Machine Learning for biology

V. Monbet

UFR de Mathématiques
Université de Rennes 1

Outline

Outline

Outline

Outline

Outline

Outline

Outline

Outline

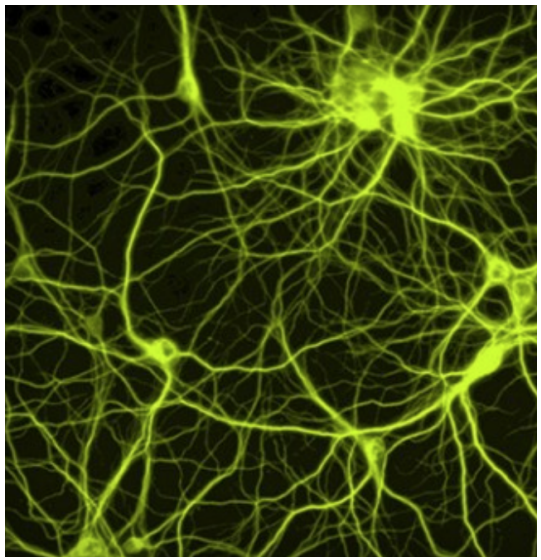
Outline

Outline

Outline

Introduction

- Artificial neural networks (ANN) are non linear regression models with a particular structure.
  They can be interpreted as a functional imitation of a simplified model of biological neurons.
  They are used to predict numerical or categorical variables.
- As for the linear model, the most important steps to calibrate an ANN are
  - choosing the structure,
  - estimation of the parameters.
- First studies of ANNs started in the 1940s. In the 1980s, computers were able to fit reasonable ANNs. Recently, there has been a renewed interest in ANNs because of deep learning.



Outline

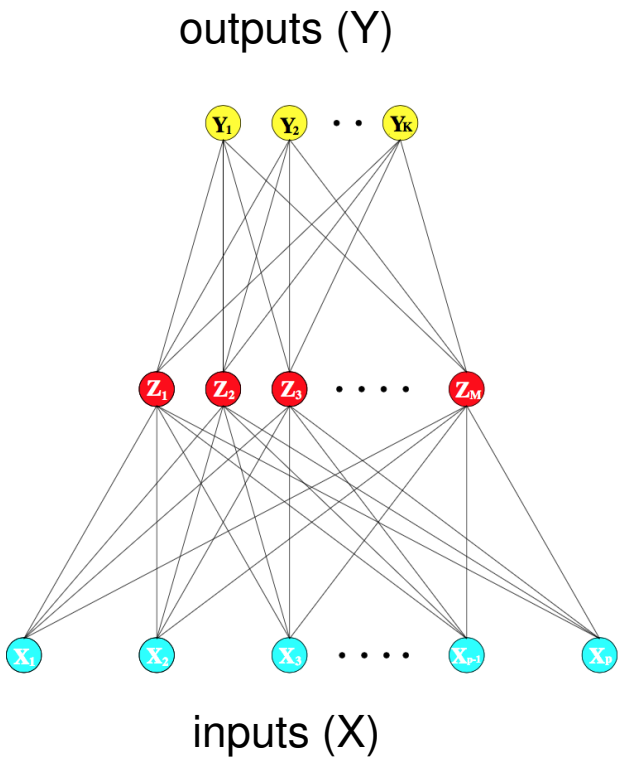
Multi layer perceptron

- A multi layer perceptron is described by the following structure

$$
\begin{aligned}
Z_m &= \sigma(\alpha_{0m} + \boldsymbol{\alpha}_m{}^T \mathbf{X}), \quad && m = 1, \cdots, M \\
T_k &= \beta_{0k} + \boldsymbol{\beta}^T Z, && k = 1, \cdots, K \\
Y_k &= g_k(T), && k = 1, \cdots, K
\end{aligned}
$$

  with M the number of neurons and K the number of outputs.

- Typically if $Y \in \mathbb{R}$, $K = 1$ and if $Y \in \{1, \cdots, M\}$, $K = M - 1$.
- In the hidden layer, each Z_m is obtained as a activation σ of a linear combination of the inputs. An ANN can have several hidden layers.
- The structure of the network is defined by the number of hidden layers and the number of neurons in each hidden layer.
- The unknown parameters are the weights $\boldsymbol{\alpha}$ and the $\boldsymbol{\beta}$.

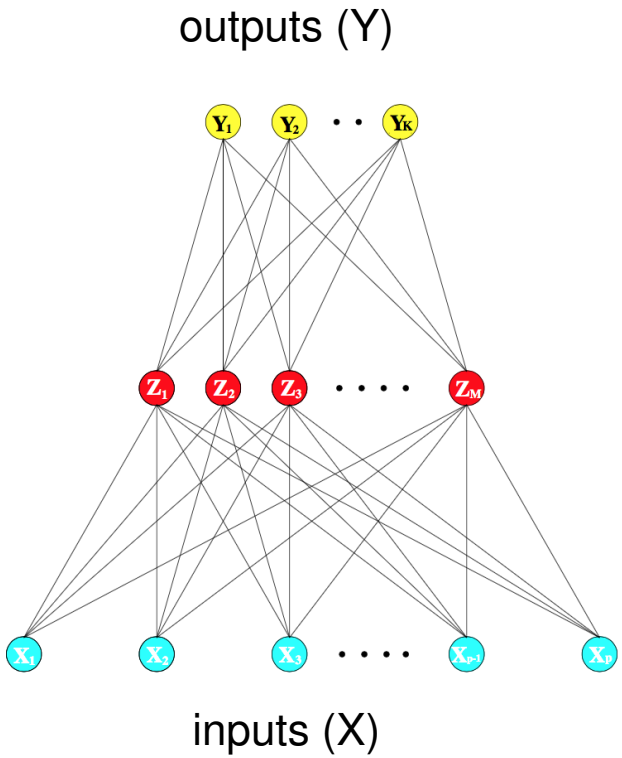


MLP inputs and outputs

Examples

- An input X_j denotes a variable.
  Examples: $T15$ for Ozone data or the expression of a given gene for the leukemia data.

- An output is the output of the problem.
  Regression task: $Y \in \mathbb{R}$
  Classification task with K classes: $Y = \{Y_1, Y_2, \cdots, Y_{K-1}\}$

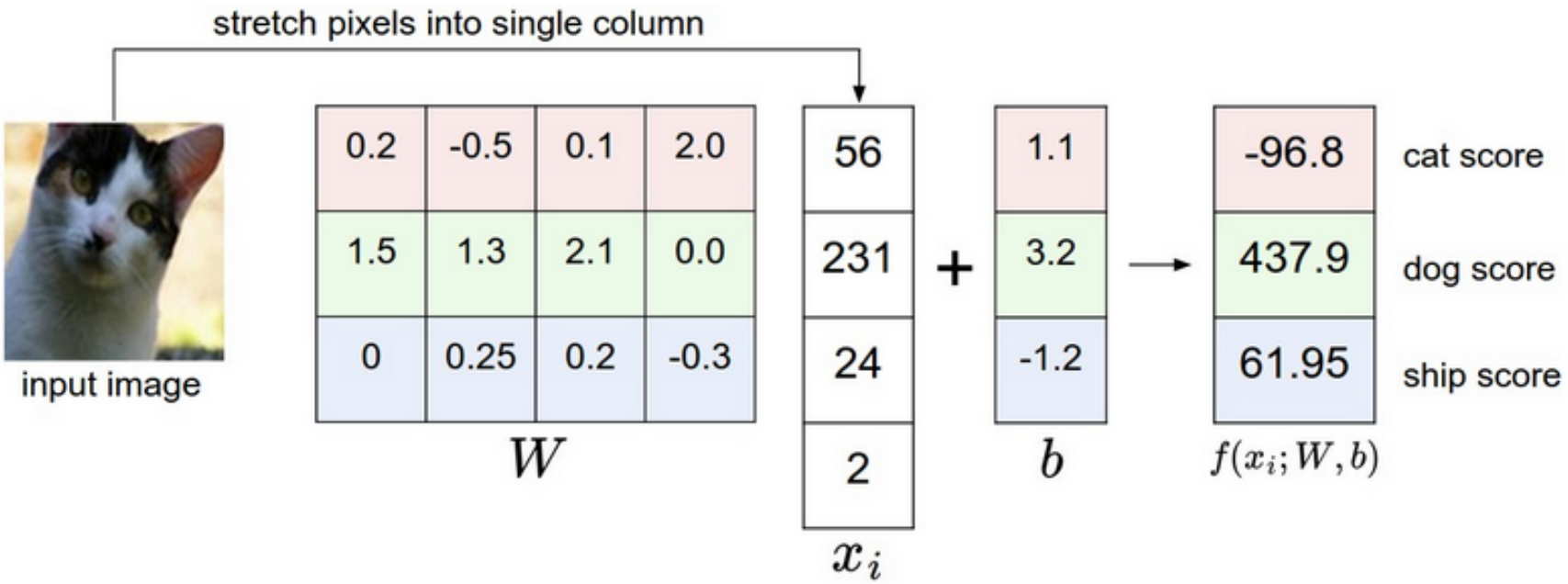


linear classifier

Each neurone is based on a linear classifier

$$\alpha_{0m} + \boldsymbol{\alpha}_m{}^T \mathbf{X}$$

For example, if the input $\mathbf{X}$ is an image (and $\boldsymbol{\alpha_m} = W$, $\alpha_{0m} = b$)



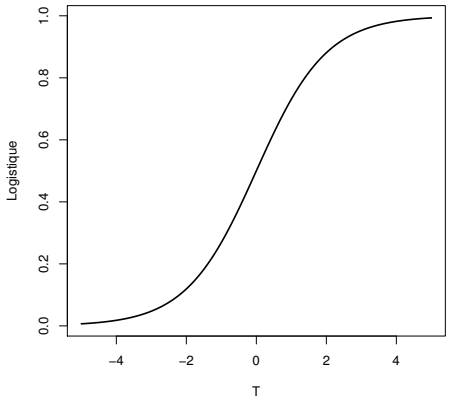
Each line of W corresponds to a set of weights fitted for a given class (cat, dog, ...).

MLP model, activation functions

- The activation function σ is generally the sigmoid function (top panel)

$$\sigma(x; s) = \frac{1}{1 + e^{-sx}},$$

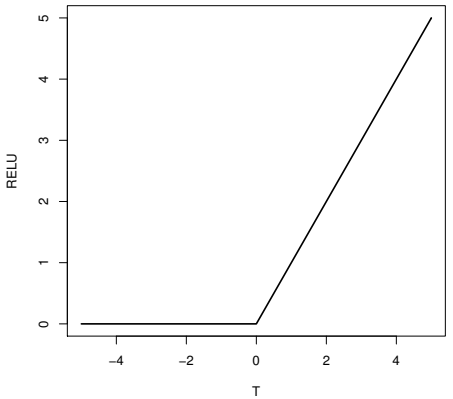
but other functions can be chosen.
Drawbacks: saturation for low/high values, computation time



- In Deep Learning, the Rectified Linear Unit (RELU) function is used (bottom panel),

$$\sigma(x) = max(0, x)$$

This function better imitates biological phenomena and its derivative is not 0 for large x.
Advantages: simple computation, linear bahaviour (a ANN is easy to fit is it is close to linear), representational sparsity (negative values are put exactly to 0 output)



MLP model, activation functions

$$
\begin{aligned}
Z_m &= \sigma(\alpha_{0m} + \boldsymbol{\alpha}_m{}^T \mathbf{X}), \quad && m = 1, \cdots, M \\
T_k &= \beta_{0k} + \boldsymbol{\beta}_k{}^T Z, && k = 1, \cdots, K \\
Y_k &= g_k(T), && k = 1, \cdots, K
\end{aligned}
$$

The output function depends on the problem

- Regression task: $g_k(T) = T_k$ is linear.
- Classification task (2 classes): $g_k(T) = \frac{1}{1+e^{-T}}$ is logistic.
- Classification task (K classes): $g_k(T) = \frac{e^{T_k}}{\sum_{j=1}^{K} e^{T_j}}$ is softmax.

  The softmax function is the generalization of the logistic function to more than 2 classes.

Summary

Now the model has all the standard components of what people usually mean when they say "neural network":

- A set of **nodes**, analogous to neurons, organized in **layers**.
- A set of **weights** representing the connections between each neural network layer and the layer beneath it. The layer beneath may be another neural network layer, or some other kind of layer.
- A set of **biases**, one for each node.
- An **activation function** that transforms the output of each node in a layer. Different layers may have different activation functions.

Outline

How to learn the weights of an ANN?

- The neural network model has unknown parameters, called weights (and denoted θ hereafter), and we seek values for them that make the model fit the training data well.
- For **regression**, we use sum-of-squared errors as the measure of fit

$$R(\theta) = \sum_{i=1}^{n} (y_i - f_\theta(\mathbf{x}_i))^2$$

- ANNs usually involve a lot of parameters so the risk of overfitting is high and a regularization of $R(\theta)$ is needed.
- The penalized objective function

$$R_J(\theta) = R(\theta) + \lambda \sum_{j=1}^{q} {\theta_j}^2$$

is usually optimized via a gradient descent, where q is the dimension of the vector of parameters.

- **Gradient descent**
  Repeat until convergence
  For $k \in \{1, \cdots, p\}$

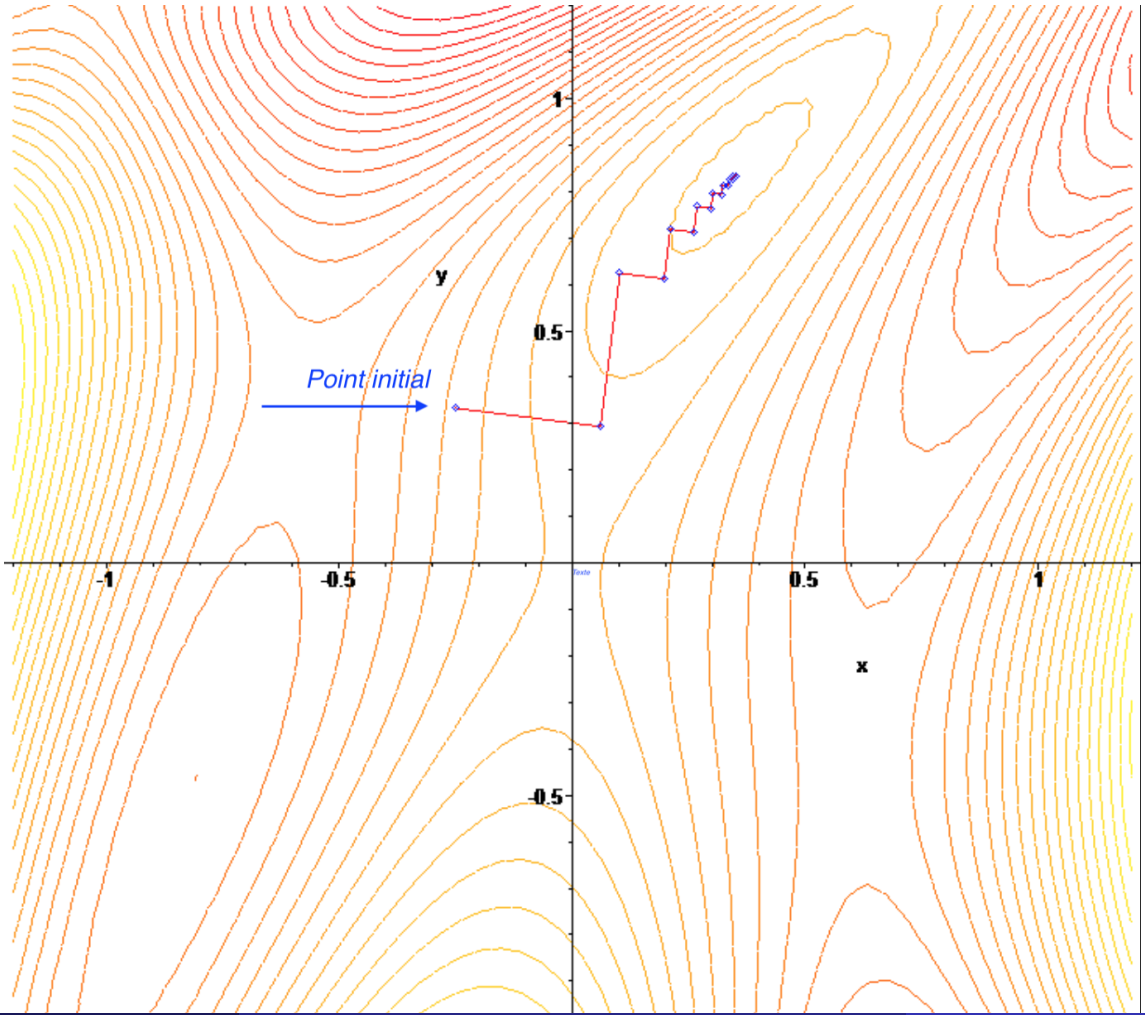
$$\theta_k^{r+1} = \theta_k^r - \gamma_r \frac{\partial R(\theta^r)}{\partial \theta_k} = \theta_k^r - \gamma_r \sum_{i=1}^{n} \frac{\partial (y_i - f_\theta(\mathbf{x}_i))^2}{\partial \theta_k}$$

with a learning rate γ_r.

Gradient Descent

Repeat until convergence
For $k \in \{1, \cdots, q\}$

$$\theta_k^{r+1} = \theta_k^r - \gamma_r \sum_{i=1}^{n} \frac{\partial (y_i - f_{\theta^r}(\mathbf{x}_i))^2}{\partial \theta_k}$$



Learning, back-propagation algorithm

- Parameters to be learned: weights
  - α_{0m}, α_m ; $j = 1, \cdots, M$, $(M+1)p$ parameters of the hidden layers
  - β_{0k}, β_k ; $k = 1, \cdots, K$, $M(K+1)$ parameters of the output
- Loss function
  Regression : $\sum_{i=1}^n (y_i - f_\theta(x_i))^2 = \sum_{i=1}^n R_i(\theta)$ (mean square error)
  Classification : $-\sum_{i=1}^n \sum_{\ell=1}^K y_{i\ell} \log f_\ell(x_i;\theta)$ (entropy)
- Backpropagation algorithm.
  Based on a gradient descent algorithm.
  At each step r,

$$\beta_m^{(r)} = \beta_m^{(r-1)} - \gamma_r \sum_{i=1}^n \frac{\partial R_i}{\partial \beta_m^{(r)}}$$

$$\alpha_{m\ell}^{(r)} = \alpha_{m\ell}^{(r-1)} - \gamma_r \sum_{i=1}^n \frac{\partial R_i}{\partial \alpha_{m\ell}^{(r)}}$$

where γ_r is the learning rate. It tends to 0 when r increases.

Gradient computation

- Let us remark that, for a regression problem,

$$\frac{\partial R_i}{\partial \beta_m} = -2(y_i - f_\theta(x_i))z_{mi} = \delta_i z_{mi}$$

$$\frac{\partial R_i}{\partial \alpha_{m\ell}} = -\sigma'(\alpha_m^T x_i)2(y_i - f_\theta(x_i))\beta_m x_{i\ell} = s_{mi} x_{i\ell}$$

  where δ_i and s_{mi} denote the "errors" associated with observation i at the output of each hidden neuron.
- The backpropogation equations apply to the errors δ_i (output layer) and s_{mi} (hidden layers)

$$s_{mi} = \sigma'(\alpha_m^T x_i)\beta_m \delta_i$$

- The particular expression of the gradients leads to a fast algorithm which is easy to parallelize.

$$\sigma(x) = \frac{1}{1+e^{-x}}, \ \sigma'(x) = \frac{d\sigma(x)}{dx} = (1-\sigma(x))\sigma(x)$$

Backpropagation algorithm

Until (convergence) repeat,

[forward] compute $\hat{f}_\theta(x_i)$ with the current weights
[backward] compute the output z_{mi} of each layer and the errors δ_i.
[update] update the gradients by computing the s_{mi} and update the weights.

Backpropagation step by step

Backpropagation example

Backpropagation in practice

- The backpropagation algorithm can be run in *batch* : all the entries are presented at the same time.
  But it is not optimal neither from a memory point of view nor for the gradient descent.

- Stochastic gradient descent: learning *online*
  At each step, only one observation (or *a minibatch* = few training examples) is considered and the associated weights are updated.
  It allows to deal with big datasets.

  Standard gradient descent

$$\theta_{r+1} = \theta_r - \gamma_r \nabla_\theta E\left(J(\theta)\right)$$

  Stochastic gradient descent (SGD)

$$\theta_{r+1} = \theta_r - \gamma_r \nabla_\theta J(\theta; \mathbf{x}_i, y_i)$$

- Using minibatch reduces the variance in the parameter update and allows the computation to take advantage of highly optimized matrix operations (vectorized computation).

$$\theta_{r+1} = \theta_r - \gamma_r \nabla_\theta J(\theta; \mathbf{x}_{i:i+k}, y_{i:i+k})$$

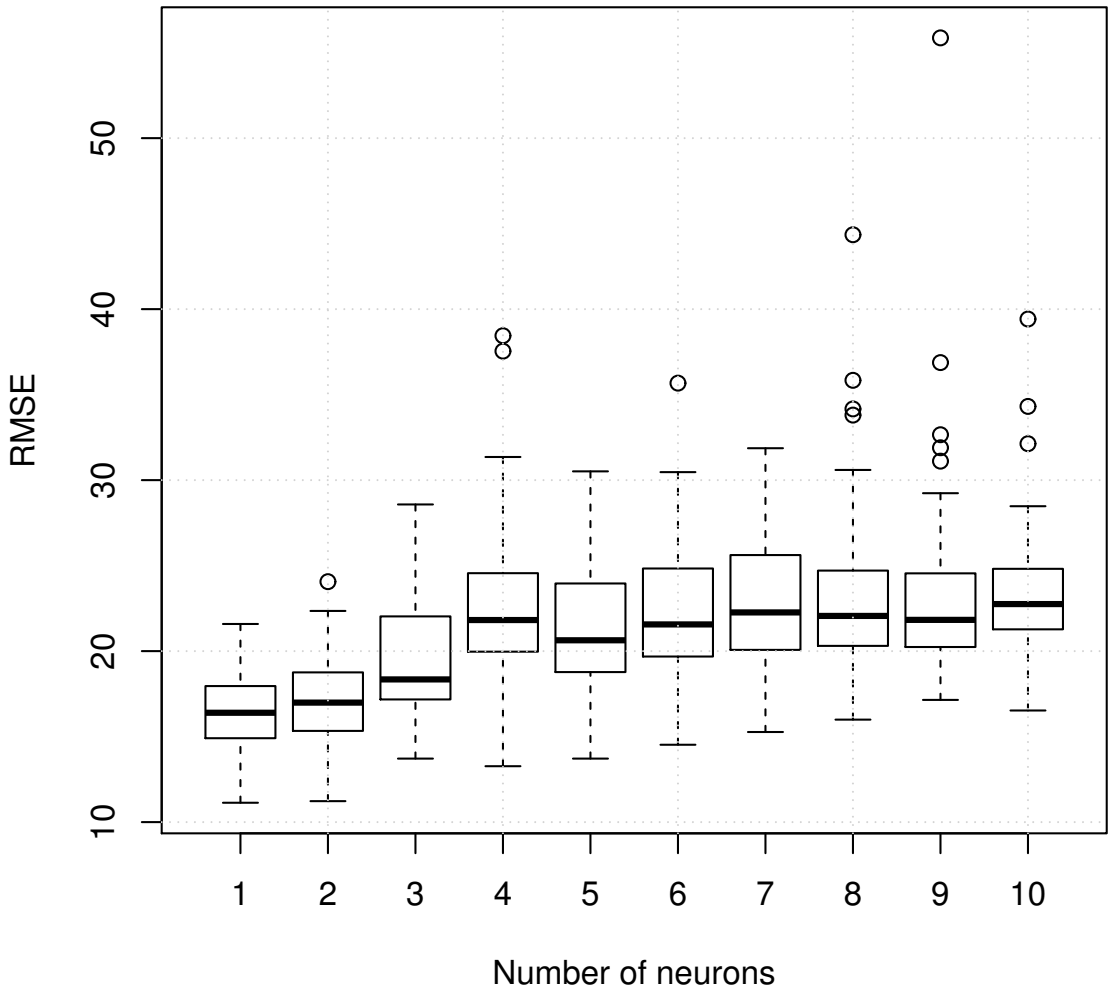
Some issues in learning ANN

- The objective function usually exhibits a lot of local minima so that the initialization of the optimization task is very important. $\rightarrow$ It is typical to take random uniform weights over the range $[0.7, +0.7]$
  Furthermore, if the weights are near 0, the sigmoid function is very close to a linear function and the ANN collapse to a linear approximation. But starting with large weights generally leads to poor solutions. $\rightarrow$ work with ReLU activation function.
  One of the main features of deep learning is to find a tricky initialization of the network weights.
- Scale the inputs: it is best to standardize all inputs to have mean zero and standard deviation one.
- For highly non linear problems, it is often more efficient to build a network with several hidden layers and a low number of neurons instead of a network with only one hidden layer and a lot of neurons.
- Backpropagation: the learning of the weights of the layer which are close to the input is slow compare to the ones which are close to the output. Indeed, in the back propagation gradient are multiplied to each other ($\max \sigma' = 0.25$), it may lead to almost 0 gradients and slow update of the weights. $\rightarrow$ work with ReLU activation function.

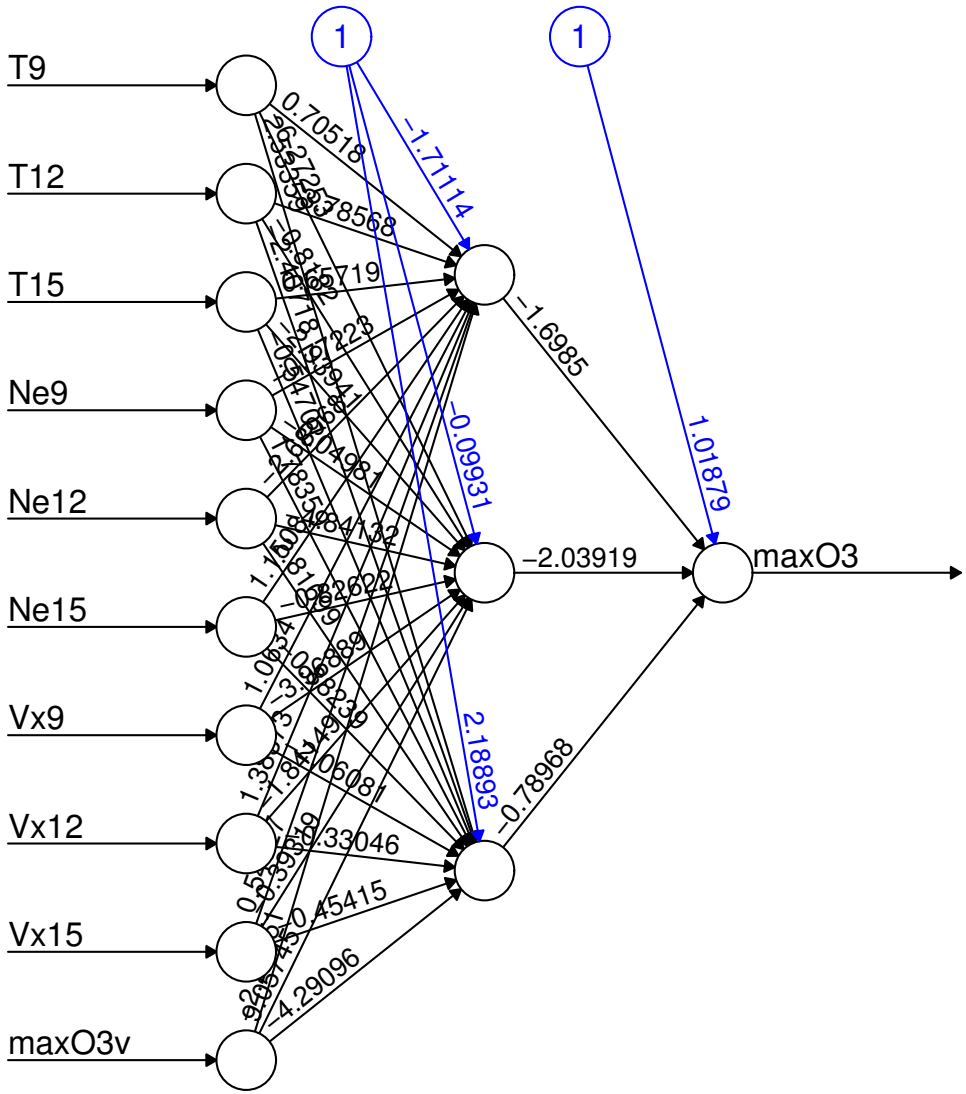
Outline

Example: Ozone (regression)

- Selection of the number of neurones



Example: Ozone (regression)

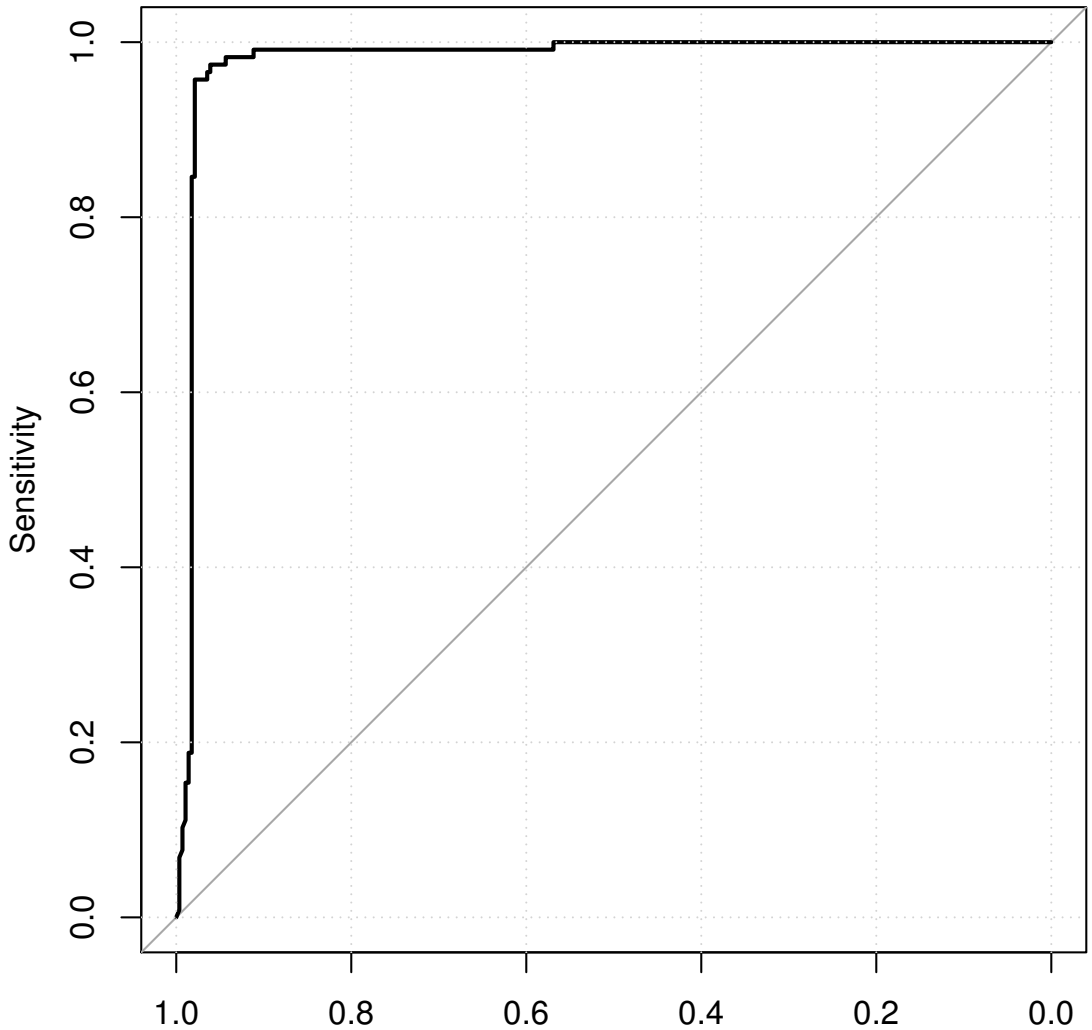


Error: 0.138995 Steps: 1

Example: Leukemia gene expression

- Selection of the number of neurones

Hidden	(1,0)	(1,1)	(2,0)	(2,1)	(2,2)	(3,0)	(3,1)	(3,2)	(3,3)
AUC	0.963	0.927	0.963	0.966	0.952	0.978	0.943	0.978	0.979



ANN in practice

- Example with keras + google colab

`https://drive.google.com/open?id=1BItUAmMXXZYPCmoct-yu8SRFheVGjRZ_`

Outline

Outline

Outline