DD2380 Artificial Intelligence
Machine Learning 1: Deep Neural Networks

Alexandre Proutiere

September 6, 2017

KTH (The Royal Institute of Technology)

Outline of today's lecture

0. Introduction: Supervised vs. Unsupervised learning

1. Supervised Learning
    A. Regression and Classification
    B. Neural Networks (Deep Learning)

2. Reinforcement Learning

Outline of today's lecture

0. **Introduction: Supervised vs. Unsupervised learning**

1. Supervised Learning
   A. Regression and Classification
   B. Neural Networks (Deep Learning)

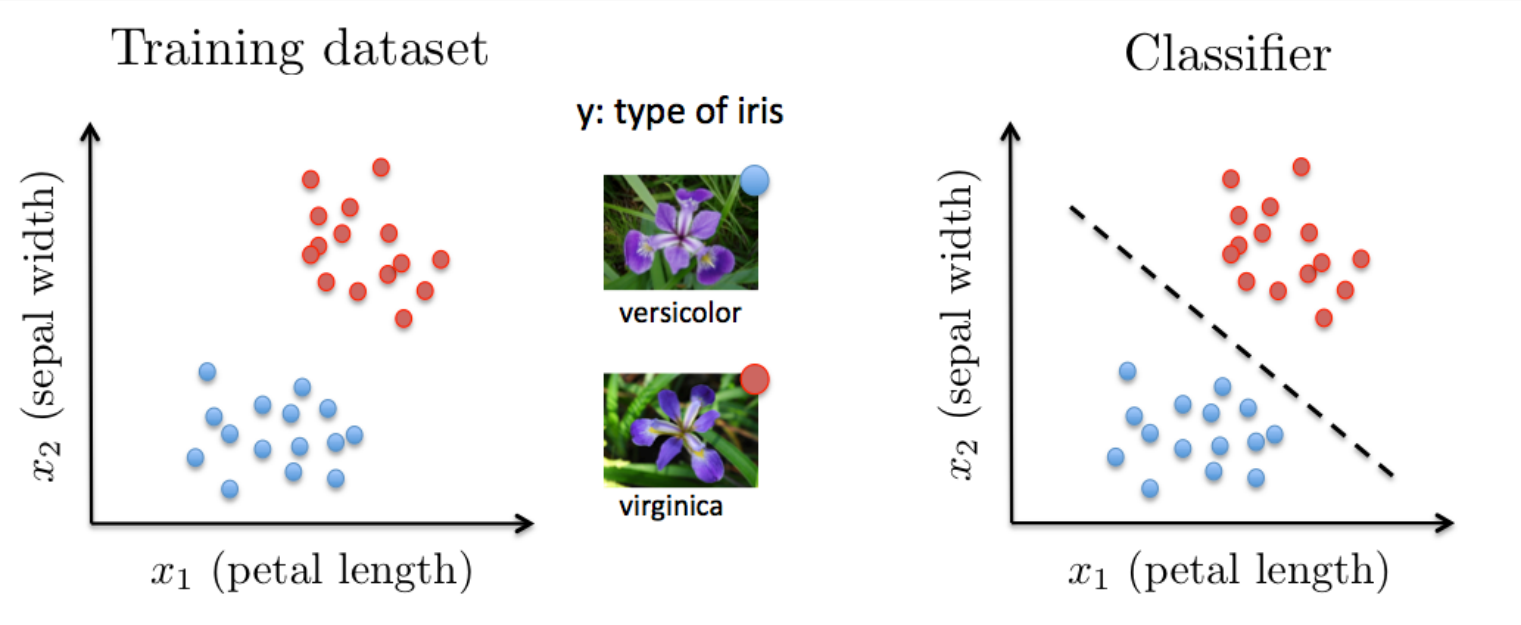
2. Reinforcement Learning

"Machine Learning explores the study and construction of algorithms that can learn from and make predictions on data" (Wikipedia)

Learning

Supervised (or Predictive) learning (or learning from examples)

Learn a mapping from inputs x to outputs y, given a labeled set of input-ouput pairs (the training set)
$D_n = \{(X_i, Y_i), i = 1, \ldots, n\}$

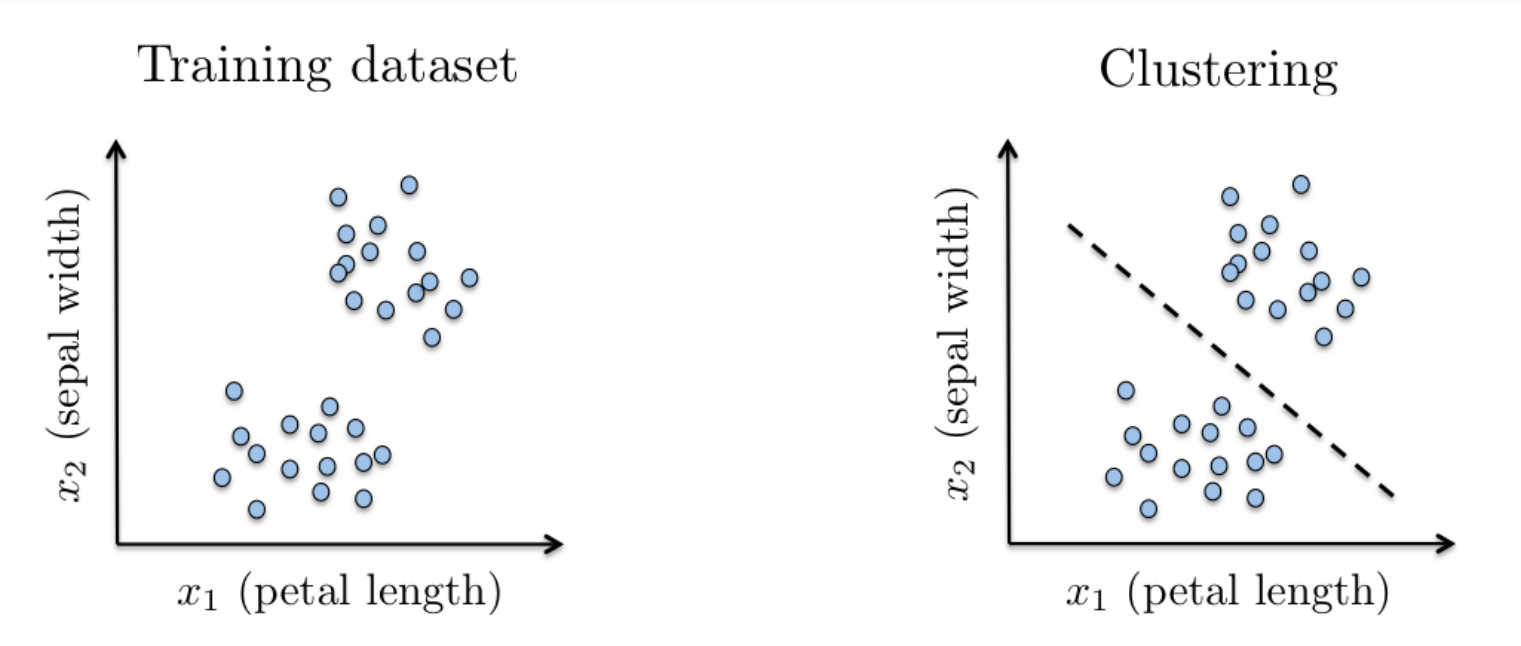


We learn the classification function $f = 1$ if versicolor, $f = -1$ if virginica

Learning

Unsupervised (or Descriptive) learning

Find interesting patterns in the data $D_n = \{X_i, i = 1, \ldots, n\}$



We learn there are 2 distinct types of iris and how to distinguish them!

Examples

Supervised learning:

- digit, flower, picture, ... recognition
- music classification
- predict prices
- predict the outcome of chemical reactions
- source separation: identify the instruments present in a recording
- ...

Unsupervised learning:

- classification (without training set): spam filter, news (google news), ...
- identify structures and causes in the data: e.g. J. Snow – cholera deaths vs pollution
- image segmentation
- ...

Outline of today's lecture

0. Introduction: Supervised vs. Unsupervised learning

1. **Supervised Learning**
   A. **Regression and Classification**
   B. Neural Networks (Deep Learning)

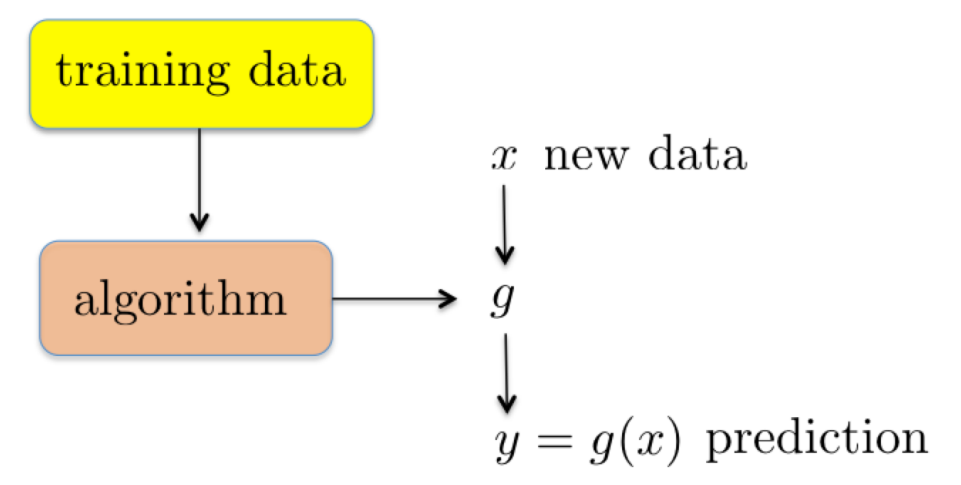
2. Reinforcement Learning

Supervised learning: regression vs. classification

- Training set: $D_n = \{(X_i, Y_i), i = 1, \ldots, n\}$
  - Input features: $X_i \in \mathbb{R}^d$
  - Output: Y_i

$$Y_i \in \mathcal{Y} \begin{cases} \mathbb{R} & \textbf{regression} \text{ (price, position, etc)} \\ \text{finite} & \textbf{classification} \text{ (type, mode, etc)} \end{cases}$$

- y is a non-deterministic and complicated function of x
  i.e., $y = f(x, z)$ where z is unknown (e.g. noise). Goal: learn f.
- Learning algorithm:



Empirical risk

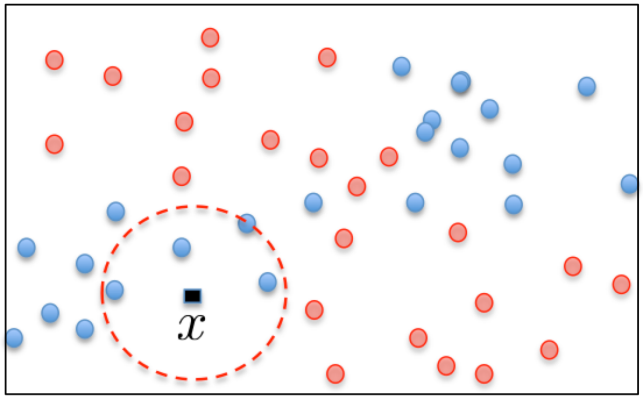
- Learning algorithm: $A : D_n \mapsto \hat{f}_n$, $\hat{f}_n$ estimates the true function f
- Performance of predictions defined through a loss function ℓ
  - **Example 1.** Regression, Least Squares (LS): $\mathcal{Y} = \mathbb{R}$, $\ell(y, y') = \frac{1}{2}|y - y'|^2$
  - **Example 2.** Classification in $\mathcal{Y} = \{0, 1\}$: $\ell(y, y') = 1_{y \neq y'}$
- **Empirical risk of estimate g:**

$$\hat{R}_n(g) := \frac{1}{n} \sum_{i=1}^{n} \ell(g(X_i), Y_i)$$

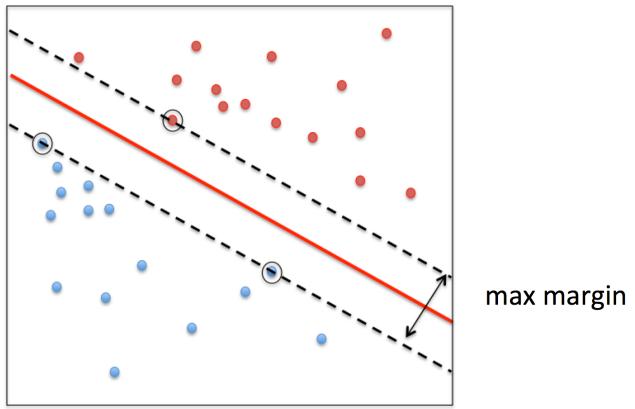
(often referred to as *in-sample* error)

Examples of classification algorithms

- Local averaging:
  k-nearest neighbors
  Nadaraya-Watson (Gaussian kernel)



- Support Vector Machine:
  Aim at finding an hyperplane
  "optimally" separating data



- Minimizing the empirical risk within a predefined set $\mathcal{F}$ of functions, called the "model". Example: Deep learning

Model selection

We look for $\hat{f}_n$, the estimate of the true function, in a particular set $\mathcal{F}$ of functions (e.g. for regression, linear or polynomial functions)

The choice of $\mathcal{F}$ is guided considering:

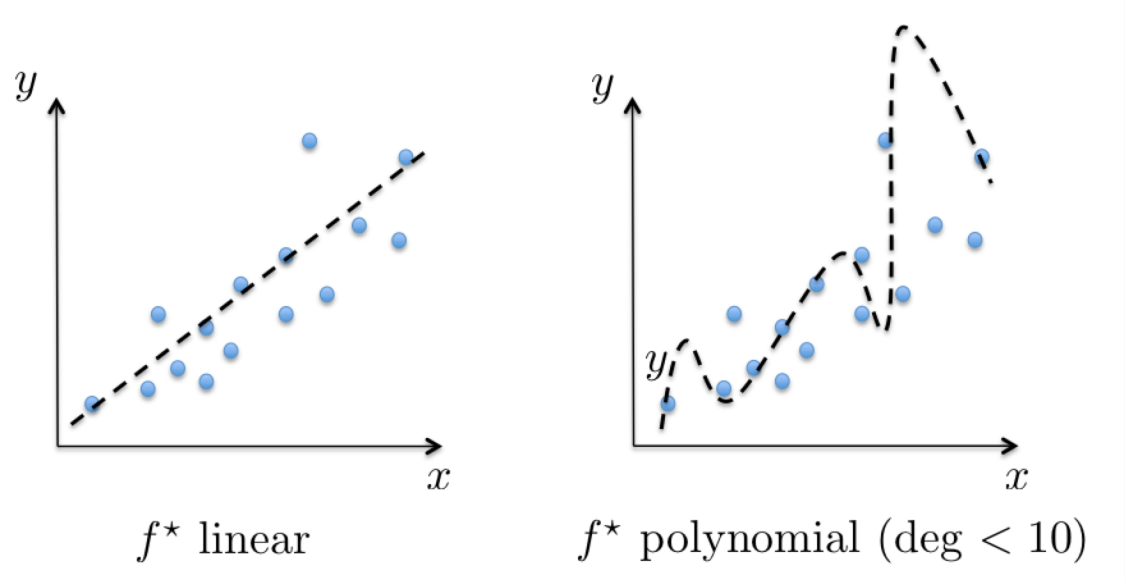
Expressibility. How large the class of functions that $\mathcal{F}$ can represent is.

Efficiency. How many parameters are required to approximate a function.

Learnability. How rapidly the model can be trained (sample complexity).

Model selection - Choosing $\mathcal{F}$

Avoid overfitting!



$f^\star$ linear $\quad$ $f^\star$ polynomial (deg < 10)

A few principles:

- **Occam**'s razor principle: choose the simplest of two models if they explain the data equally well
- **Hadamard**'s well posed problems: unique solution, smooth in the parameters

Regularization: minimize error$(\hat{f}_n)+\Omega(\hat{f}_n)$ – Penalizes the model complexity

Linear regression: Minimizing the empirical risk

- $\mathcal{F} =$ set of linear functions from $\mathbb{R}^d$ ($X_i \in \mathbb{R}^d$) to $\mathbb{R}$ ($Y_i \in \mathbb{R}$)
- Function $f_\theta \in \mathcal{F}$ parametrized by $\theta \in \mathbb{R}^{d+1}$: (by convention $x_0 = 1$)

$$f_\theta(x) = \theta_0 + \theta_1 x_1 + \ldots + \theta_d x_d = \theta^\top x$$

- Least square model. Empirical risk of θ:
  $\hat{R}_n(\theta) = \frac{1}{2n} \sum_{i=1}^n (f_\theta(X_i) - Y_i)^2$
- Minimal risk achieved for $\theta^\star = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$ where
  $\mathbf{y} = [Y_1 \ldots Y_n]^\top$ and $\mathbf{X}$ is a matrix whose i-th line is $X_i^\top$

Gradient Descents for Regression

Alternative methods to find $\theta^\star$: sequential algorithms.

Batch **gradient descent.** Repeat:

$$\forall k \in \{0, 1, \ldots, d\}, \theta_k := \theta_k - \alpha \sum_{i=1}^{n} (Y_i - f_\theta(X_i)) X_{ik}$$

Stochastic **gradient descent.** Repeat:

1. Select a sample i uniformly at random
2. Perform a descent using (X_i, Y_i) only

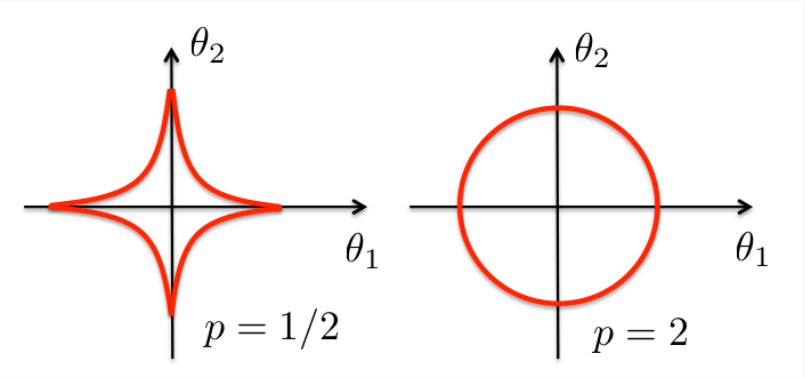
$$\forall k \in \{0, 1, \ldots, d\}, \theta_k := \theta_k - \alpha (Y_i - f_\theta(X_i)) X_{ik}$$

Linear regression: Regularization

- For $d > n$, $\mathbf{X}^\top \mathbf{X} \in \mathbb{R}^{d \times d}$ is not invertible, so $\theta^\star$ is not uniquely defined. Need to put additional constraints on the model to get a well-posed problem
- Regularization: add a cost for the magnitude of θ

$$\min_\theta \frac{1}{2n} \sum_{i=1}^{n} (f_\theta(X_i) - Y_i)^2 + \lambda \Omega(\theta)$$

Ridge: $\Omega(\theta) = \|\theta\|_2^2$
LASSO: $\Omega(\theta) = \|\theta\|_1$
ℓ_p: $\Omega(\theta) = \|\theta\|_p$



p **small** (< 1)**:** sparse solutions but hard optimization problems
p **large** (≥ 1)**:** less sparse solutions but convex optimization problems

Linear regression: Regularization

- Regularization: add a cost for the magnitude of θ

$$\min_{\theta} \frac{1}{2n} \sum_{i=1}^{n} (f_\theta(X_i) - Y_i)^2 + \lambda \Omega(\theta)$$

- $\lambda > 0$ controls the **bias-variance trade-off**
  - High value of λ: the data has a low weight in the objective function (low variance but high bias)
  - Low value of λ: the data has a high weight in the objective function (low bias but high variance)
- Solution for Ridge regression: $\theta^\star = (\mathbf{X}^\top \mathbf{X} + n\lambda I)^{-1} \mathbf{X}^\top \mathbf{y}$
  Prediction: For all $x \in \mathbb{R}^d$, $\hat{f}_\lambda(x) = \mathbf{y}^\top (\mathbf{X}^\top \mathbf{X} + n\lambda I)^{-1} \mathbf{X}^\top x$

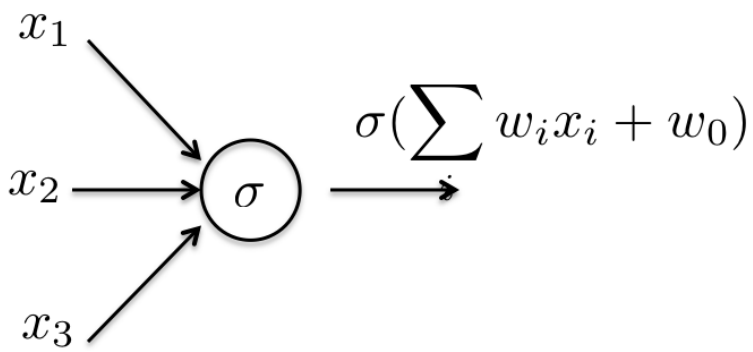
Outline of today's lecture

0. Introduction: Supervised vs. Unsupervised learning

1. Supervised Learning
   A. Regression and Classification
   B. **Neural Networks (Deep Learning)**
      (i) **From perceptron to deep networks**
      (ii) Computing with and training neural nets
      (iii) Why deep? Why does it work?

2. Reinforcement Learning

Neural networks

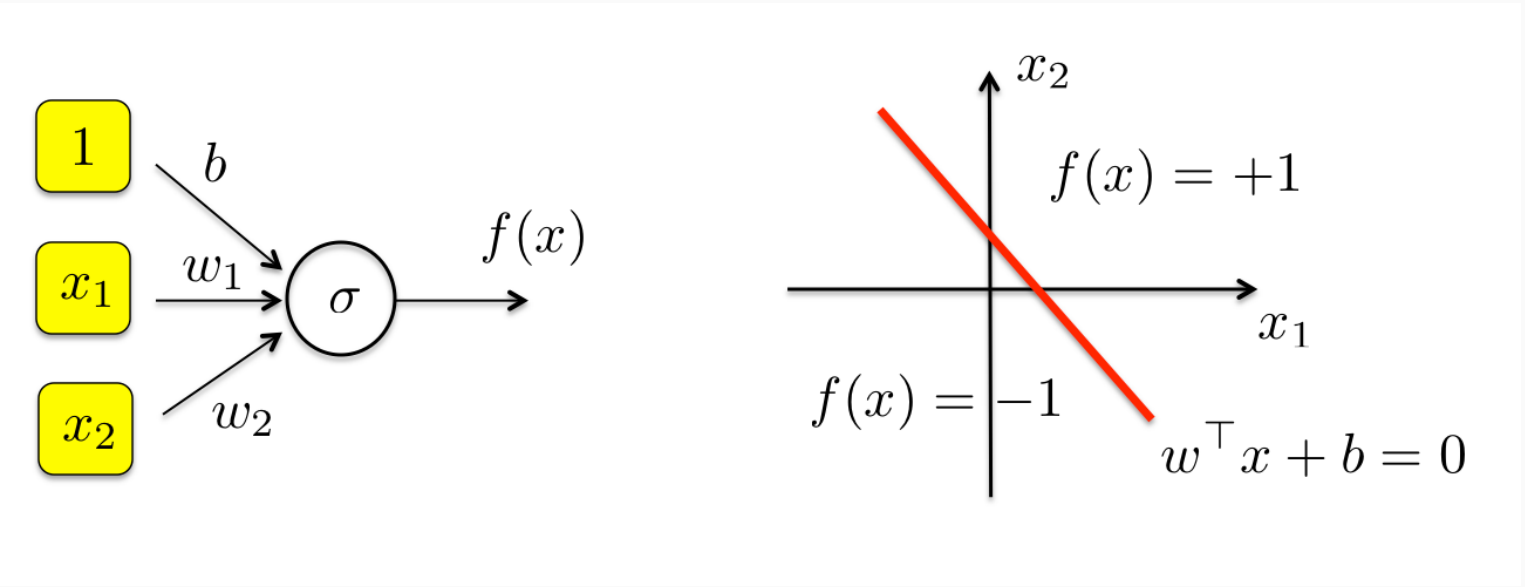


Loosely inspired by how the brain works[1]. Construct a network of simplified neurones, with the hope of approximating and learning any possible function

[1] Mc Culloch-Pitts, 1943

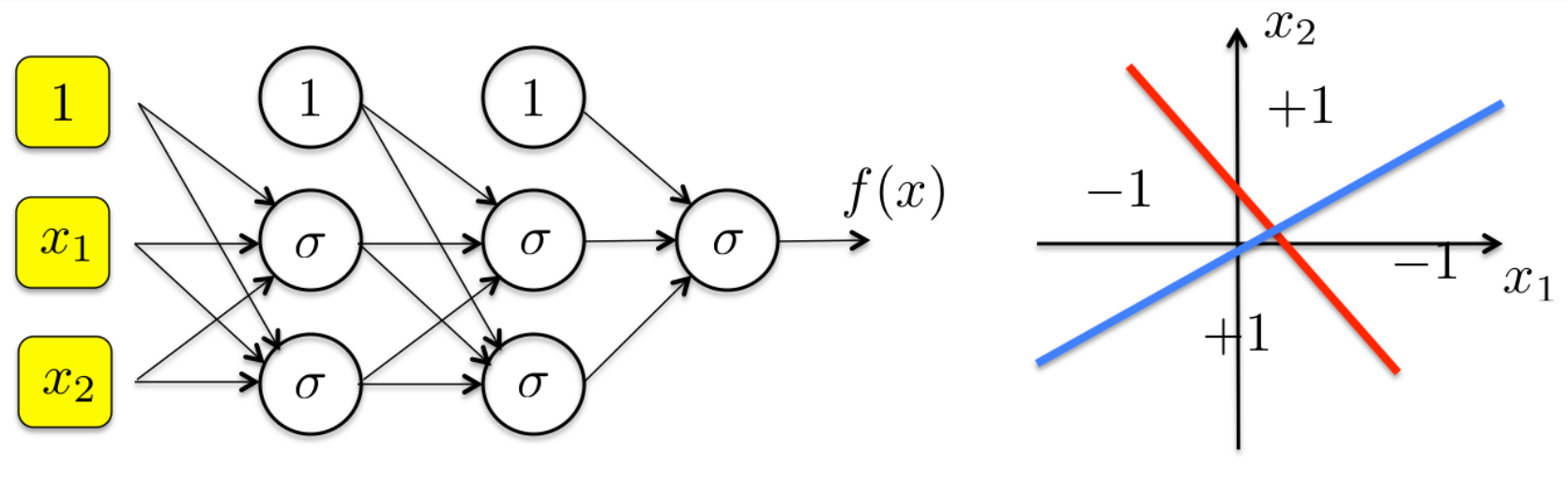
The perceptron

The first artificial neural network with one layer, and $\sigma(x) = \text{sgn}(x)$ (classification)
Input $x \in \mathbb{R}^d$, output in $\{-1, 1\}$. Can represent separating hyperplanes.



Multilayer perceptrons

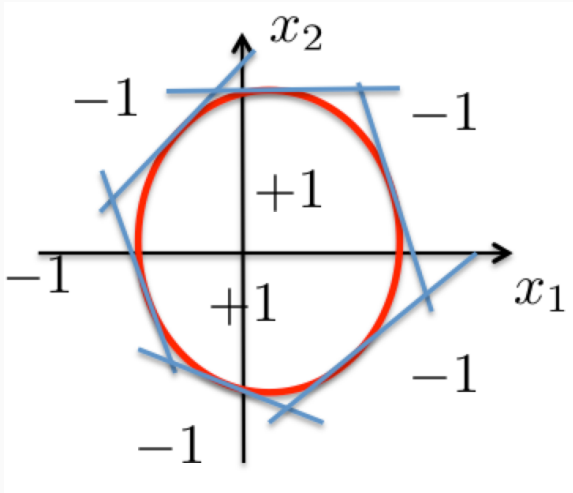
They can represent any function of $\mathbb{R}^d$ to $\{-1, 1\}$



... but the structure depends on the **unknown** target function f, and is difficult to optimise

From perceptrons to neural networks

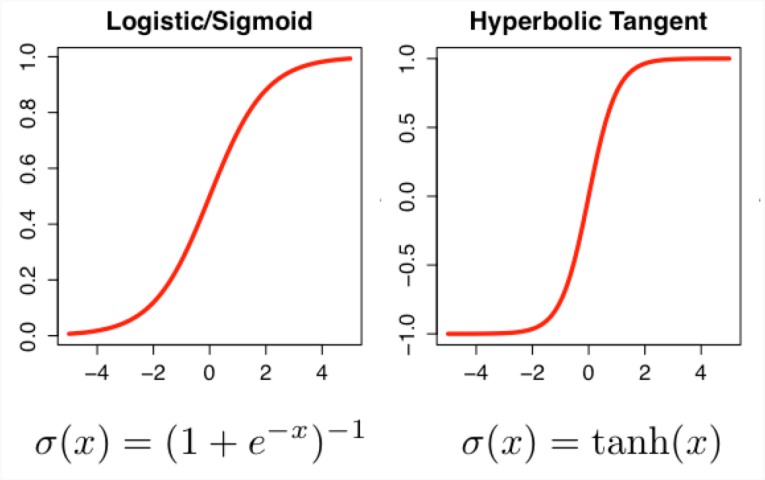
... and the number of layers can rapidly grow with the complexity of the function



A key idea to make neural networks practical: **soft-thresholding** ...

Soft-thresholding

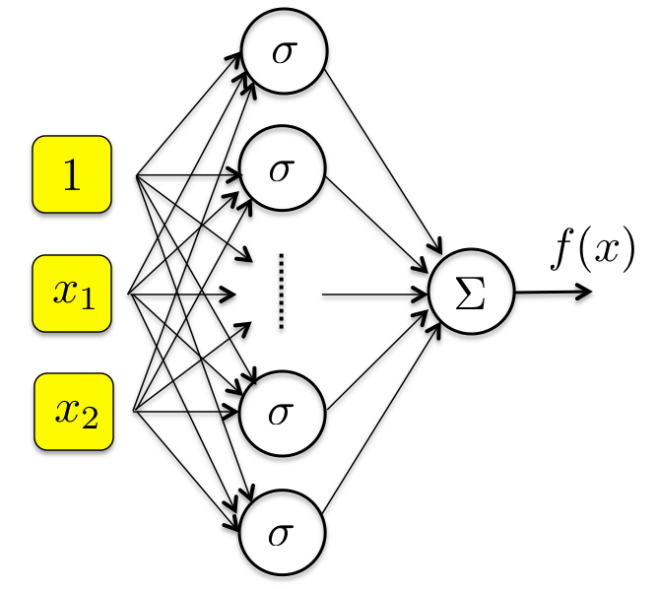
Replace hard-thresholding function σ by smoother functions



Theorem (Cybenko 1989) *Any continuous function f from $[0,1]^d$ to $\mathbb{R}$ can be approximated as a function of the form: $\sum_{j=1}^{N} \alpha_j \sigma(w_j^\top x + b_j)$, where σ is any sigmoid function.*

Soft-thresholding

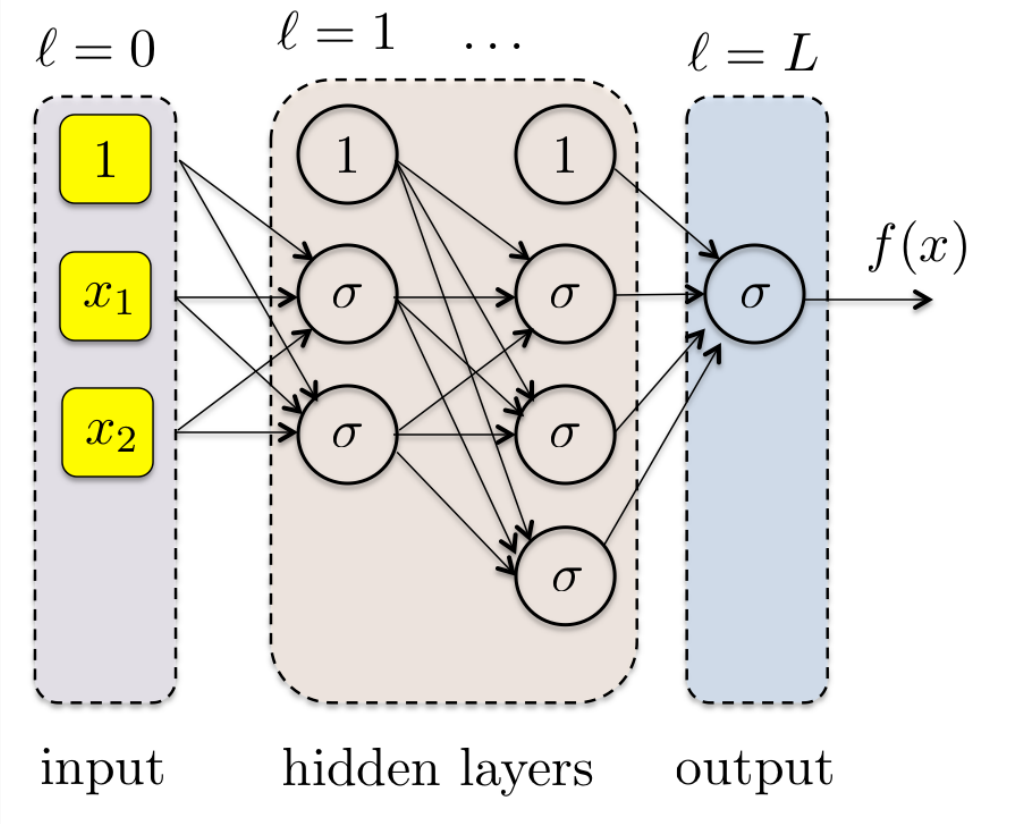
Cybenko's theorem tells us that f can be represented using a single hidden layer network ...



A non-constructive proof: how many neurones do we need? Might depend on f ...

Neural networks

A feedforward layered network (deep learning = enough layers)



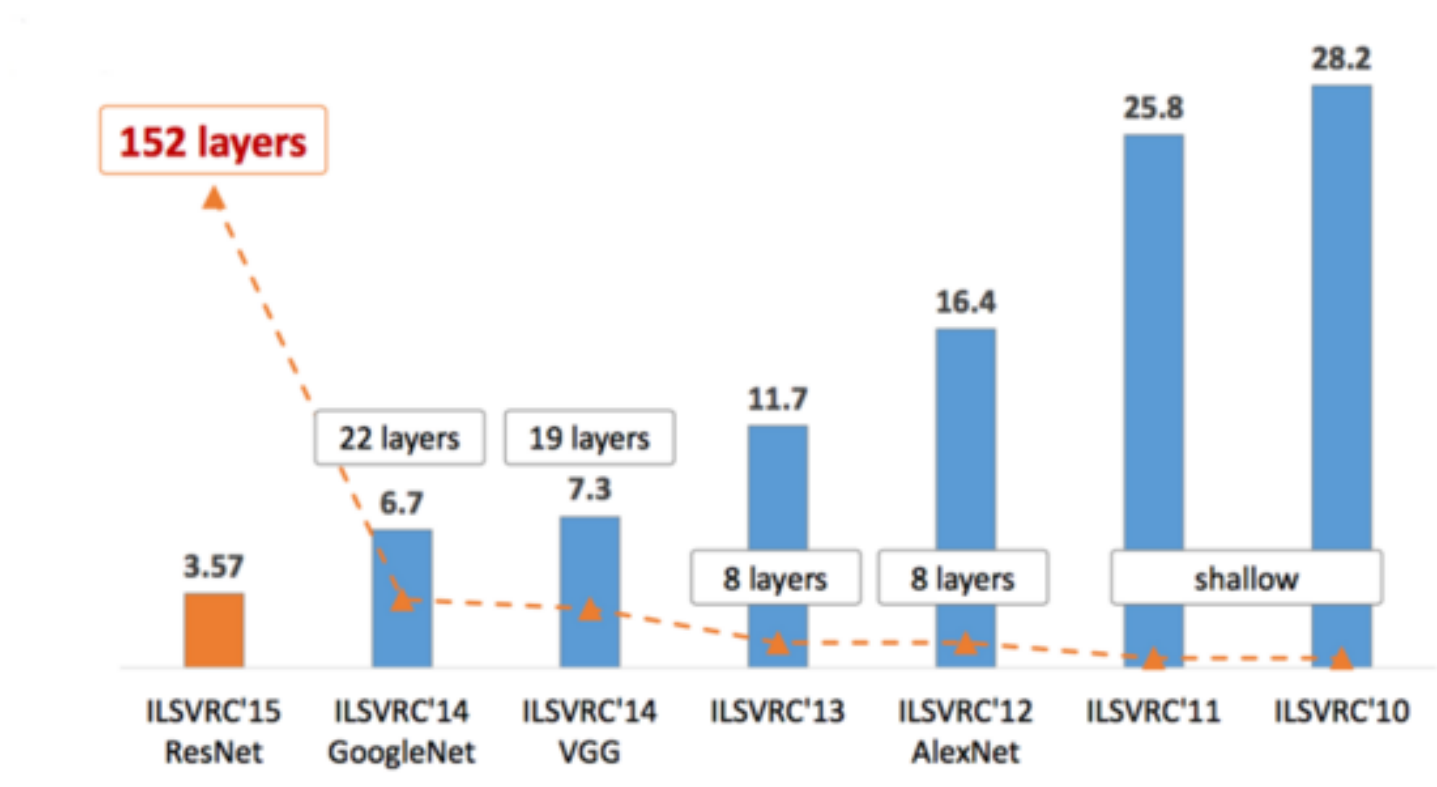
Deep Learning and the ILSVR challenge

Deep learning outperformed any other techniques in all major machine learning competitions (image classification, speech recognition and natural language processing)

The ImageNet Large Scale Visual Recognition Challenge (ILSVRC).

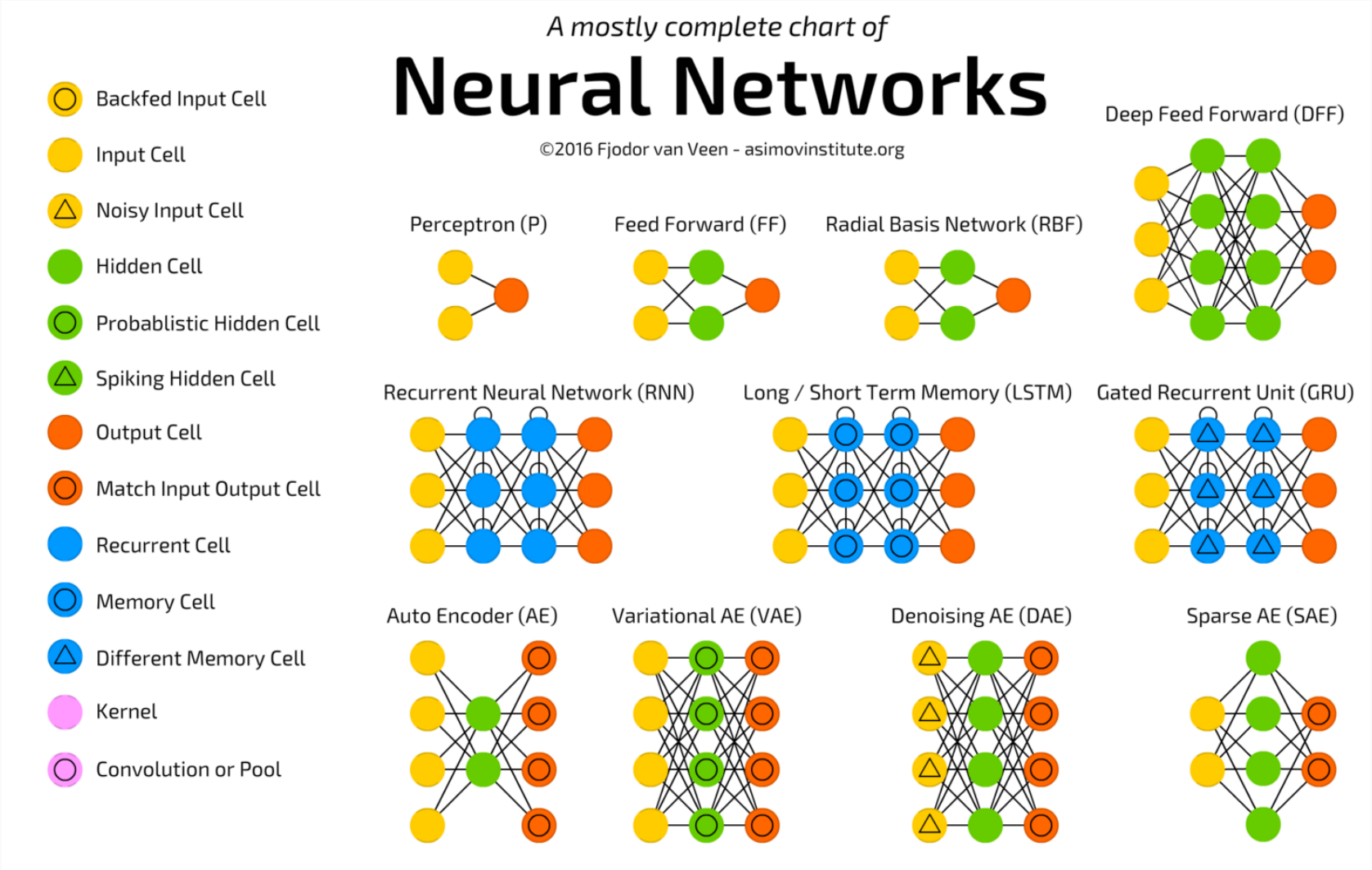
1. Training: 1.2 million images (227×227), labeled one out of 1000 categories
2. Test: 100.000 images (227×227)
3. Error measure: The teams have to predict 5 (out of 1000) classes and an image is considered to be correct if at least one of the predictions is the ground truth.

ILSVR challenge[2]

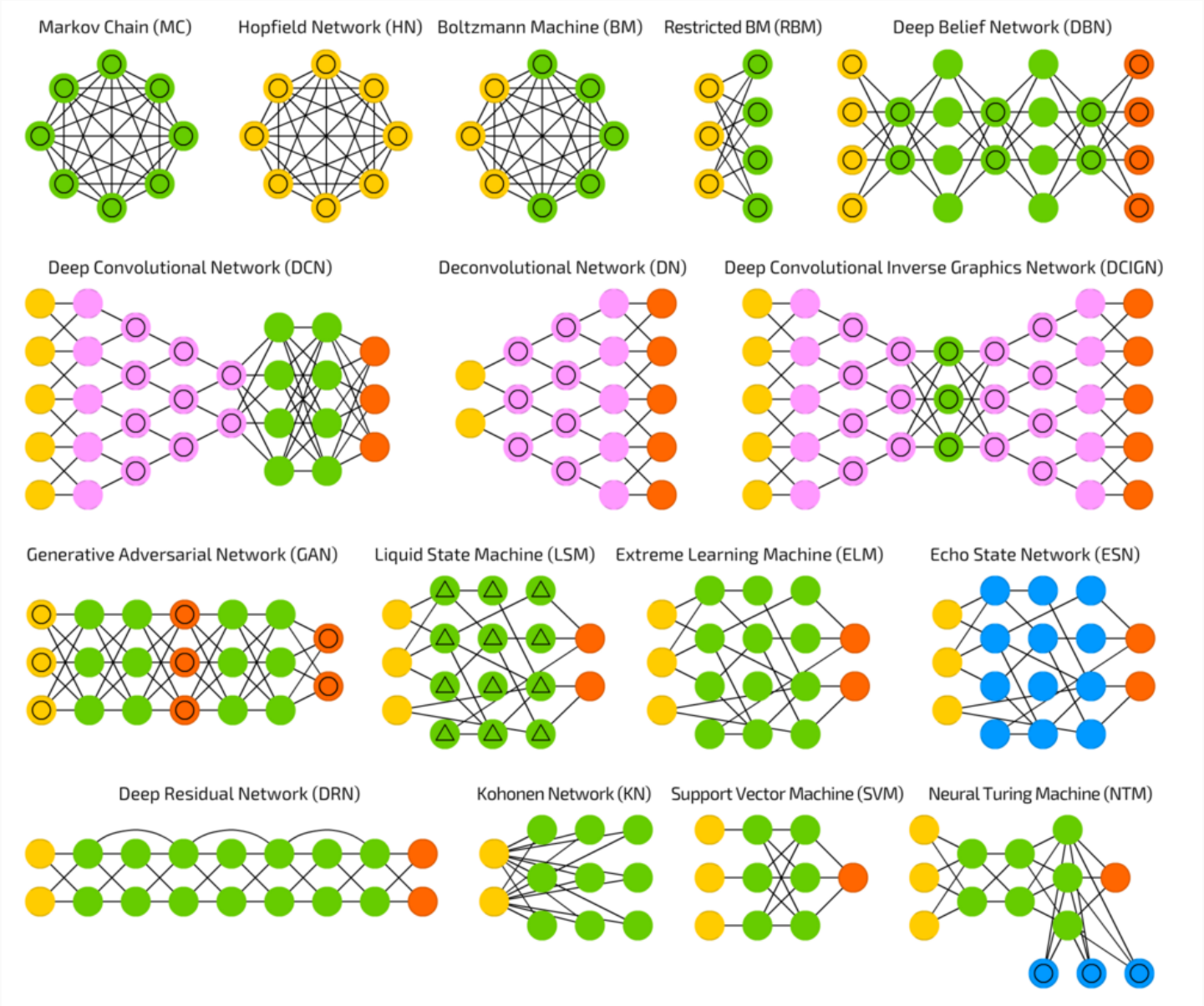


[2]From Stanford CS231n lecture notes

Architectures



Architectures



Outline of today's lecture

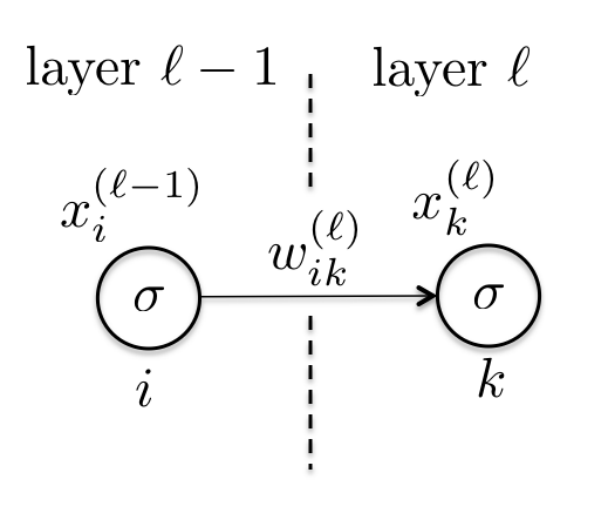
0. Introduction: Supervised vs. Unsupervised learning

1. Supervised Learning
   A. Regression and Classification
   B. **Neural Networks (Deep Learning)**
      (i) From perceptron to deep networks
      (ii) **Computing with and training neural nets**
      (iii) Why deep? Why does it work?

2. Reinforcement Learning

Computing with neural networks

- Layer 0: inputs $x = (x_1^{(0)}, \ldots, x_d^{(0)})$ and $x_0^{(0)} = 1$
- Layer $1, \ldots, L-1$: hidden layer ℓ, $d^{(\ell)} + 1$ nodes, state of node i, $x_i^{(\ell)}$ with $x_0^{(\ell)} = 1$
- Layer L: output $y = x_1^{(L)}$



Signal at k: $s_k^{(\ell)} = \sum_{i=0}^{d^{(\ell-1)}} w_{ik}^{(\ell)} x_i^{(\ell-1)}$

State at k: $x_k^{(\ell)} = \sigma(s_k^{(\ell)})$

Output: the state of $y = x_1^{(L)}$

Training neural networks

The output of the network is a function of $\mathbf{w} = (w_{ij}^{(\ell)})_{i,j,\ell}$: $y = f_{\mathbf{w}}(x)$
We wish to optimise over $\mathbf{w}$ to find the most accurate estimation of the target function

Training data: $(X_1, Y_1), \ldots, (X_n, Y_n) \in \mathbb{R}^d \times \{-1, 1\}$

Objective: find $\mathbf{w}$ minimising the empirical risk:

$$E(\mathbf{w}) := R(f_{\mathbf{w}}) = \frac{1}{2n} \sum_{l=1}^{n} |f_{\mathbf{w}}(X_l) - Y_l|^2$$

Stochastic Gradient Descent

$E(\mathbf{w}) = \frac{1}{2n} \sum_{l=1}^{n} E_l(\mathbf{w})$ where $E_l(\mathbf{w}) := |f_{\mathbf{w}}(X_l) - Y_l|^2$
In each iteration of the SGD algorithm, only one function E_l is considered ...

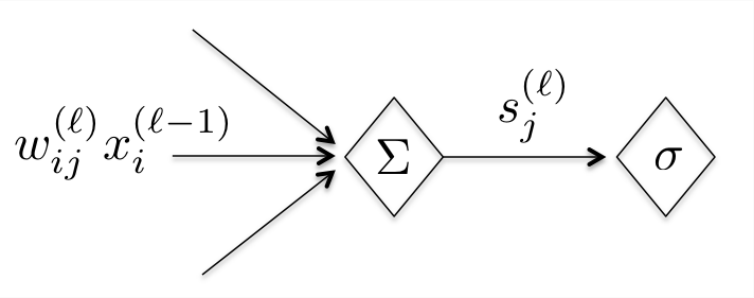
Parameter. learning rate $\alpha > 0$

1. **Initialization.** $\mathbf{w} := \mathbf{w}_0$
2. **Sample selection.** Select l uniformly at random in $\{1, \ldots, n\}$
3. **GD iteration.** $\mathbf{w} := \mathbf{w} - \alpha \nabla E_l(\mathbf{w})$, go to 2.

Is there an efficient way of computing $\nabla E_l(\mathbf{w})$?

Backpropagation

We fix l, and introduce $e(\mathbf{w}) = E_l(\mathbf{w})$.
Let us compute $\nabla e(\mathbf{w})$:



$$\frac{\partial e}{\partial w_{ij}^{(\ell)}} = \underbrace{\frac{\partial e}{\partial s_j^{(\ell)}}}_{:=\delta_j^{(\ell)}} \times \underbrace{\frac{\partial s_j^{(\ell)}}{\partial w_{ij}^{(\ell)}}}_{=x_i^{(\ell-1)}}$$

The sensitivity of the error w.r.t. the signal at node j can be computed recursively ...

Backward recursion

Output layer. $\delta_1^{(L)} := \frac{\partial e}{\partial s_1^{(L)}}$ and $e(\mathbf{w}) = (\sigma(s_1^{(L)}) - Y_l)^2$

$$\delta_1^{(L)} = 2(x_1^{(L)} - Y_l)\sigma'(s_1^{(L)})$$

From layer ℓ to layer $\ell - 1$.

$$\delta_i^{(\ell-1)} := \frac{\partial e}{\partial s_i^{(\ell-1)}} = \sum_{j=1}^{d^{(\ell)}} \underbrace{\frac{\partial e}{\partial s_j^{(\ell)}}}_{:=\delta_j^{(\ell)}} \times \underbrace{\frac{\partial s_j^{(\ell)}}{\partial x_i^{(\ell-1)}}}_{=w_{ij}^{(\ell)}} \times \underbrace{\frac{\partial x_i^{(\ell-1)}}{\partial s_i^{(\ell-1)}}}_{=\sigma'(s_i^{(\ell-1)})}$$

Summary.

$$\frac{\partial E_l}{\partial w_{ij}^{(\ell)}} = \delta_j^{(\ell)} x_i^{(\ell-1)}, \quad \delta_i^{(\ell-1)} = \sum_{j=1}^{d^{(\ell)}} \delta_j^{(\ell)} w_{ij}^{(\ell)} \sigma'(s_i^{(\ell-1)})$$

Backpropagation algorithm

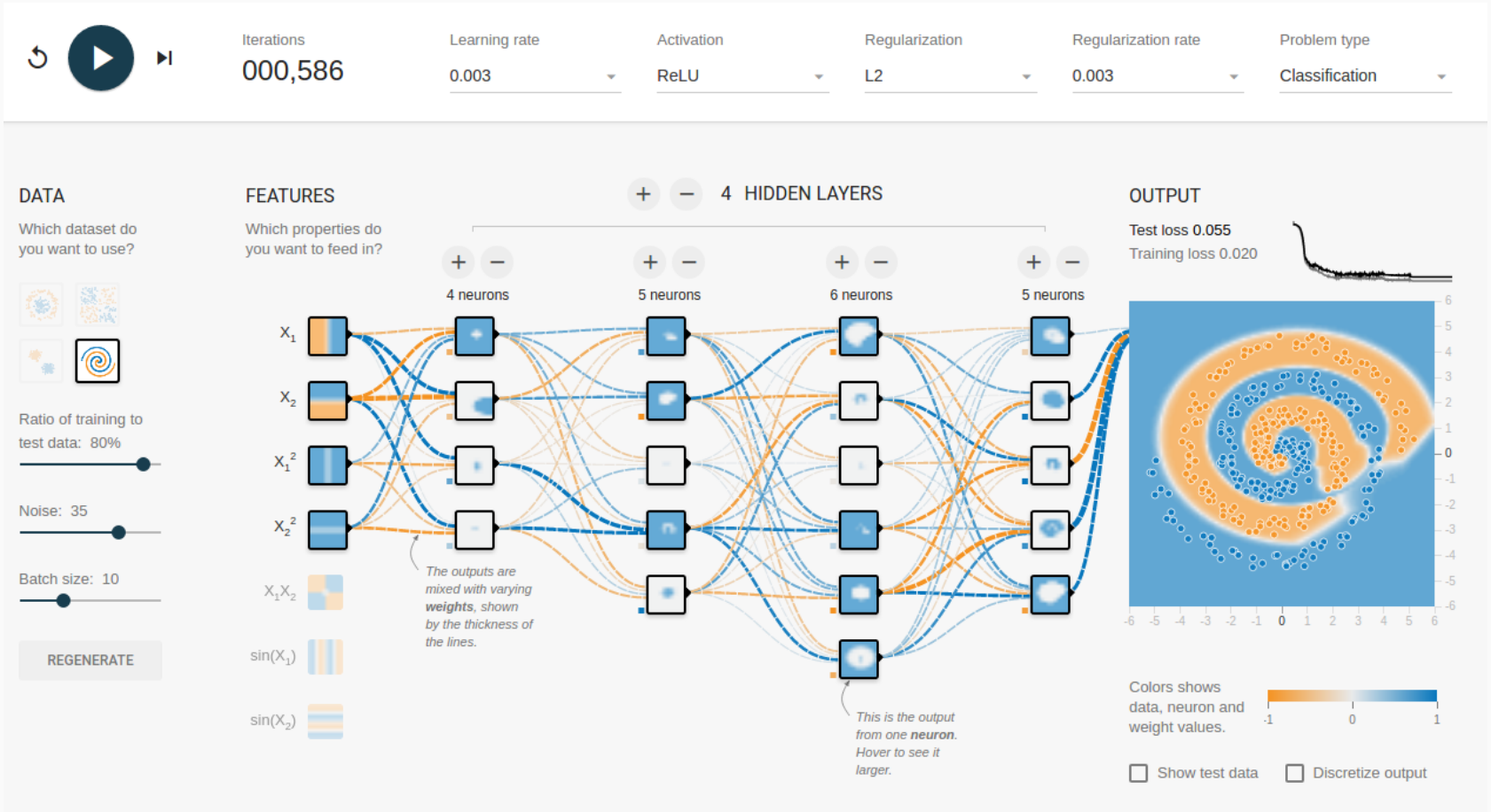
Parameter. Learning rate $\alpha > 0$

Input. $(X_1, Y_1), \ldots, (X_n, Y_n) \in \mathbb{R}^d \times \{-1, 1\}$

1. **Initialization.** $\mathbf{w} := \mathbf{w}_0$
2. **Sample selection.** Select l uniformly at random in $\{1, \ldots, n\}$
3. **Gradient of E_l.**
   - $x_i^{(0)} := X_{li}$ for all $i = 1, \ldots d$
   - Forward propagation: compute the state and signal at each node $(x_i^{(\ell)}, s_i^{(\ell)})$
   - Backward propagation: propagate back Y_l to compute $\delta_i^{(\ell)}$ at each node and the partial derivative $\frac{\partial E_l}{\partial w_{ij}^{(\ell)}}$
4. **GD iteration.** $\mathbf{w} := \mathbf{w} - \alpha \nabla E_l(\mathbf{w})$, go to 2.

Example: tensorflow

http://playground.tensorflow.org/



Outline of today's lecture

0. Introduction: Supervised vs. Unsupervised learning

1. Supervised Learning
   A. Regression and Classification
   B. **Neural Networks (Deep Learning)**
      (i) From perceptron to deep networks
      (ii) Computing with and training neural nets
      (iii) **Why deep? Why does it work?**

2. Reinforcement Learning

Deep learning and model selection

A given network represents a **model**. $\mathcal{F}$ encodes the architecture and topolgy of the network.

$$\mathcal{F} = \{f_{\mathbf{w}} : \mathbf{w} \in \mathcal{W} \subset \mathbb{R}^b\}$$

Deep learning = find the weights or equivalently the function in $\mathcal{F}$ minimising the empirical risk. Why should it be good?

Expressibility. How large the class of functions that $\mathcal{F}$ can represent is.

Efficiency. How many parameters are required to approximate a function.

Learnability. How rapidly the model can be trained (sample complexity).

Expressibility and Efficiency of Neural Nets[3]

- Neural nets can approximate any continuous function (Cybenko's theorem).
- Neural nets can compute polynomials efficiently.

Lemma *Let σ be a sigmoid function with non-zero second derivative at 0. Using 4 σ-neurones, we can approximation the multiplication.*

Corollary *The class of polynomials involving n multiplications can be represented (with arbitrary precision) with a neural network of size slightly larger than $4n$.*

[3]Why does deep and cheap learning work so well? Lin-Tegmark, 2016

Why deep?

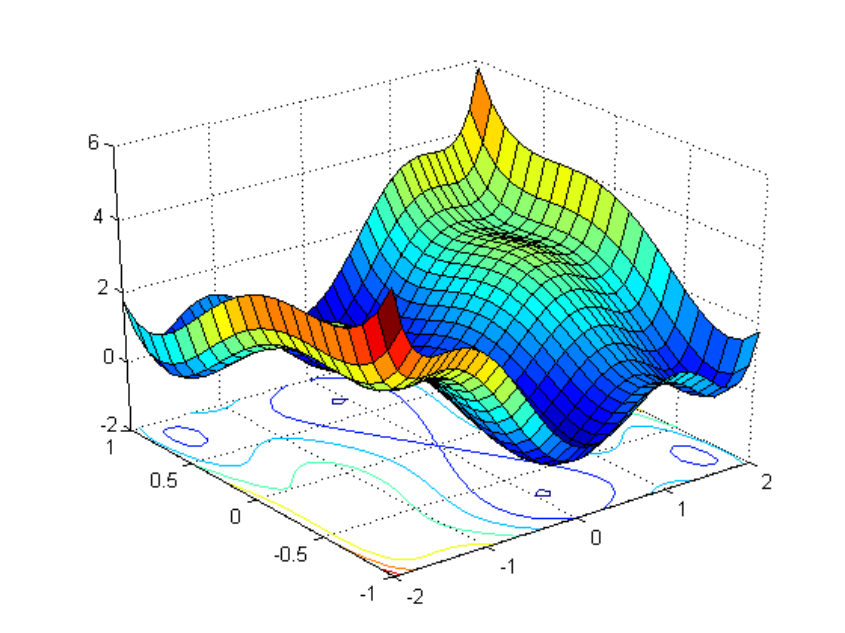
No flattening theorem:

Theorem *To compute the product of n variables in a single-layer network, we need 2^n neurones.*

Extensions: with two layers we need $2 \times 2^{n/2}$ neurones.

Learnability

Critical question: The SGD algorithm will converge to a global minimum of the risk, if we can guarantee that local minima have the same risk as a global minimum. What does the loss surface look like?



The output of a neural network is a non-linear function of the weights, and possibly has many bad local minima where the SGD gets trapped.

Notations

- Data: $X_i \in \mathbb{R}^{d_x}$, $Y_i \in \mathbb{R}^{d_y}$, m data points
  X: $d_x \times m$ matrix whose columns are the X_is
  Y: $d_y \times m$ matrix whose columns are the Y_i's
- H hidden layers
- Layer k with d_k neurons, input weight matrix $W_k \in \mathbb{R}^{d_k \times d_{k1}}$
- $p = \min\{d_1, \ldots, d_H\}$
- Output:

$$\hat{Y}(W, X) = q\sigma_{H+1}(W_{H+1}\sigma(W_H\sigma(W_{H1} \ldots \sigma(W_2\sigma(W_1 X)\ldots)$$

  Linear activation function: $\hat{Y}(W, X) = W_{H+1} \ldots W_1 X$.

1-hidden layer networks

- Linear regression: fitting a linear model to the data. $X_i \in \mathbb{R}^{d_x}$, $Y_i \in \mathbb{R}_{d_y}$
  Find the matrix $L^\star \in \mathbb{R}^{d_y \times d_x}$ minimizing

$$\mathcal{L}(L) = \sum_{i=1}^{m} \|Y_i - LX_i\|^2$$

  When $XX^\top$ is invertible, $L^\star = YX^\top(XX^\top)^{-1}$ Convexity of $\mathcal{L}$
- Now in a 1-hidden layer network, we are looking for L that can be factorized as W_2W_1 where $W_1 \in \mathbb{R}^{p \times d_x}$ and $W_2 \in \mathbb{R}^{d_y \times p}$.
  In particular the rank of L is at most p. Non uniqueness:
  $W_1' = CW_1$ and $W_2' = W_2C^{-1}$ work as well.

1-hidden layer networks

Introduce the $d_y \times d_y$ matrix $\Sigma = YX^\top (XX^\top)^{-1} XY^\top$ as the covariance matrix of the best unconstrained linear approximation of Y

Theorem (Baldi-Hornik, 1989) *Assume that Σ is full rank with distinct eigenvalues. Up to C, the global minimizer is unique, and is the projection on the subspace spanned by the p top eigenvectors of Σ of the ordinary least square regression matrix.*

H-hidden layer networks

Theorem (Kawaguchi, 2016) *Assume that $XX^\top$ and $YY^\top$ are full rank, and $d_x \geq d_y$. Assume that Σ is full rank with distinct eigenvalues. The loss function $\mathcal{L}(W_1, \ldots, W_{H+1})$ satisfies:*

(i) *it is non-convex and non-concave.*

(ii) *Every local minimum is a global minimum.*

(iii) *Every critical point that is not a minimum is a saddle point.*

(iv) *If rank$(W_H, \ldots, W_2) = p$, then the Hessian at any saddle point has at least one strictly negative eigenvalue (we can escape local saddle points).*

Neural networks

Open questions:

- What is the role of the regularization term?
- What about sigmoid functions, and ReU?
- Interpretability of the role of the various layers?
- ...

A few references

- P. Baldi, K. Hornik. Neural Networks and PCA: Learning from Examples without Local Minima. Neural Networks, 1989.
- I. Goodfellow, Y. Bengio, A. Courville. Deep Learning, http://www.deeplearningbook.org
- A. Choromanska et al.. The Loss Surface of Multilayer Networks. ICML 2015.
- K. Kawaguchi. Deep Learning without Poor Local Minima. NIPS 2016