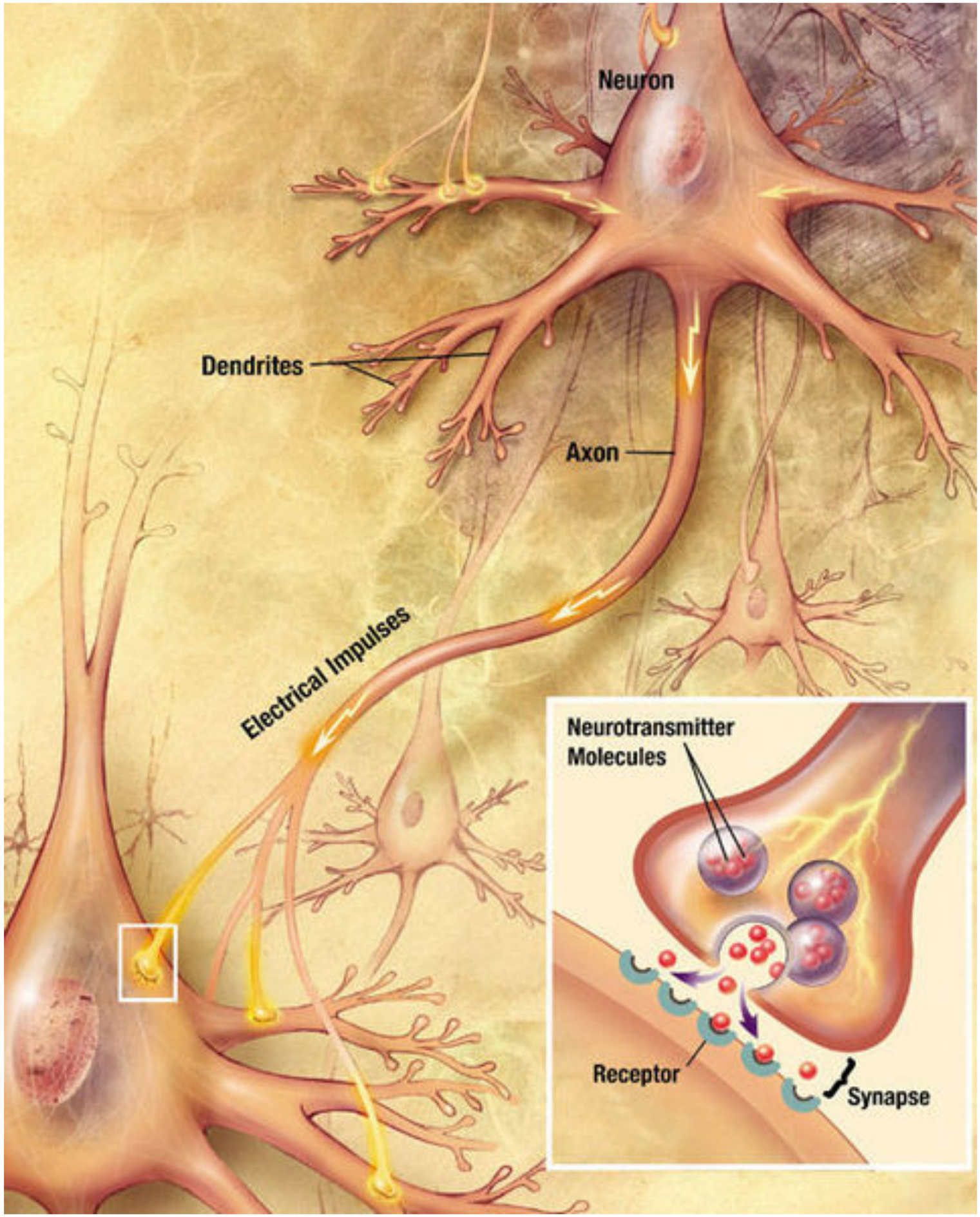
IFT6390
Fondements de l'apprentissage machine

Neural Networks

Professor: Ioannis Mitliagkas
Slides: Pascal Vincent

Human Brain

- 100 billions neurons
- complex network
- each neuron connected to thousands of others

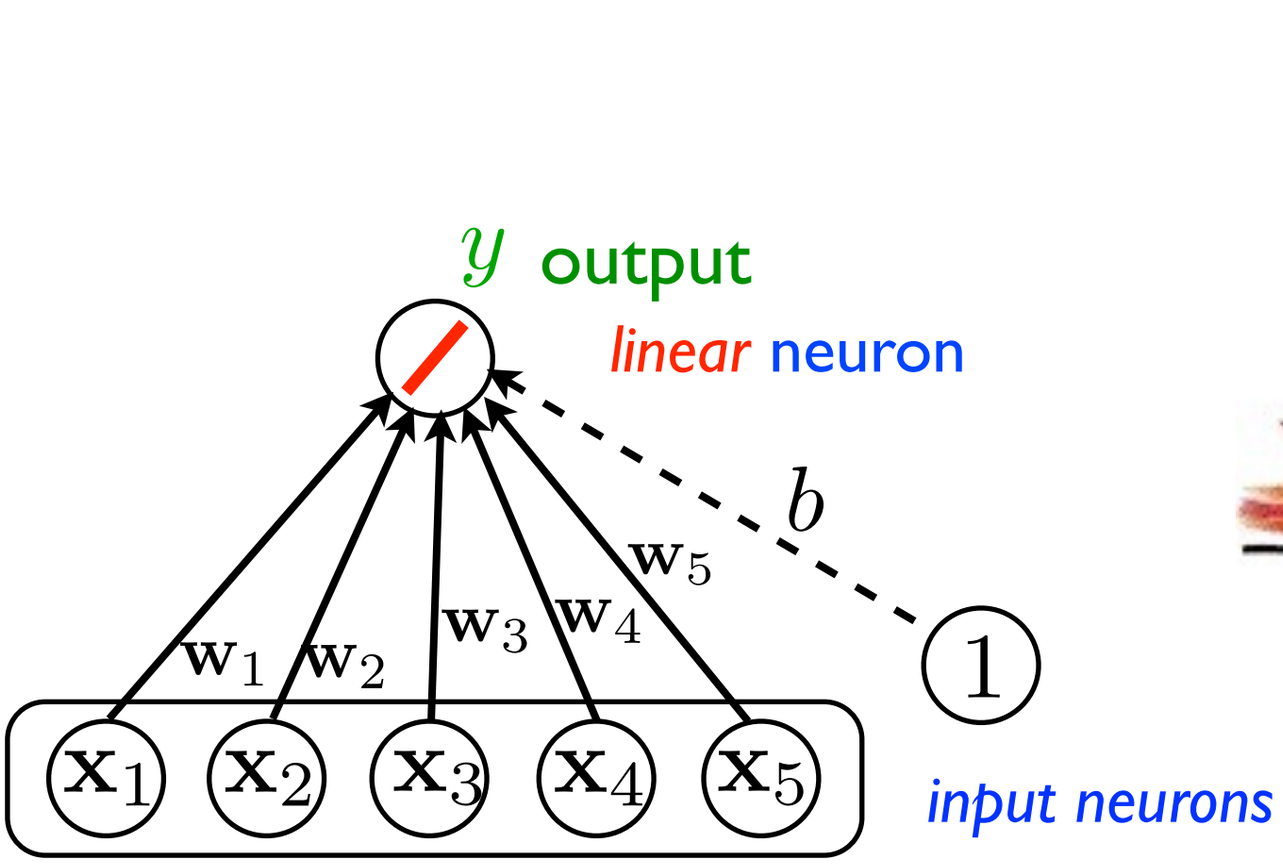

Neuron
Dendrites
Axon
Electrical Impulses
Neurotransmitter Molecules
Receptor
Synapse

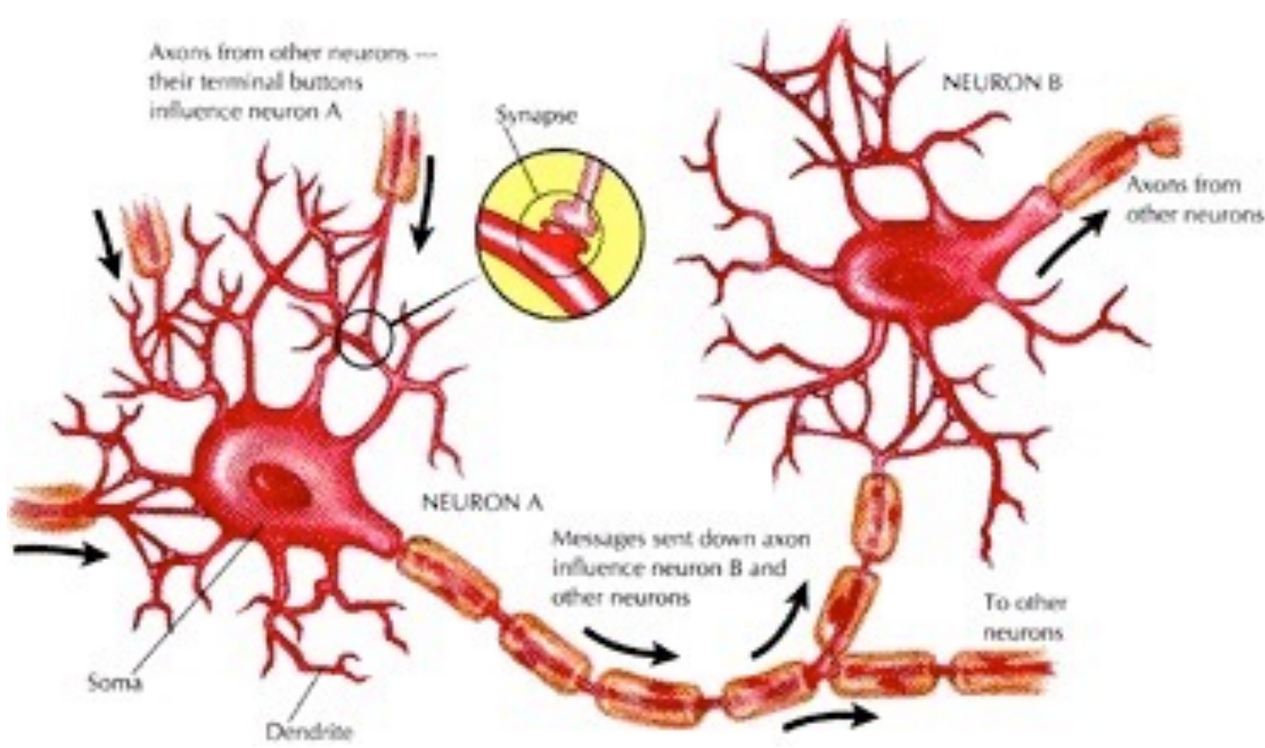
Linear neuron

Intuition behind the dot product:

each component of *x* has a (weighted) influence on the output *y*

$$y = f_\theta(\mathbf{x}) = \mathbf{w}_1\mathbf{x}_1 + \mathbf{w}_2\mathbf{x}_2 + \ldots + \mathbf{w}_d\mathbf{x}_d + b$$



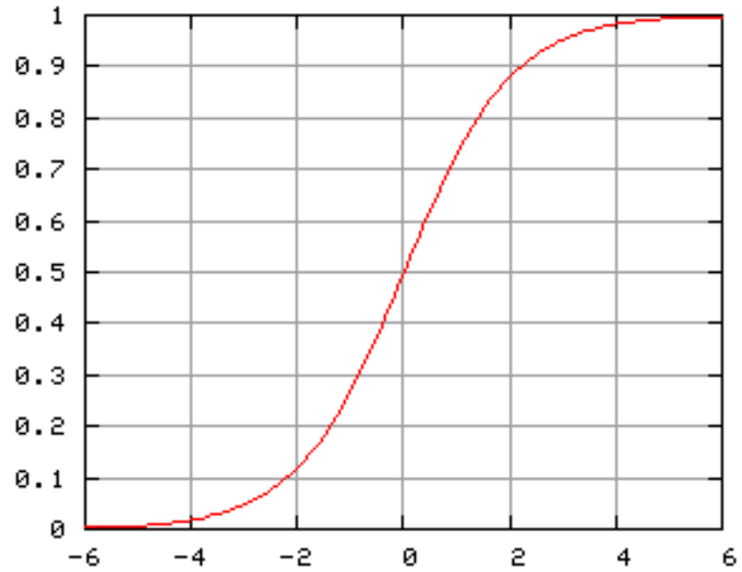


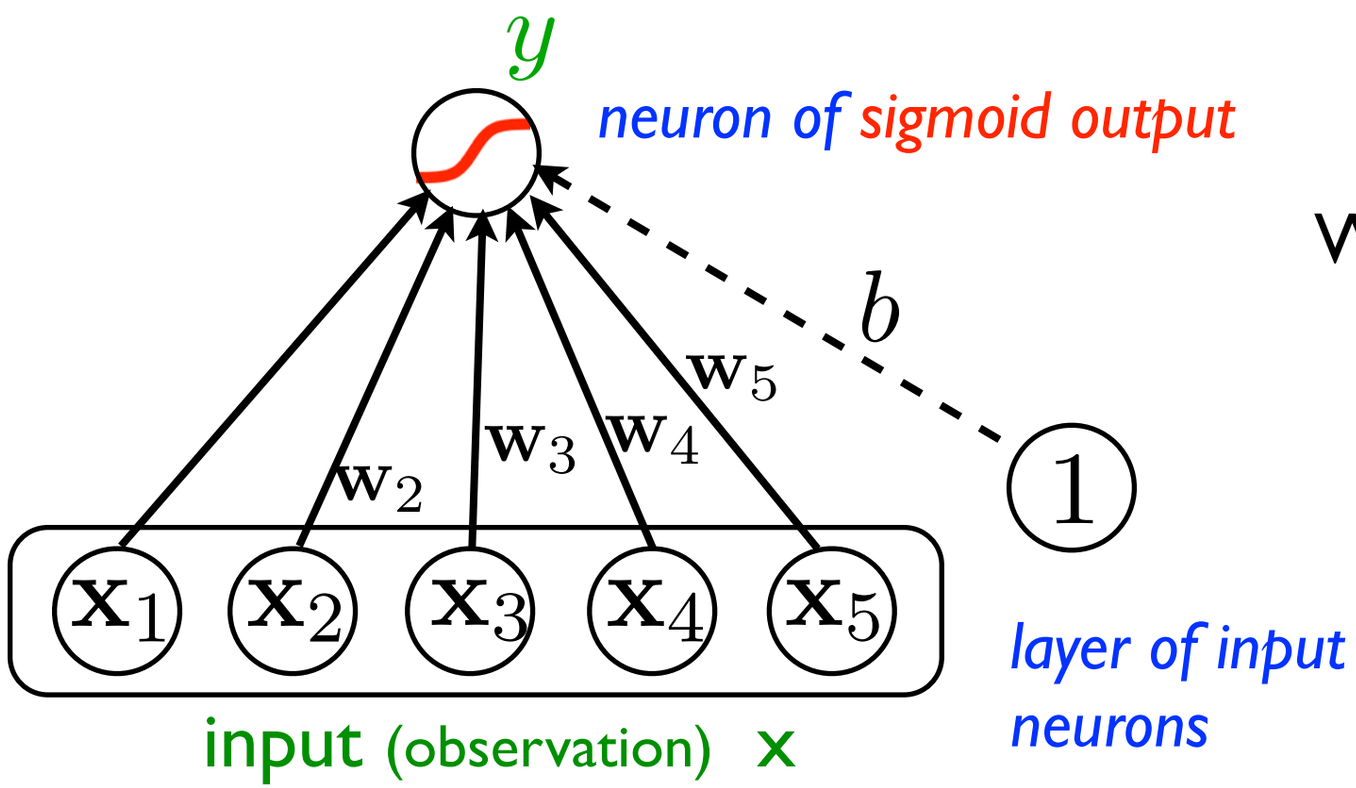
Neuron with sigmoid activation

$$f_\theta(\mathbf{x}) = f_{\mathbf{w},b}(\mathbf{x}) = \underbrace{\text{sigmoid}}(\langle \mathbf{w}, \mathbf{x} \rangle + b)$$

non-linearity, transfert function, activation

logistic $$\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$$

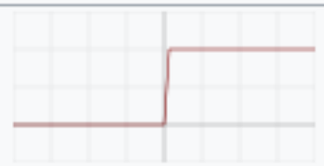
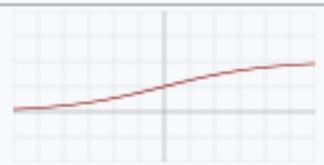
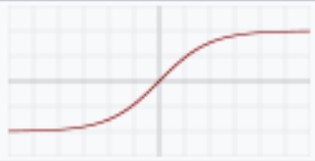
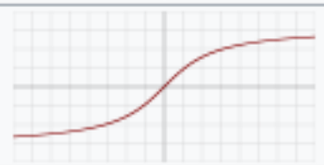
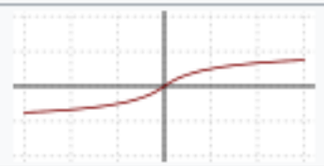
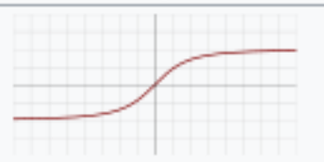
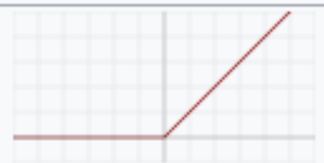
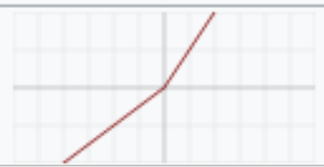




We can see the sigmoid as

- A differentiable alternative to the indicator function (step function)
- An approximation of the "pulse rate" response in biological neurons

Other activation functions

Name	Plot	Equation
Identity		$f(x) = x$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Logistic (a.k.a. Sigmoid or Soft step)		$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}}$ [1]
TanH		$f(x) = \tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$
ArcTan		$f(x) = \tan^{-1}(x)$
Softsign[8][9]		$f(x) = \frac{x}{1 + \lvert x \rvert}$
Inverse square root unit (ISRU)[10]		$f(x) = \frac{x}{\sqrt{1 + \alpha x^2}}$
Rectified linear unit (ReLU)[11]		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$
Parameteric rectified linear unit (PReLU)[13]		$f(\alpha, x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

Reminder: Logistic regression
(case of two classes)

Attention: despite the name it is a classification algorithm

For a binary classification task: $t \in \{0, 1\}$

We want to estimate the conditional **probability**: $y \simeq P(t = 1|\mathbf{x})$

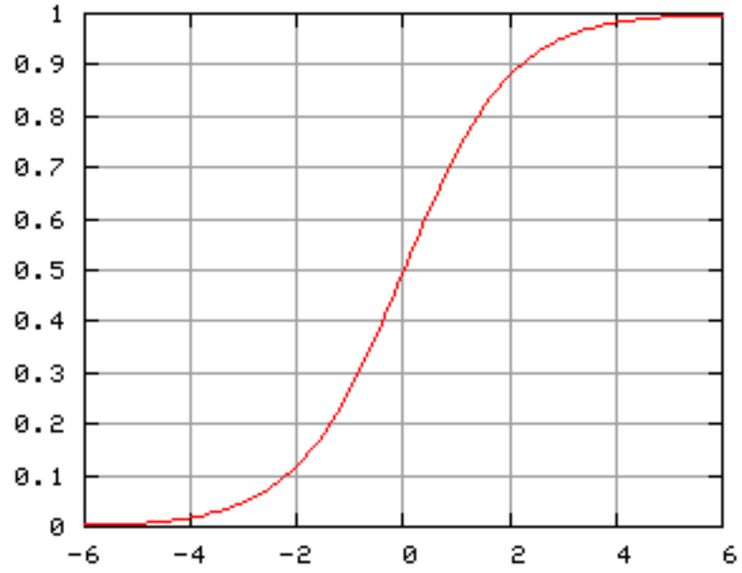
We choose $y \in [0, 1]$

A non-linear prediction function giving a probability

$$y = f_\theta(\mathbf{x}) = f_{\mathbf{w},b}(\mathbf{x}) = \underbrace{\text{sigmoid}(}\ \langle \mathbf{w}, \mathbf{x} \rangle + b)$$

non-linearity, transfer or activation function

logistic



The logistic sigmoid is the inverse of the logit link function in the terminology of Generalized Linear Models (GLMs).

Cross-entropy loss

$$L(y, t) = -(t \ln(y) + (1 - t) \ln(1 - y))$$

Here, no analytical solution
but the optimization is convex

Empirical risk minimization

We must specify:

- A parametric form for our functions, f_θ
- A specific cost (loss) function $L(y, t)$

So we define the empirical risk as:

$$\hat{R}(f_\theta, D_n) = \sum_{i=1}^{n} L(f_\theta(\mathbf{x}^{(i)}), t^{(i)})$$

i.e. total loss on the training set

Learning amounts to finding the optimal values for the parameters:

$$\theta^\star = \arg\min_\theta \hat{R}(f_\theta, D_n)$$

It is the principle of empirical risk minimization.

Other possibility:

optimization by gradient descent

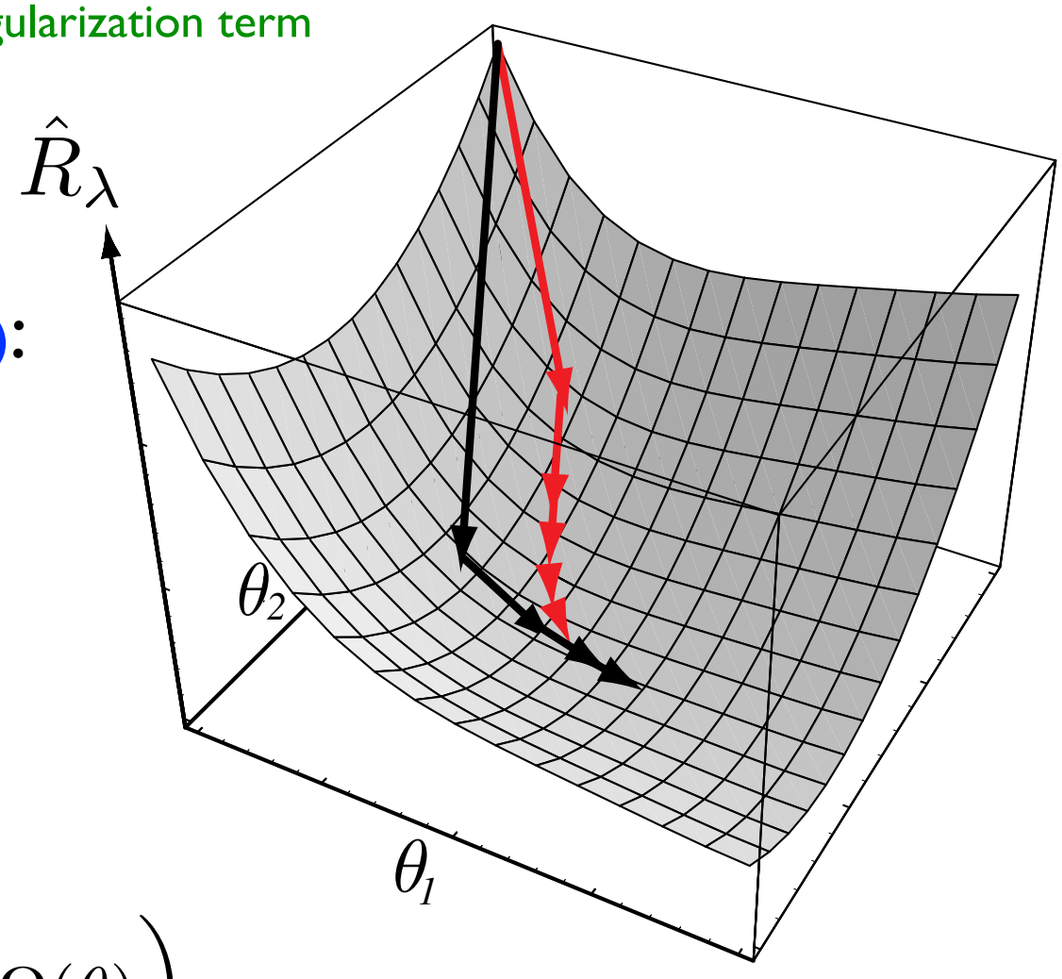
$$D_n = \{(x^{(1)}, t^{(1)}), \ldots, (x^{(n)}, t^{(n)})\} \qquad t^{(i)} \in \{0, 1\}$$

$$\hat{R}_\lambda(f_\theta, D_n) = \left(\sum_{i=1}^{n} L(f_\theta(\mathbf{x}^{(i)}), t^{(i)}) \right) + \underbrace{\lambda \Omega(\theta)}$$

empirical risk

regularization term

- we initialize the parameters randomly
- we update them iteratively following the gradient



Either batch gradient descent (whole dataset):

Loop: $\theta \leftarrow \theta - \eta \frac{\partial \hat{R}_\lambda}{\partial \theta}$

$$= \left(\sum_{i=1}^{n} \frac{\partial}{\partial \theta} L(f_\theta(\mathbf{x}^{(i)}), t^{(i)}) \right) + \lambda \frac{\partial}{\partial \theta} \Omega(\theta)$$

Or stochastic gradient descent:

Loop:

For i in 1...n

$$\theta \leftarrow \theta - \eta \frac{\partial}{\partial \theta} \left(L(f_\theta(\mathbf{x}^{(i)}), t^{(i)}) + \frac{\lambda}{n} \Omega(\theta) \right)$$

Or other variants of the gradient descent idea

(conjugate gradient, Newton's method, natural gradient, ...)

Limitation of logistic regression:

it remains a linear classifier!

We decide class 1 if P(t=1|x) > 0.5 (class 0 otherwise).

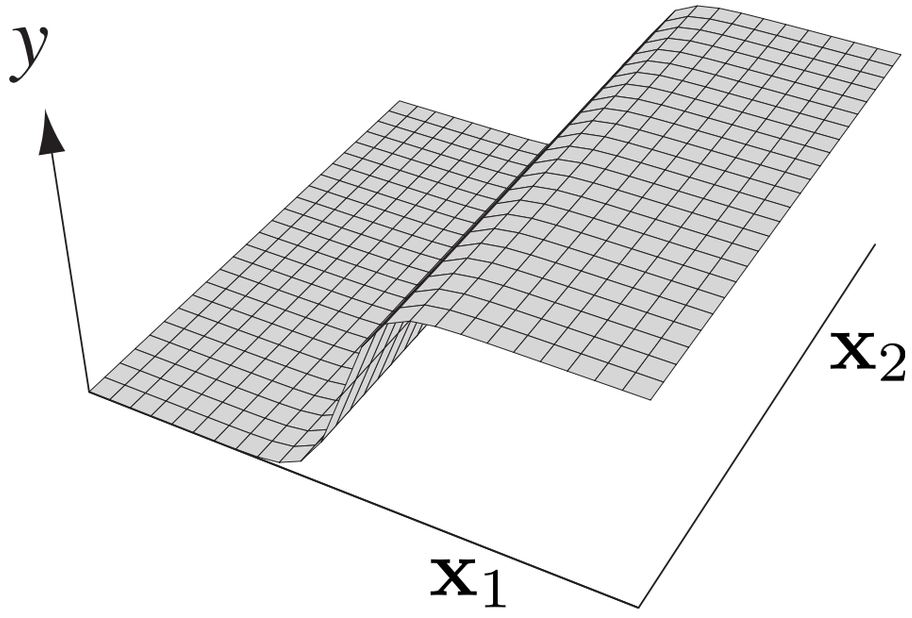
Or sigmoid($\langle \mathbf{w},\mathbf{x} \rangle + b$) > 0.5 $\equiv \langle \mathbf{w},\mathbf{x} \rangle + b > 0$.

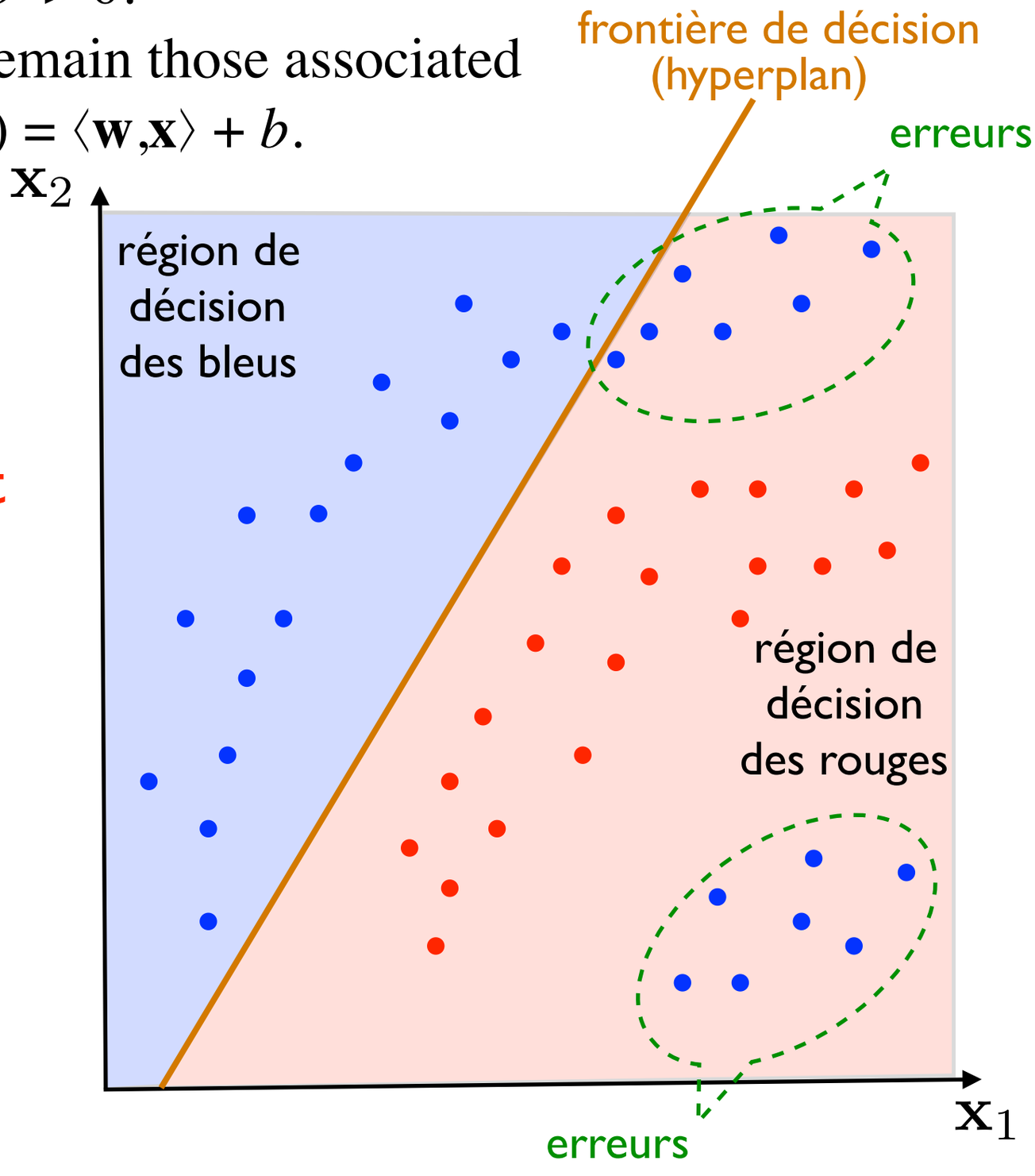
So the regions and decision boundaries remain those associated with the linear discriminant function $g(\mathbf{x}) = \langle \mathbf{w},\mathbf{x} \rangle + b$.

The decision boundary (surface) is a hyperplane

➠ inappropriate if the classes are not linearly separable

(eg: figure)





How can we build a *non-linear* classifier using a linear classifier?

An old trick...

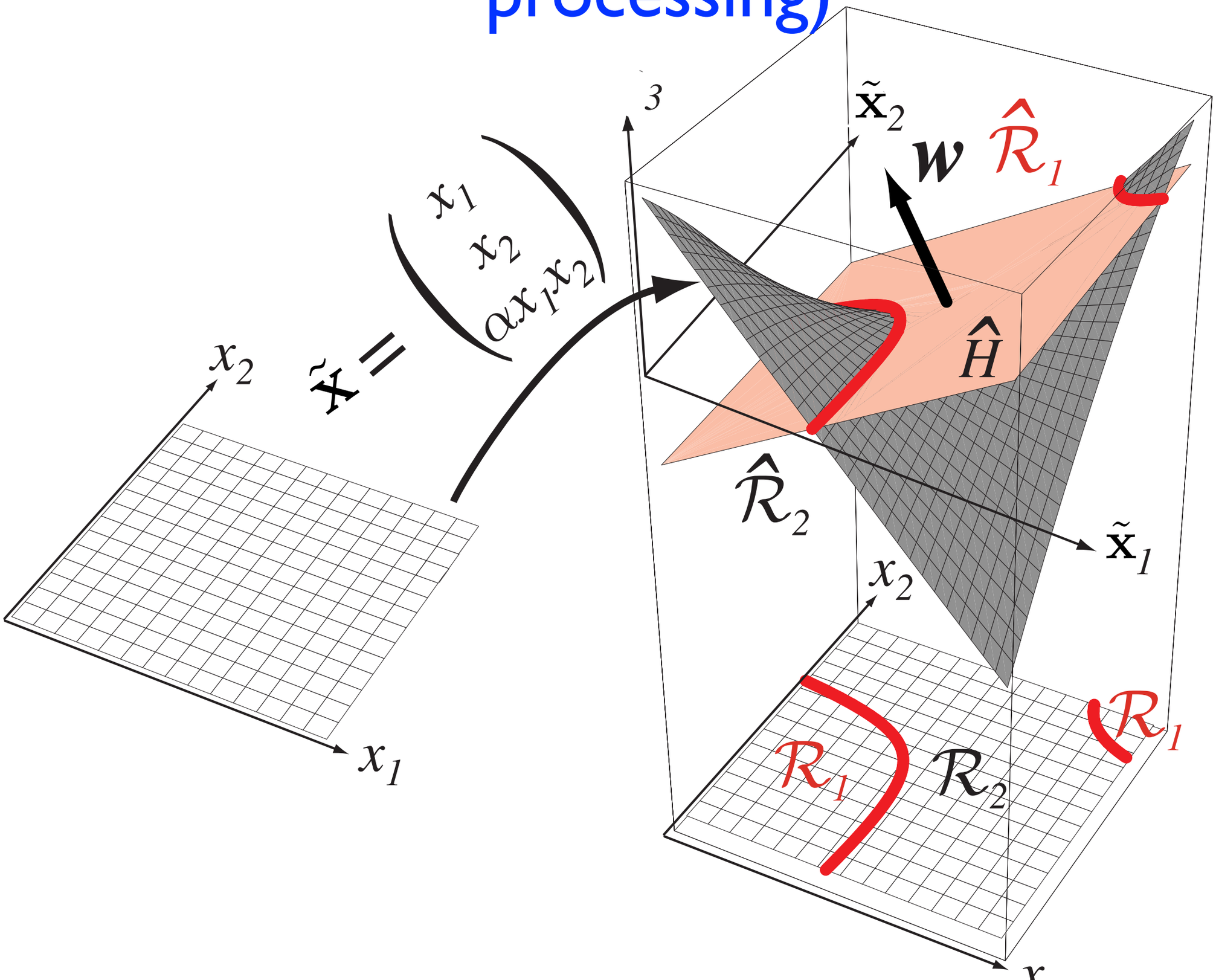
- **First,** apply a non-linear transform to *x* (projection into a new feature space)

$$\tilde{\mathbf{x}} = \phi(\mathbf{x})$$

non-linear feature map

- Learn a linear discriminant function in this new space
  (using one of the many algorithms we saw for learning linear classifiers)
- The separating hyperplan in this new space corresponds to a non-linear decision boundary in the original space!
- And voilà! ⇨ A non-linear classifier (with increased capacity).

Ex. *a priori* linear transform (pre-processing)



But which transformation?

Three way to obtain a non-linear feature map $\tilde{\mathbf{x}} = \phi(\mathbf{x})$

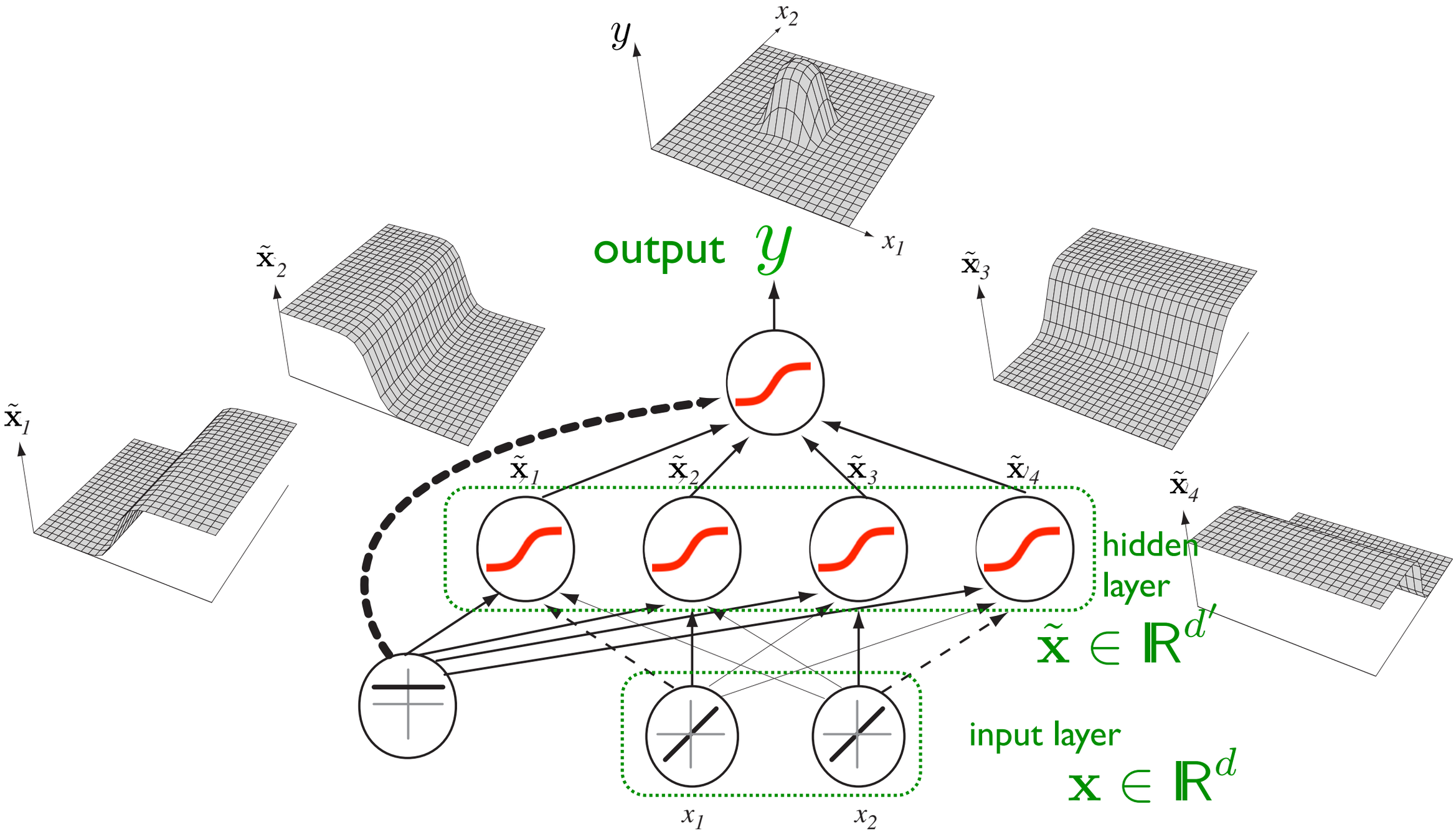
- Explicitly choose a fixed transformation *a priori*
  - ➠ Previous example
- Implicitly choose a fixed transformation *a priori*
  - ➠ Kernel method (kernelized SVM, kernelized logistic regression, ...)
- Learn the parameters of a parametrized feature map:
  - ➠ Neural networks

    Multilayer perceptron (MLP)

Neural network:

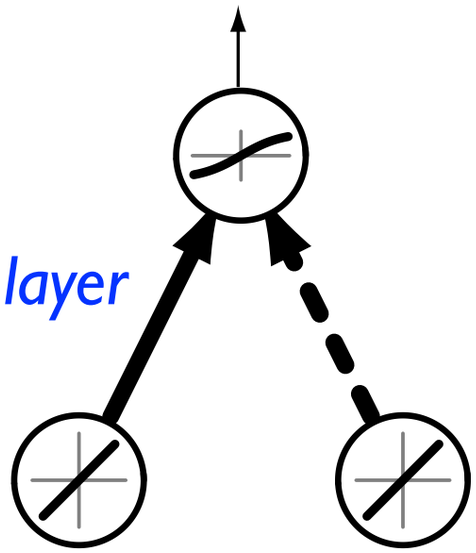
MLP with a hidden layer with 4 hidden units

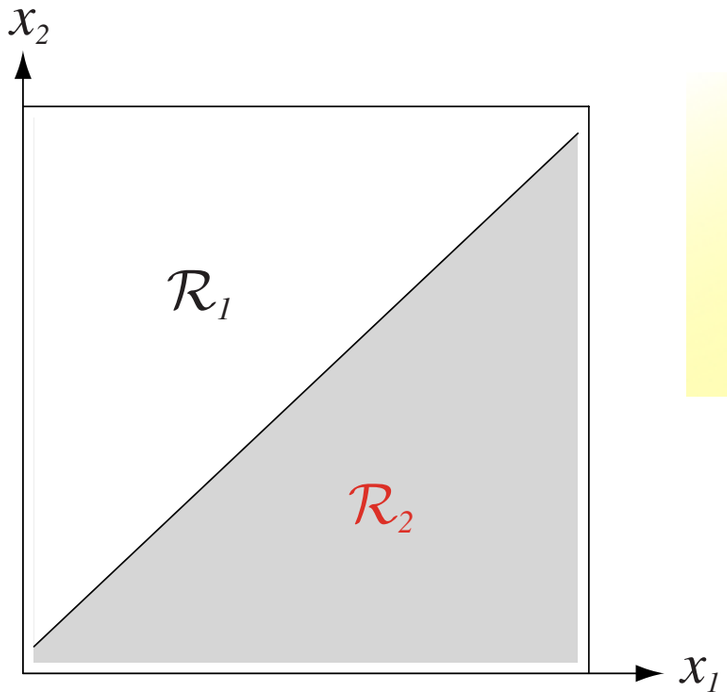
$$L(y,t) = -(t\ln(y) + (1-t)\ln(1-y))$$



Expressiveness/Capacity of MLPs with one hidden layer

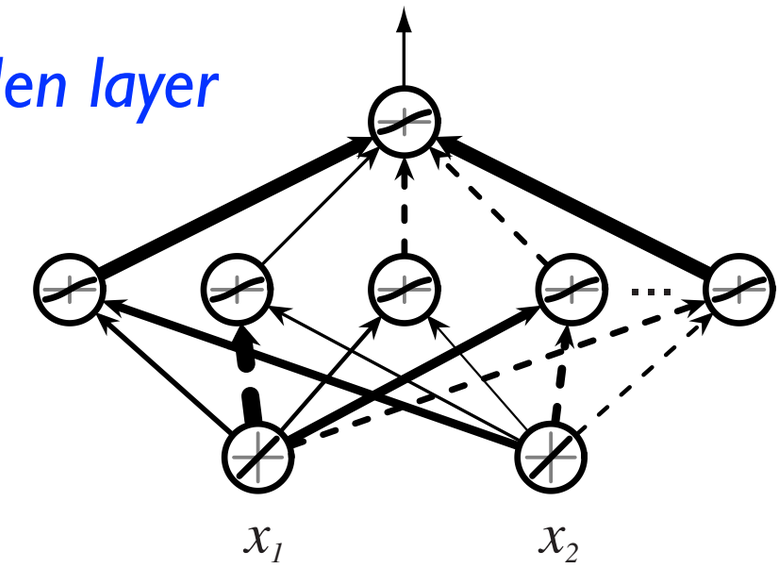
without hidden layer

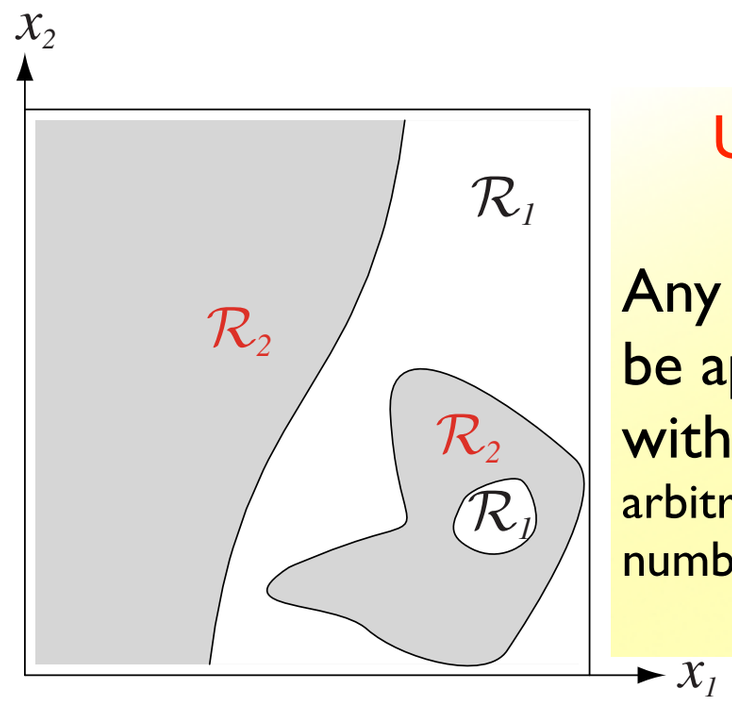




== Logistic regression boundary decision is limited to an hyperplan

one hidden layer





Universal approximation Theorem

Any continuous function can be approximated by an MLP with 1 hidden layer (to an arbitrary precision, increasing the number of hidden neurons)

Neural networks (MLP)

with *d'* (sigmoid) hidden neurons and one (sigmoid) output neuron.

Function:

$$y = f_\theta(\mathbf{x}) = \text{sigmoid}\left(\langle \mathbf{w}, \underbrace{\tilde{\mathbf{x}}}\rangle + b\right)$$

$$\tilde{\mathbf{x}} = \text{sigmoid}(\underbrace{\mathbf{W}^{\text{hidden}}}_{d' \times d}\mathbf{x} + \underbrace{\mathbf{b}^{\text{hidden}}}_{d' \times 1})$$

Parameters:

$$\theta = \{\mathbf{W}^{\text{hidden}}, \mathbf{b}^{\text{hidden}}, \mathbf{w}, b\}$$

Parameters optimization on the training data *(network training)*:

$$\theta^\star = \arg\min_\theta \underbrace{\hat{R}_\lambda(f_\theta, D_n)}_{\left(\underbrace{\sum_{i=1}^{n} L(f_\theta(\mathbf{x}^{(i)}), t^{(i)})}_{\text{empirical risk}}\right) + \underbrace{\lambda\Omega(\theta)}_{\substack{\text{regularization term} \\ \text{(weight decay)}}}}$$

Neural networks (MLP)

with *d'* (sigmoid) hidden neurons and *m* (sigmoid) output neurons.

Function:

$$y = f_\theta(\mathbf{x}) = \text{sigmoid}(\underbrace{\mathbf{W}^{\text{out}}}_{m \times d'} \tilde{\mathbf{x}} + \mathbf{b}^{\text{out}})$$

$$\tilde{\mathbf{x}} = \text{sigmoid}(\underbrace{\mathbf{W}^{\text{hidden}}}_{d' \times d} \mathbf{x} + \underbrace{\mathbf{b}^{\text{hidden}}}_{d' \times 1})$$

Parameters:

$$\theta = \{\mathbf{W}^{\text{hidden}}, \mathbf{b}^{\text{hidden}}, \mathbf{W}^{\text{out}}, \mathbf{b}^{\text{out}}\}$$

Parameters optimization on the training data *(network training)*:

$$\theta^\star = \arg\min_\theta \underbrace{\hat{R}_\lambda(f_\theta, D_n)}_{\left(\underbrace{\sum_{i=1}^{n} L(f_\theta(\mathbf{x}^{(i)}), t^{(i)})}_{\text{empirical risk}}\right) + \underbrace{\lambda\Omega(\theta)}_{\substack{\text{regularization term} \\ \text{(weight decay)}}}}$$

Which loss to use for the output (sigmoid) neurons?

Quadratic loss

$$L(y,t) = \|y-t\|^2 = \sum_{k=1}^{m}(y_k - t_k)^2$$

OR Cross-entropy

$$L(y,t) = -\sum_{k=1}^{m} t_k \ln(y_k) + (1-t_k)\ln(1-y_k)$$

Hyper-parameters controlling the capacity

✳ A network has a set of *parameters:* θ

➠ optimized on the training data using gradient descent.

✳ One also needs to tune *hyper-parameters*

controlling the network's capacity:

- number of hidden neurons d'
- amount of regularization λ (*weight decay*)
- early stopping (number of training epochs)

and other inherhent to the optimisation algorithms (e.*g. learning rate*)

➠ tuned using model selection techniques **ona validation set** disjoint from the training data.

Hyper-parameter tuning

$D=$

$(\mathbf{x}^{(1)}, t^{(1)})$

$(\mathbf{x}^{(2)}, t^{(2)})$

$\vdots$

$(\mathbf{x}^{(N)}, t^{(N)})$

Split available data in 3 parts

Training set (size n)

Validation set (size n')

Test set (size m)

For each combination of values for the hyper-parameters:

1) *Train the network: find the parameters minimizing the regularized empirical risk on the training data using gradient descent.*

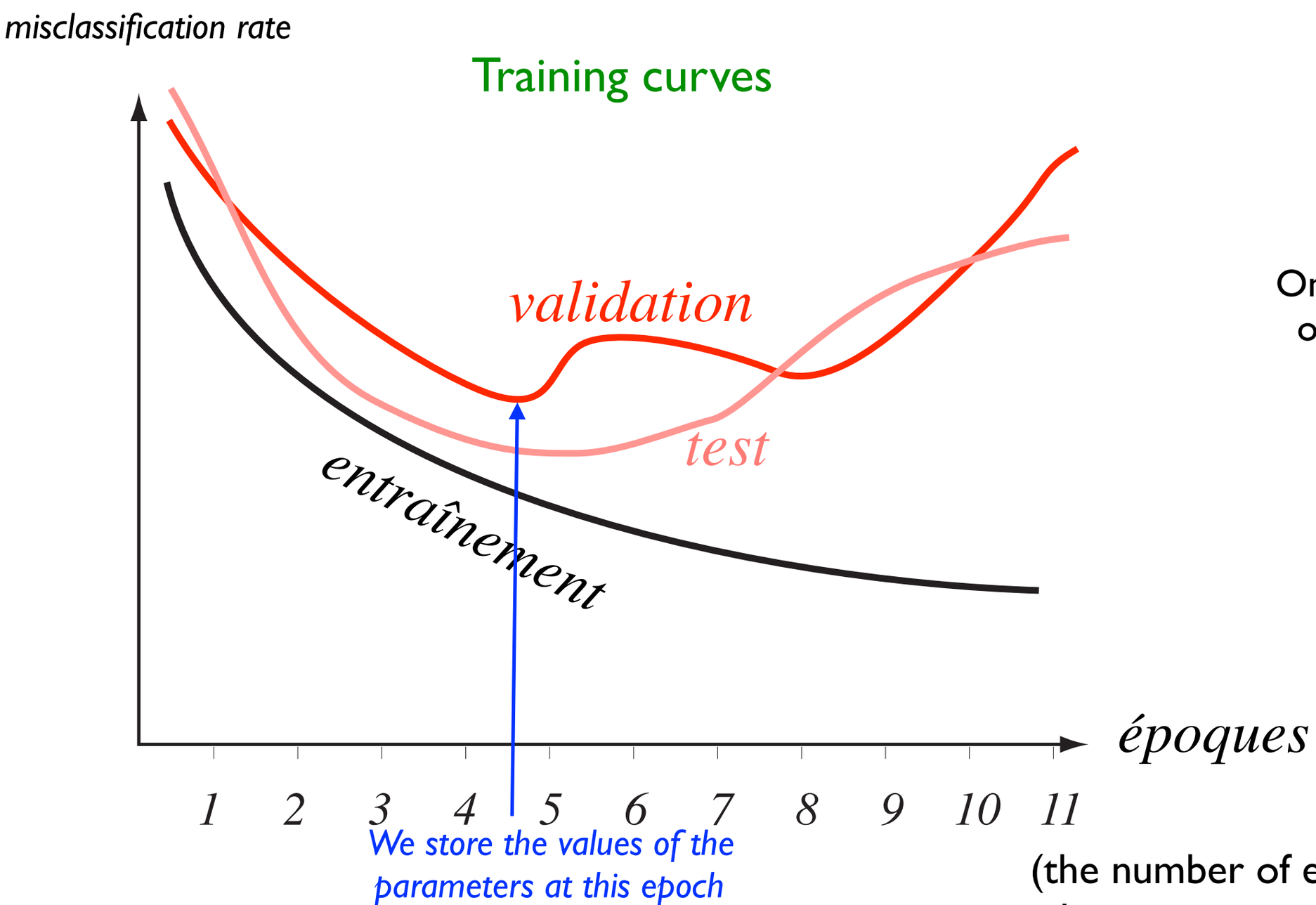
2) *Evaluate the performance on the validation set using the criterion/loss we actually want to minimize (ex. classification error instead of cross-entropy)*

Store the value of the hyper-parameters and the network that obtained the best validation performance.
(Potentially retrained on both training and validation data).

Evaluate the generalization performance on the test set, which has never been used, neither to find parameters nor hyper-parameters (in order to get a non-biased estimate of the generalization error).

If there is not enough training data, we can use *k*-fold cross-validation or leave-one-out ("jack-knife")

Ex: Early stopping



One *epoch* I = I pass of the optimizer over the whole training data.

(the number of epochs can be seen as an hyper-parameter controlling capacity)

Summary

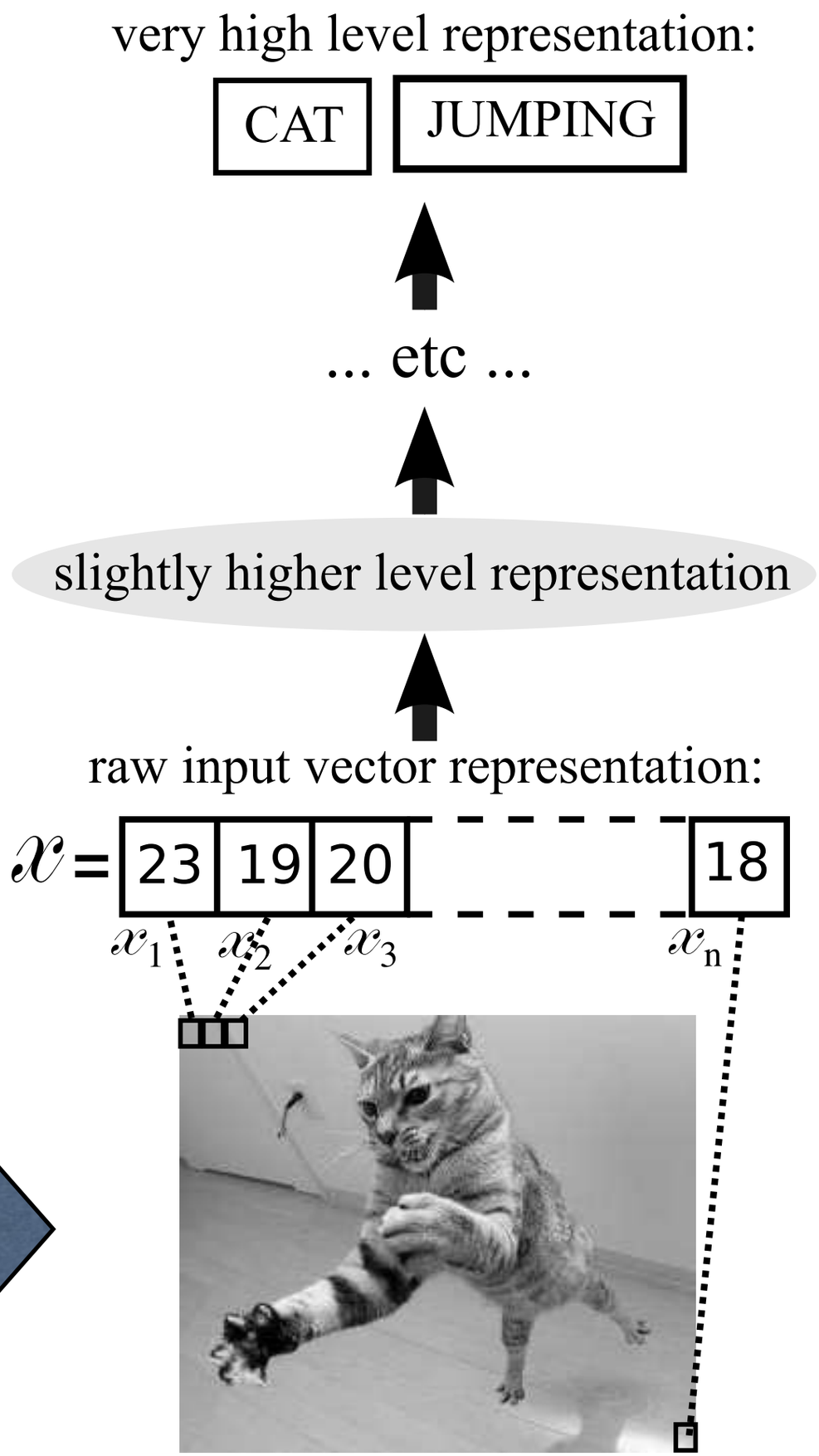
- *Feed-forward* neural networks (such as MLPs) are parametrized non-linear functions.
- ... trained using gradient descent to minimize some objective function on the training set.
- There is a huge number of architecture choices (ex: size of hidden layers, activation type) and other hyper-parameters controlling capacity, need to be tuned using a rigorous model selection techniques.

  (ex: cross-validation, using separate validation data).

Note: there many different kinds of artificial neural networks...

Deep MLPs

Multi-level representations



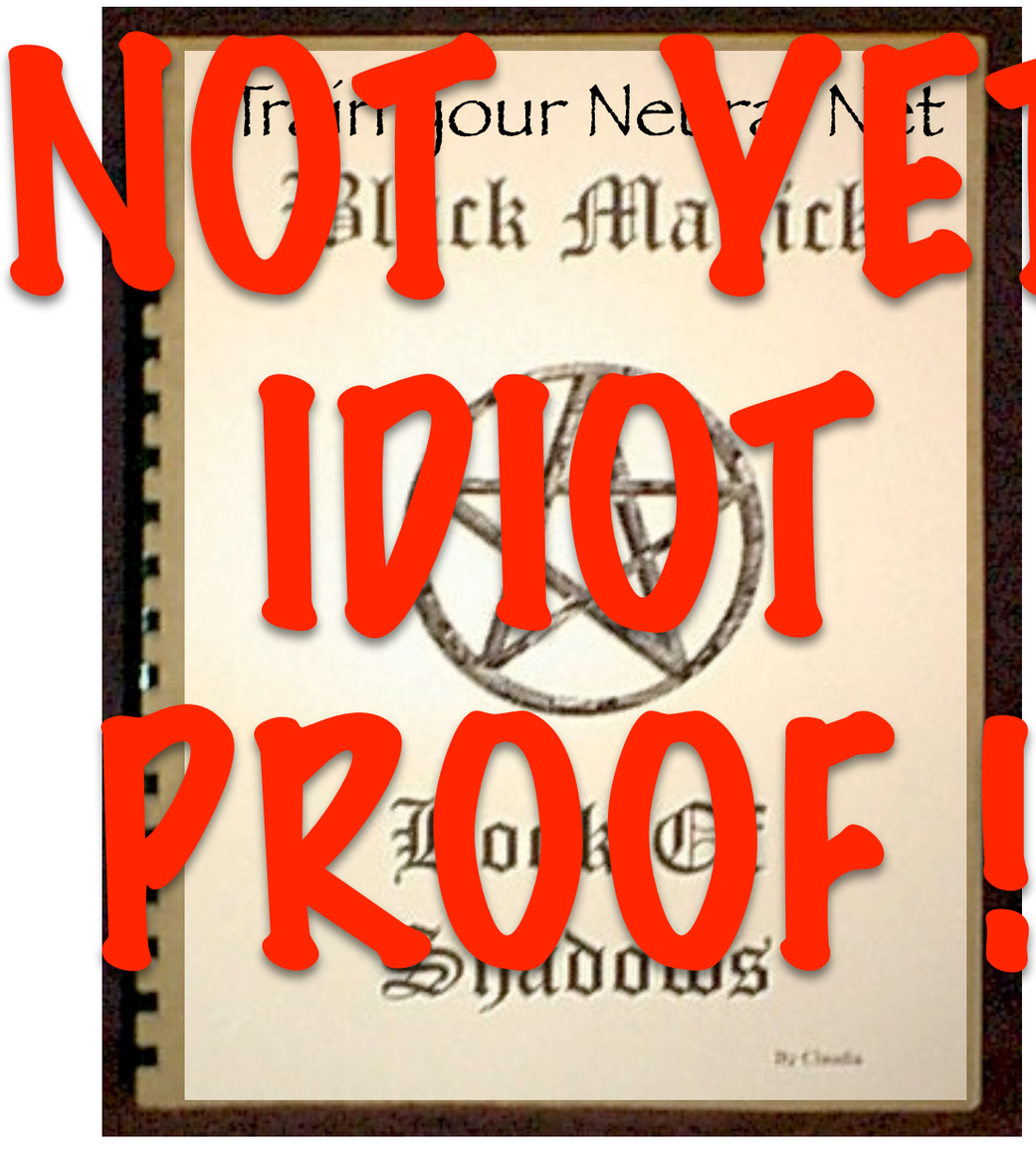
Important practical details

- Efficient computation of the gradient is done using the back-propagation algorithm (clever way to use the chain rules in order to optimize memory usage and avoid repeating computations).
- It is important to start with the good representation of the input data (categorical attributes with one-hot encoding, continuous attributes need to be normalized)
- It is important to use an appropriate range to randomly initializae the weights for the network.
- Gradient descent can be very sensitive to the values of hyper-parameters (learning rate, ...)

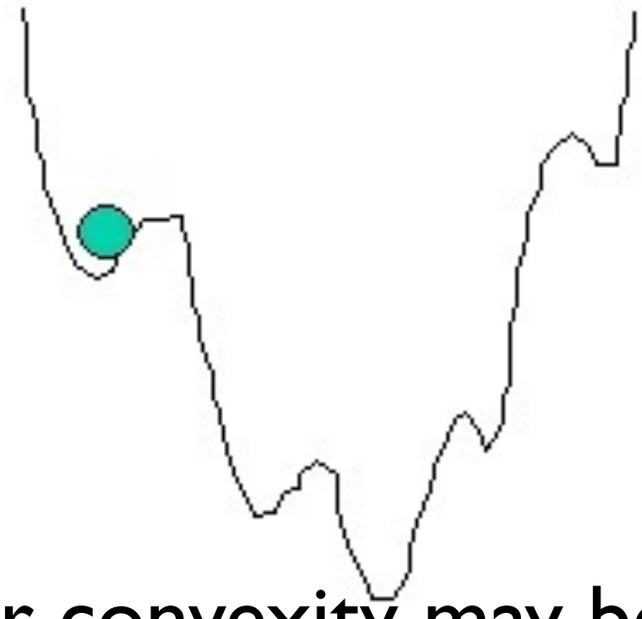
=> More details in the following demos.

Neural networks are back!

- Difficult to train
  (lots of hyper-parameters to tune)



- Non-convex optimization
  ➠ local minima: solution depends where we start...



However convexity may be a too restrictive assumption

Convex optimization is easier and simpler, but real-world challenging problems may necessitate non-convex models.