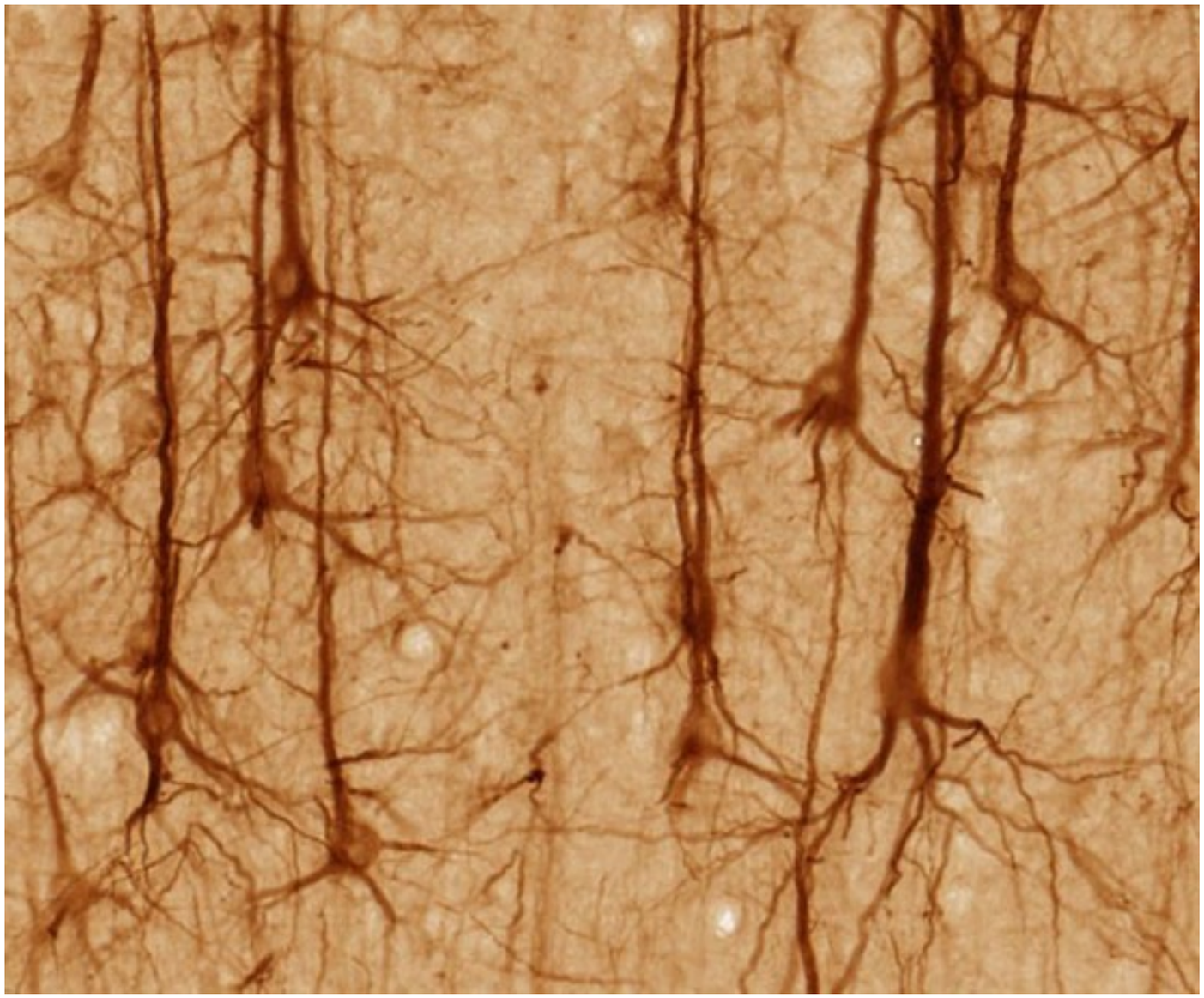
Neural Networks

B. Mehlig, Department of Physics, University of Gothenburg, Sweden

Thanks to E. Werner and H. Linander (Zenuity)

FFR135 Artificial Neural Networks Chalmers/Gothenburg University, 7.5 credits

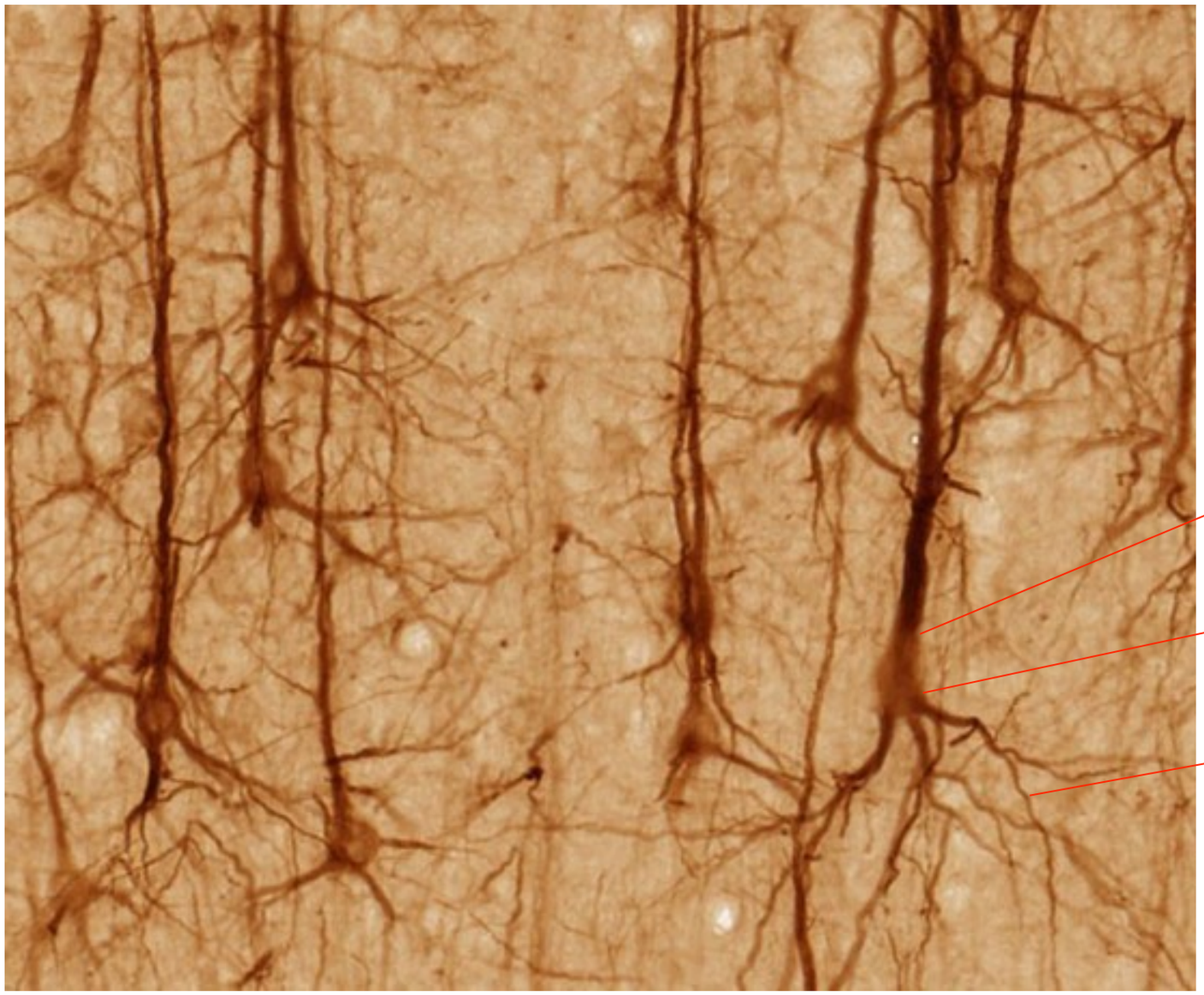
Neurons in the cerebral cortex



brainmaps.org

Neurons in the *cerebral cortex* (outer layer of the *cerebrum*, the largest and best developed part of the Human brain)

Neurons in the cerebral cortex

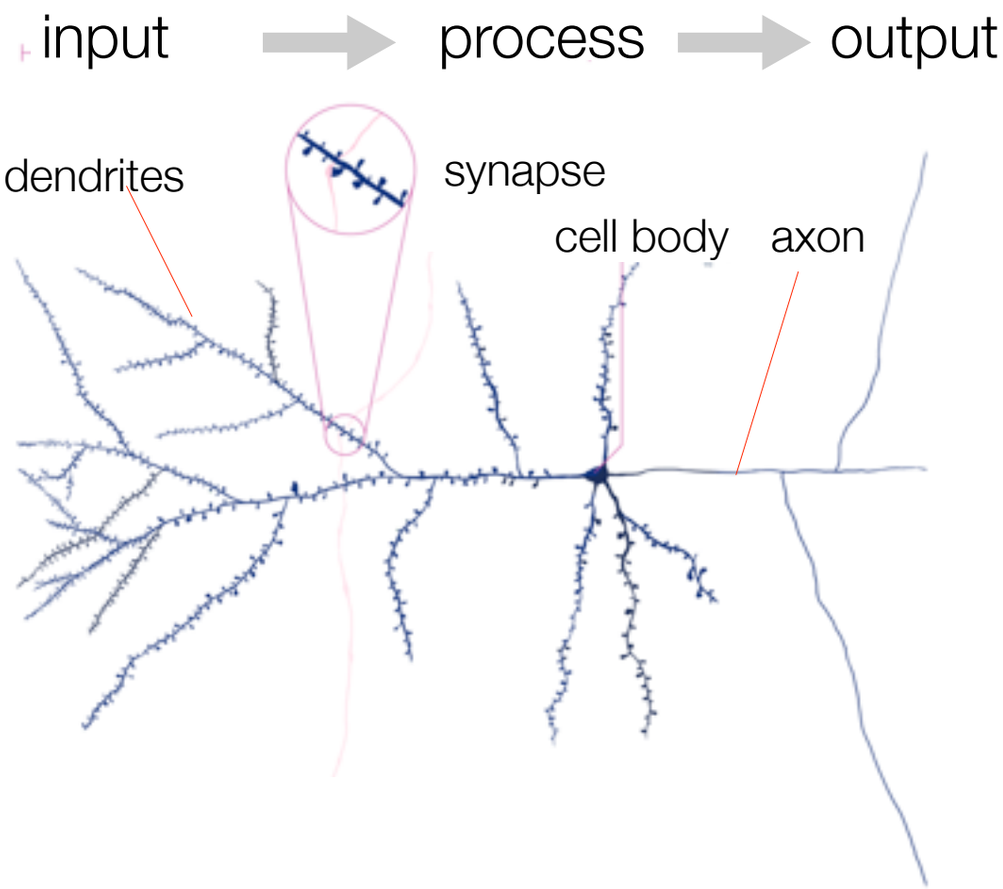


brainmaps.org

Neurons in the *cerebral cortex* (outer layer of the *cerebrum*, the largest and best developed part of the Human brain)

Neuron anatomy and activity

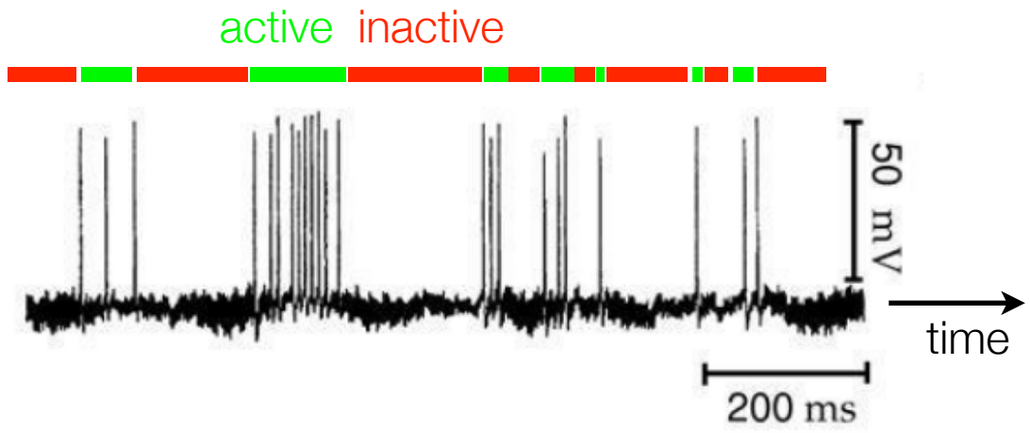
Schematic drawing of a neuron



Total length of dendrites
up to $\sim$ cm

synapseweb.clm.utexas.edu/dimensions-dendrites
wikipedia.org/wiki/Neuron

Output of a neuron: *spike train*



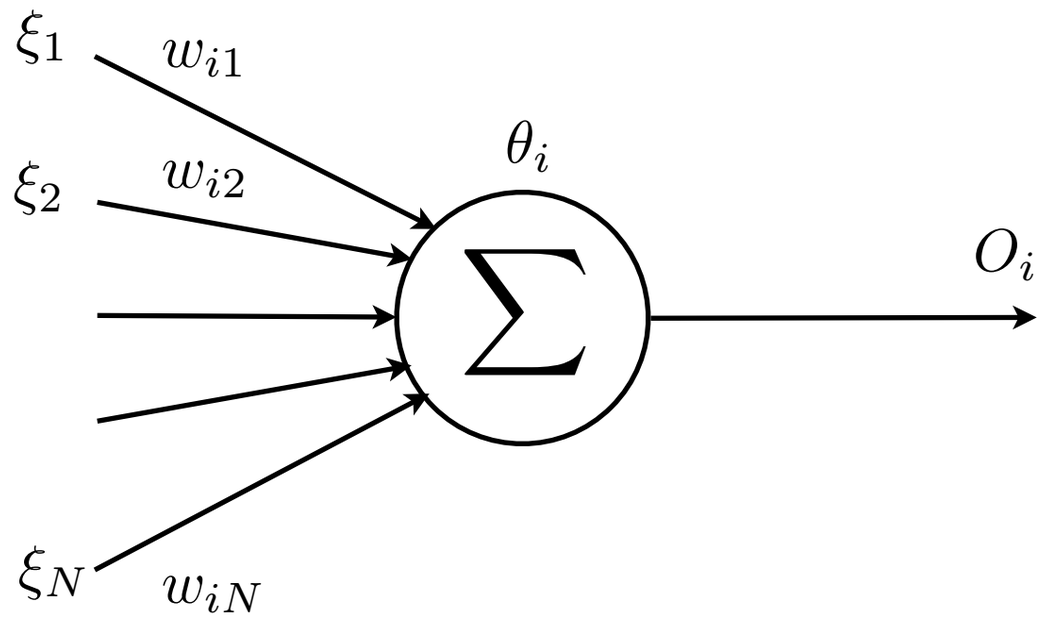
Spike train in electrosensory pyramidal neuron in fish (*Eigenmannia)*

Gabbiani & Metzner, J. Exp. Biol. **202** (1999) 1267

McCulloch-Pitts neuron

McCulloch & Pitts, Bull. Math. Biophys. **5** (1943) 115

Simple model for a neuron:



Incoming signals ξ_j $j=1,\ldots,N$

Weights w_{ij} (synaptic couplings)

Threshold θ_i Output O_i

Neuron number i

Signal processing: weighted sum of inputs
Activation function $g(b_i)$

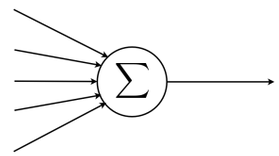
$$O_i = g\Big(\underbrace{\sum_{j=1}^{N} w_{ij}\xi_j}_{= b_i \text{ (local field)}} - \theta_i\Big)$$

$$= g\big(w_{i1}\xi_1 + \ldots + w_{iN}\xi_N - \theta_i\big)$$

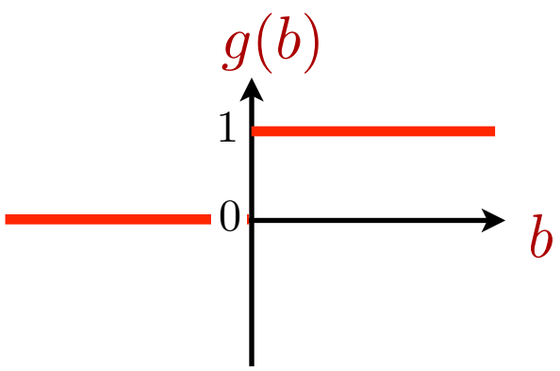
Activation function

Signal processing of McCulloch-Pitts neuron: weighted sum of inputs with activation function $g(b_i)$:

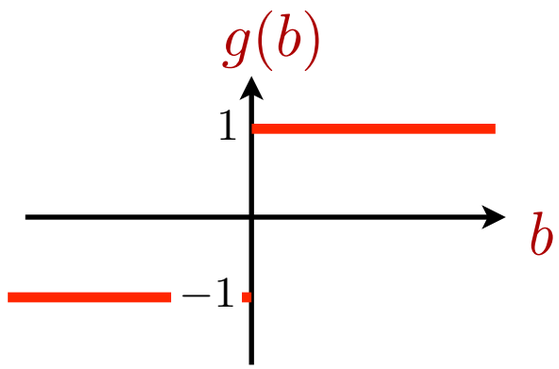
$$O_i = g\Big(\underbrace{\sum_{j=1}^{N} w_{ij}\xi_j - \theta_i}_{= b_i}\Big) = g\big(w_{i1}\xi_1 + \ldots + w_{iN}\xi_N - \theta_i\big)$$



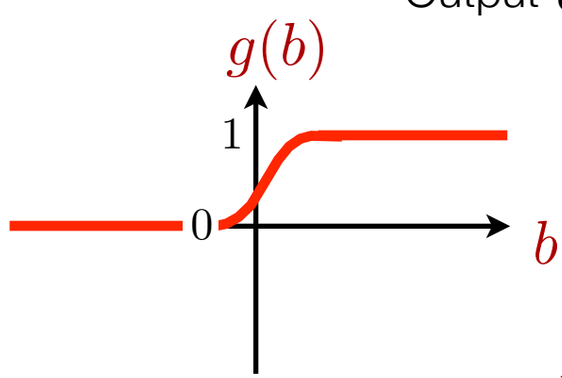
Inputs ξ_j
Synaptic couplings w_{ij}
Threshold θ_i
Output O_i



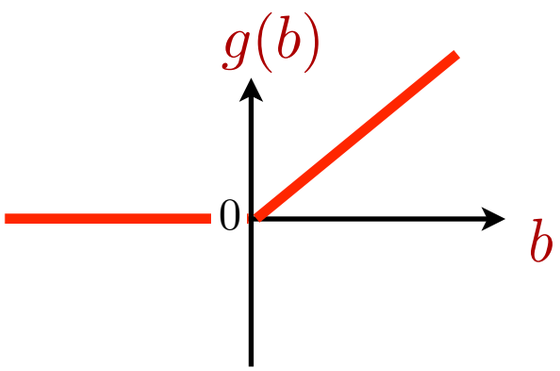
Step function $\theta(b)$



Signum function $\mathrm{sgn}(b)$



Sigmoid function $g(x) = \dfrac{1}{\mathrm{e}^{-\beta b} + 1}$
with parameter $\beta > 0$



ReLU function $\max(0, b)$

Highly simplified idealisation of a neuron

Take activation function to be signum function $\mathrm{sgn}(b_i)$.
Output of neuron i can assume only two values, ± 1,
representing the level of activity of the neuron.

$$\begin{cases} 1 & \textit{active} \\ -1 & \textit{inactive} \end{cases}$$

This is an idealisation.

-real neurons fire spike trains

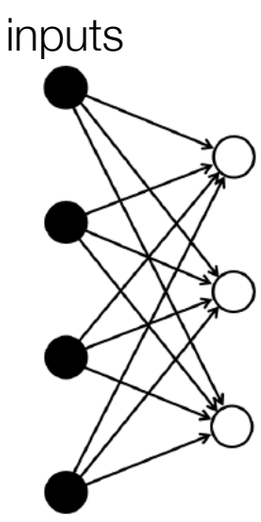
-switching *active* ⟷ *inactive* can be delayed (*time delay*)

-real neurons may perform non-linear summation of inputs

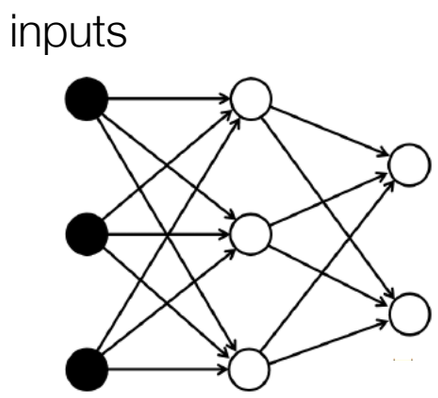
-real neurons subject to noise (*stochastic dynamics*)

Neural networks

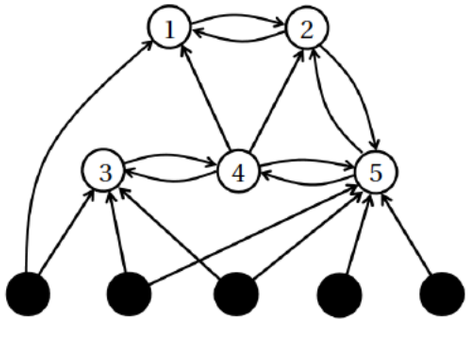
Connect neurons into networks that can perform computing tasks: for example image analysis, object identification, pattern recognition, classification, clustering, data compression.



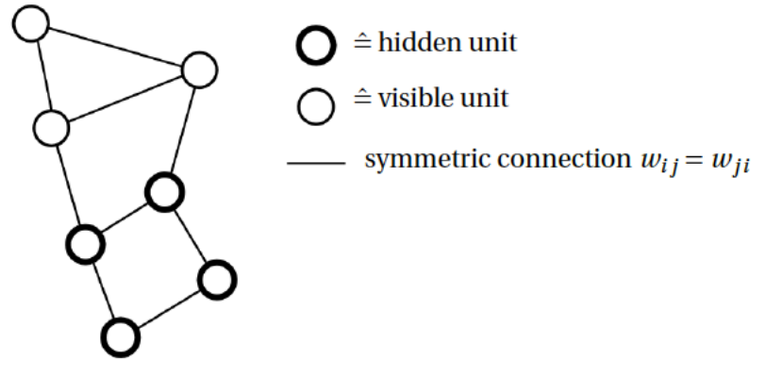
Simple perceptrons



Two-layer perceptron



Recurrent network



Boltzmann machine

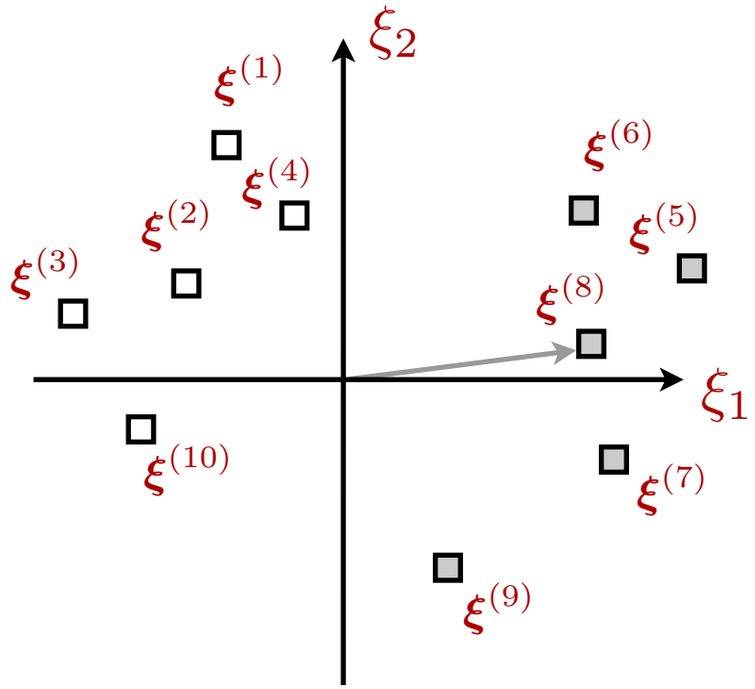
A classification task

Input patterns $\boldsymbol{\xi}^{(\mu)}$. Index $\mu = 1, \ldots, p$ labels patterns.

Each pattern has two components, $\xi_1^{(\mu)}$ and $\xi_2^{(\mu)}$.

Arrange components into vector, $\boldsymbol{\xi}^{(\mu)} = \begin{bmatrix} \xi_1^{(\mu)} \\ \xi_2^{(\mu)} \end{bmatrix}$, shown in the Figure.

We see: the patterns fall into two classes: □ on the left, and ■ on the right.



A classification task

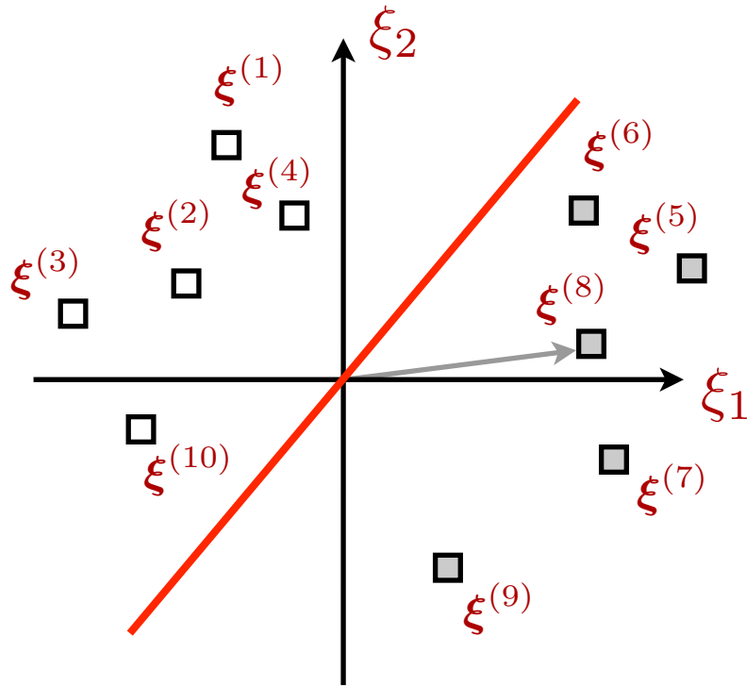
Input patterns $\boldsymbol{\xi}^{(\mu)}$. Index $\mu = 1, \ldots, p$ labels patterns.

Each pattern has two components, $\xi_1^{(\mu)}$ and $\xi_2^{(\mu)}$.

Arrange components into vector, $\boldsymbol{\xi}^{(\mu)} = \begin{bmatrix} \xi_1^{(\mu)} \\ \xi_2^{(\mu)} \end{bmatrix}$, shown in the Figure.

We see: the patterns fall into two classes: □ on the left, and ■ on the right.

Draw a red line (*decision boundary*) to distinguish the two types of patterns.



A classification task

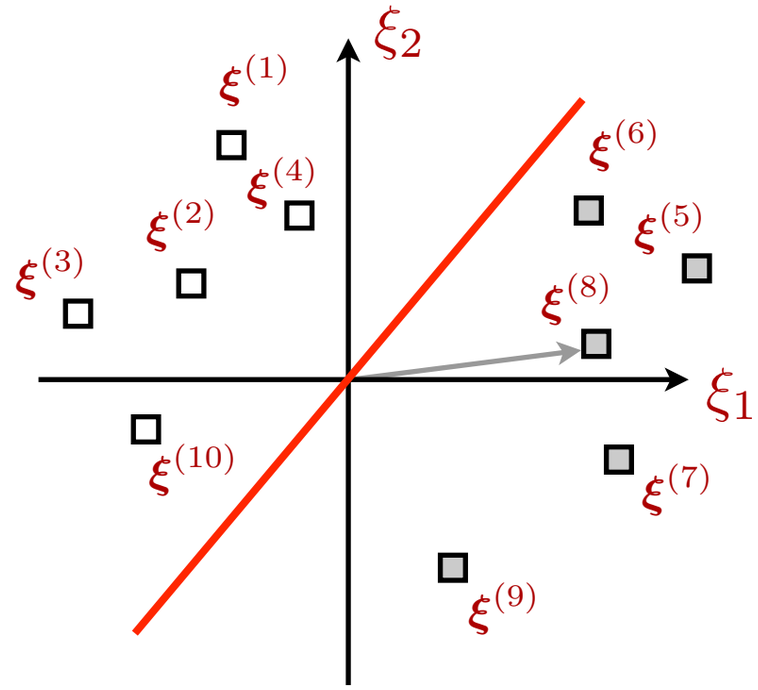
Input patterns $\boldsymbol{\xi}^{(\mu)}$. Index $\mu = 1, \ldots, p$ labels patterns.

Each pattern has two components, $\xi_1^{(\mu)}$ and $\xi_2^{(\mu)}$.

Arrange components into vector, $\boldsymbol{\xi}^{(\mu)} = \begin{bmatrix} \xi_1^{(\mu)} \\ \xi_2^{(\mu)} \end{bmatrix}$, shown in the Figure.

We see: the patterns fall into two classes: □ on the left, and ■ on the right.

Draw a red line (*decision boundary*) to distinguish the two types of patterns.



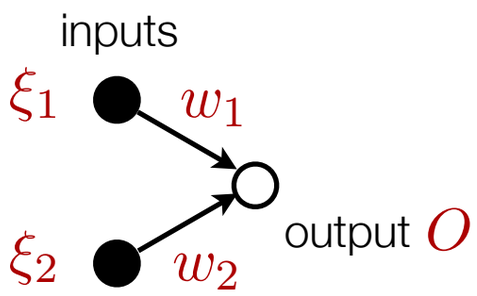
Aim: train a neural network to compute the decision boundary. To do this, define *target values*:

$t^{(\mu)} = 1$ for ■, and $t^{(\mu)} = -1$ for □

Training set $(\boldsymbol{\xi}^{(\mu)}, t^{(\mu)}), \mu = 1, \ldots, p$.

Simple perceptron

Simple perceptron: one neuron. Two input units ξ_1 and ξ_2. Activation function $\mathrm{sgn}(b)$.

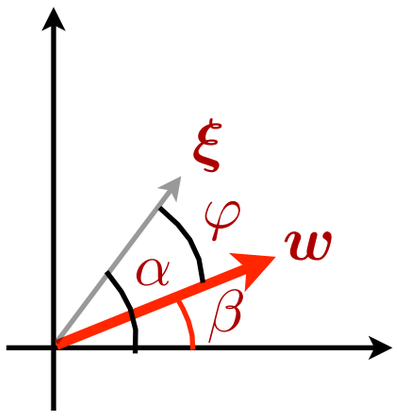


No threshold, $\theta = 0$.

Output $O = \mathrm{sgn}(w_1\xi_1 + w_2\xi_2) = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi})$

Vectors $\boldsymbol{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} = |\boldsymbol{w}| \begin{bmatrix} \cos\beta \\ \sin\beta \end{bmatrix}$ $\qquad \boldsymbol{\xi} = \begin{bmatrix} \xi_1 \\ \xi_2 \end{bmatrix} = |\boldsymbol{\xi}| \begin{bmatrix} \cos\alpha \\ \sin\alpha \end{bmatrix}$

denotes length of vector $\boldsymbol{w}$



Scalar product between vectors:

$$\boldsymbol{w} \cdot \boldsymbol{\xi} = w_1\xi_1 + w_2\xi_2 = |\boldsymbol{w}|\,|\boldsymbol{\xi}|(\cos\alpha\cos\beta + \sin\alpha\sin\beta)$$

So $\qquad = |\boldsymbol{w}|\,|\boldsymbol{\xi}|\cos(\alpha - \beta)$

$$\boldsymbol{w} \cdot \boldsymbol{\xi} = |\boldsymbol{w}|\;|\boldsymbol{\xi}|\;\cos\varphi$$

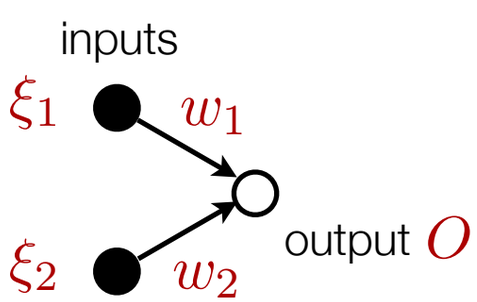
angle between $\boldsymbol{w}$ and $\boldsymbol{\xi}$

Aim: adjust the weights $\boldsymbol{w}$ so that network outputs correct target value for all patterns:

$$O^{(\mu)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(\mu)}) = t^{(\mu)} \quad \text{for} \quad \mu = 1, \ldots, p$$

Geometrical solution

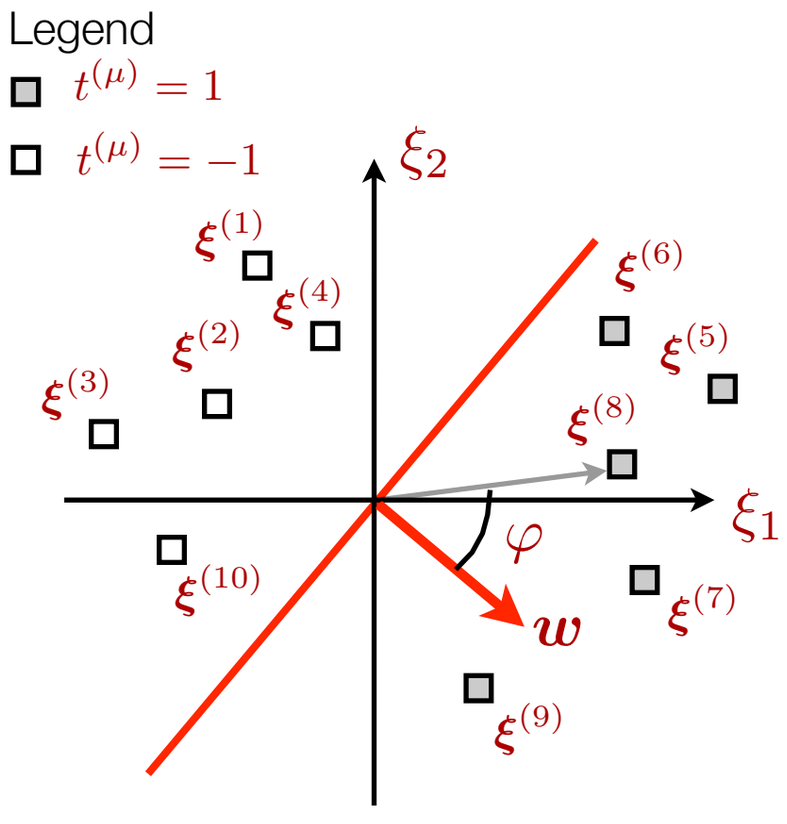
Simple perceptron: One neuron. Two input units ξ_1 and ξ_2. Activation function $\mathrm{sgn}(b)$.



No threshold, $\theta = 0$.

Output $O = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi})$ with $\boldsymbol{w} \cdot \boldsymbol{\xi} = |\boldsymbol{w}|\ |\boldsymbol{\xi}|\ \cos\varphi$ ← angle between $\boldsymbol{w}$ and $\boldsymbol{\xi}$

Aim: adjust the weights $\boldsymbol{w}$ so that network outputs correct target values for all patterns:



$$O^{(\mu)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(\mu)}) = t^{(\mu)} \quad \text{for} \quad \mu = 1, \ldots, p$$

where $t^{(\mu)} = 1$ for ◻ and $t^{(\mu)} = -1$ for ◻ .

Solution:

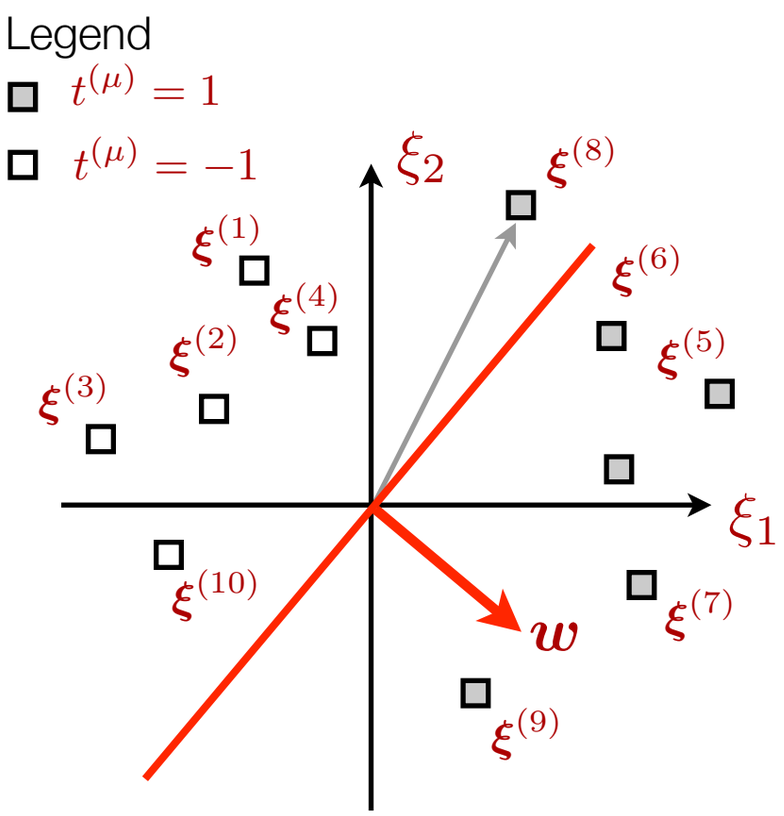
$\boldsymbol{w} \perp$ decision boundary

Check: $\boldsymbol{w} \cdot \boldsymbol{\xi}^{(8)} > 0$ so $O^{(8)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(8)}) = 1$. Correct since $t^{(8)} = 1$.

Hebb's rule

The pattern $\boldsymbol{\xi}^{(8)}$ is on the wrong side of the red line. So $O^{(8)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(8)}) \neq t^{(8)}$.

Move the red line so that $O^{(8)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(8)}) = t^{(8)}$ by rotating the weight vector $\boldsymbol{w}$:

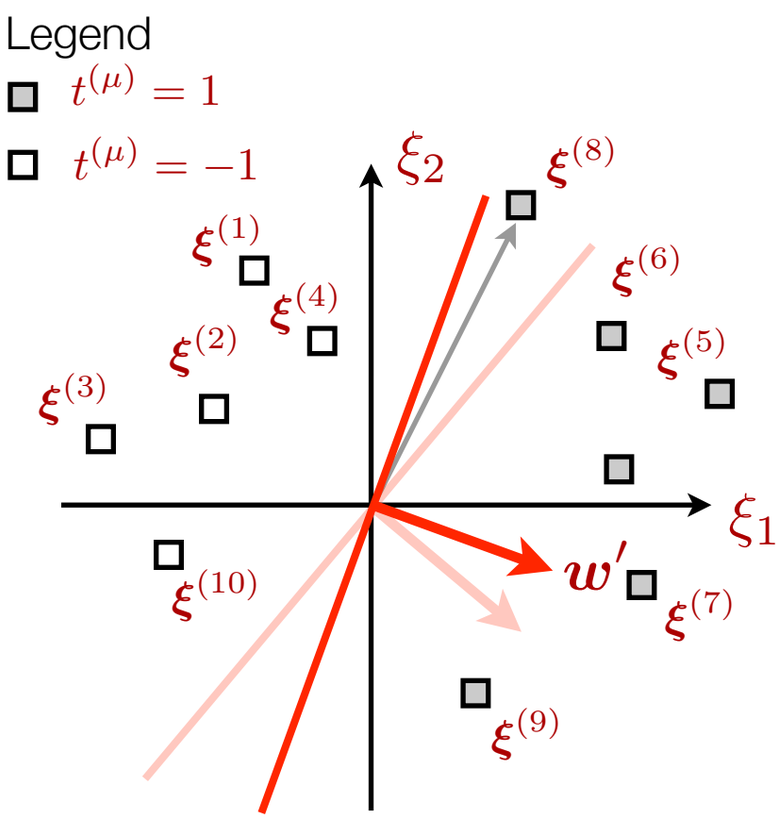


Hebb's rule

The pattern $\boldsymbol{\xi}^{(8)}$ is on the wrong side of the red line. So $O^{(8)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(8)}) \neq t^{(8)}$.

Move the red line so that $O^{(8)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(8)}) = t^{(8)}$ by rotating the weight vector $\boldsymbol{w}$:

$\boldsymbol{w}' = \boldsymbol{w} + \eta \boldsymbol{\xi}^{(8)}$ (small parameter $\eta > 0$) so that $O^{(8)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(8)}) = t^{(8)}$.

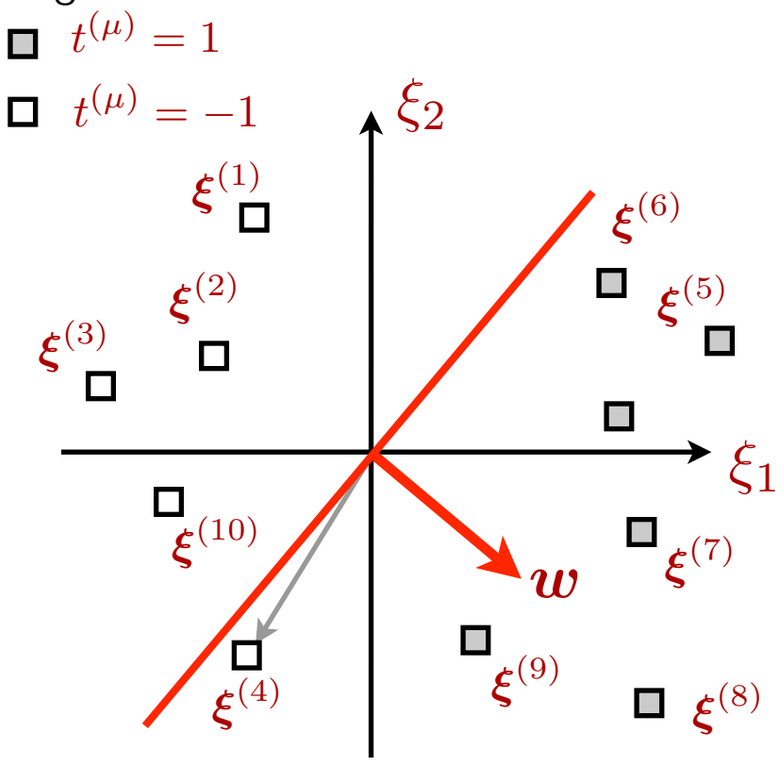


Hebb's rule

The pattern $\boldsymbol{\xi}^{(4)}$ is on the wrong side of the red line. So $O^{(4)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(4)}) \neq t^{(4)}$.

Move the red line so that $O^{(4)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(4)}) = t^{(4)}$ by rotating the weight vector $\boldsymbol{w}$:

$\boldsymbol{w}' = \boldsymbol{w} - \eta \boldsymbol{\xi}^{(4)}$ (small parameter $\eta > 0$) so that $O^{(4)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(4)}) = t^{(4)}$.

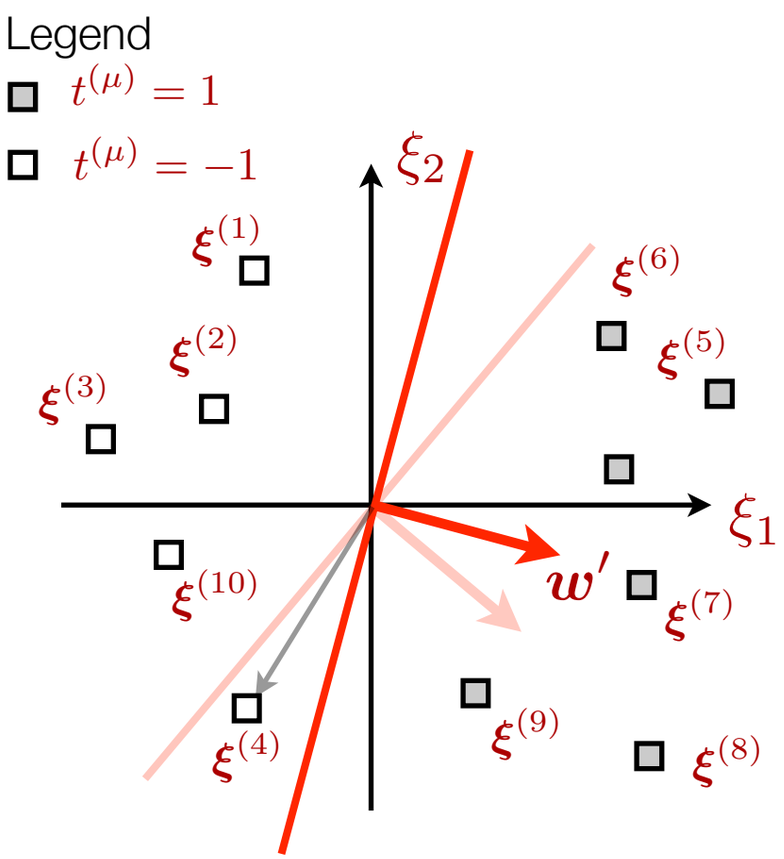


Hebb's rule

The pattern $\boldsymbol{\xi}^{(4)}$ is on the wrong side of the red line. So $O^{(4)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(4)}) \neq t^{(4)}$.

Move the red line so that $O^{(4)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(4)}) = t^{(4)}$ by rotating the weight vector $\boldsymbol{w}$:

$\boldsymbol{w}' = \boldsymbol{w} - \eta \boldsymbol{\xi}^{(4)}$ (small parameter $\eta > 0$) so that $O^{(4)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(4)}) = t^{(4)}$.

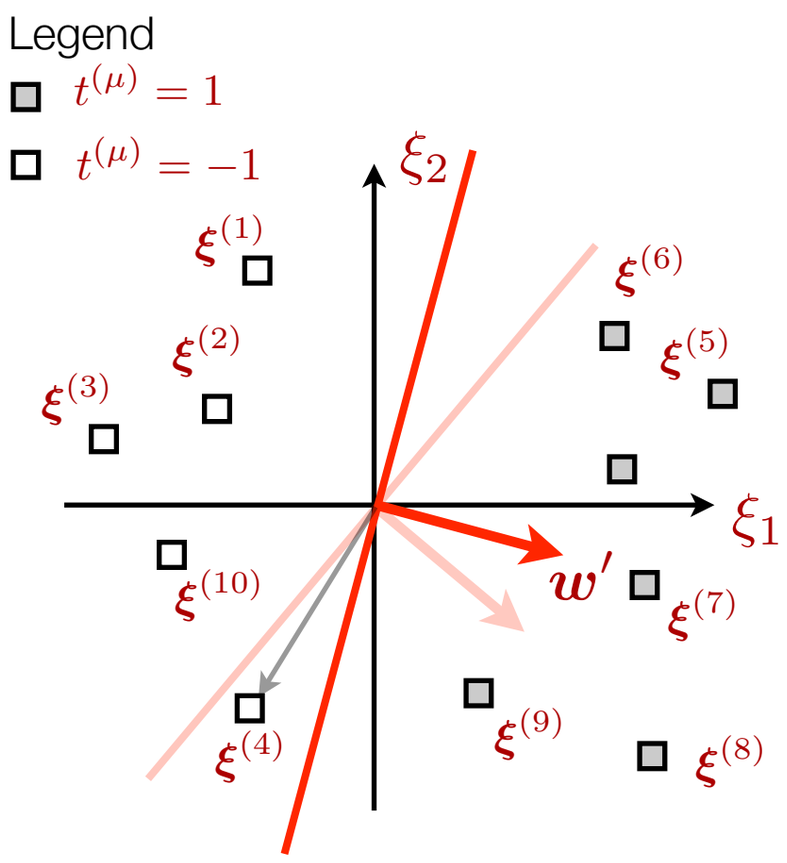


Hebb's rule

The pattern $\boldsymbol{\xi}^{(4)}$ is on the wrong side of the red line. So $O^{(4)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(4)}) \neq t^{(4)}$.

Move the red line so that $O^{(4)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(4)}) = t^{(4)}$ by rotating the weight vector $\boldsymbol{w}$:

$\boldsymbol{w}' = \boldsymbol{w} - \eta \boldsymbol{\xi}^{(4)}$ (small parameter $\eta > 0$) so that $O^{(4)} = \mathrm{sgn}(\boldsymbol{w} \cdot \boldsymbol{\xi}^{(4)}) = t^{(4)}$.



Note the minus sign:

$$\boldsymbol{w}' = \boldsymbol{w} + \eta \boldsymbol{\xi}^{(8)} \quad \text{where } t^{(8)} = 1$$

$$\boldsymbol{w}' = \boldsymbol{w} - \eta \boldsymbol{\xi}^{(4)} \quad \text{where } t^{(4)} = -1$$

Learning rule (*Hebb's rule*)

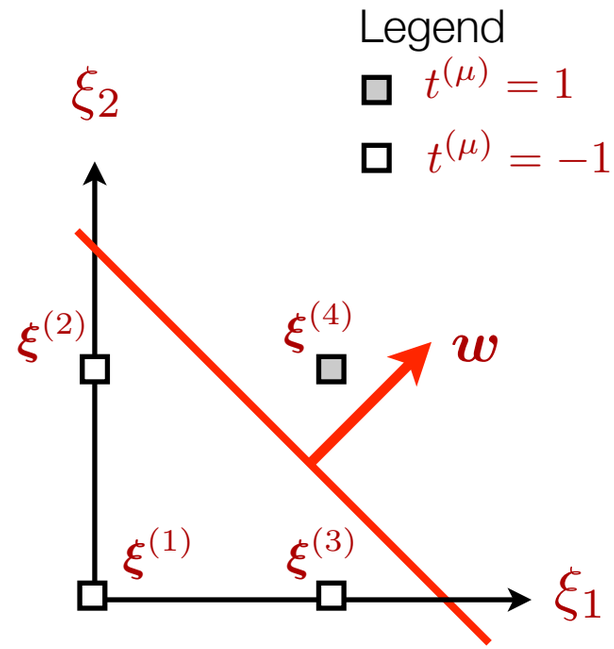
$$\boldsymbol{w}' = \boldsymbol{w} + \delta \boldsymbol{w} \quad \text{with} \quad \delta \boldsymbol{w} = \eta \, t^{(\mu)} \xi^{(\mu)}$$

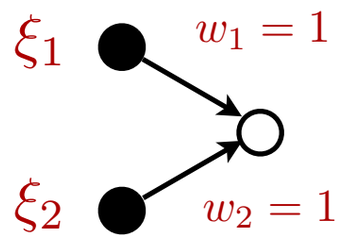
Apply learning rule many times until problem is solved.

Example - AND function

Logical AND

ξ_1	ξ_2	t
0	0	-1
1	0	-1
0	1	-1
1	1	1

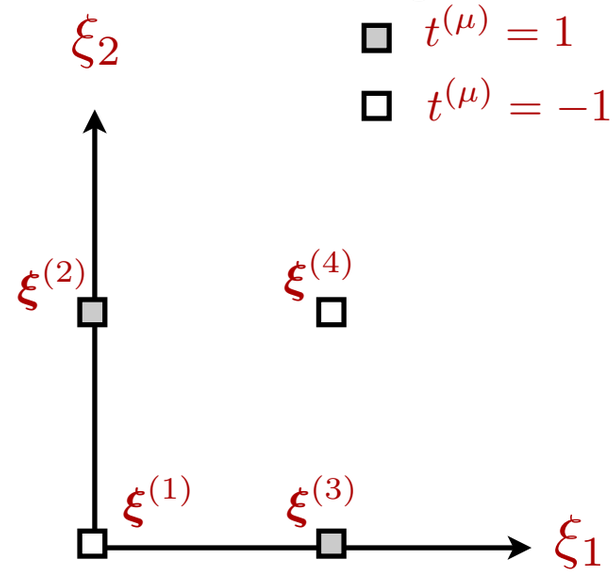




Example - XOR function

Logical XOR

ξ_1	ξ_2	t
0	0	-1
1	0	1
0	1	1
1	1	-1



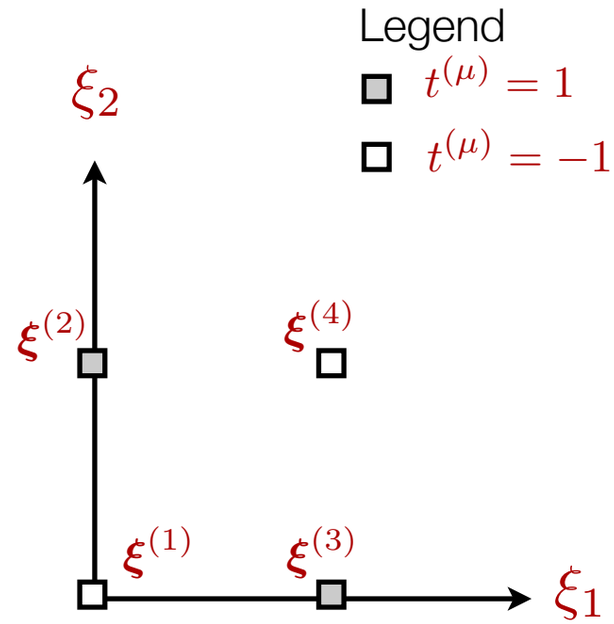
Legend

- $t^{(\mu)} = 1$
- $t^{(\mu)} = -1$

Example - XOR function

Logical XOR

ξ_1	ξ_2	t
0	0	-1
1	0	1
0	1	1
1	1	-1



Legend

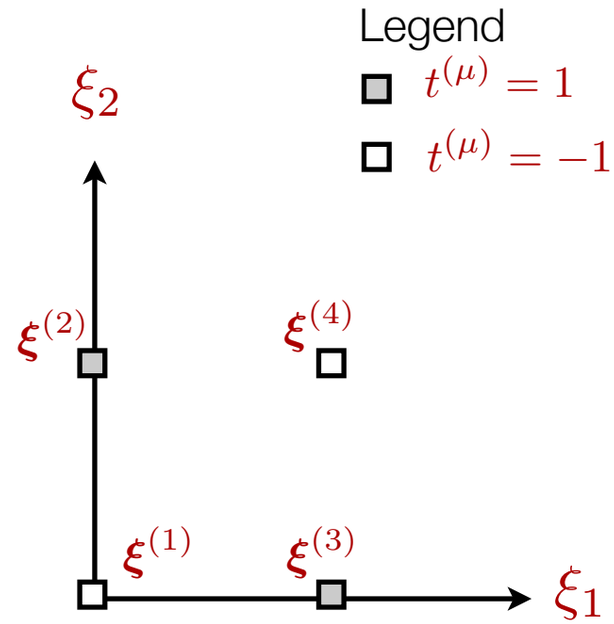
- $t^{(\mu)} = 1$
- $t^{(\mu)} = -1$

This problem is not *linearly separable* because we cannot separate ■ from □ by a single red line.

Example - XOR function

Logical XOR

ξ_1	ξ_2	t
0	0	-1
1	0	1
0	1	1
1	1	-1



Legend

- $t^{(\mu)} = 1$
- $t^{(\mu)} = -1$

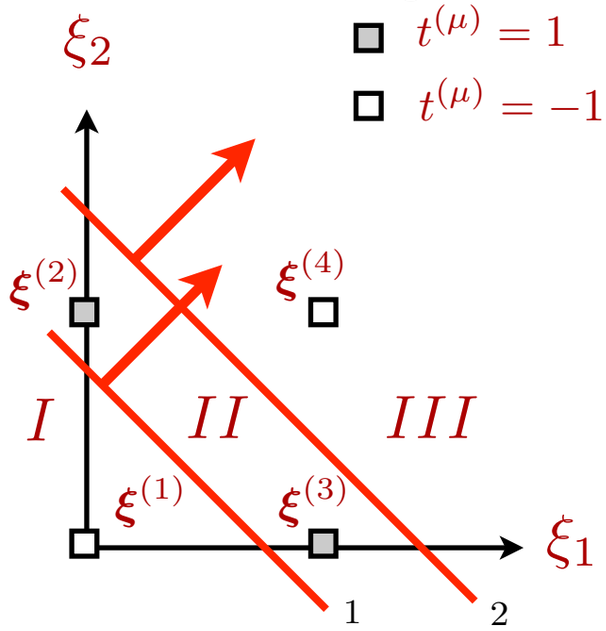
This problem is not *linearly separable* because we cannot separate ■ from □ by a *single* red line.

Solution: use *two* red lines.

Example - XOR function

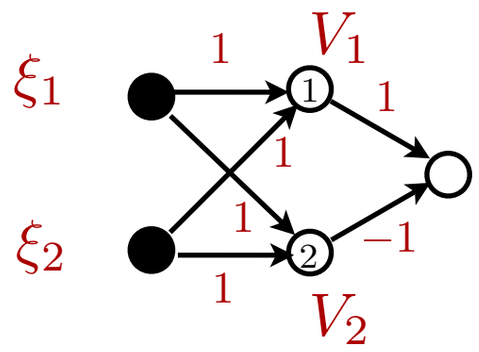
Logical XOR

ξ_1	ξ_2	t
0	0	-1
1	0	1
0	1	1
1	1	-1



Legend

- $t^{(\mu)} = 1$
- $t^{(\mu)} = -1$



layer of hidden neurons V_1 and V_2 (neither input nor output)

all *hidden weights* equal to 1

Two *hidden* neurons, each one defines one red line.

Together they define three regions, I, II, and III.

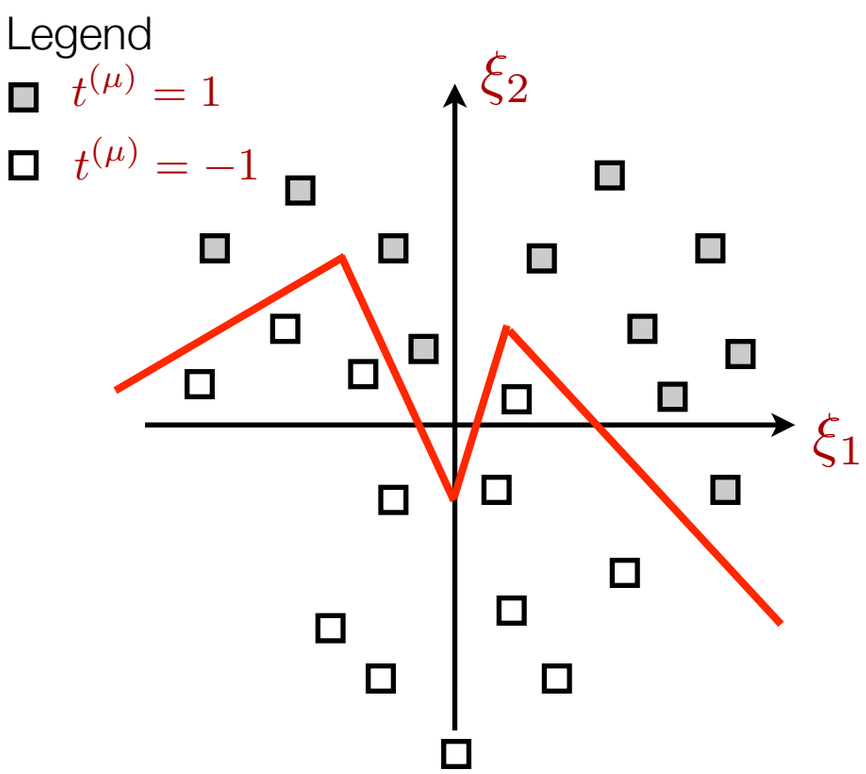
We need a third neuron (with weights W_1 and W_2 to associate region II with ◻, and regions II and III with ◻.

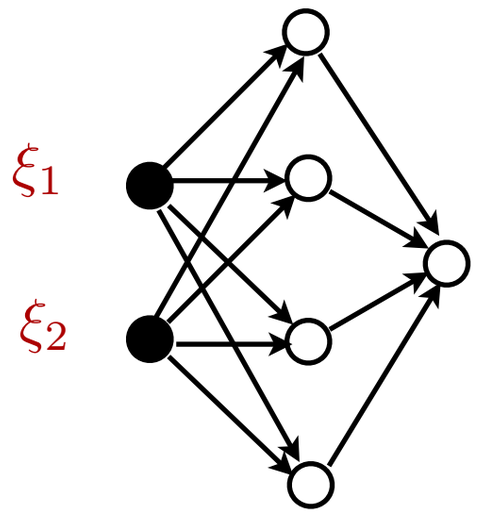
$$O = \mathrm{sgn}(V_1 - V_2 - 1)$$

threshold

Non-(linearly) separable problems

Solve problems that are not linearly separable with a hidden layer of neurons

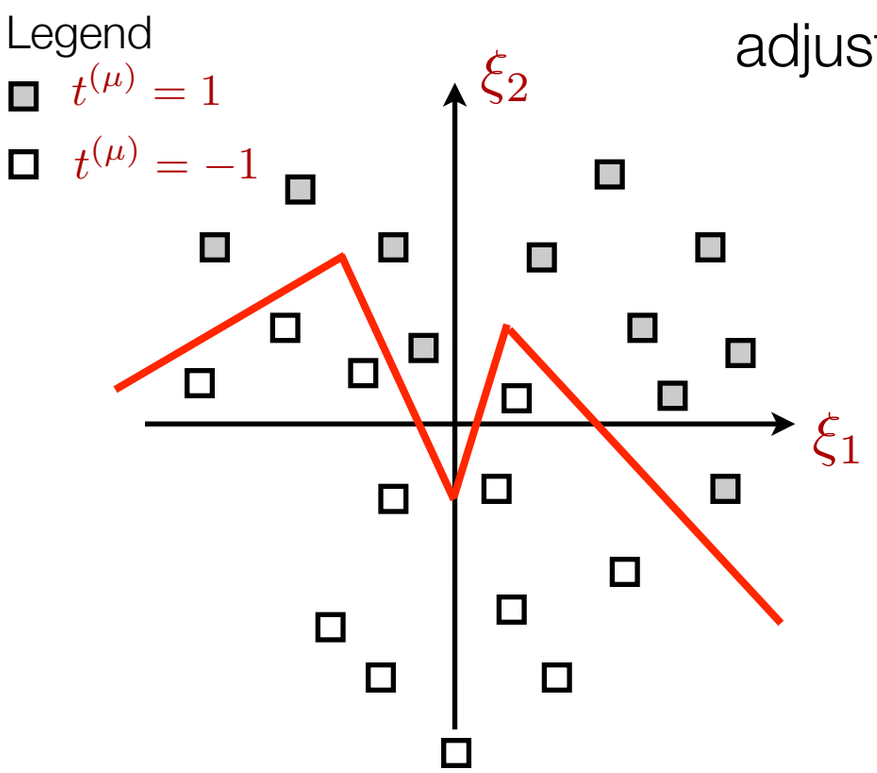


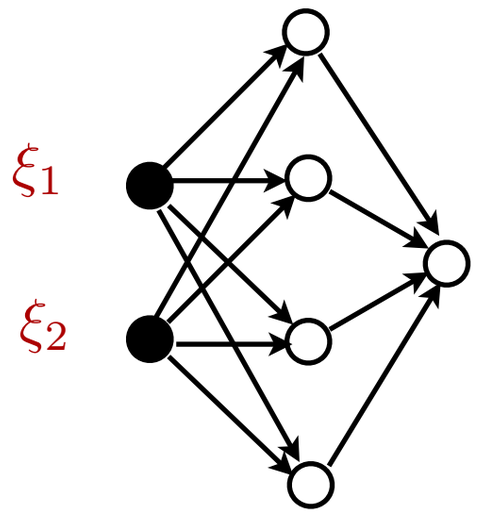


Four hidden neurons - one for each red line-segment. Move the red lines into the correct configuration by repeatedly using Hebb's rule until the problem is solved (a fifth neuron assigns regions and solves the classification problem).

Generalisation

Train the network on a *training set* $(\boldsymbol{\xi}^{(\mu)}, t^{(\mu)}), \mu = 1, \ldots, p$: move red lines into the correct configuration by repeatedly applying Hebb's rule to adjust all weights. Usually many iterations necessary.

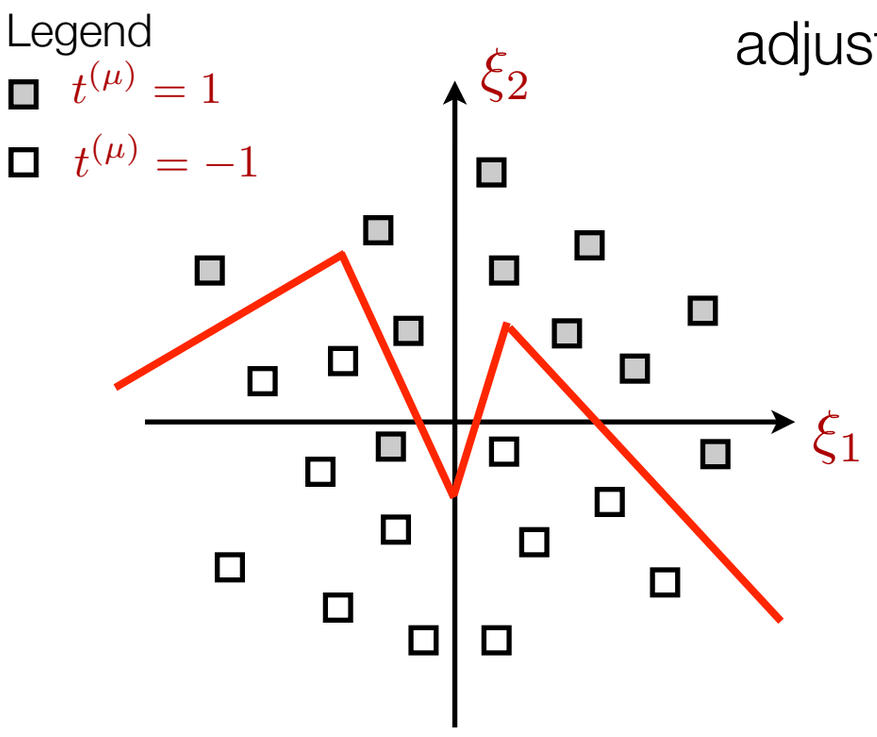


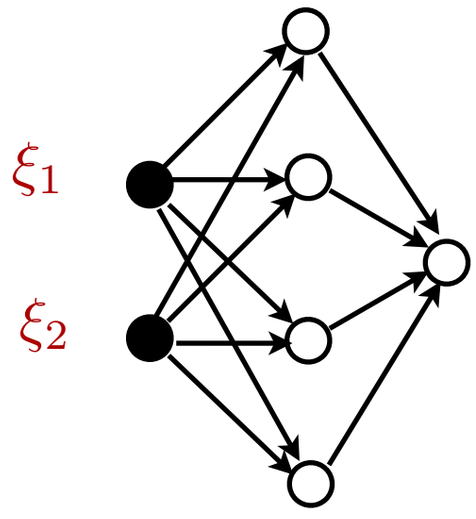


Once all red lines are in the right place (all weights determined), apply network to a new data set. If the training set was reliable, then the network has learnt to classify the new data, it has learnt to *generalise*.

Training

Train the network on a *training set* $(\boldsymbol{\xi}^{(\mu)}, t^{(\mu)}), \mu = 1, \ldots, p$: move red lines into the correct configuration by repeatedly applying Hebb's rule to adjust all weights. Usually many iterations necessary.

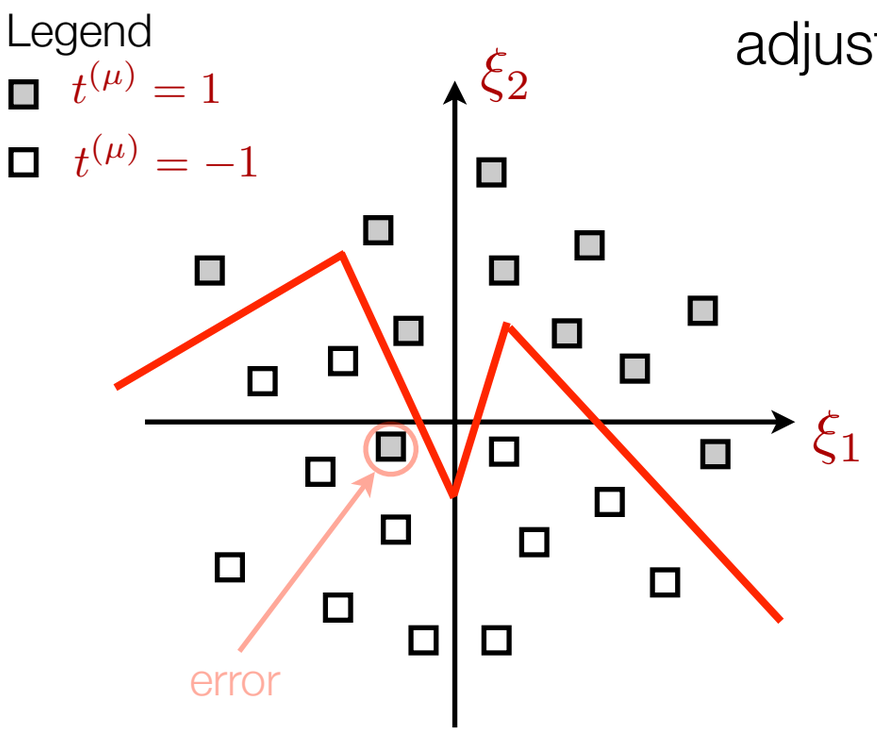


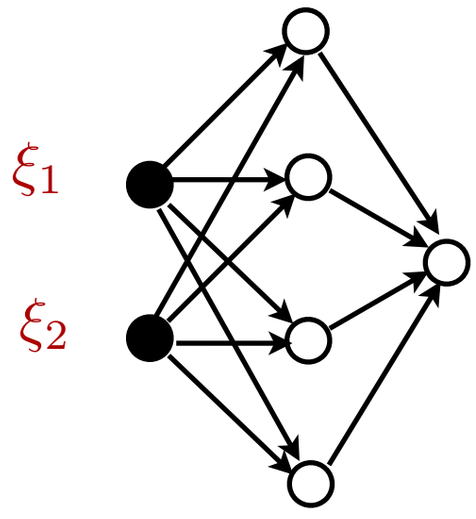


Once all red lines are in the right place (all weights determined), apply network to a new data set. If the training set was reliable, then the network has learnt to classify the new data, it has learnt to *generalise*.

Training

Train the network on a *training set* $(\boldsymbol{\xi}^{(\mu)}, t^{(\mu)}), \mu = 1, \ldots, p$: move red lines into the correct configuration by repeatedly applying Hebb's rule to adjust all weights. Usually many iterations necessary.

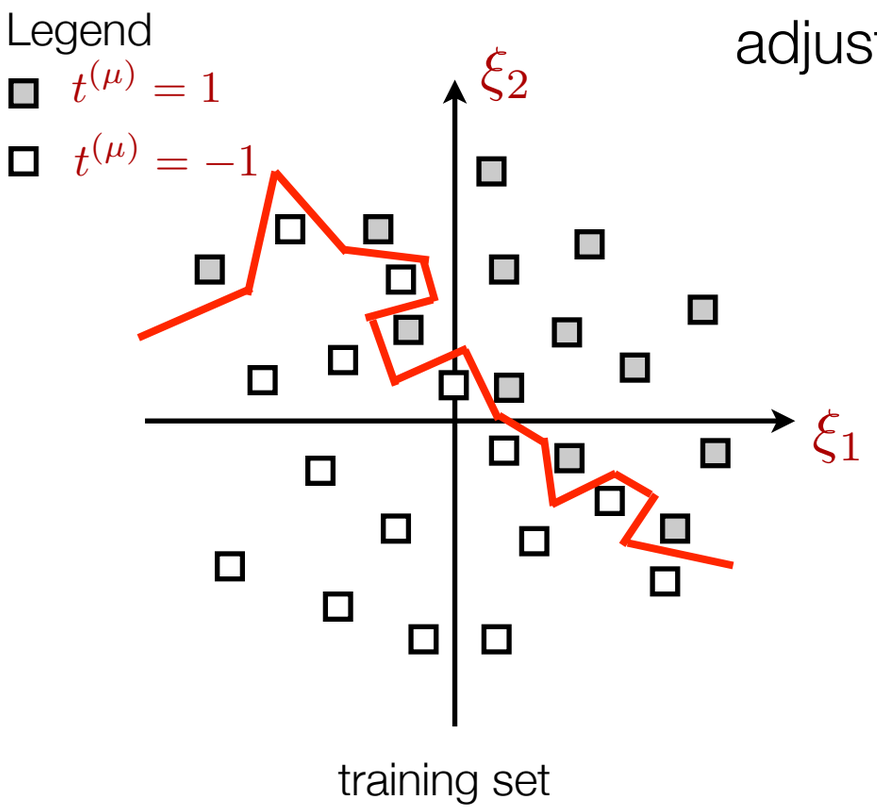




Usually a small number of errors is acceptable. It is often not meaningful to try to fine-tune very precisely (input patterns are subject to noise).

Overfitting

Train the network on a *training set* $(\boldsymbol{\xi}^{(\mu)}, t^{(\mu)}), \mu = 1, \ldots, p$: move red lines into the correct configuration by repeatedly applying Hebb's rule to adjust all weights. Usually many iterations necessary.



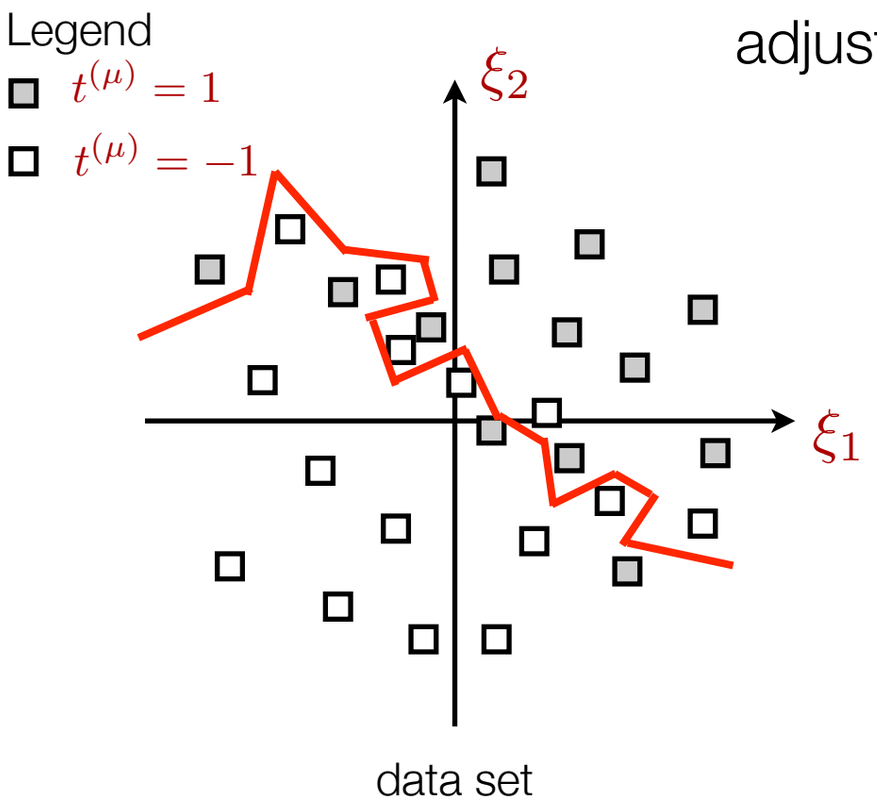
training set

Here: used 15 hidden neurons to fit decision boundary decision boundary very precisely.

But often the inputs are affected by noise, which can be very different in different data sets

Overfitting

Train the network on a *training set* $(\boldsymbol{\xi}^{(\mu)}, t^{(\mu)}), \mu = 1, \ldots, p$: move red lines into the correct configuration by repeatedly applying Hebb's rule to adjust all weights. Usually many iterations necessary.



data set

Here: used 15 hidden neurons to fit decision boundary decision boundary very precisely.

But often the inputs are affected by noise, which can be very different in different data sets.

Here the network just learnt noise in the training set. Avoid this by reducing number of weights (*dropout*).

Multilayer perceptrons

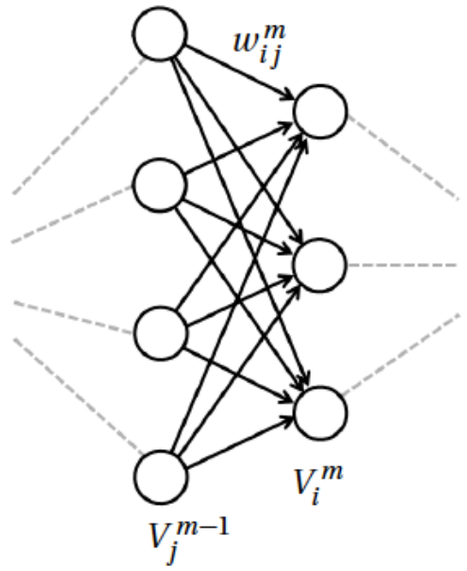
All examples had a two-dimensional input space and one-dimensional outputs.
Reason: so that we could draw the problem and understand its geometrical form.

In reality problems are often high-dimensional. For example image recognition: input dimension $N = \#\text{pixels} \times 3$. Output dimension: number of classes to be recognised (car, lorry, road sign, person, bicycle,...)

Multilayer perceptron: network with many hidden layers

Two problems: (i) expensive to train if many weights
(ii) slow convergence (weights get stuck)

Questions: (i) how many hidden layers necessary?
(ii) how many hidden layers efficient?



hidden layers
number $m-1$ and m

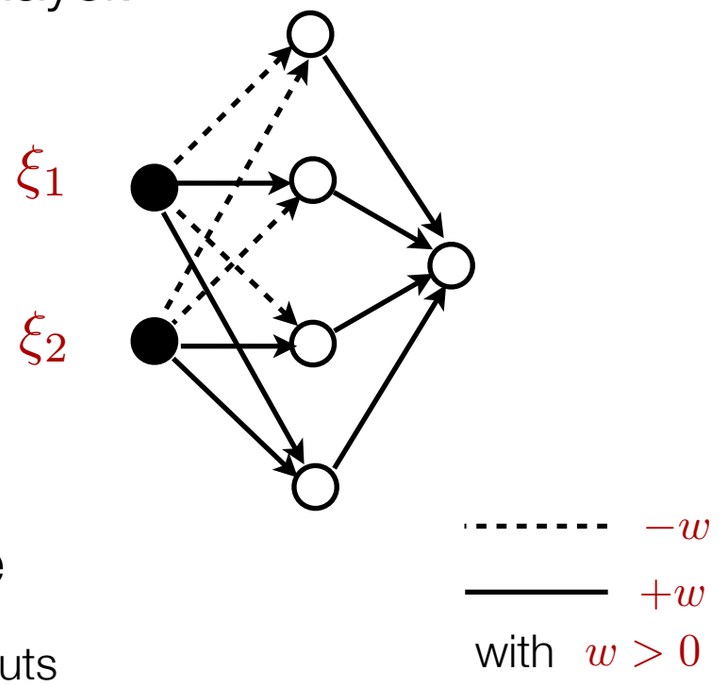
How many hidden layers?

All Boolean functions with N inputs can be trained/learned with a single hidden layer.

Proof by construction. Requires 2^N neurons in the hidden layer.

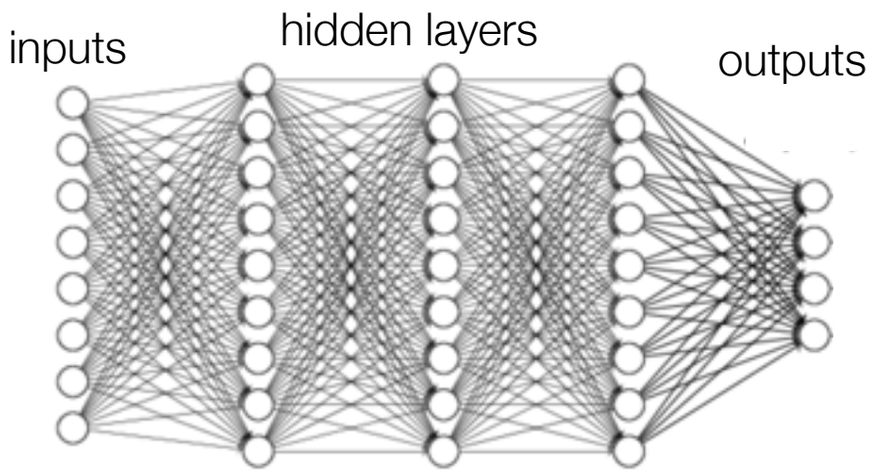
Example: XOR function ($N = 2$).

For large N, this architecture is not practical, because the number of neurons increases exponentially with N.



Better: use more layers. Intuition: the more layers the more powerful is the network.

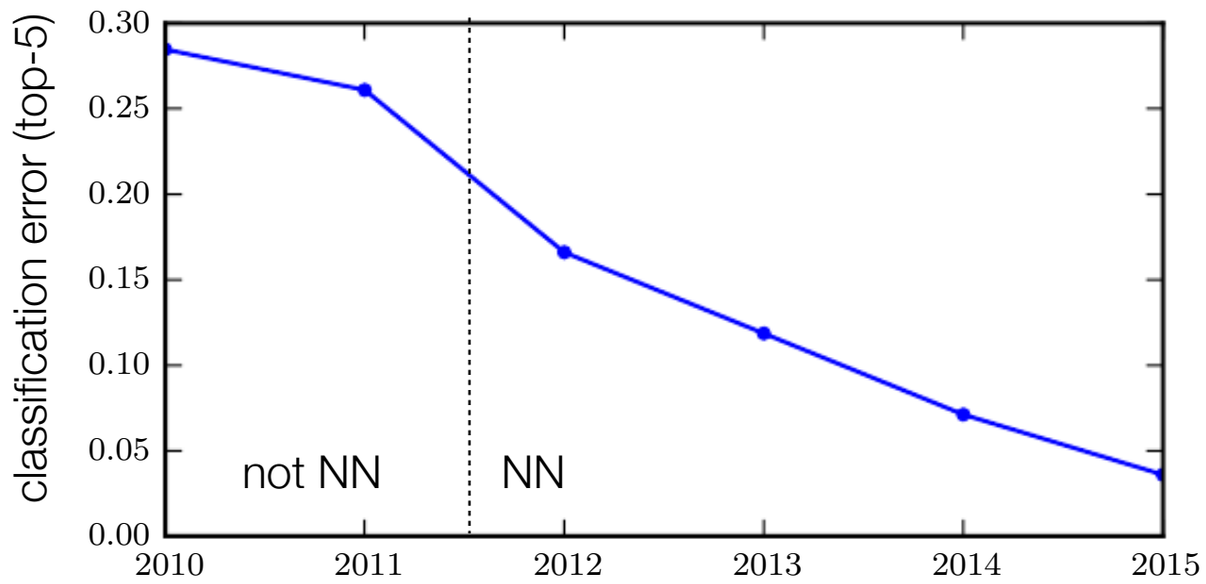
But such *deep networks* are in general hard to train.



Deep learning

Deep learning. Perceptrons with large numbers of layers.

In the last few years, interest in neural-network algorithms has exploded, and their performance has significantly improved. Why?



Goodfellow *et al. Deep Learning* (2016)

Better training sets

Improved architecture

- better activation functions, $g(b) = \max(0, b)$ Giorot, Bordes & Bengio (2011)
- a bunch of other tricks (dropout, batch normalisation)

Better hardware

- GPUs
- dedicated chips by Google en.wikipedia.org/wiki/Tensor_processing_unit

ImageNet

To obtain training set: must collect and manually annotate data.
Questions: (i) efficient data collection and annotation
(ii) aim for large variety in collected data

Example: Image-net (publicly available data set): $> 10^7$ images, classified into 10^3classes.

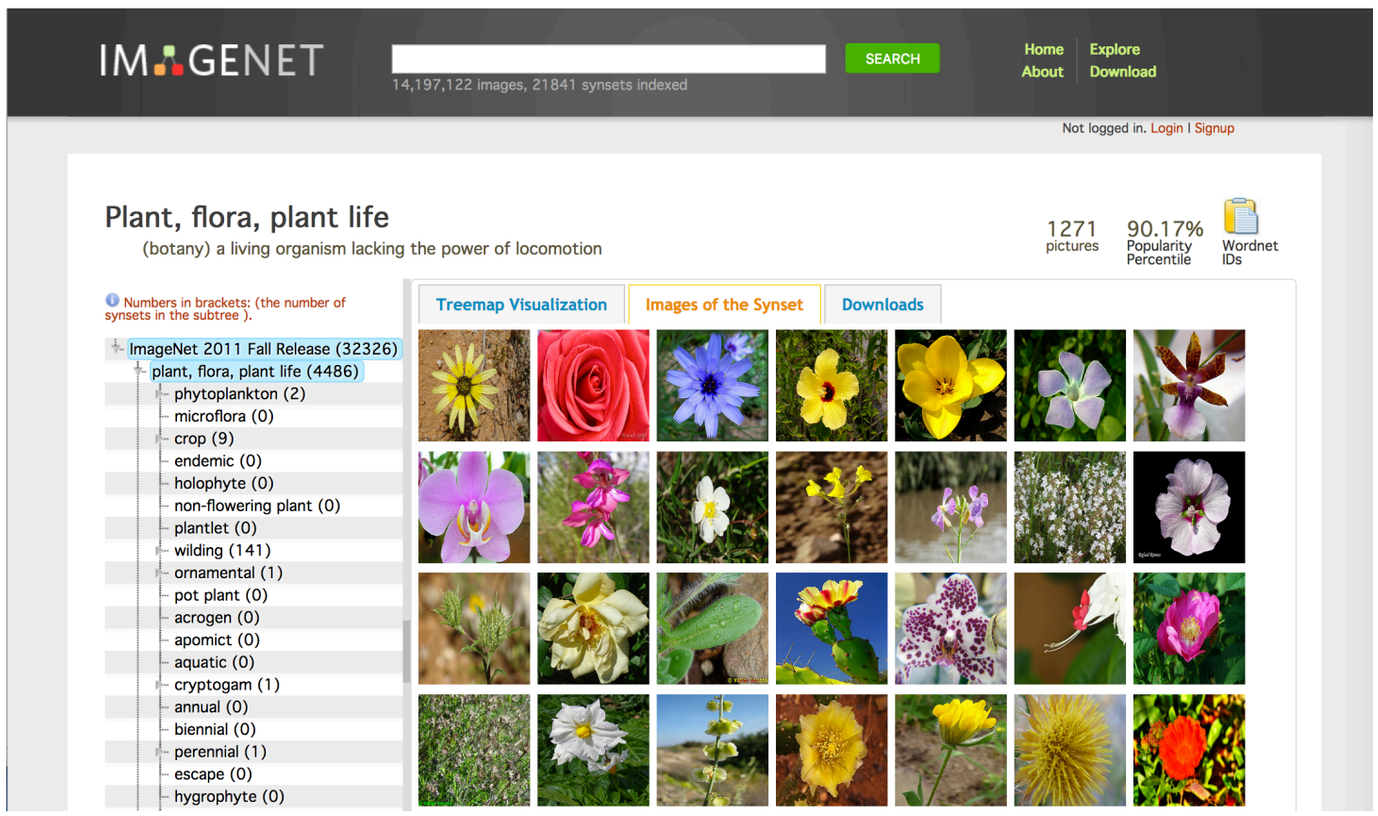
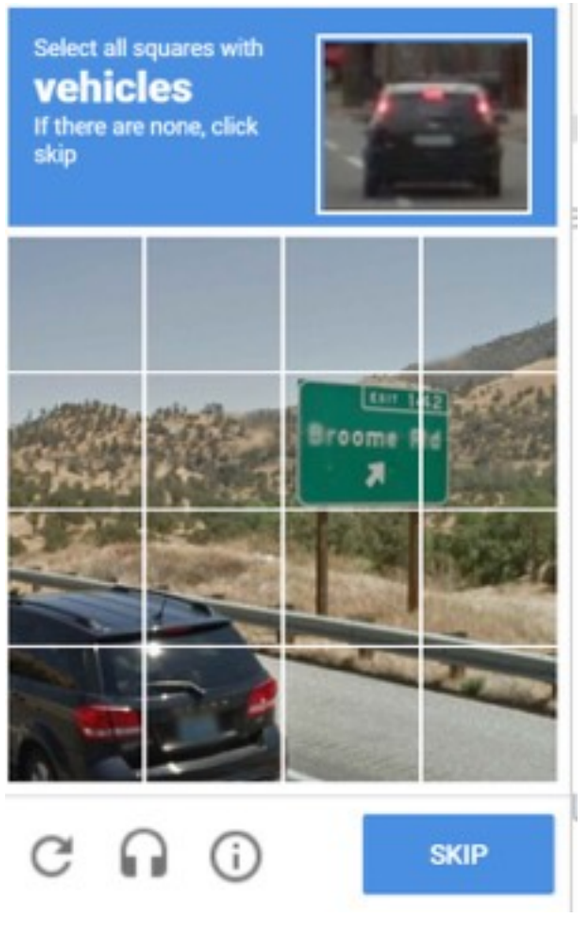


image-net.org

reCAPTCHA

github.com/google/recaptcha

Two goals. (i) protects websites from bot
(ii) users who click correctly provide correct annotation.



Convolutional networks

Convolutional network. Each neuron in first hidden layer is connected to a small region of inputs. Here 3×3 pixels.

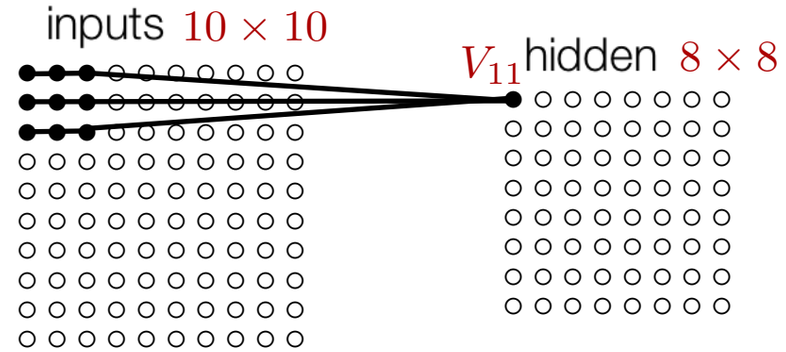
Slide the region over input image. Use *same* weights for all hidden neurons. Update V_{mn}

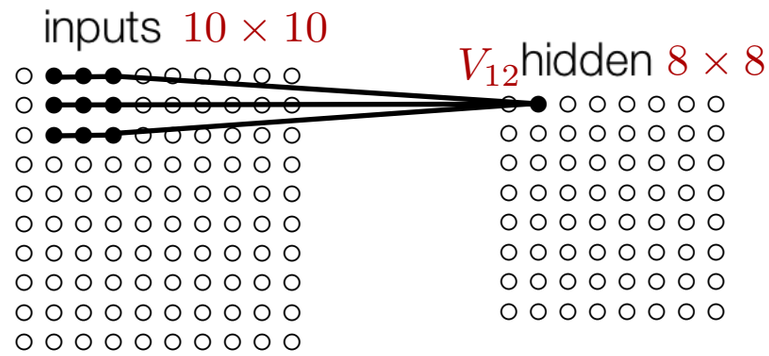
$$V_{mn} = g\Big(\sum_{k=1}^{3}\sum_{l=1}^{3} w_{kl}\xi_{m+k,n+l}\Big)$$

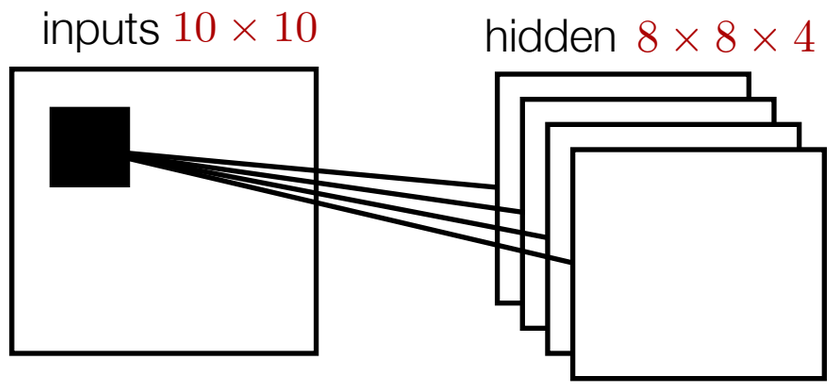
- detects the *same local feature* everywhere in input image (edge, corner,...). *Feature map*.

- the form of the sum is called *convolution*

- efficient because fewer weights



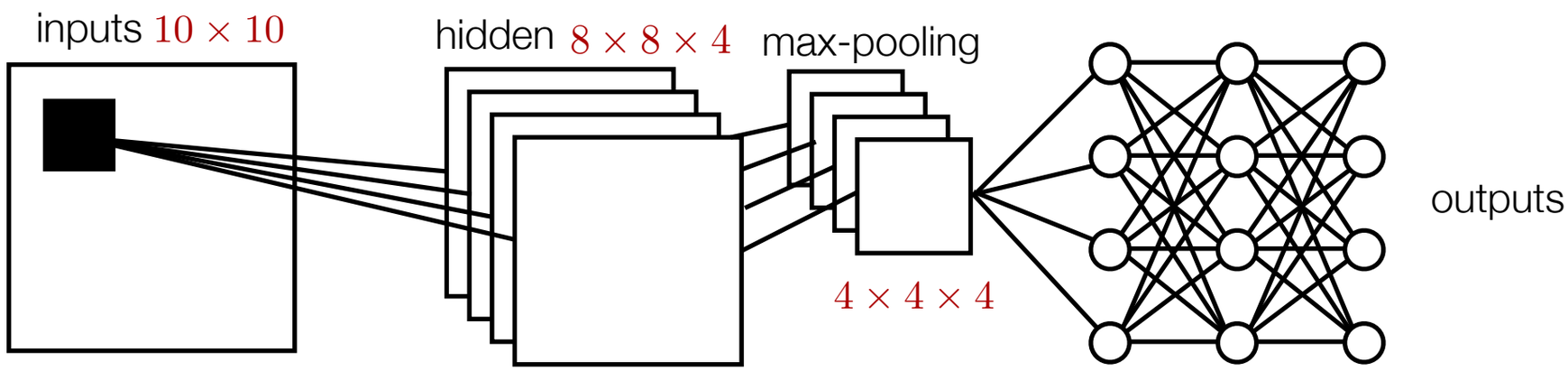




- use several feature maps to detect different features in input image

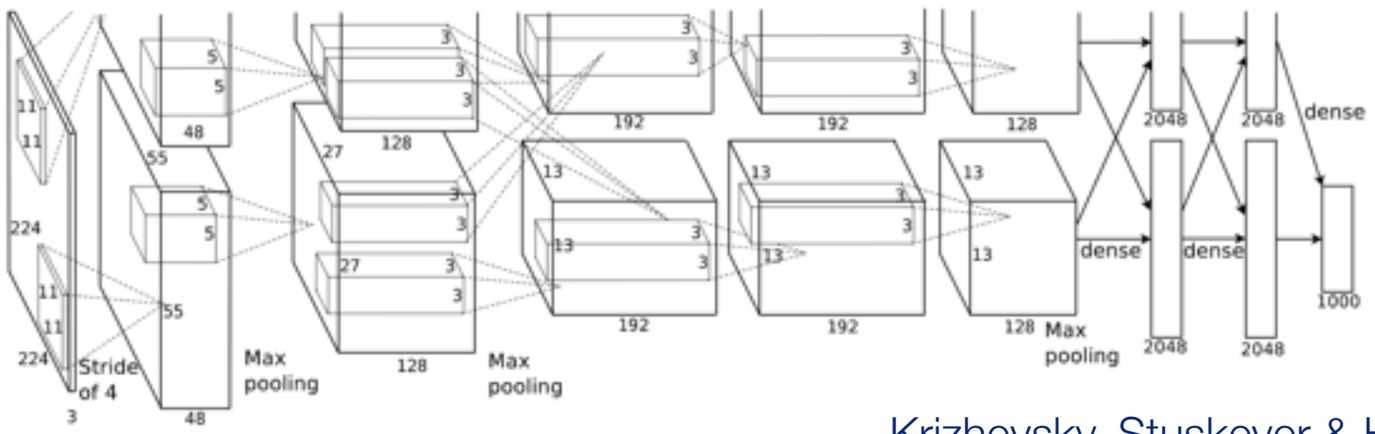
Convolutional networks

- *max-pooling*: take maximum of V_{mn} over small region (2×2) to reduce # of parameters



- add fully connected layers to learn more abstract features

- Example



Krizhevsky, Stuskever & Hinton (2012)

Object detection - You only look once

www.google.com/recaptcha/intro/android.html

Deep learning with a convolutional network. Here Yolo-9000 with tiny number of weights (so that the algorithm runs on my laptop). Pretrained on VOC training set

Redmon *et al.* www.arxiv.org/abs/1612.08242
github.com/pjreddie/darknet/wiki/YOLO:-Real-Time-Object-Detection

Installation instructions: github.com/philipperemy/yolo-9000

Yolo-9000

Object detection from video frames using Yolo-9000 github.com/philipperemy/yolo-9000

Film recorded with Iphone 6 on my way home from work.

Tiny number of weights (so that the algorithm runs on my laptop)

Pretrained on Pascal VOC data set
host.robots.ox.ac.uk/pascal/VOC/

Deep learning for self-driving cars

Zenuity: joint venture between Autoliv and Volvo Cars. Goal: develop software for assisted and autonomous driving. Founded 2017. 500 employees, mostly in Gothenburg.

Conclusions

Artificial neural networks were already studied in the 80ies

In the last few years: *deep-learning revolution* due to

- improved algorithms (focus on object detection)
- better hardware (GPUs, dedicated chips)
- better training sets

Applications: google, facebook, pinterest, tesla, zenuity,...

Perspectives

- close connections to statistical physics (*spin glasses*, random magnets)
- related methods in mathematical statistics (*big data* analysis)
- *unsupervised* learning (clustering and familiarity of input data)

Conclusions

xkcd.com/1897

Literature

Neural networks

Hertz, Krogh & Palmer, I*ntroduction to the theory of neural computation* (Santa Fe Institute Series)

Mehlig, *Artificial Neural Networks* physics.gu.se/~frtbm

Deep learning

Nielsen *Neural Networks and Deep Learning* neuralnetworksanddeeplearning.com/

Goodfellow, Bengio & Courville *Deep Learning* MIT Press

LeCunn, Bengio & Hinton, *Deep learning*, Nature **521** (2015) 436

Convolutional networks

Nielsen *Neural Networks and Deep Learning* neuralnetworksanddeeplearning.com/