = \nabla_{\Theta} L|_{\Theta=\Theta_k}$$

Element-wise vector product

$$\mathbf{r}_{k+1} = \mathbf{r}_k + \mathbf{g}_k \odot \mathbf{g}_k$$

$$\Theta_{k+1} = \Theta_k - \frac{\varepsilon}{\delta + \sqrt{\mathbf{r}_{k+1}}} \odot \mathbf{g}_k$$

Small value, avoids divisions by zero

The learning rate is shrunk based on the entire history of the gradient.

RMSProp (Hinton, 2012)

Same as AdaGrad but with an exponentially weighted average of gradient...

$$\begin{aligned}\mathbf{g}_k &= \nabla_{\Theta} L|_{\Theta=\Theta_k} \\ \mathbf{r}_{k+1} &= \rho \mathbf{r}_k + (1 - \rho) \mathbf{g}_k \odot \mathbf{g}_k \\ \Theta_{k+1} &= \Theta_k - \frac{\varepsilon}{\delta + \sqrt{\mathbf{r}_{k+1}}} \odot \mathbf{g}_k\end{aligned}$$

where $0 < \rho < 1$. Initial values of the gradient are not relevant when we are close to a minimum. Better performance. The concept can be applied to momentum algorithms.

Adam (Kingma, 2014)

Accelerates the gradient algorithm by introducing unbiased mean and correlation of the gradient...

$$\mathbf{g}_k = \nabla_{\Theta} L|_{\Theta=\Theta_k}$$

$$\mathbf{s}_{k+1} = \rho_1 \mathbf{s}_k + (1 - \rho_1) \mathbf{g}_k$$

$$\mathbf{r}_{k+1} = \rho_2 \mathbf{r}_k + (1 - \rho_2) \mathbf{g}_k \odot \mathbf{g}_k$$

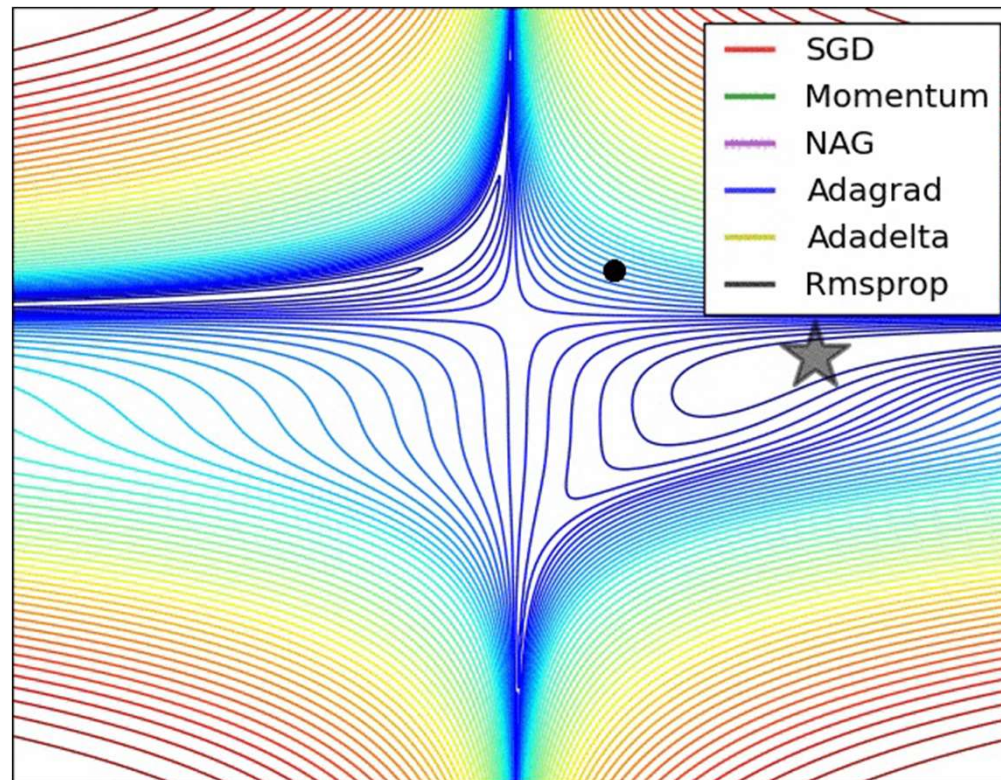
$$\hat{\mathbf{s}} = \mathbf{s}_{k+1} / (1 - \rho_1^k)$$

$$\hat{\mathbf{r}} = \mathbf{r}_{k+1} / (1 - \rho_2^k)$$

$$\Theta_{k+1} = \Theta_k - \frac{\varepsilon}{\delta + \sqrt{\hat{\mathbf{r}}}} \odot \hat{\mathbf{s}}$$

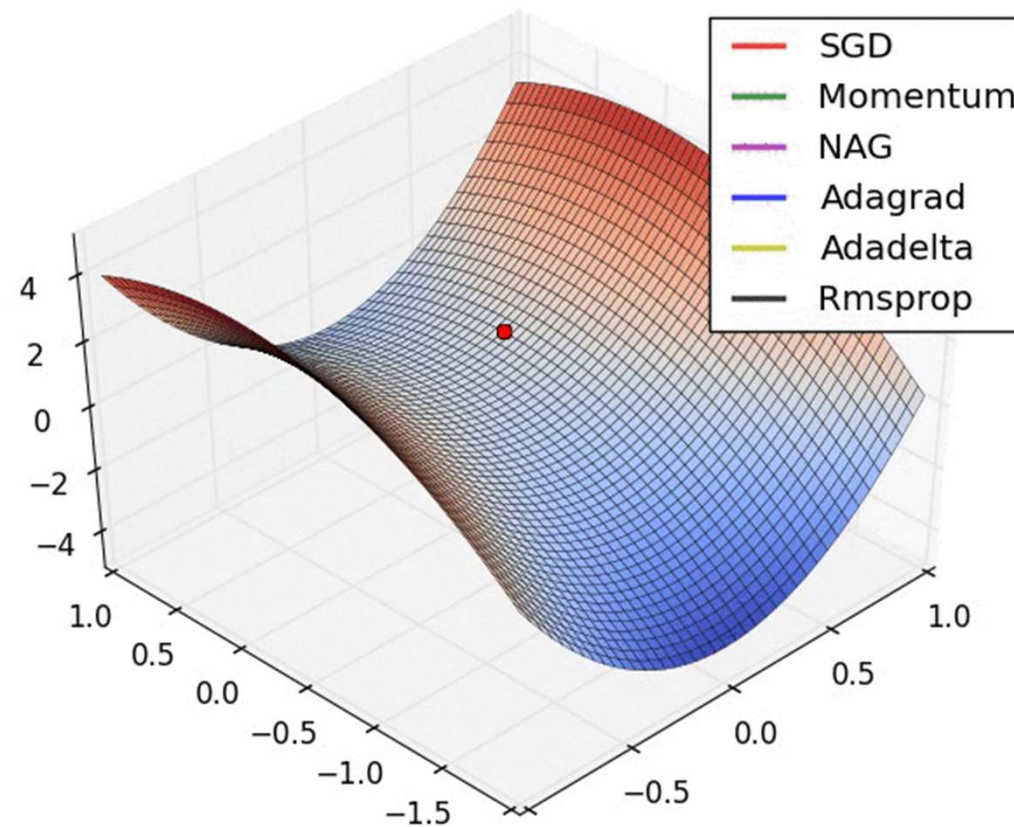
It is quite robust to the selection of parameters.

Behaviour of several algorithms over time towards the minimum



By S. Ruder in <https://ruder.io/optimizing-gradient-descent/>

Behaviour of several algorithms at a saddle point



By S. Ruder in <https://ruder.io/optimizing-gradient-descent/>

8 . Rules for the improved convergence

- **Scaling input**
 - Average of each feature over the training set is zero.
 - The full data should be scaled to have the same variance in each feature component (1 with the sigmoid function).
 - Any test set must be subjected to the same transformation before its classification.
- **Output values for classification** +1, 0 for normalization purposes.
- Gaussian d-dimensional noise added to increase the train set size.
- **Number of hidden units h**
 - It represents the number of degrees of freedom and governs the expressive power of the NN.
 - Few hidden units are needed if the patterns are well separated.
 - Better choose a large value ($h < N/10$) when using regularization in the optimisation.

- Initializing weights
 - Different to zero
 - Uniform learning (all weights reach final equilibrium simultaneously) and all category are learnt at the same time
 - Choose them randomly from a uniform distribution $(-W, +W)$ with small initial values
 - Input-to-hidden: choose $W = 1/\sqrt{d}$
 - Hidden-to-output: choose $W = 1/\sqrt{h}$
- Other iterative solutions:
 - Nelder-Mead
 - Conjugate gradient method

9. The power of neural networks

1. Universal function approximation

Kolmogorov Theorem: Any continuous function $g(\mathbf{x})$ defined on the unit hypercube $I^d = [0,1]$, $d \geq 2$ can be represented as:

$$g(\mathbf{x}) = \sum_{j=1}^h \Xi \left(\sum_{i=1}^d \psi_{ij}(x_i) \right)$$

with properly chosen functions $\Xi_j \quad \psi_{ij} = \lambda^i \psi(x_i + j\varepsilon) + j$

Practical conclusions

Choose λ and ψ depend on d and are independent of the mapping function. Function Ξ depends on the mapping function to be implemented.

More theoretical than practical interest

A. Kowalczyk, “Can multilayer mapping networks with finite number of real parameters harness the computational power of Kolmogorov's representation theorem?”, *IEEE International Joint Conference on Neural Networks*, Nov 1991. Page(s): 2722-2728, vol.3

G. Cybenko, “Approximation by superpositions of a sigmoidal function”, *Mathematics of control, signals and systems*, 2(4):303–314, 1989

2. Backpropagation and Bayesian decision

MLNN trained with MMSE criterion provide a least squares fit to the Bayes discriminant functions:

- Assume for simplicity in the development $t_k = \begin{cases} +1 & \mathbf{x} \in \text{class } k \\ 0 & \mathbf{x} \notin \text{class } k \end{cases}$
- The contribution of a single output unit $g_k(\mathbf{x})$ to the error J is

$$\begin{aligned} J(\mathbf{w}) &= \sum_{\mathbf{x}} \left(\overset{z_k}{g_k(\mathbf{x}; \mathbf{w})} - t_k \right)^2 = \sum_{\mathbf{x} \in \omega_k} (g_k(\mathbf{x}; \mathbf{w}) - 1)^2 + \sum_{\mathbf{x} \notin \omega_k} (g_k(\mathbf{x}; \mathbf{w}) - 0)^2 = \\ &= n \left\{ \frac{n_k}{n} \frac{1}{n_k} \sum_{\mathbf{x} \in \omega_k} (g_k(\mathbf{x}; \mathbf{w}) - 1)^2 + \frac{n - n_k}{n} \frac{1}{n - n_k} \sum_{\mathbf{x} \notin \omega_k} (g_k(\mathbf{x}; \mathbf{w}) - 0)^2 \right\} \end{aligned}$$



In the limiting case, the terms in the previous slide can be written as
(**prove it as an exercise**):

$$\begin{aligned}\lim_{n \rightarrow \infty} \frac{1}{n} J(\mathbf{w}) &= \Pr(\omega_k) \int (g_k(\mathbf{x}; \mathbf{w}) - 1)^2 f(\mathbf{x} | \omega_k) d\mathbf{x} + \Pr(\omega_{i \neq k}) \int g_k(\mathbf{x}; \mathbf{w})^2 f(\mathbf{x} | \omega_{i \neq k}) d\mathbf{x} = \\ &= \int g_k(\mathbf{x}; \mathbf{w})^2 f(\mathbf{x}) d\mathbf{x} - 2 \int g_k(\mathbf{x}; \mathbf{w}) f(\mathbf{x}, \omega_k) d\mathbf{x} + \int f(\mathbf{x}, \omega_k) d\mathbf{x} = \\ &= \int (g_k(\mathbf{x}; \mathbf{w}) - \Pr(\omega_k | \mathbf{x}))^2 f(\mathbf{x}) d\mathbf{x} + \underbrace{\int \Pr(\omega_k | \mathbf{x}) \Pr(\omega_{i \neq k} | \mathbf{x}) f(\mathbf{x}) d\mathbf{x}}_{\text{Independent of } \mathbf{w}}\end{aligned}$$

The MMSE rule minimizes the left hand side integral, and hence

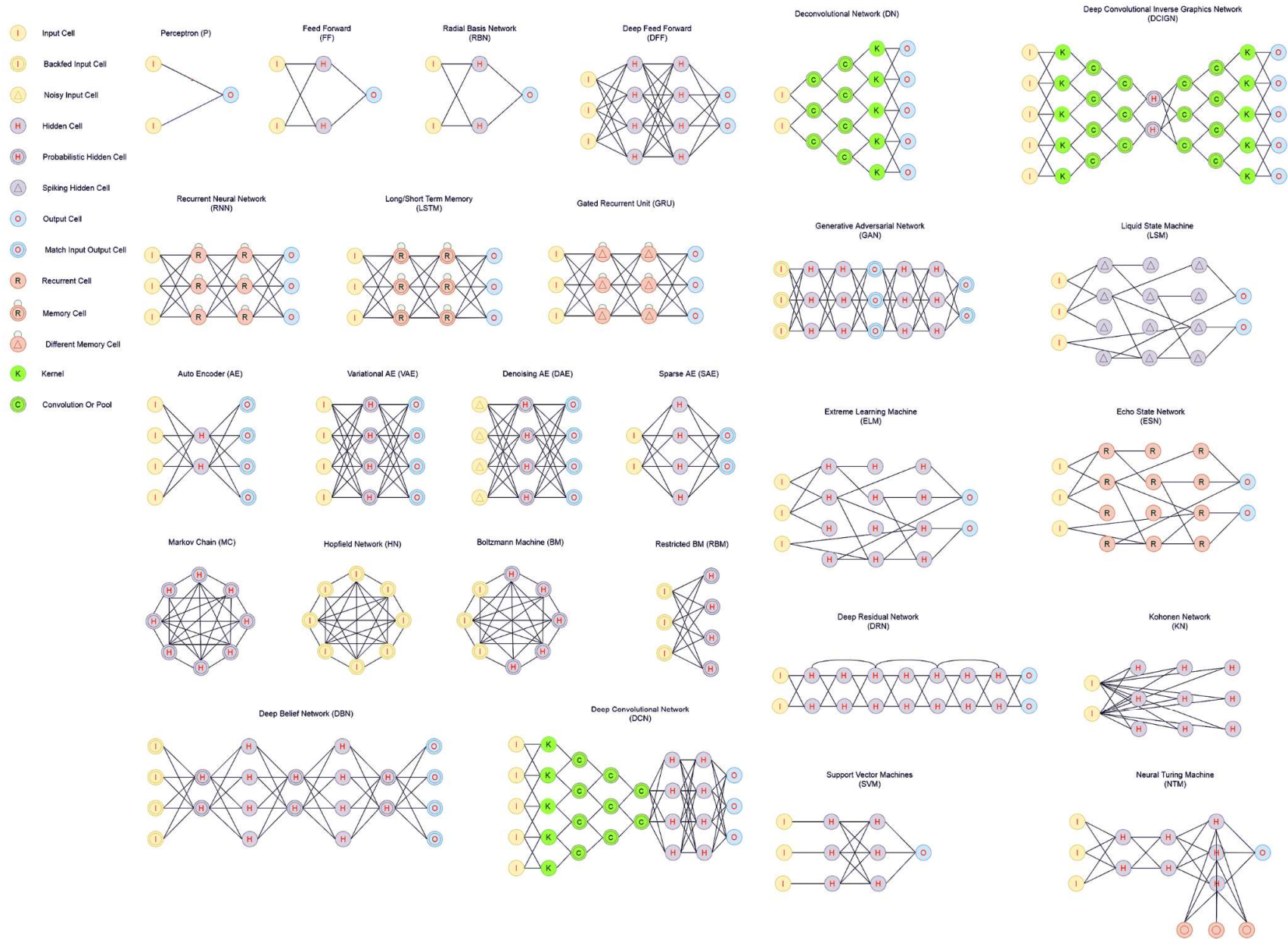
$$g_k(\mathbf{x}; \mathbf{w}) \cong \Pr(\omega_k | \mathbf{x})$$

if the number of training data n is large enough and the number of hidden units is large enough to represent the a-posteriori probability.

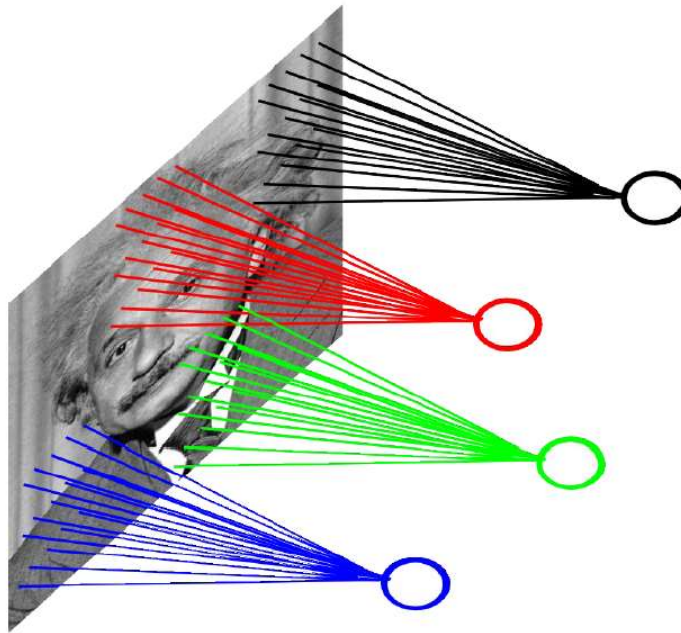
10. Other NN architectures

- Convolutional neural networks
- Recurrent neural networks
- Long short-term memory neural networks
- Autoencoders
- Generative neural networks
- Hopfield networks

... and many more



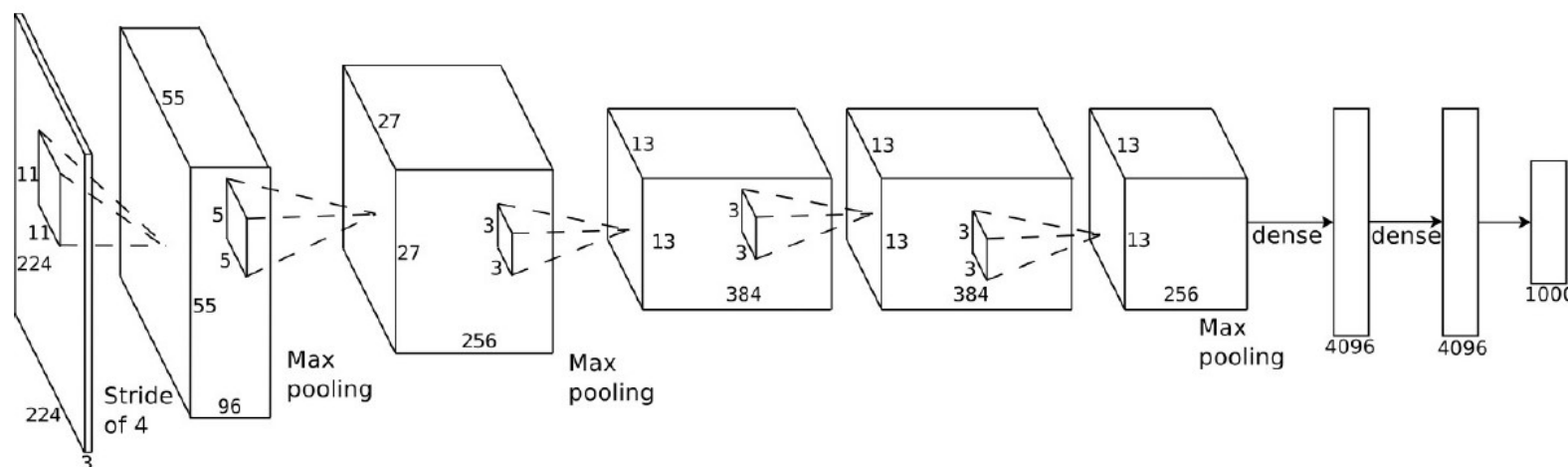
- Convolutional neural networks



2D CNN (Figure credit: Ranzatto)

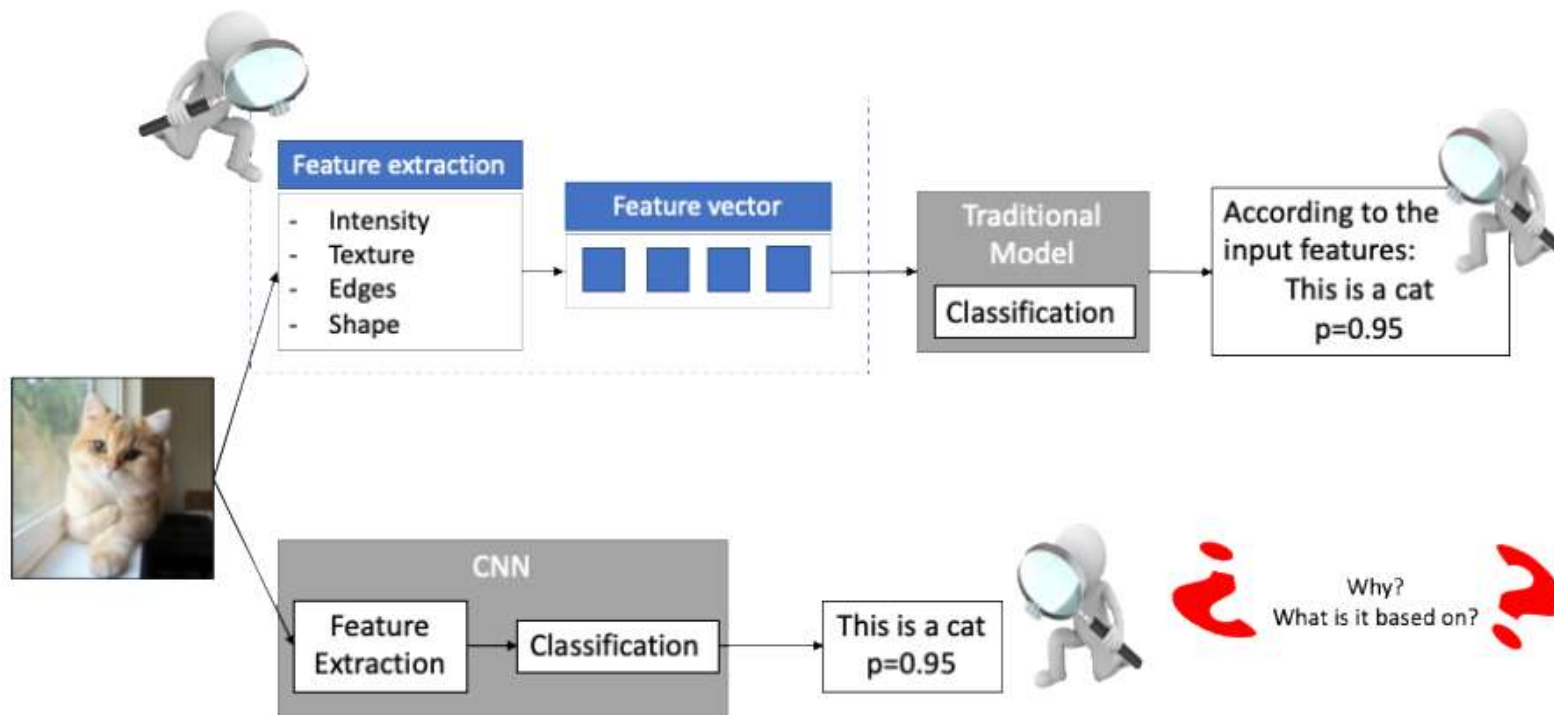
- **Convolutional neural networks**

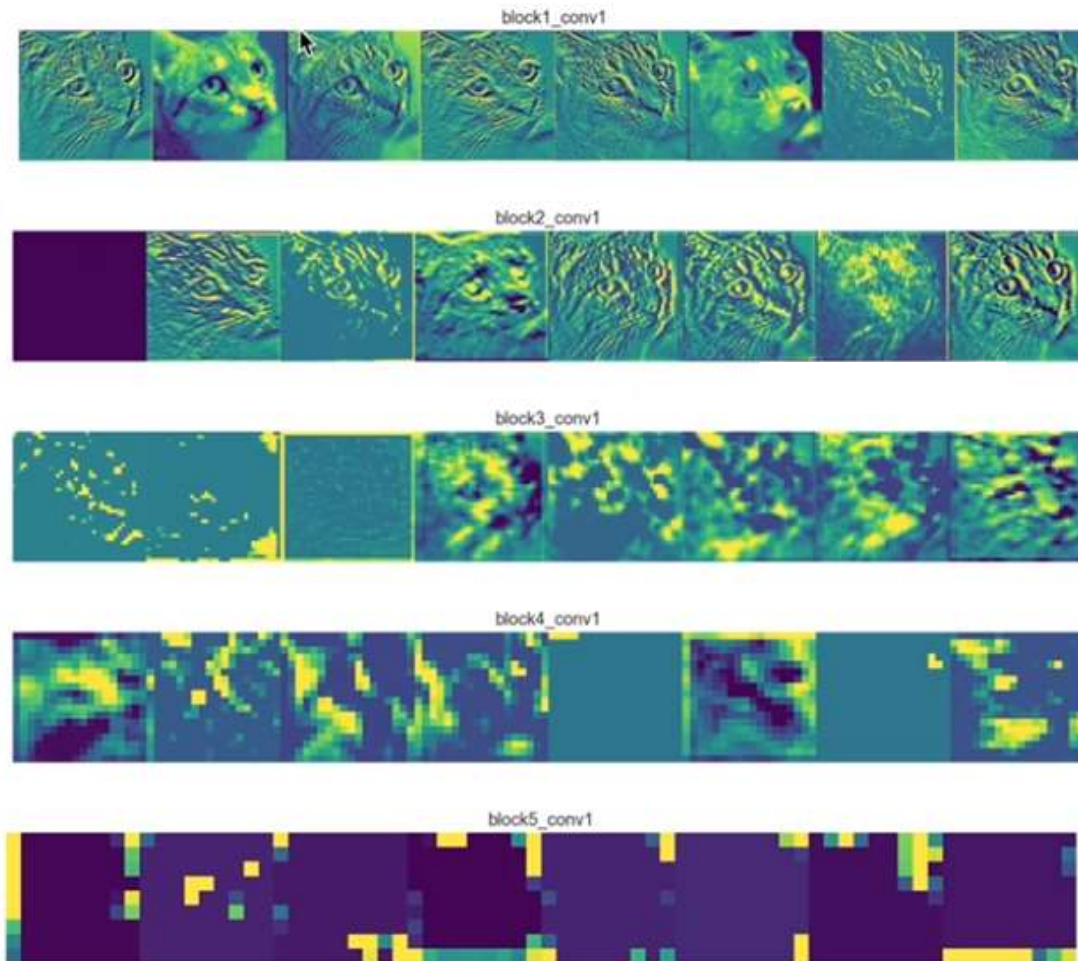
Rather than seeking a limited number of features, let us allow the network combine pixels in the most appropriate way and finding the best features. For a picture recognition task, with 1000 classes:



The number of neurons in each layer is given by 253440, 186624, 64896, 64896, 43264, 4096, 4096, 1000.

Training is hard, only if very large database is available.



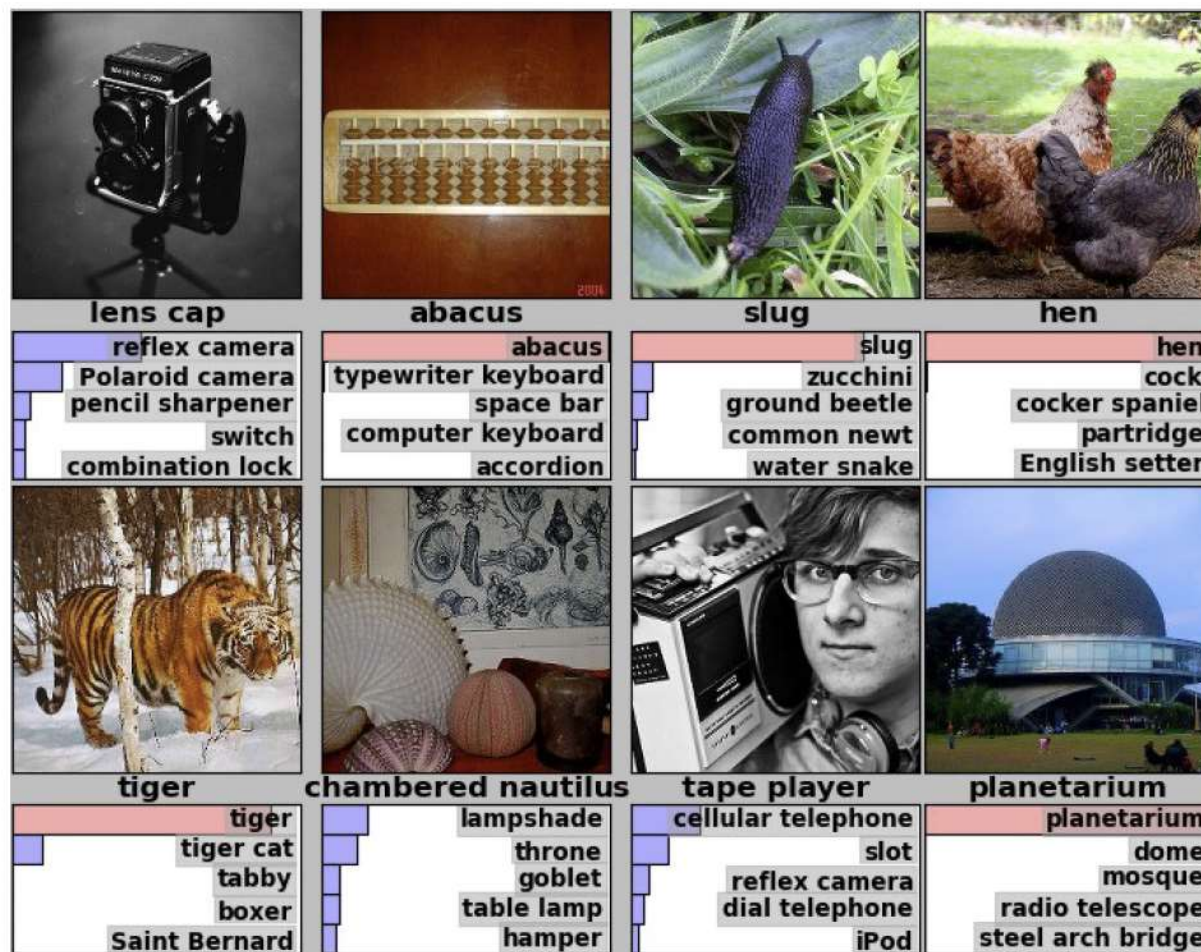


Better performance than any other method based on feature extraction on images!

Method	mean Accuracy
HSV [27]	43.0
SIFT internal [27]	55.1
SIFT boundary [27]	32.0
HOG [27]	49.6
HSV+SIFTi+SIFTb+HOG(MKL) [27]	72.8
BOW(4000) [14]	65.5
SPM(4000) [14]	67.4
FLH(100) [14]	72.7
BiCos seg [7]	79.4
Dense HOG+Coding+Pooling[2] w/o seg	76.7
Seg+Dense HOG+Coding+Pooling[2]	80.7
CNN-SVM w/o seg	74.7
CNNaug-SVM w/o seg	86.8

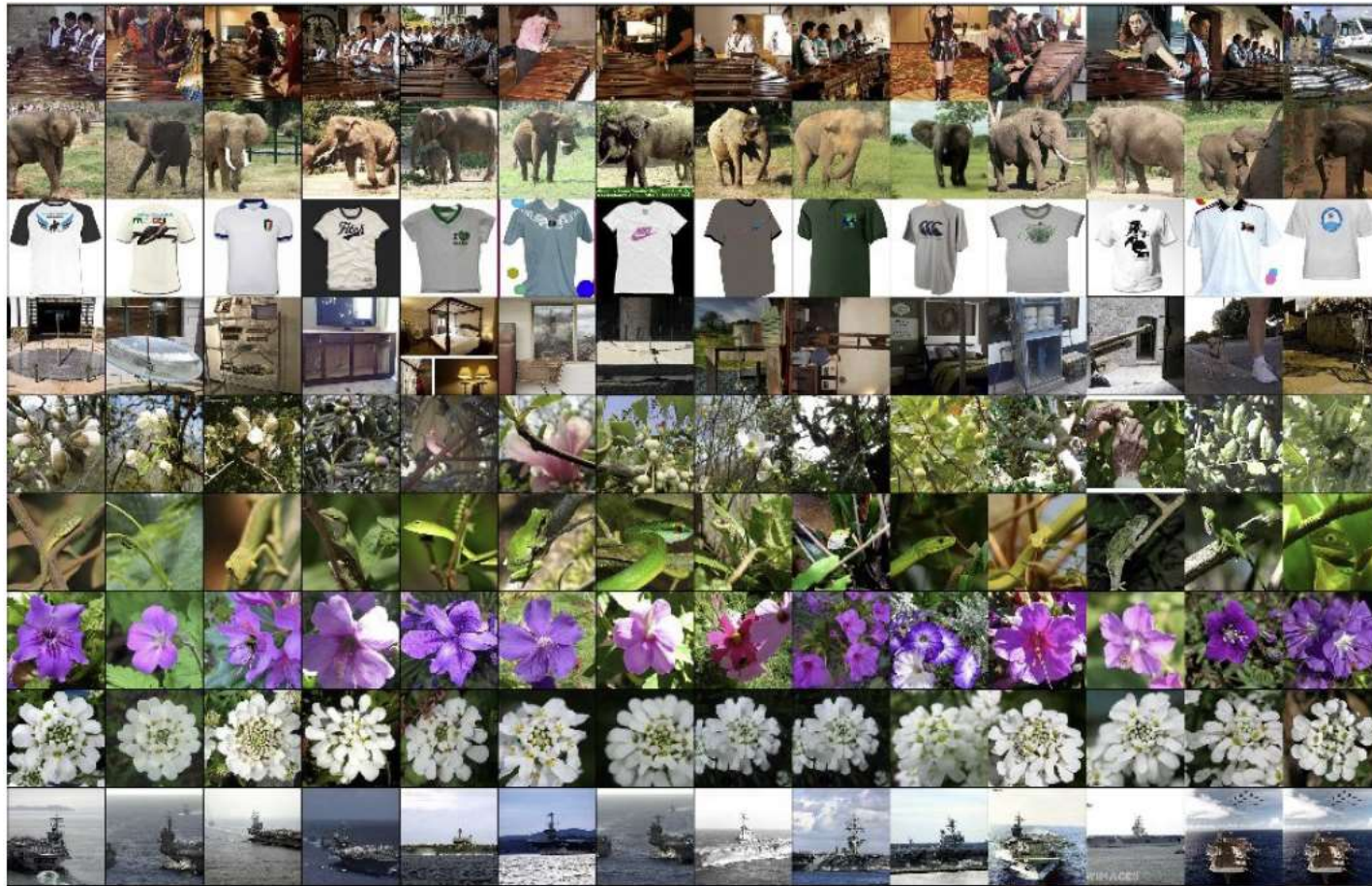
Oxford 102 flowers dataset

Validation classification in MNIST task

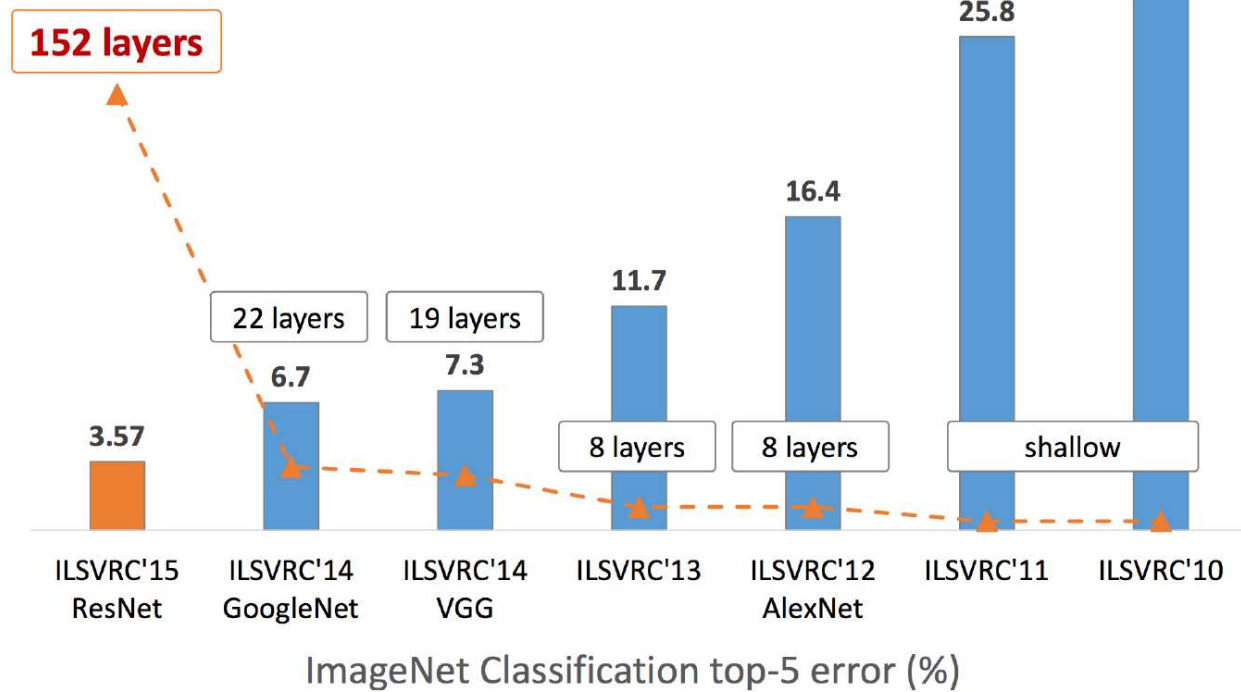


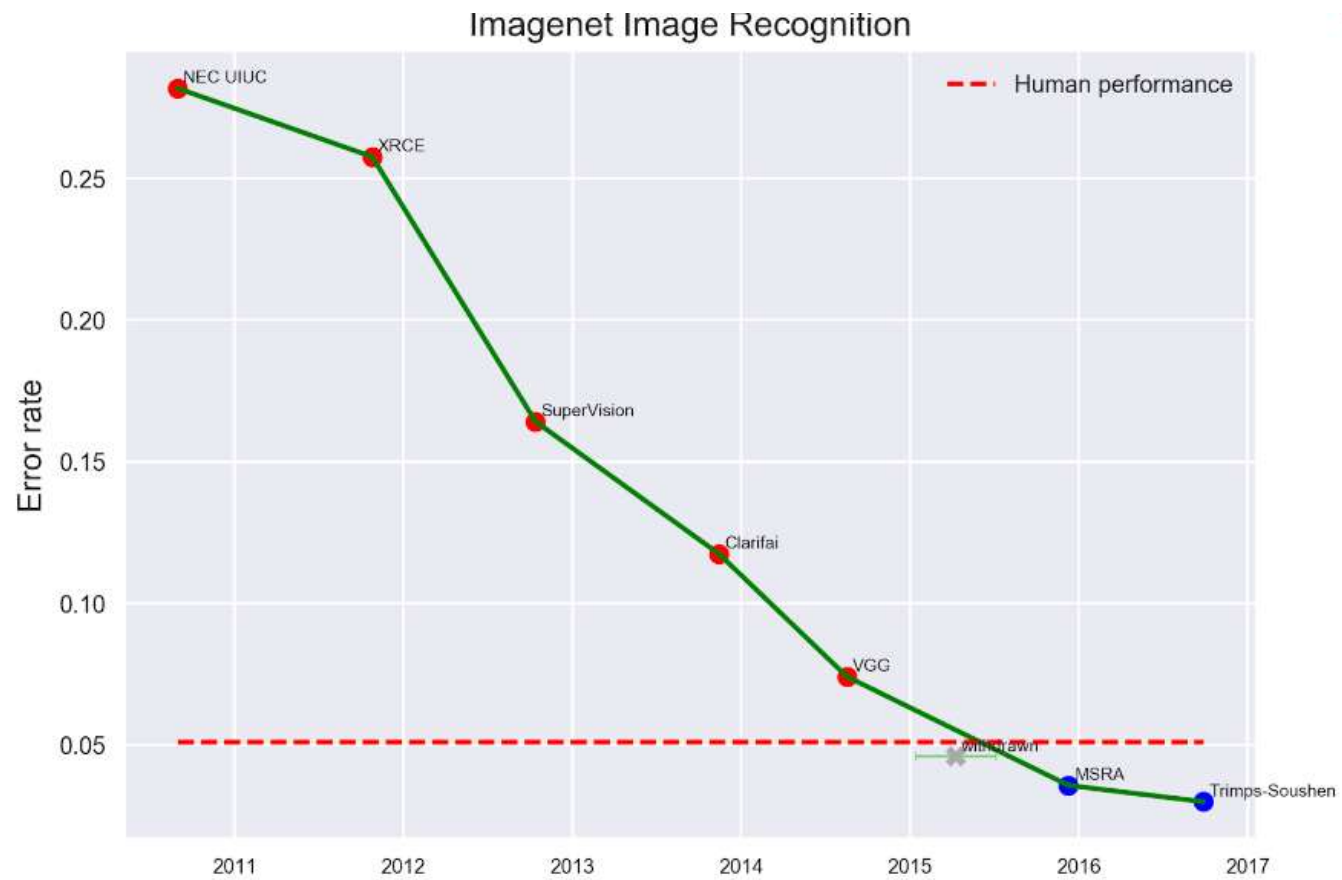
Retrieval experiments...

First column contains query images from ILSVRC-2010 test set, remaining columns contain retrieved images from training set.



Revolution of Depth



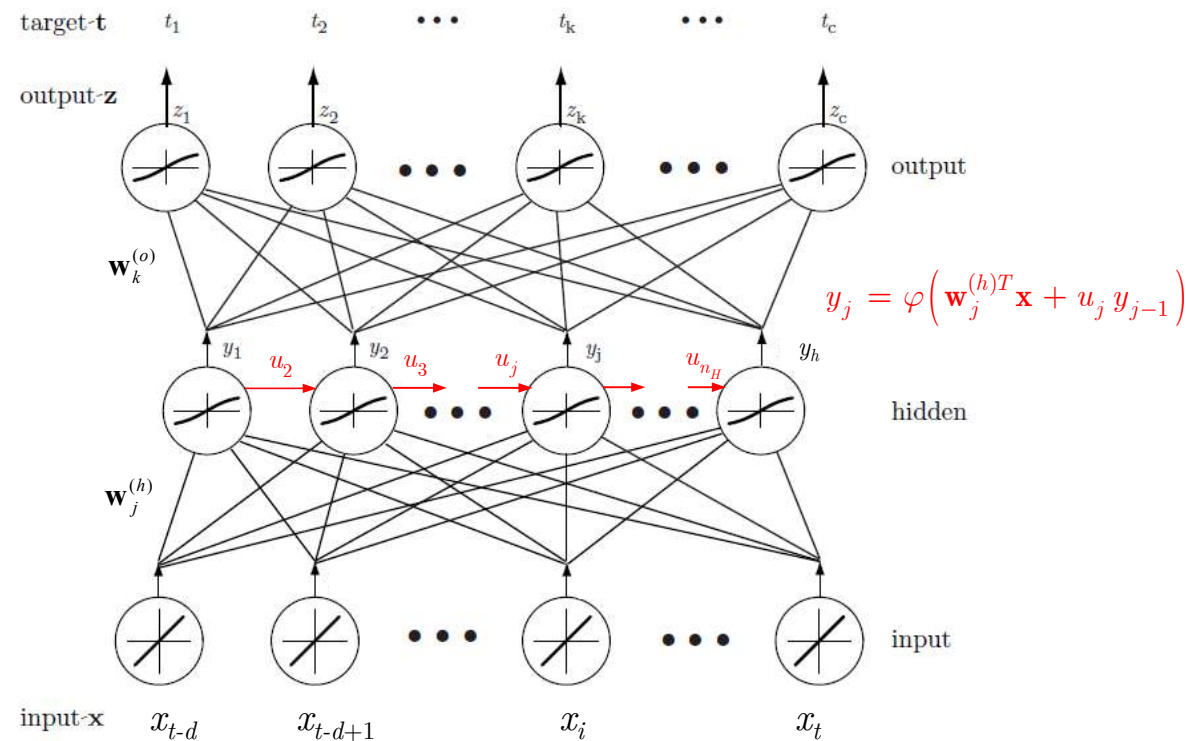


11. Other NN architectures

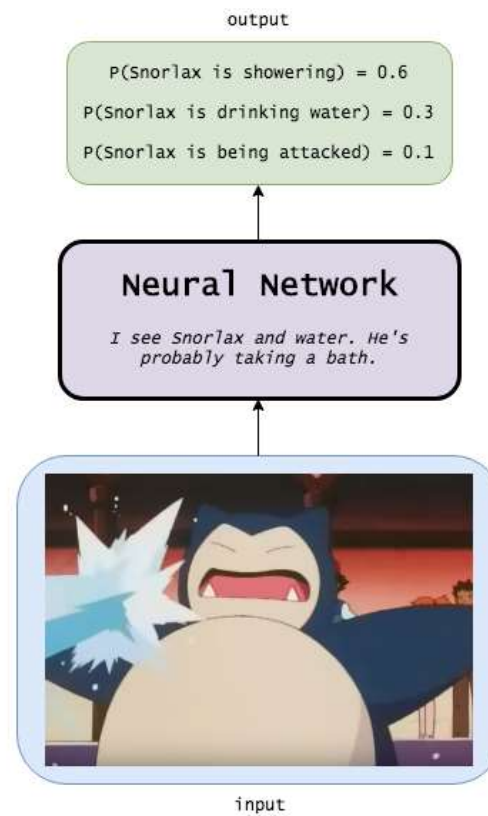
- Convolutional neural networks
- **Recurrent neural networks**
- Long short-term memory (LSTM) neural networks
- Autoencoders
- Generative neural networks
- Hopfield networks

- **Recurrent neural networks**

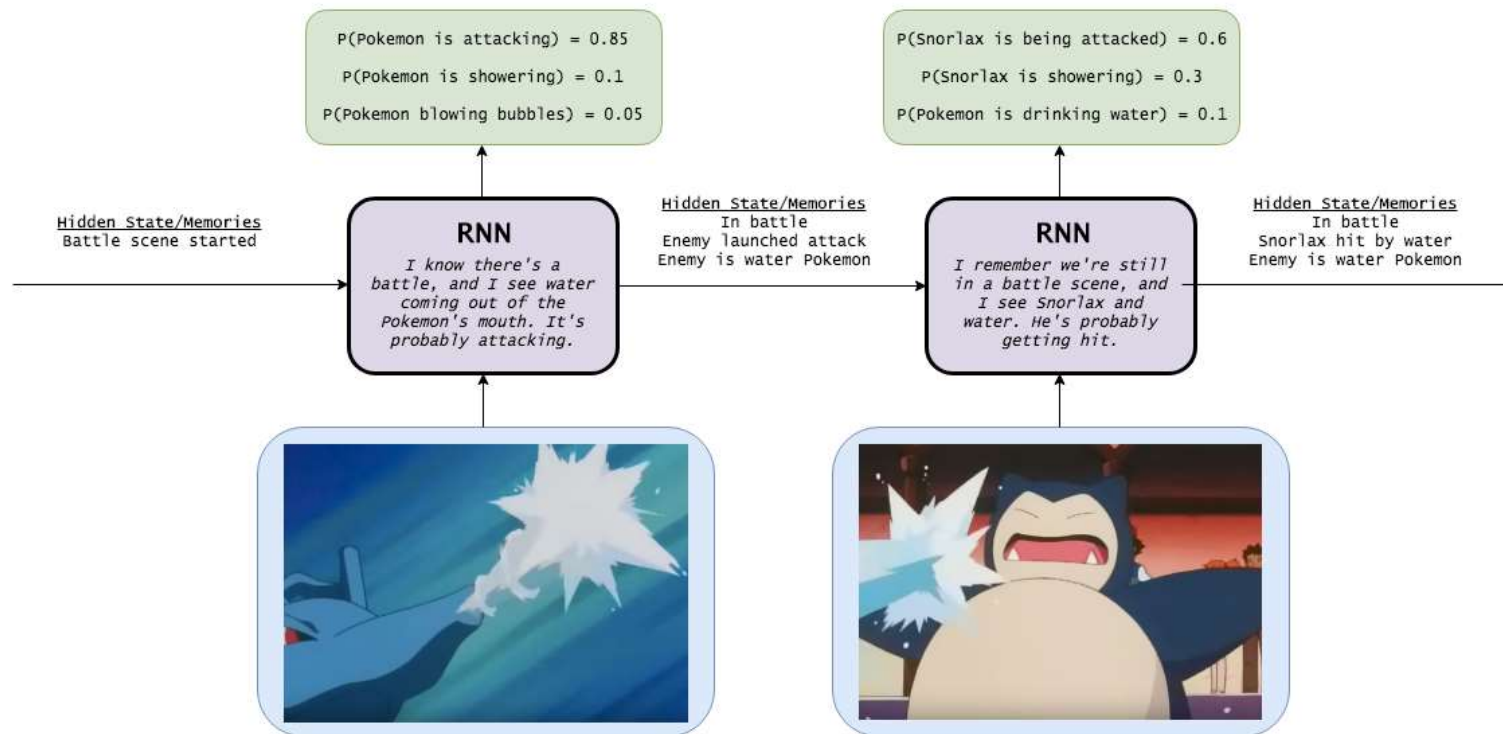
In some problems, like video analysis, inputs are signals in time. The hidden layers of neural networks encode useful information, so why not use these hidden layers as the memories we pass from one time step to the next?



In conventional NN...



In recurrent NN...



Takeaway message

Useful for non-linear classification problems...

- The best NN topology is tightly related to the nature of the problem
- Number of neurons and other parameters, obtained as a function of the training database

The back-propagation algorithm (or its variants) computes the approximation to the Bayes discriminant function if the number of training samples goes to infinity.

Learn more: Deep Learning course at NYU by Yann LeCun & Alfredo Canziani

[DEEP LEARNING · Deep Learning \(atcold.github.io\)](#)

[\(26\) Deep Learning \(with PyTorch\) - YouTube](#)