

# Introduction to Machine Learning



**Tommaso Dorigo, INFN-Padova**  
**Kolympari, July 19, 2023**

# Suggested reading

These lectures are based on a number of different sources, as well as on some personal alchemy

I tried to provide references but sometimes I failed [apologies...]

A formal treatment of most of the covered material, and more in-depth than what I can go here, is offered in a couple of excellent textbooks:

Springer Series in Statistics

Trevor Hastie  
Robert Tibshirani  
Jerome Friedman

## The Elements of Statistical Learning

Data Mining, Inference, and Prediction

Second Edition

Springer

Hastie, Tibshirani, Friedman:  
The elements of statistical learning

→ AVAILABLE ONLINE FOR FREE!

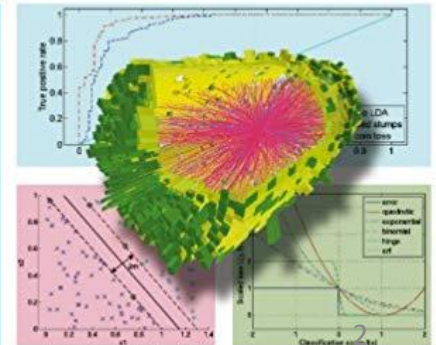
Narsky, Porter: Statistical Analysis  
techniques in Particle Physics,  
Wiley

Ilya Narsky, Frank C. Porter

WILEY-VCH

## Statistical Analysis Techniques in Particle Physics

Fits, Density Estimation and Supervised Learning

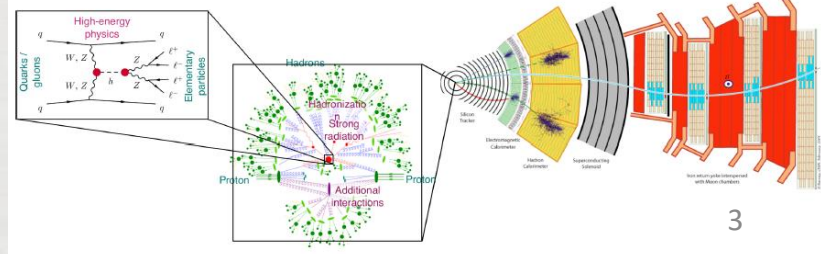


# Contents - 1

## Lecture 1: An introduction to Machine Learning

- Introduction
  - Map of ML problems
  - Supervised and unsupervised learning
- Density estimation
  - kNN
  - Divergence measures
- Resampling techniques
- The data
- The model

### Particle physics



# Contents - 2

## Lecture 2: Classification and decision trees

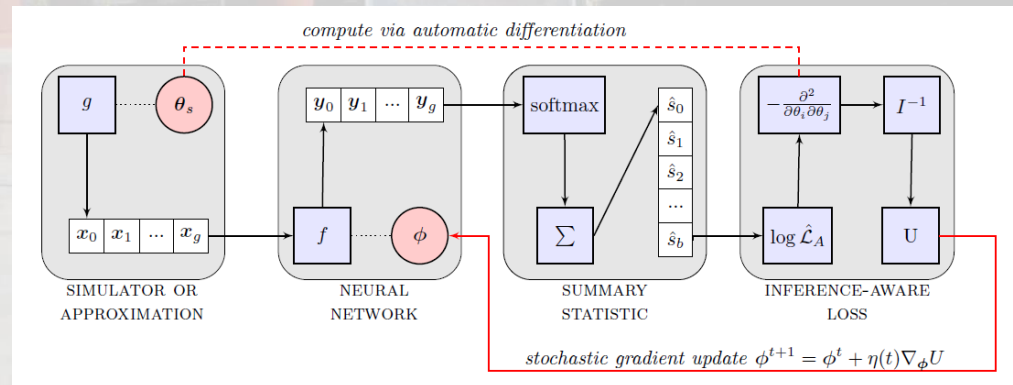
- Classification
- Loss minimization
- Decision trees
  - random forests
  - boosting techniques



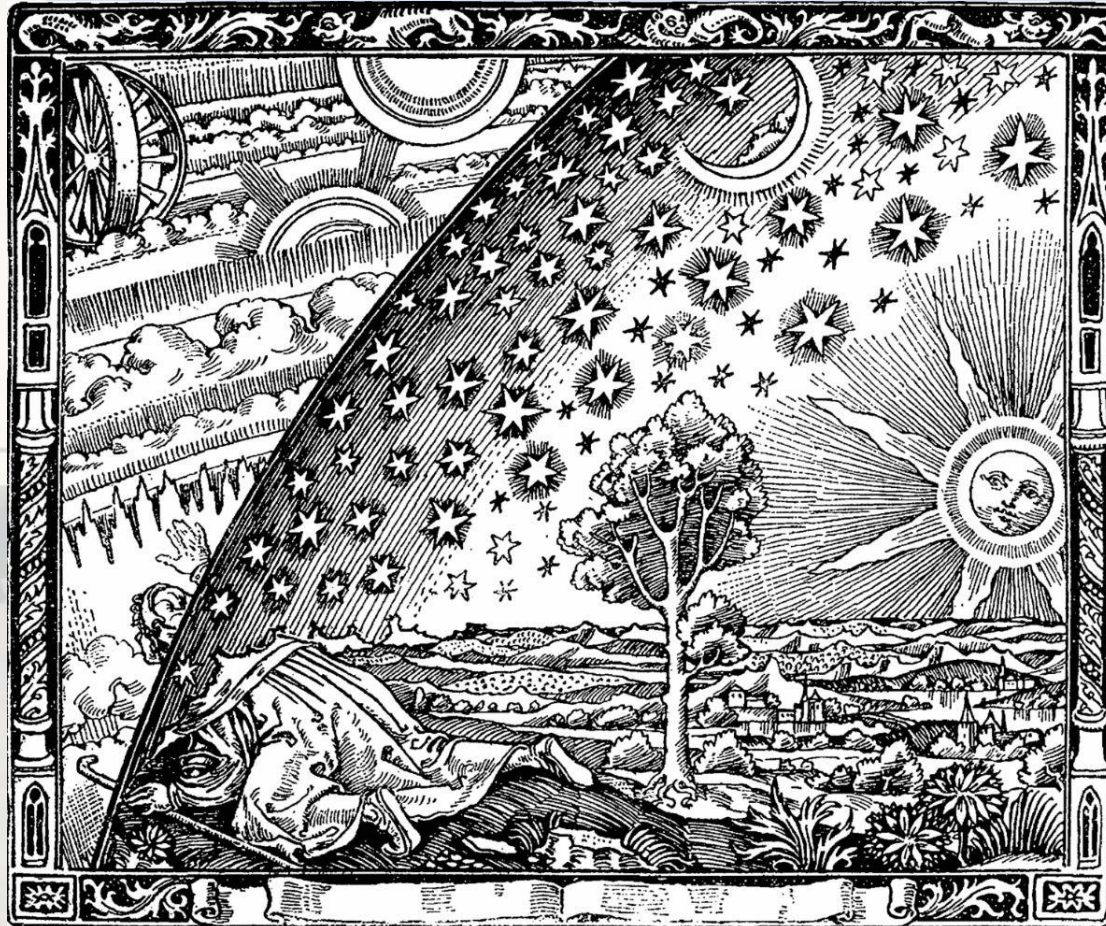
# Contents - 3

## Lecture 3: Neural networks

- Neural networks
- Playing with NNs
- Practical tips
- Conclusions



# Lecture 1 - INTRODUCTION



# Classes of Statistical Learning algorithms

## Supervised:

if we know the probability density of S and B, or if at least we can estimate it  
→ E.g. we use "labeled" training events ("Signal" or "Background")  
to estimate  $p(x|S)$ ,  $p(x|B)$  or their ratio

## Semi-supervised:

it has been shown that even knowing the labels for part of the data is sufficient to construct a classifier

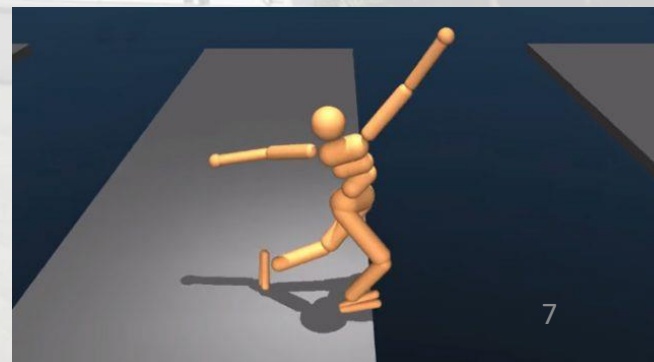
## Un-supervised:

if we lack an a-priori notion of the structure of the data, and we let an algorithm discover it without e.g. labeling classes → **cluster analysis, anomaly detection, unconditional density estimation.**

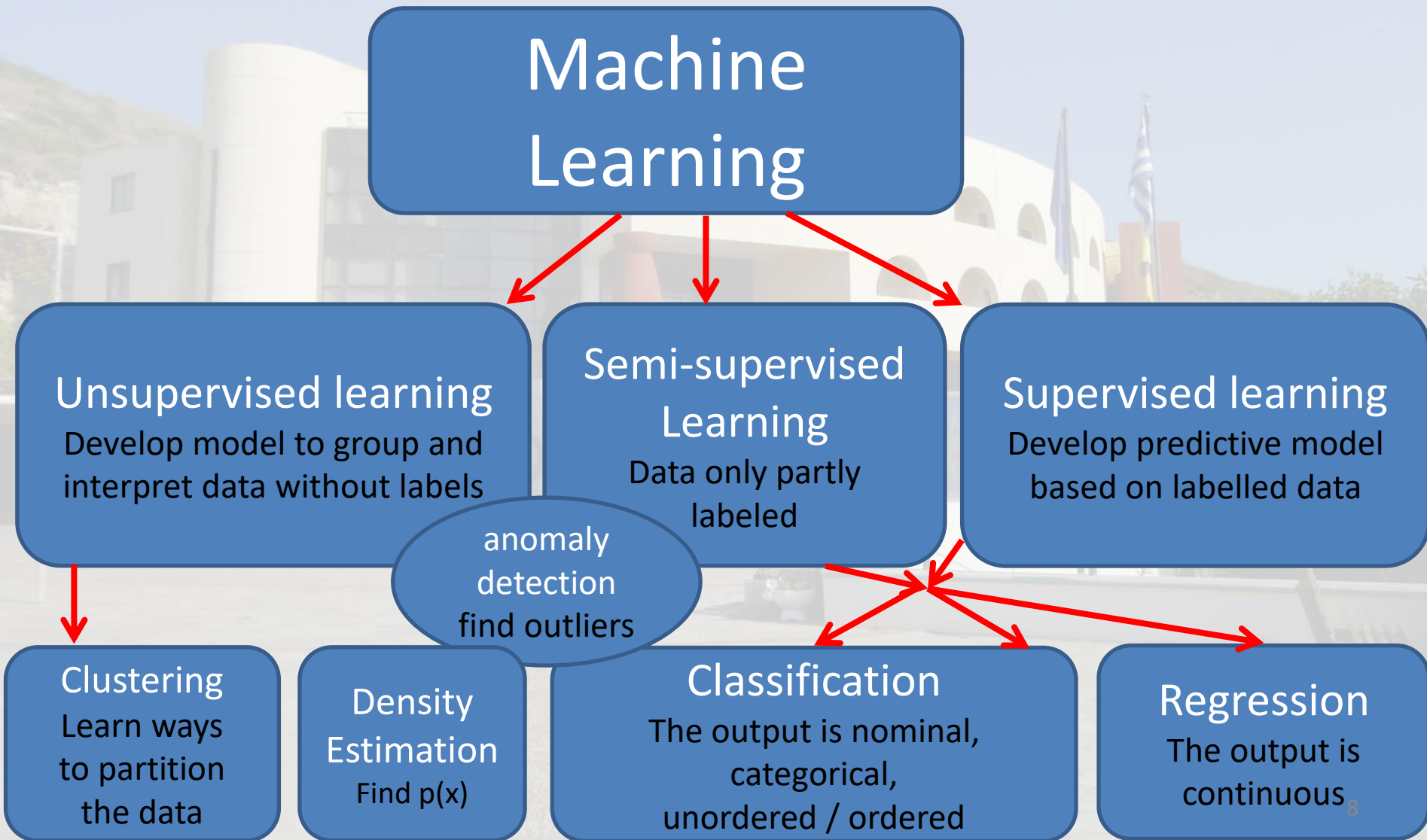
## We may also single out:

### Reinforcement learning:

the algorithm learns from the success or failure of its own actions  
→ E.g. a robot reaches its goal or fails



# A map to clarify the players role

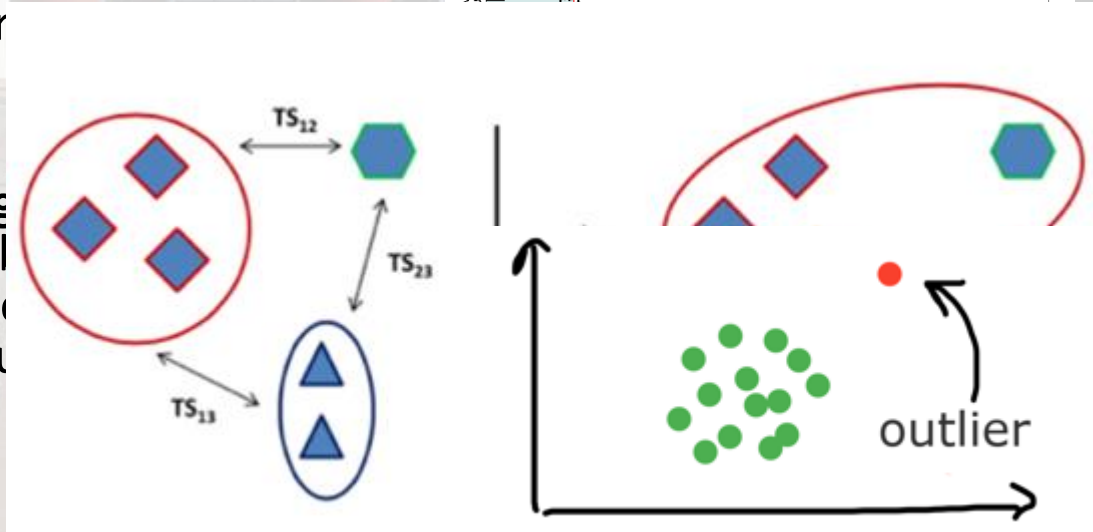
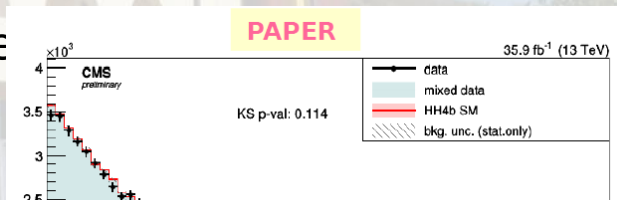




# A more complete list of ML tasks /1

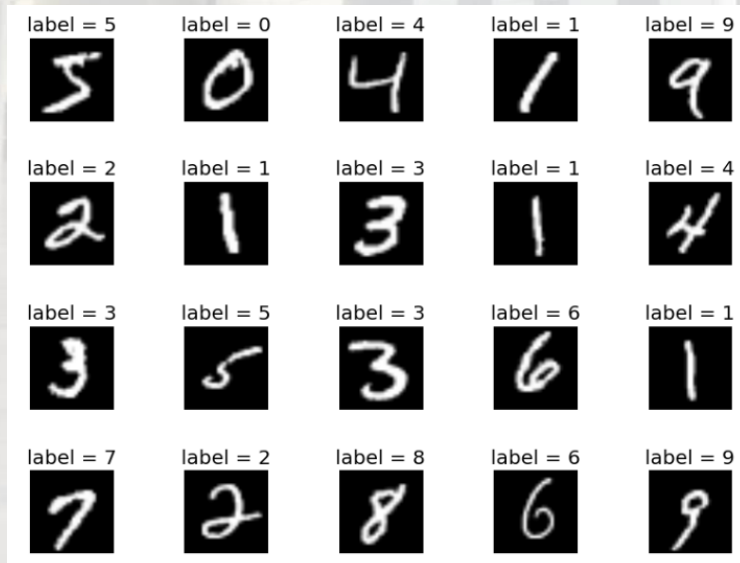
Classification and Regression are important tasks belonging to the "supervised" realm. But there are many other tasks:

- **Density estimation:** this is usually an ingredient of classification, but can be a task of its own.
- **Clustering:** find structures in the data, organize be a useful input to other tasks
- **Anomaly detection** (e.g. for purchasing habits; or new
- **Classification with missing** from simple classification by each mapping into the category of missing components (but handle it)



# A more complete list of ML tasks /2

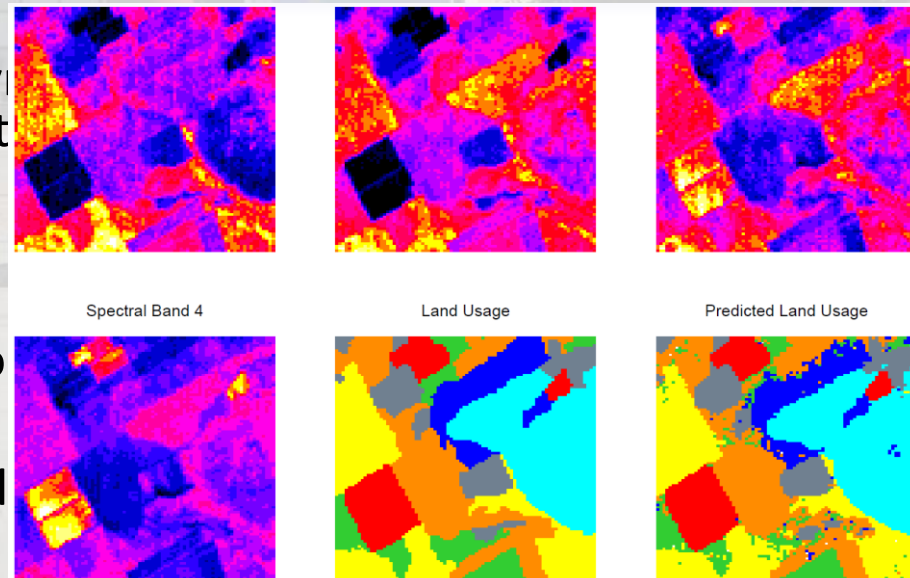
- **Structured output:**
  - **Transcription:** e.g. transform unstructured representation of data into discrete textual form; e.g. images of handwriting or numbers (Google Street view does it with DNN for street addresses), or speech recognition from audio stream
  - **Machine translation**
  - **parsing sentences** into grammatical structures
  - **image segmentation**, e.g. aerial pictures → road positions or land usage
  - **image captioning**



speech sy  
ch input

this can  
nd custo

an exampl

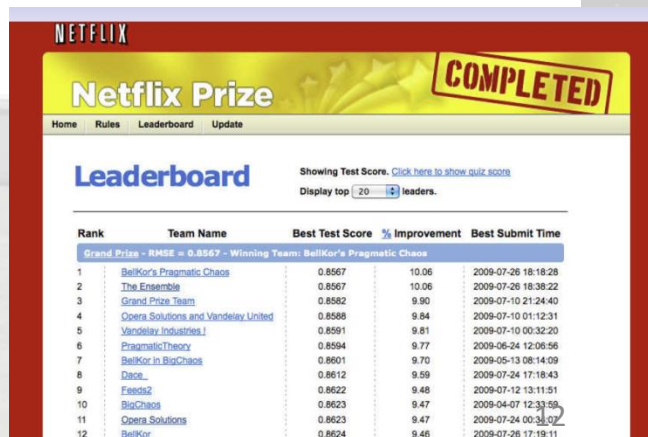


# The supervised learning problem

- **Starting point:**
  - A vector of **n predictor measurements X** (a.k.a. inputs, regressors, covariates, features, independent variables).
  - One has training data  $\{(\mathbf{x}, \mathbf{y})\}$ : events (or examples, instances, observations...)
  - The **outcome measurement Y** (a.k.a. dependent variable, or response, or target)
    - In classification problems Y can take a discrete, unordered set of values (signal/background, index, type of class)
    - In regression, Y has a continuous value
- **Objective:**
  - Using the data at hand, we want to predict  $y^*$  given  $x^*$ , when  $(x^*, y^*)$  does not necessarily belong to the training set.
  - The prediction should be **accurate**:  $|f(x^*) - y^*|$  must be small according to some useful metric (se later)

# Example: the Netflix challenge

- competition started in October 2006. Training data is ratings for 18,000 movies by 400,000 Netflix customers, each rating between 1 and 5.
- training data is very sparse— about 98% missing.
- objective is to predict the rating for a set of 1 million customer-movie pairs that are missing in the training data.
- Netflix's original algorithm achieved a root MSE of 0.953. The first team to achieve a 10% improvement wins one million dollars.



The screenshot shows the Netflix Prize Leaderboard page. At the top, there is a yellow banner with the text "Netfli Prize" and a red stamp that says "COMPLETED". Below the banner, there are navigation links for "Home", "Rules", "Leaderboard", and "Update". The main heading is "Leaderboard", followed by a link to "Showing Test Score. Click here to show quiz score" and a dropdown menu for "Display top 20 leaders". The table below lists the top 12 teams with their Rank, Team Name, Best Test Score, % Improvement, and Best Submit Time.

Rank	Team Name	Best Test Score	% Improvement	Best Submit Time
Grand Prize - RMSE = 0.8567 - Winning Team: BellKor's Pragmatic Chaos				
1	BellKor's Pragmatic Chaos	0.8567	10.06	2009-07-26 18:18:28
2	The Ensemble	0.8567	10.06	2009-07-26 18:38:22
3	Grand Prize Team	0.8582	9.90	2009-07-10 21:24:40
4	Coers Solutions and Vandelay United	0.8588	9.84	2009-07-10 01:12:31
5	Vandelay Industries I	0.8591	9.81	2009-07-10 00:32:20
6	PragmaticTheory	0.8594	9.77	2009-06-24 12:06:56
7	BellKor in BioChaos	0.8601	9.70	2009-05-13 08:14:09
8	Dace	0.8612	9.59	2009-07-24 17:18:43
9	Feeds2	0.8622	9.48	2009-07-12 13:11:51
10	BioChaos	0.8623	9.47	2009-04-07 12:33:58
11	Opera Solutions	0.8623	9.47	2009-07-24 00:33:02
12	BellKor	0.8624	9.46	2009-07-26 17:19:11

# The unsupervised learning problem

- **Starting point:**
  - A vector of **n predictor measurements X** (a.k.a. inputs, regressors, covariates, features, independent variables).
  - One has training data **{x}**: events (or examples, instances, observations...)
  - There is **no outcome variable Y**
- **Objective** is much fuzzier:
  - Using the data at hand, find groups of events that behave similarly, find features that behave similarly, find linear combinations of features exhibiting largest variation
- Hard to find a metric to see how well you are doing
- Result can be **useful as a pre-processing step** for supervised learning

# Ideal predictions, a' la Bayes

For a regression problem, the best prediction we can make for  $Y$  based on the input  $X=x$  is given by the function

$$f(x) = \text{Ave}(Y|X=x)$$

This is the conditional expectation: you just derive the  $Y$  average for all examples having the relevant  $x$ .

- It is the best predictor if we want to minimize the average squared error,  $\text{Ave}(Y-f(X))^2$ .
- But it is NOT the best predictor if you use other metrics. E.g., if you wish to minimize  $\text{Ave}|Y-f(X)|$  you should rather pick... Who can guess it?

$$f(x) = \text{Median}(Y|X=x)$$

If we instead are after a qualitative output  $Y$  in  $\{1\dots M\}$  (a discrete one, as in multi-class classification tasks) what we can do is to compute

$$P(Y=m|X=x)$$

for each  $m$ : conditional probability of class  $m$  at position  $X=x$ ; then we take as the class prediction

$$C(x) = \text{Arg max}_j \{P(Y=j|X=x)\}.$$

The above is the majority vote classifier.

**Problem solved?** Let us try and see how to implement these ideas.

# Implementation

To predict  $Y$  at  $X=x^*$ , collect all pairs  $(x^*, y)$  in your training data, then

- For regression, get  
 $f(x^*) = \text{Ave}(y | X=x^*)$
- For classification, get  
 $c(x^*) = \text{Arg max}_j \{P(Y=j | X=x^*)\}$

Alas, this would be good, but...

We usually have sparse training data, obtained by forward simulation. Our simulator gives  $p(x|y)$  but the process is stochastic. That means we cannot invert the simulator, extracting  $p(y|x)$ !

In most cases we have NO observations with  $X=x^*$ .

Who you're gonna call ?

**Density estimation methods!**

# DENSITY ESTIMATION





# Density Estimation

Given a sample of data  $X$ , one wishes to determine their prior PDF  $p(X)$ .

One can solve this with parametric or non-parametric approaches.

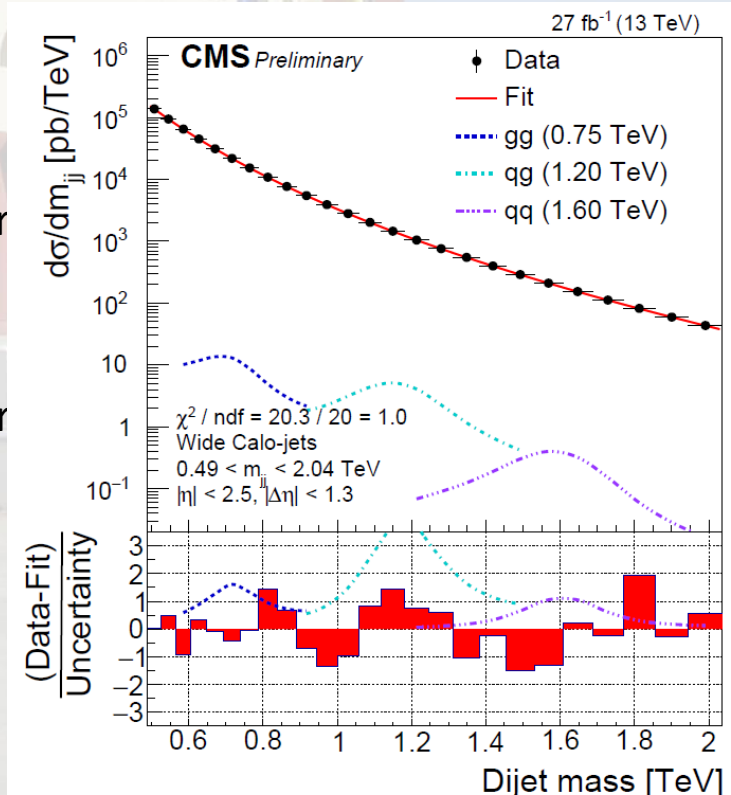
**Parametric:** find a model within a class, which fits the observed density

→ an assumption is necessary as the starting point

**Non-parametric:** sample-based estimators.

The most common is the histogram. NP density estimator have a number of attractive properties for experimental sciences:

- are easy to use for two things dear to HEP/astro-HEP: efficiency estimates (e.g. from Bernoulli trials) and for background subtraction
- lend themselves to be good inputs to unfolding methods
- are an excellent visualization tool, both in 1 and 2D



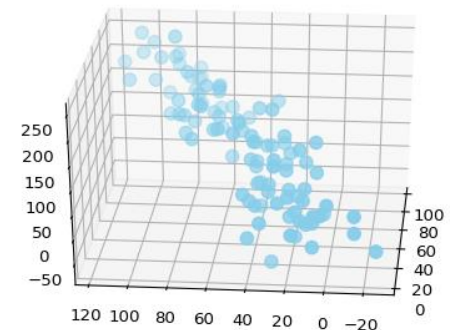
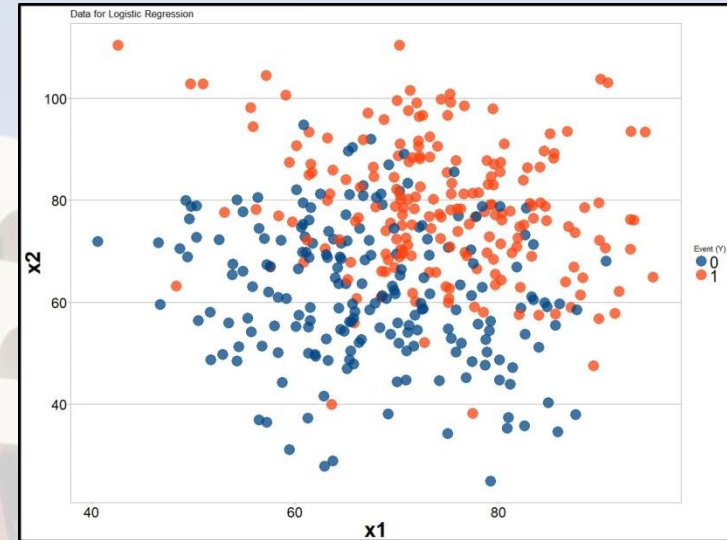
# The empirical density estimate

With sparse data, the most obvious estimate of the density from which i.i.d. data  $\{x_i\}$  are drawn is called "empirical probability density function" (EPDF), which can be obtained by placing a Dirac delta function at each observation  $x_i$ :

$$\hat{f}(x) = \frac{1}{N} \sum_{i=1}^N \delta(x - x_i)$$

Of course, the EPDF is rarely useful in practice as a computation tool.

Common way of visualizing 2D or 3D data (scatterplots)



# Histograms

Histograms are a versatile, intuitive, ubiquitous way to get a quick density estimate from data points:

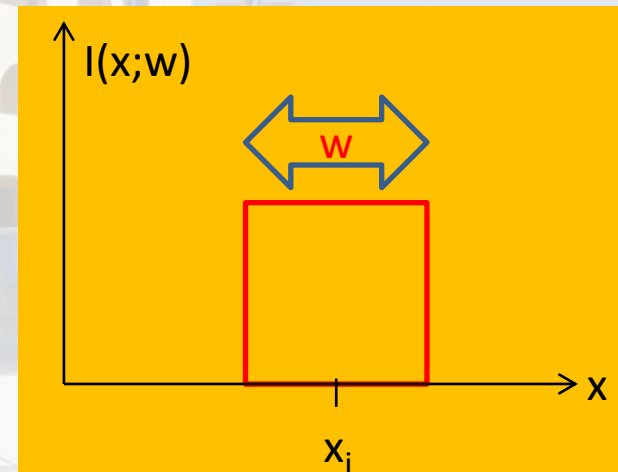
$$h(x) = \text{Sum}_{i=1 \dots N} I(x-x_i;w)$$

where  $w$  is the width of the bin, and  $I$  is a uniform interval function (indicator function),

$$I(x;w) = \begin{cases} 1 & \text{for } x \text{ in } [-w/2, w/2], \\ 0 & \text{otherwise} \end{cases}$$

The density estimate provided by  $h(x)$  is then

$$f(x) = h(x)/(Nw)$$



Yet **histograms have drawbacks:**

- they are *discontinuous*
- they *lose information* on the true location of each data points through the use of a "regularization", the bin width  $w$
- *not unique*: there is a 2D infinity of histogram-based PDF estimates possible for each dataset, depending on binning and offset.

# Kernel density estimation

A useful generalization of the histogram is obtained by substituting the indicator function with a suitable "kernel":

$$\hat{f}(x) = \frac{1}{N} \sum_{i=1}^N k(x - x_i; w)$$

The kernel function  $k(x, w)$  is normalized to unity. It is typically of the form

$$k(x - x_i; w) = \frac{1}{w} K\left(\frac{x - x_i}{w}\right)$$

The advantage of using a kernel instead than a delta is evident: **we obtain a continuous, smooth function**. This comes, of course, at the cost of a modeling assumption.

A common kernel is the **Gaussian distribution**; the results however depend more on the smoothing parameter  $w$  than on the choice of the specific form of  $k$ .

The "kernelization" of the data can be operated in multiple dimensions, too. The kernels operating in each component are identical but may have different smoothing parameters:

$$\hat{f}(x, y) = \frac{1}{Nw_xw_y} \sum_{i=1}^N K\left(\frac{x - x_i}{w_x}\right) K\left(\frac{y - y_i}{w_y}\right)$$

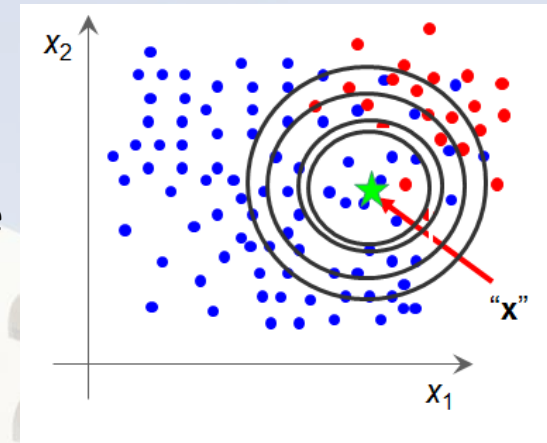
A different extension of the KDE idea is to adapt the smoothness parameter  $w$  to reflect the precision of the local estimate of the data density → adaptive kernel estimation

# Going multi-D: K-Nearest Neighbours

The kNN algorithm tries to determine the local density of multi-dimensional data by finding how many data points are contained in the **surroundings** of every point in feature space

One usually "weights" data points with a suitable function of the distance from the test point

Problem: **how to define the distance in an abstract space?**



Also, your features might include real numbers, categories, days of the week, etcetera...  
In general, it is useful to remove the dimensional nature of the features

General recipe: **standardization**

- find variance  $\sigma^2$ , obtain standardized variable  $y^2 = x^2 / \sigma^2$

# More on kNN

Once the data is properly standardized one can construct an Euclidean distance:

$$D(y, y') = \sum_{i=1}^D (y_i - y'_i)^2$$

kNN density estimates can be endowed with several parameters to improve their performance

**Most obvious is k:** how many events in the ball?

Rule of thumb: estimating a mean → if target varies a lot, can use small k; if variation small, k needs to allow precise estimate

**Common to have k=20-50**, but it of course depends on dimensionality D and size N of training data

# kNN, continued

## Also crucial to assess:

- **relative importance of variables:** assign larger weight to more meaningful components in the feature space (ones along which target has largest variance)
- **local gradients-aware:** one may try and adapt the shape of the "hyperellipsoid" to reflect how much target  $y(x)$  is variable in each direction, at test point  $x^*$

In general, kNN estimates suffer from a couple of shortcomings:

- The evaluation of local density requires to use all data for each point of feature space → **CPU expensive** (but there are shortcuts)
- The **curse of dimensionality:** for  $D > 8-9$ , they become insensitive to local density (see next slide)

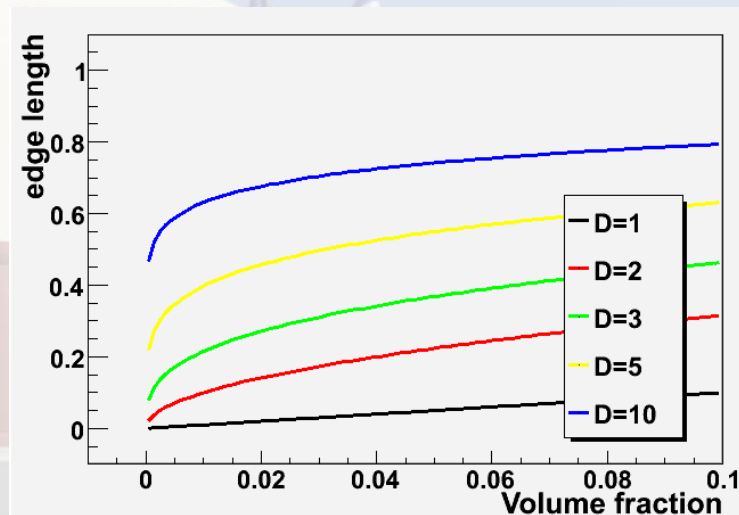
# The Curse of Dimensionality

The estimate of density in a D-dimensional space is usually impossible, due to the lack of sufficient labeled data for the calculation to be meaningful. An adequate amount of data must grow exponentially with D for a good representation

For high problem dimensionality, the k "closest" events are in no way close to the point where an estimate of the density is sought:

$$\text{edge length} = (\text{fraction of volume})^{1/D}$$

This makes kNN impractical for  $D > 8-10$  dimensions



In 10 dimensions, if a hypersphere captures 1% of the feature space, it has a radius of 63% in each variable span

HEP analyses often have  $>10$  important variables so the kNN has limited use as a generative algorithm for S/B discrimination



# What If We Ignore Correlations?

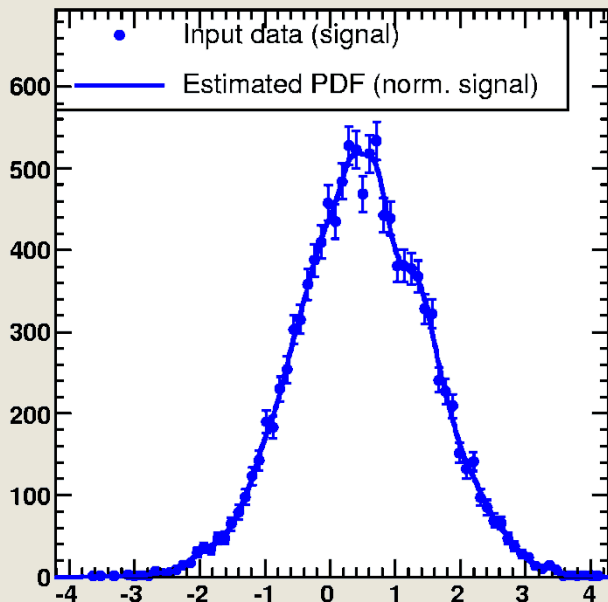
All PDF-estimation methods similarly fail when  $D$  is large.

A "**Naïve Bayesian**" approach consists in ignoring altogether the correlations between the variables of the feature space

That is, one only looks at the "marginals": 
$$P(\mathbf{x}) \cong \prod_{i=0}^D P_i(\mathbf{x})$$

$P$  is a product of "marginal PDFs"  $P_i$

One usually models the  $P_i$  with a smoothing of histograms



The  $P(x)$  thus obtained can be used to construct a discriminant (a likelihood ratio), or more simply, to assign the class label to the highest  $p(x)$  given  $x$

The method works if correlations are unimportant.  
**In HEP, however, this is not usually the case!**

# RESAMPLING TECHNIQUES



# Resampling Techniques

**Resampling techniques are pivotal for a number of ML tasks:**

- hypothesis testing (construction of discriminative methods)
- estimation of bias and variance (optimization of predictors)
- cross-validation (estimate accuracy of predictors)

**Resampling allows you to avoid modeling assumptions**, as you construct a non-parametric model from the data themselves. The benefits can be huge  
(as measured e.g. by performance of boosting methods)

Here we only briefly flash the generalities of three basic ingredients:

- permutation tests
- bootstrap
- ~~jackknife~~
- ~~cross-validation techniques~~ ← will treat later

# Permutation Sampling

Permutation sampling is mainly used in hypothesis tests; usually the question is: **are datasets A  $\{x_1 \dots x_{N_A}\}$  and B  $\{x_1 \dots x_{N_B}\}$  sampled from the same parent PDF?**

The way to answer is to form a sensitive statistic  $T$  (say: the mean of the observations) and **compare quantitatively** the difference between  $T_A$  and  $T_B$

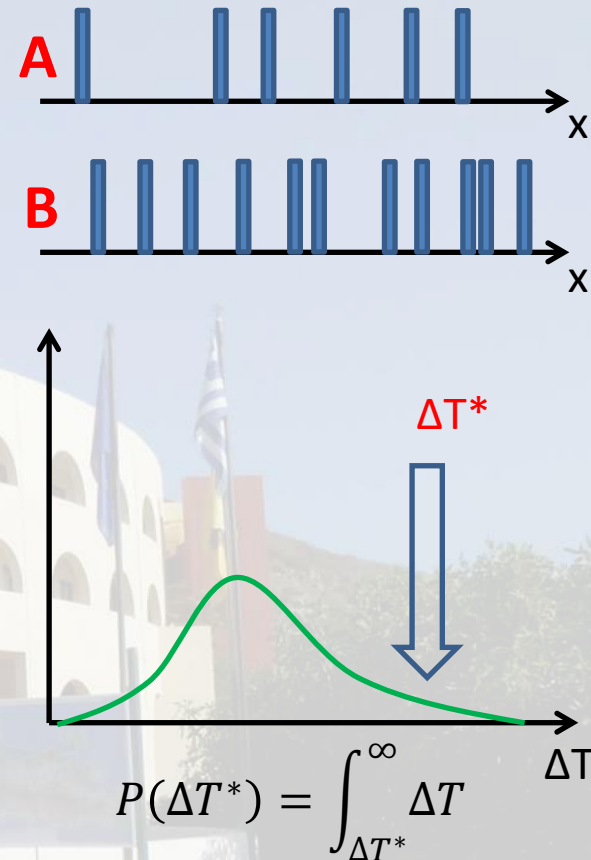
$$\Delta T^* = T_A - T_B$$

with what one would expect if the PDF were the same.

The comparison requires to **know how  $T$  distributes under the null hypothesis** (green curve).

Here permutation comes in: we assume  $A=B$ , merge  $A+B=C$ , and sample the  $N_A+N_B$  observations creating all possible pairs of splits  $A', B'$  **still of size  $\{N_A, N_B\}$** , computing distances  $\Delta T = T_{A'} - T_{B'}$ .

This provides **all the information in the data** about the distribution of  $\Delta T$ . **The permutation test makes no model assumptions, so it is called an exact test.**



# The Bootstrap

The Bootstrap (B.Efron, 1979) is called this way because it allows to "*pull oneself up from one's own bootstraps*".



Motivating problem: get the variance of an estimator  $\hat{\theta}(x)$  of a parameter  $\theta$ , for a sample of i.i.d. observations  $\{x_1 \dots x_N\}$ .

We may **generate  $M$  replicas of the dataset  $X$**  by repeatedly picking  $N$  observations at random from  $X$ , with replacement.

Then we estimate  $\theta$  in each replica, and proceed to obtain a sample mean and variance with the  $\hat{\theta}_i(x)$ ,

$$\bar{\theta} = \frac{1}{M} \sum_{i=1}^M \theta_i$$
$$s^2_{\theta} = \frac{1}{M-1} \sum_{i=1}^M (\hat{\theta}_i - \bar{\theta})^2$$

You get an estimate of variance without assumptions of the distribution

# THE DATA



# The data

A set of multi-dimensional data is made of  $N$  individual **events** (AKA cases or examples), made up each of  $D$  variables (or **features**, or attributes, or predictors)

We can think of our data as a table, where each column is an observed feature, each line a different event: we can organize them in a  **$N \times D$  matrix**

**In physics and astrophysics, typically  $N$  is large and  $D$  is small:** these data are called "**tall**"

In other disciplines one instead frequently encounters "**wide**" data, with few examples and many features: e.g. in DNA testing one may have thousands of gene sequences

The distinction is useful as different ML algorithms apply more successfully to the analysis of data depending on their shape; wide data is often problematic

- BDTs can handle wide data just as well as tall data
- resampling techniques may help with wide data
- but kNN and other simple methods do not work well with wide data
- also, linear discriminant analysis encounters difficulties with wide data as inversion matrix gets singular for  $N < D$

# Data preprocessing

An important part of data analysis concerns its preprocessing – a sometimes annoying chore, which forces you to fiddle with non-high-tech tools

You have to preprocess your data if

- some of the features are **missing** in a few of the events
- there are **outliers** that spoil the accuracy of your model
- some of the features are **categorical**

A preprocessing is not mandatory:

- you can **remove incomplete events** (but see below)
- you may decide to **ignore the effect of outliers**
- you may split the data in subsets with homogeneous categories and proceed with each, or **use methods that are robust** to their existence

In general, the proper handling of missing data, outliers, and categorical features can significantly improve your model



# More preprocessing: Centering, scaling, reflections

Some ML algorithms benefit from preprocessing of the features by **standardization procedures**, operated with univariate transforms

## Centering:

Centering consists in subtracting means off the marginals.

If  $X = \{x_1, \dots, x_N\}$  are the relevant coordinates of your  $N$  data examples, centering produces

$$X' = \{x_1 - \mu^*, \dots, x_N - \mu^*\},$$

where  $\mu^*$  is the observed mean.

Note that since  $\mu^* \neq \mu$ ,  $E[X] \neq 0$  in general.

# Scaling and reflections

**Scaling:** Multiplying each feature by a positive constant

**Reflection:** multiplication by negative constant

The main application of scaling is to force all features to have the same variance (usually chosen to be 1.0 → standardization).

By scaling one can "remove" the dimensional character of different features, to facilitate the interpretation of the resulting Euclidean distance

When should you use these preprocessing steps?

- Centering is **useless for decision trees, random forests, BDTs**
- Centering can instead **improve training stability for neural networks** (when applied with scaling)
- **Scaling is useful in kNN applications**, which are instead insensitive to centering or reflection. The same is true for distance-based methods

# Data with unbalanced classes

In classification applications, it is usually the case that the amount of training data for each class differs. Most algorithms confronted with unbalanced training samples will learn more about one class than the other → smaller classification error for the oversampled class

To handle this, one can **use Bayes theorem**, obtaining from the learned  $p(x|c_i)$  a posterior probability  $p(c_i|x)$  by accounting for the class population in the training:

$$p(c_1|x) = \frac{p(x|c_1)p(c_1)}{p(x|c_1)p(c_1)+p(x|c_2)p(c_2)}$$

This procedure is also known as "weighting" the training data.

If one wants to mix data in equal proportions, one may **undersample the majority class** or **oversample the minority class**. The former reduces CPU but also information; the latter is effective IF one does it by synthesizing new observations. This can be done e.g. using local density estimates (e.g. kNN)

# THE MODEL



# The mathematical model

Machine learning relies on building a model of your data: a mathematical characterization of the studied system, in probabilistic terms

**To build a model**, you need

- to understand the structure of your data
- to clarify the problem you want to solve: e.g. regression, classification, clustering, ...

Based on the above inputs, you may choose the most appropriate ML method

E.g.

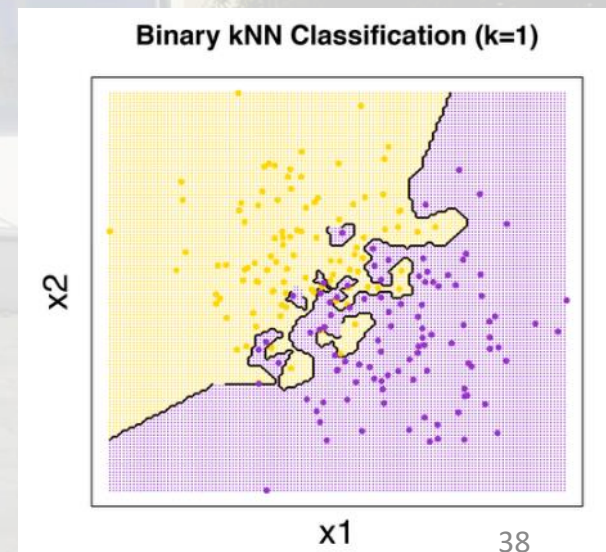
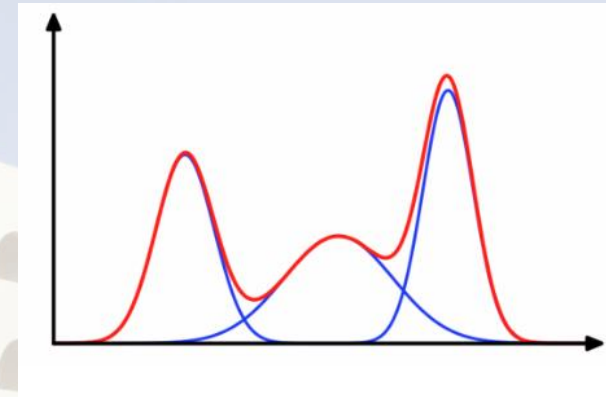
- for a low-D regression task, you might want to specify a family of parametric functions, and proceed to find the best choice of parameters
- for a complex classification task, you might focus on designing a proper architecture for a DNN

The method will **learn the model parameters** from available training data

The learned model will allow to **make predictions or inference** on previously unseen data

# Parametric or non-parametric?

- **Parametric models** are fully defined by a function, with a fixed set of parameters
  - $f(x;\theta) = \dots$ 
    - They can be a good choice when you want to retain insight in what is learned
    - They involve an **assumption** on the behaviour of the data  $\rightarrow$  bias
- **Non-parametric models** do not have a fixed set of parameters, and they may become arbitrarily more complex as you let them learn more information from training data
  - Assumption free (almost)
  - Higher variance, less bias



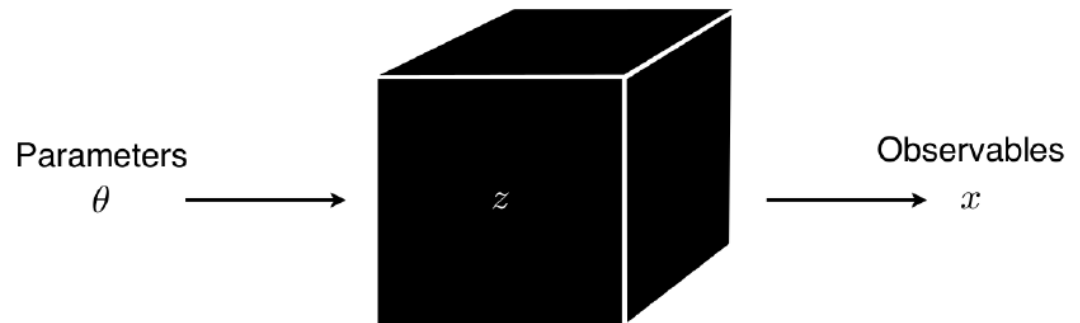
# Going forward or backward

In most applications of interest to fundamental science we observe natural phenomena and try to decrypt them by constructing a model, then doing inference on its parameters

**We are helped by simulations** that, based on the model, allow to artificially generate observations based on chosen parameter values.

Problem: the generative process is usually affected by stochasticity  $\rightarrow$  cannot be reversed trivially, as same  $\theta$  lead to different  $x$  at random

We have no analytic likelihood!  
The inference process becomes intractable, forcing us to use work-arounds.



**Prediction (simulation):**

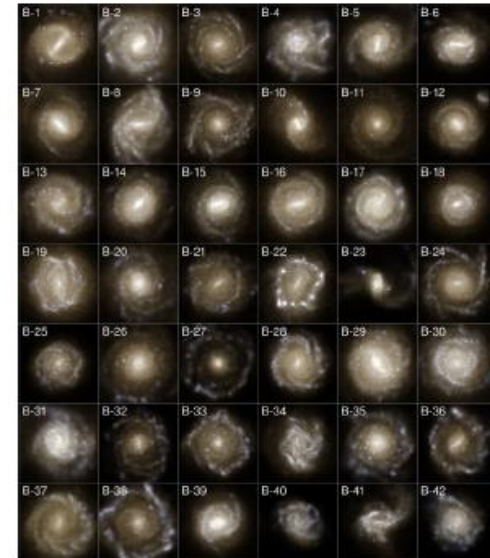
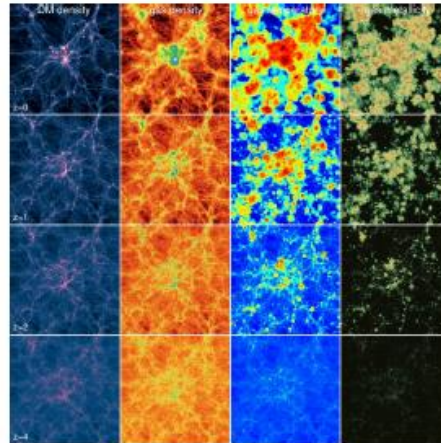
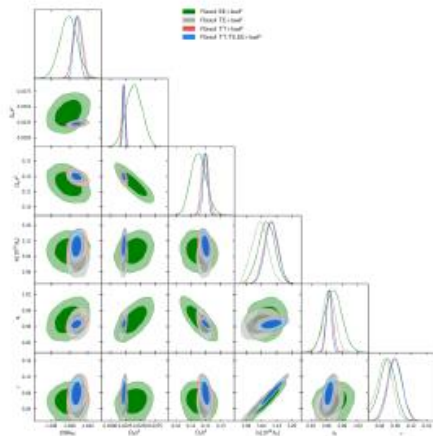
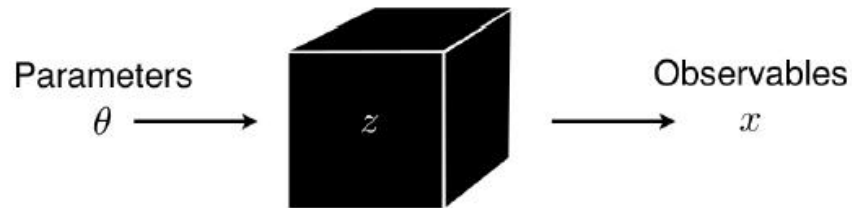
- Well-understood mechanistic model
- Simulator can generate samples

**Inference:**

- Likelihood function  $p(x|\theta)$  is intractable
- Goal: estimator  $\hat{p}(x|\theta)$

# Cases when we can only go forward / 1

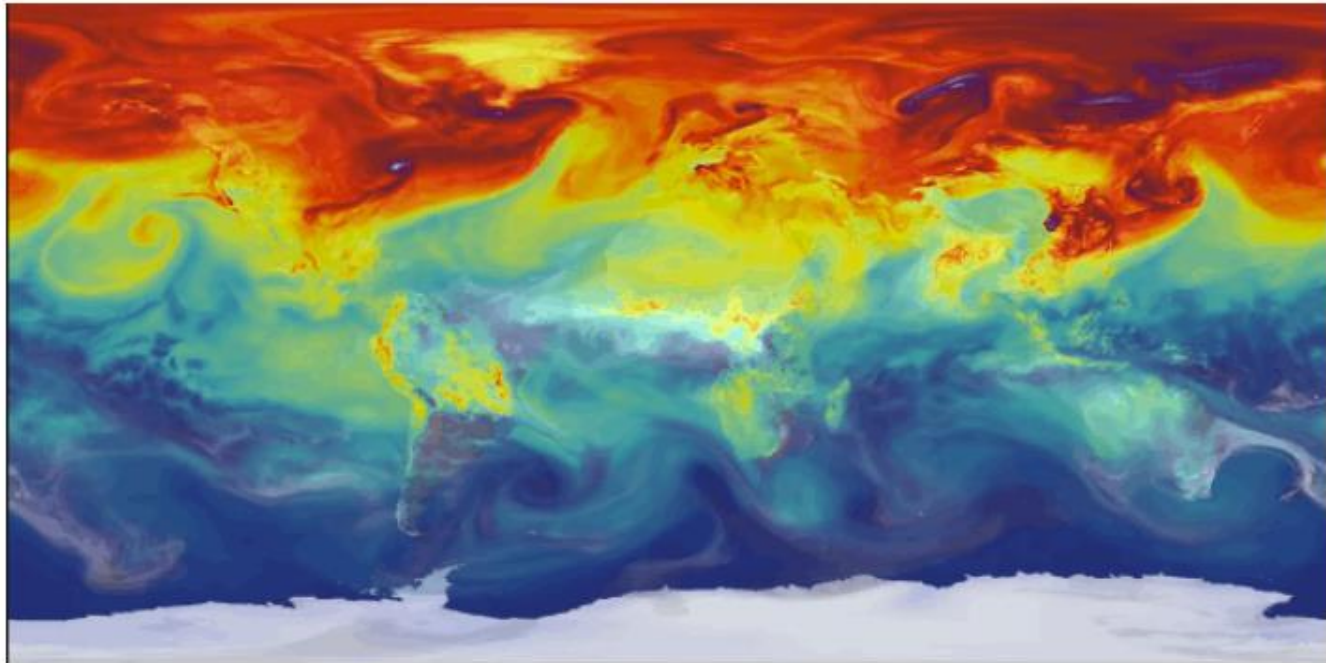
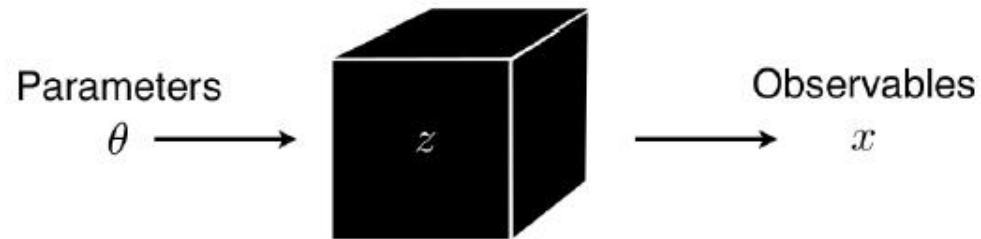
## Cosmological N-body simulations





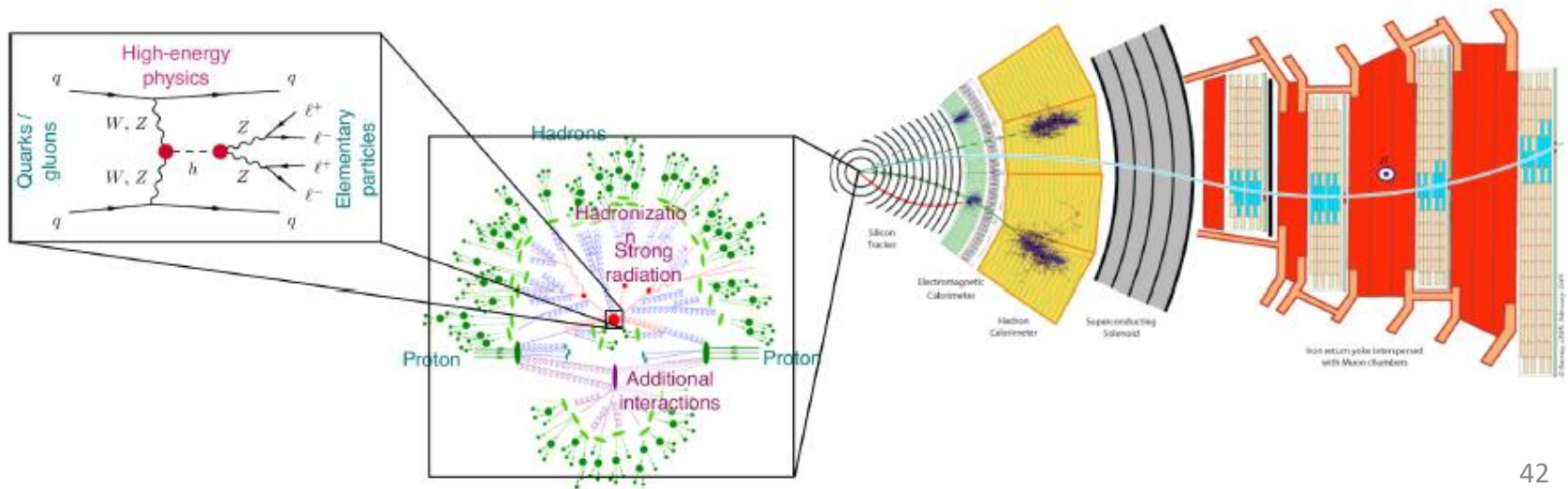
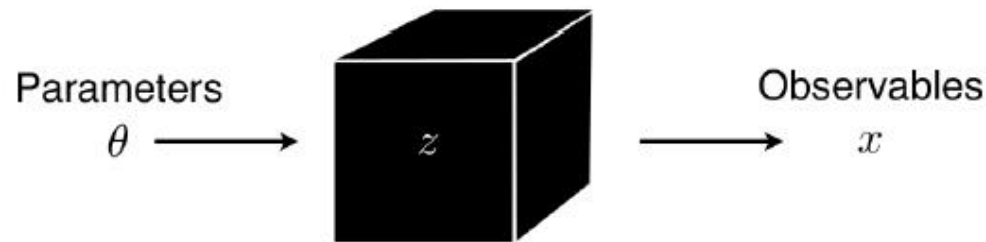
# Cases when we can only go forward / 2

## Climatology



# Cases when we can only go forward / 3

## Particle physics



# How to deal with this?

We resort to the construction of **proper summary statistics**, which have a much lower dimensionality than the observed data.

This in general throws away information, unless  $T$  is sufficient

**Statistical sufficiency:** the definition stems from the factorization theorem of Fisher-Neyman.

$T$  is sufficient for  $X$  if

$$f(X|\theta) = h(X) g(T(X)|\theta)$$

in other words, *all* the information on the unknown parameters  $\theta$  provided by data  $X$  is accessible through the function  $T$

The problem is that **finding sufficient statistics is very hard**, when at all possible. Machine learning methods are however capable of extracting summaries from the data that offer useful surrogates

# Generative and Discriminative models

In the light of the ill-defined nature of the PDFs we deal with in our physics problems, and bearing in mind the Neyman-Pearson lemma, the goal then becomes: **estimate**  $p(x|S)$  and  $p(x|B)$ , and then **construct their ratio  $r(x)$**

Many MVA algorithms do precisely that: they approximate multivariate densities. Among them are kernel density estimators, nearest neighbors...

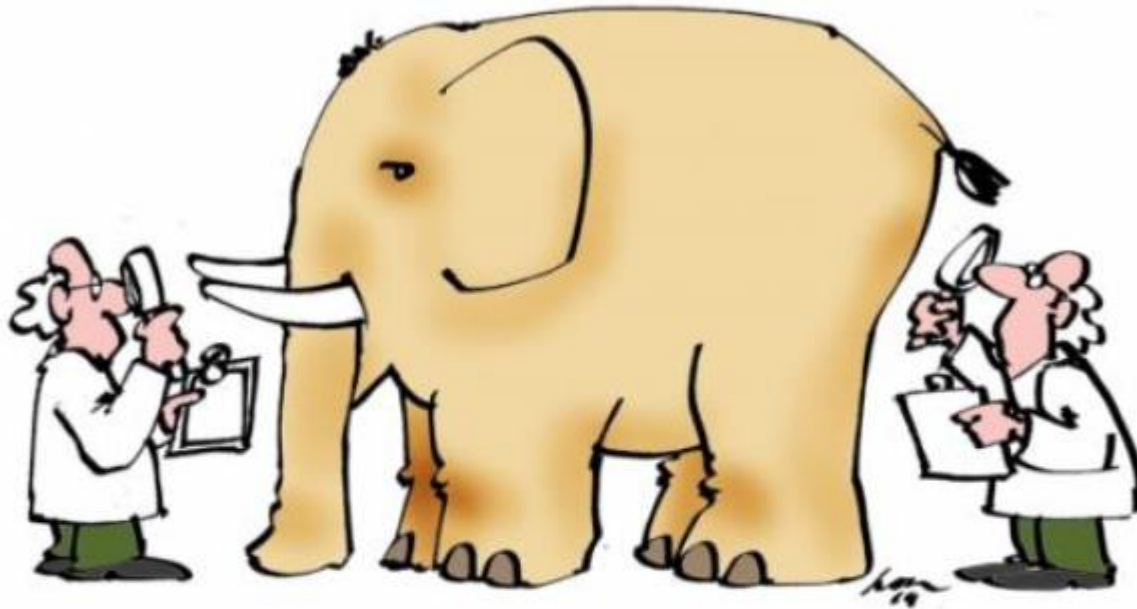
→ **generative algorithms**

or one may **approximate** the “likelihood ratio”, or a monotonous function of it, directly: finding hyperplanes in the observation space where  $r(x)$  is constant allows then to separate S from B pseudo-optimally

There is a bunch of ways to learn a monotonous function of the LR: linear discriminators, BDTs, neural networks. These are globally called

→ **discriminative algorithms**

# LECTURE 1 CONCLUSIONS



"Statistics: The only science that enables different experts using the same figures to draw different conclusions."

Evan Esar

# Conclusions for lecture 1

- Machine learning has a large overlap with statistical learning, which has been around for much longer.
  - Emphasis is on **large-scale applications**, and on **prediction accuracy** (as opposed to emphasis on models and their uncertainty)
- Density estimation is an important ingredient of many ML methods
  - especially when they require pdfs as inputs
- Data preprocessing may be an essential step that pays dividends in performance later on
- In fundamental science we often deal with the lack of analytic likelihoods
  - ML methods can provide **effective approximations to summary statistics** to carry out the inference work

# Lecture 2

## Classification and Decision Trees



# CLASSIFICATION



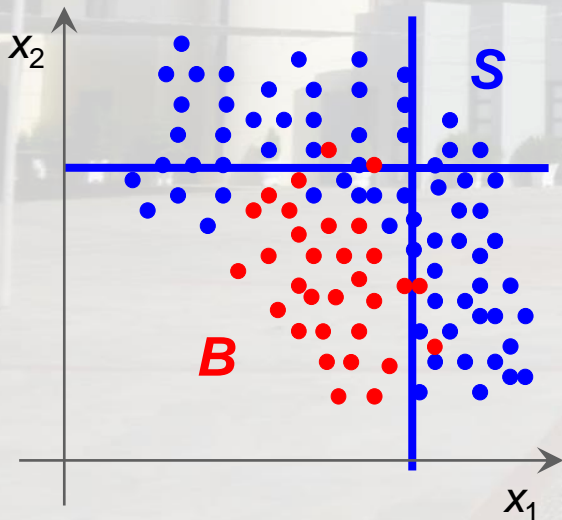


# Binary Classification

**Signal** and **Background**: a common problem in many setups

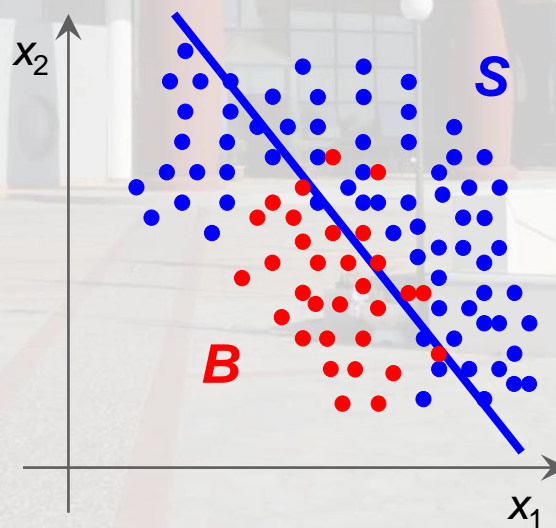
Generally we know the characteristics of two classes of events, and we wish to use them to find the best possible distinction with a fixed rule  $\rightarrow$  a "decision boundary" in the feature space of the event characteristics

A "rectangular" rule

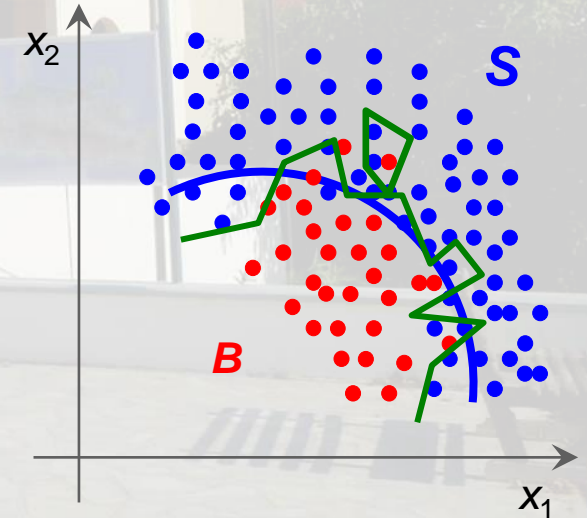


Low variance (stable), high bias methods

A linear rule



A non linear rule



High variance, small bias methods

# Feature Space and Output Function

Every **signal** or **background** event has “D” measured variables

Test statistic:

$$y(x): \mathbb{R}^D \rightarrow \mathbb{R}$$

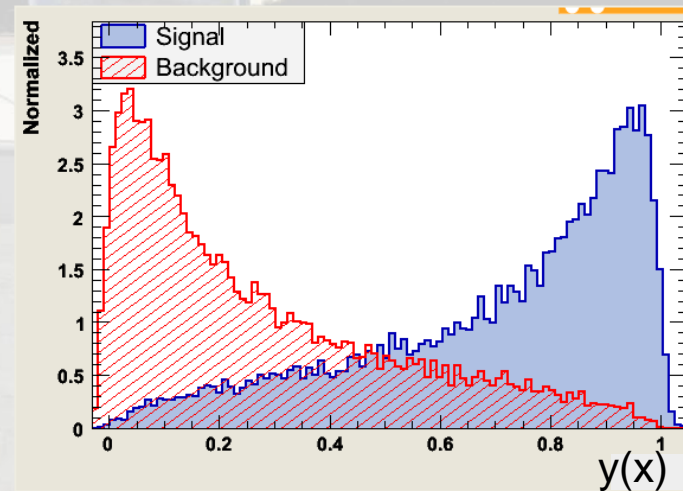
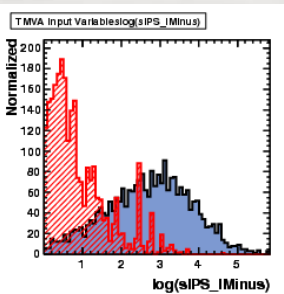
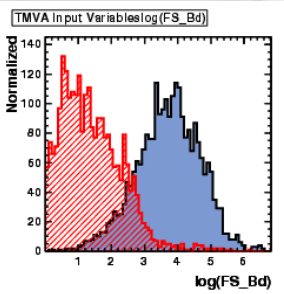
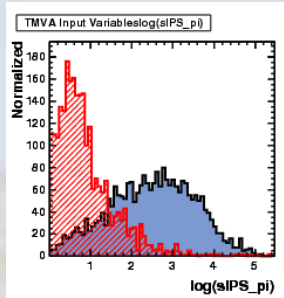
$$y = y(x); \mathbf{x} \text{ in } \mathbb{R}^D$$

$$\mathbf{x} = \{x_1, \dots, x_D\}: \text{input variables}$$

One wishes to find a map of the multi-D space of features, into a real variable that separates in a pseudo-optimal way the two classes

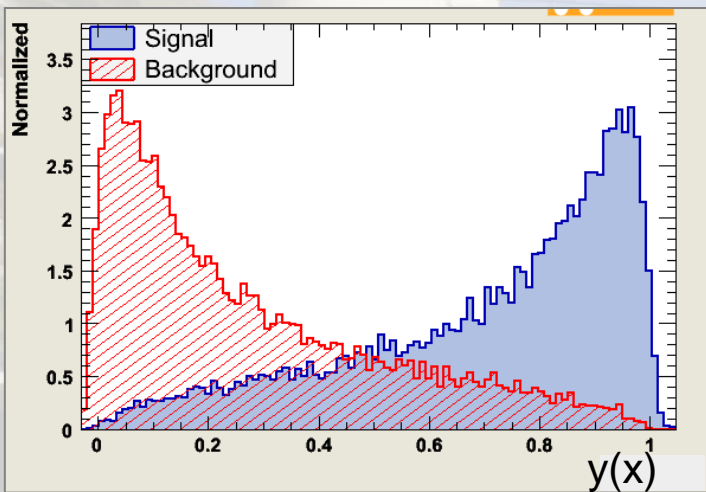
- D-dimensional “feature space”

One can construct a histogram of the resulting values of  $y(x)$  taken by **S** and **B**



# The "ROC" Curve

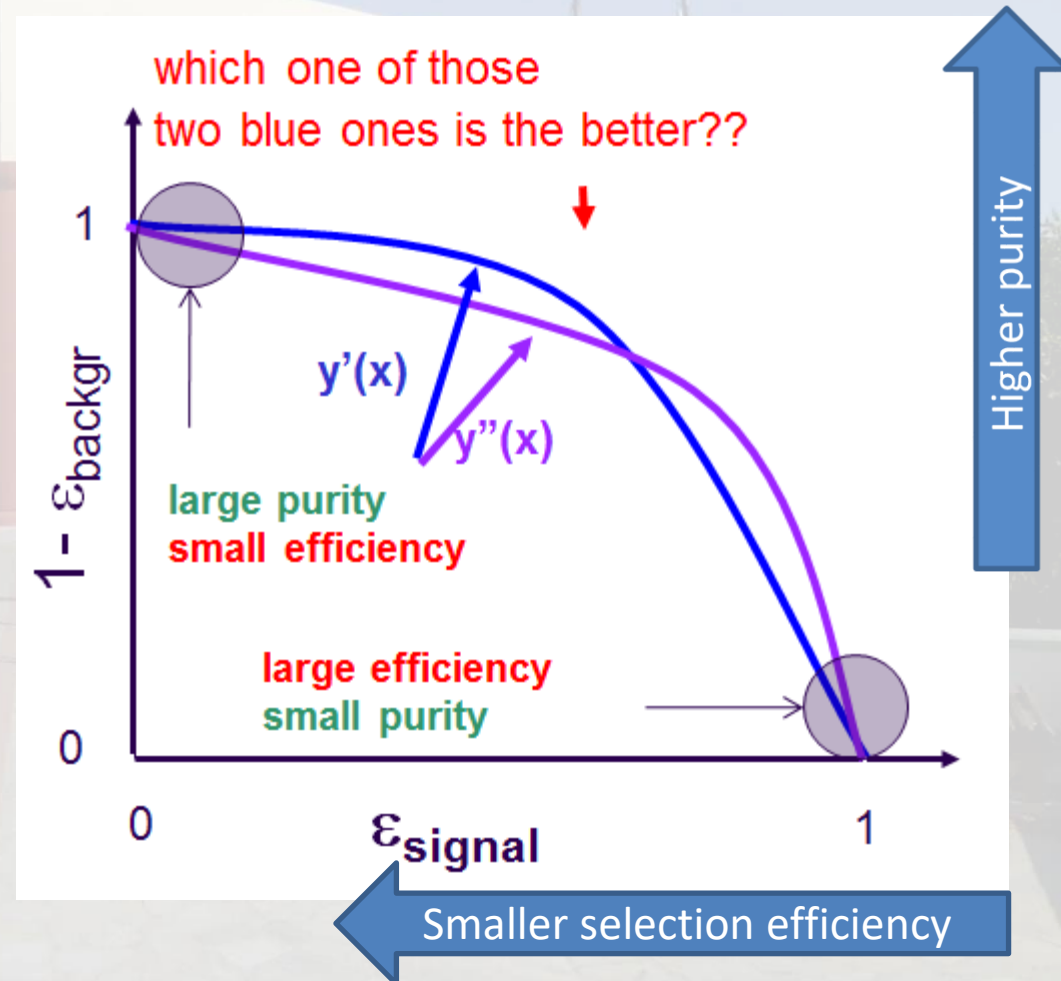
The ROC curve (receiver operating characteristic) is a way to summarize how well you are doing with your estimated  $y(x)$  in the classification problem



Cutting harder on  $y(x)$

How to quantify ?

- look at rejection at fixed eff
- compute AUC
- many other metrics available



# The AUC metric

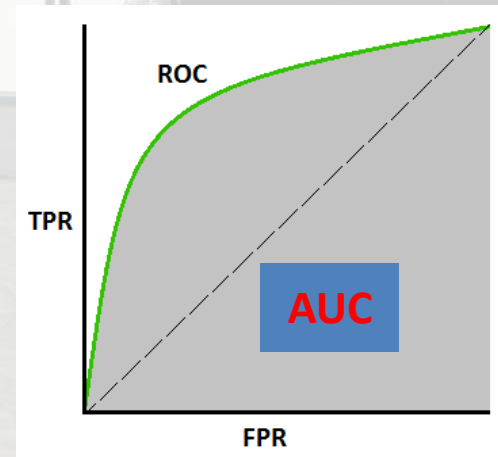
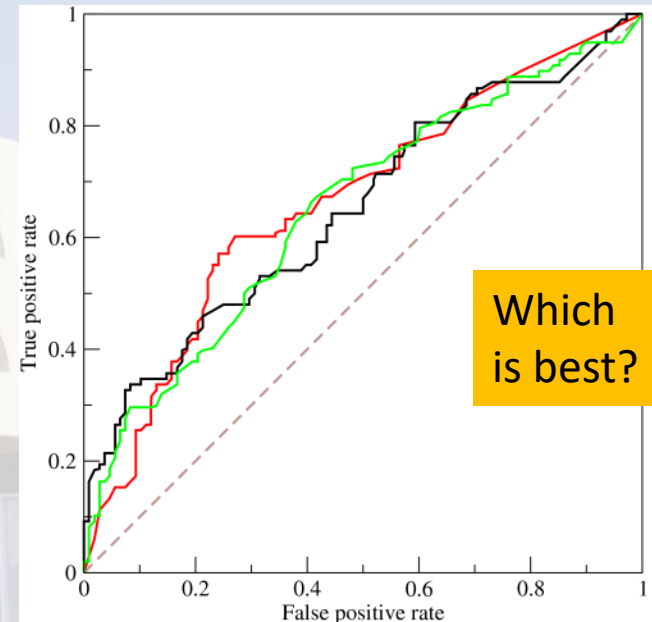
In some cases it is not trivial to rank classifiers by their performance.

In particular if one does not (yet) have an estimate of the proportion of signal and background (class probabilities), one cannot decide!

A simple criterion is to compute the **area under the ROC curve (AUC)**.

The AUC has a **clean statistical interpretation**: taken two events at random, one from each of the two classes, **AUC is the probability that the signal event has higher score than the background event.**

**AUC is a coherent measure of predictive power of  $y(x)$  if we have no information on the relative misclassification cost of the two classes – i.e. if we do not know the operating point. The more we know of that, the less useful AUC is.**



# The Neyman-Pearson Lemma

In a 1932 paper by J. Neyman and E. S. Pearson, "**On the Problem of the most Efficient Tests of Statistical Hypotheses**", the following notable result is demonstrated:

*Given two simple hypotheses, parametrized by densities  $p(x|\theta_1)$ ,  $p(x|\theta_0)$  that depend on specific values of a parameter, the likelihood ratio*

$$r(x|\theta_0, \theta_1) = p(x|\theta_1) / p(x|\theta_0)$$

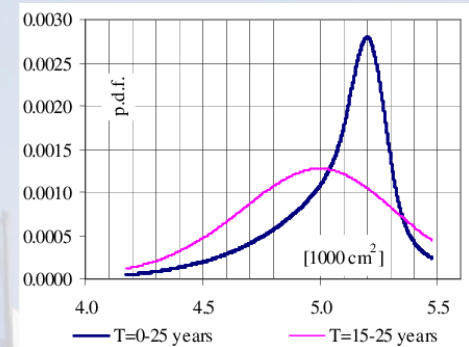
*is the most powerful test statistic to discriminate between them.*

The importance of this lemma cannot be overstated – it is the ultimate weapon for solving inference problems.

# How to Build a Classifier

The function

$$r(\mathbf{x}) = \frac{PDF(\mathbf{x}|S)}{PDF(\mathbf{x}|B)}$$



is therefore the best possible classifier of S versus B → **PROBLEM SOLVED! ?**

...NO:

- $p(x|S)$ ,  $p(x|B)$  are usually not perfectly known; in typical cases of interest in HEP and astro-HEP they may be **only estimated by forward simulation**, affected by stochastic phenomena
- hypotheses are not usually "simple vs simple" – **nuisance parameters** affect their determination

In general: One knows examples of S and B, but not their precise density

# The loss function

The **mean squared error** is a sound measure of the model accuracy, but it is not necessarily the metric most appropriate for our problem

E.g., in physics analyses we are concerned with the maximum sensitivity to a small signal, and we often use as a figure of merit the ratio  $s/\sqrt{b+s}$  → **don't do that**

In general, the quality of a predictive model can be quantified by constructing a function  $l(y, f(x))$ , a measure of the distance between the true class label  $y$  and the predicted response  $f(x)$ .

One may then define the expected distance

$$L(X, Y) = E_{X, Y} l(Y, f(X)) = \sum_{y \in Y} \int_X l(y, f(x)) P(x, y) dx$$

over the full space of  $X$  and  $Y$ . This can be computed empirically, using labelled data drawn from the pdf  $p(x, y)$ , by averaging  $l(y, f(x))$ :

$$\hat{L} = \frac{1}{N} \sum_{n=1}^N l(y_n, f(x_n))$$

# Regularizing the loss

The parameters of a model, learned through loss minimization, can sometimes diverge / be unstable

Often one wishes to regularize the loss by adding some constraining term, e.g.

$$\arg \min_{\mathbf{w}} L(\mathbf{w}, \mathbf{x}) = \arg \min_{\mathbf{w}} \frac{1}{N} \sum_{i=1}^N L(y_i, f(\mathbf{x}_i, \mathbf{w})) + \lambda \Omega(\mathbf{w})$$

The function  $\Omega$ , moderated by a **strength parameter  $\lambda$** , penalize some values of the parameters  $w$



# Ridge and Lasso

The regularization of the loss is often done with a L2 or a L1 norm on the parameters. The behavior one obtains is different.

A L2 norm

$$\Omega(\mathbf{w}) = \sum w_i^2$$

is equivalent to having a Gaussian prior in the PDF of the parameters

A L1 norm

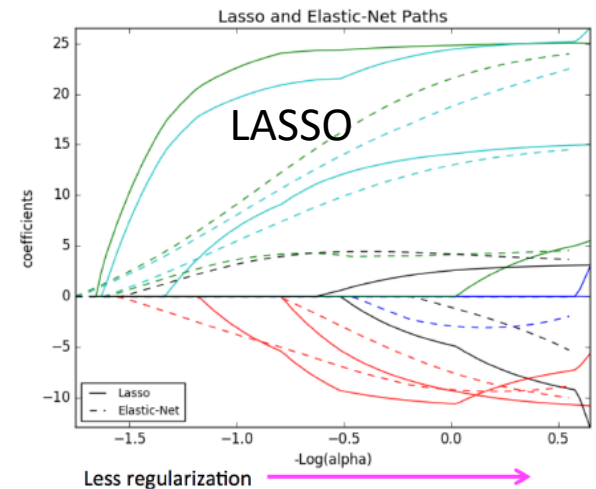
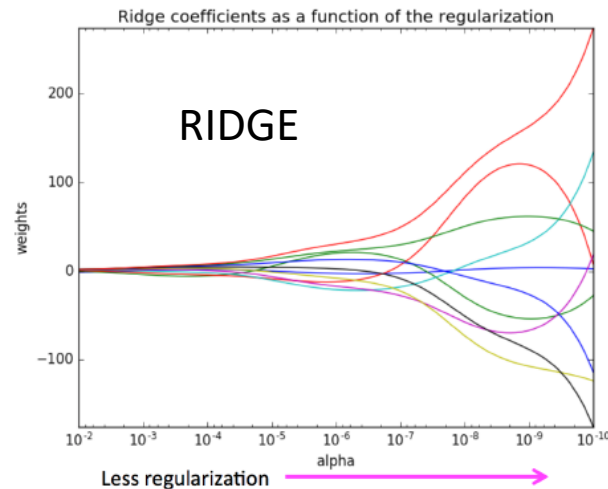
$$\Omega(\mathbf{w}) = \sum |w_i|$$

corresponds instead to a prior having a Laplace distribution

$$L(\mathbf{w}) = \frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^2 + \alpha\Omega(\mathbf{w})$$

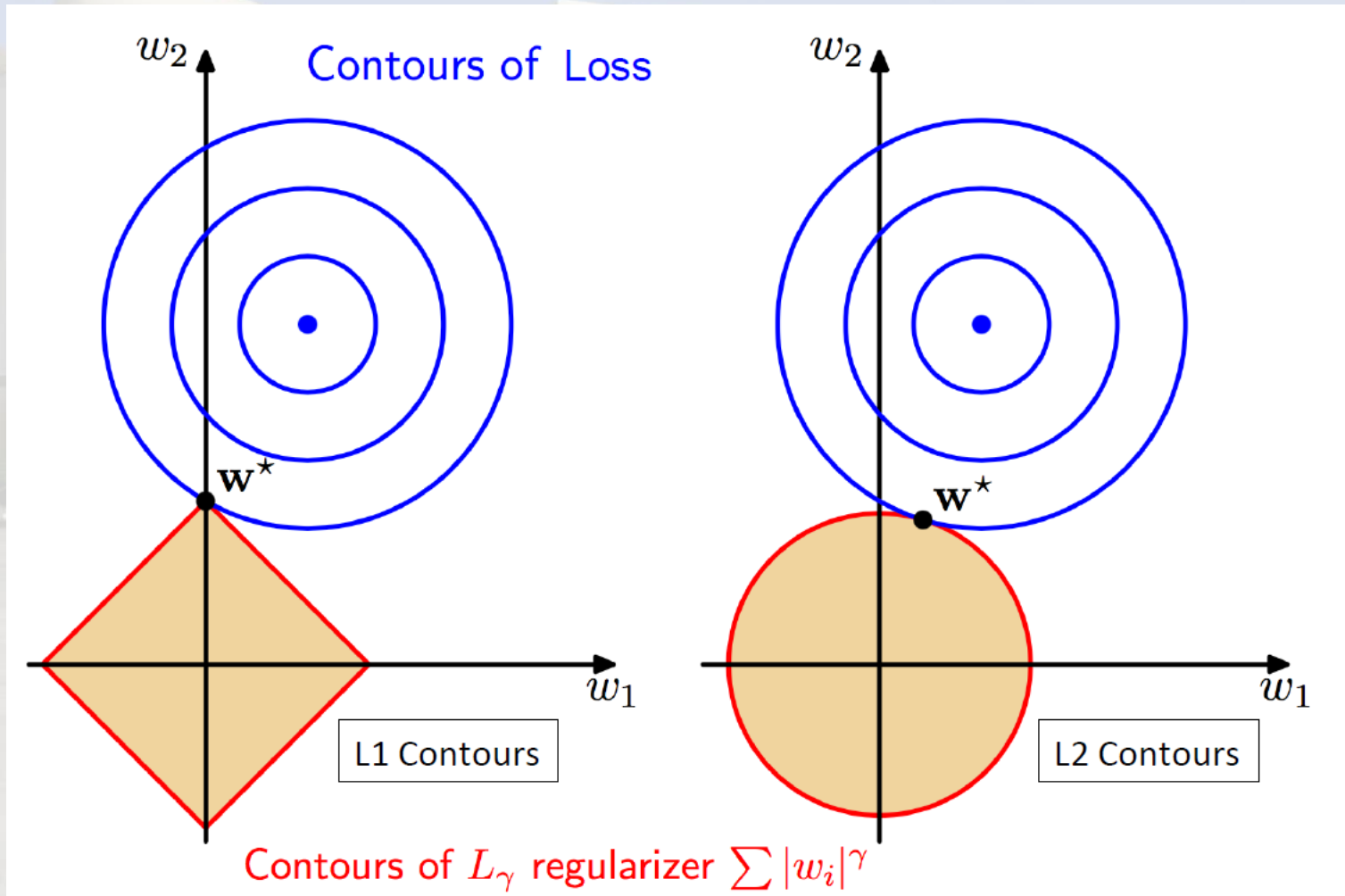
$$L2 : \Omega(\mathbf{w}) = \|\mathbf{w}\|^2$$

$$L1 : \Omega(\mathbf{w}) = \|\mathbf{w}\|$$



- L2 keeps weights small, L1 keeps weights sparse!

# What it means



# Binary Cross-Entropy

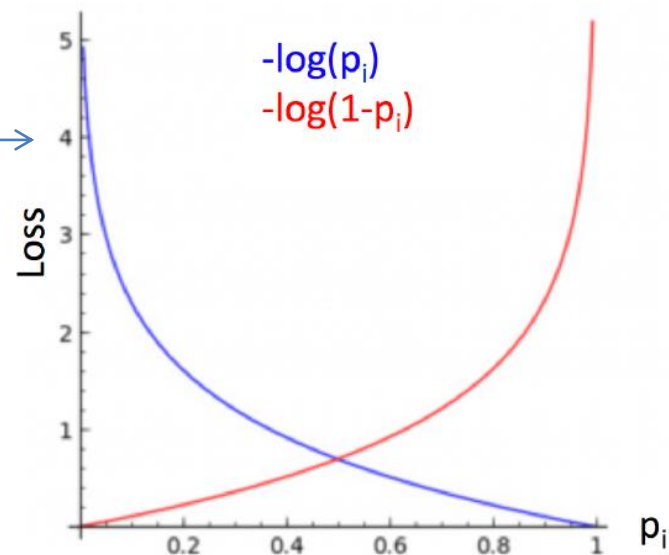
For binary classification, class assignment is modeled by a Bernoulli trial probability  $p$  following the Binomial distribution

$$p(y_i|x_i) = (p_i)^{y_i} (1 - p_i)^{1-y_i} \quad y_i = \{0, 1\}$$

A commonly used loss function is correspondingly defined as

$$L(\mathbf{w}) = - \sum_i (y_i \ln p_i + (1 - y_i) \ln(1 - p_i))$$

High loss for events with low probability of being in predicted class



# The pains of generalization

Let us investigate what is going on quantitatively when one computes a squared loss. Suppose you have a model  $\mathbf{f}(\mathbf{x})$  trained on data  $\mathbf{x}$ , that approximates a random variable  $\mathbf{t}$ . So this is a regression task.

In terms of expectation values you can write

$$E[t] = \hat{t}$$

$$E[f(x)] = \hat{f}(x)$$

One can study the generalization error on  $\mathbf{t}$  at any value  $\mathbf{x}$  by expanding

$$E[(f(x) - t)^2] =$$

Squared bias: this can be reduced with a more complex model



$$E[(t - \hat{t})^2] + (\hat{t} - \hat{f}(x))^2 + E[(f(x) - \hat{f}(x))^2]$$

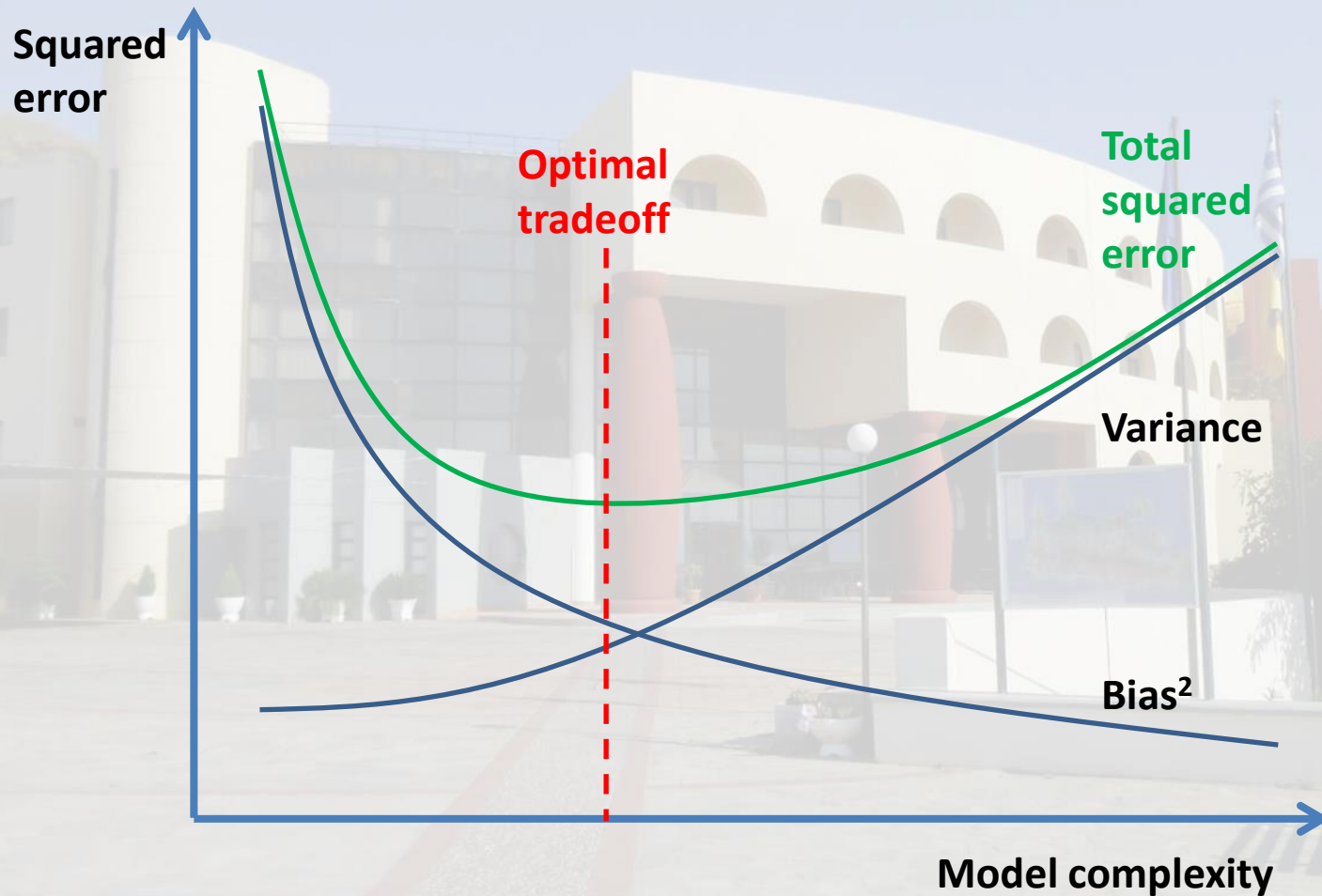


Noise term – cannot be improved with modeling



Variance term: the more complex the model, the higher this term, as it tries to capture erratic behavior

# Bias-Variance Tradeoff



With larger amounts of data the model can become more complex (smaller bias), because the variance diminishes → can live further on the right in the graph

# Training and testing

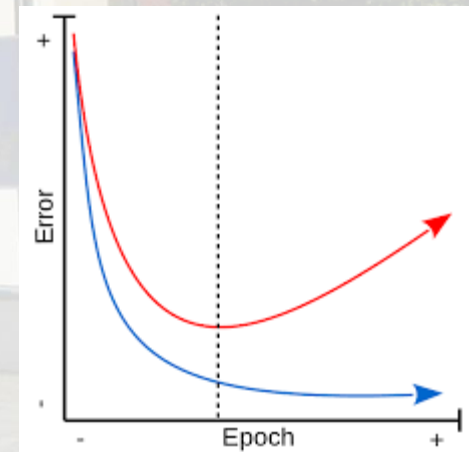
In order to construct and study a classifier built with a supervised algorithm – e.g. a decision tree, one needs labelled data from the two classes. **The more data, the better!**

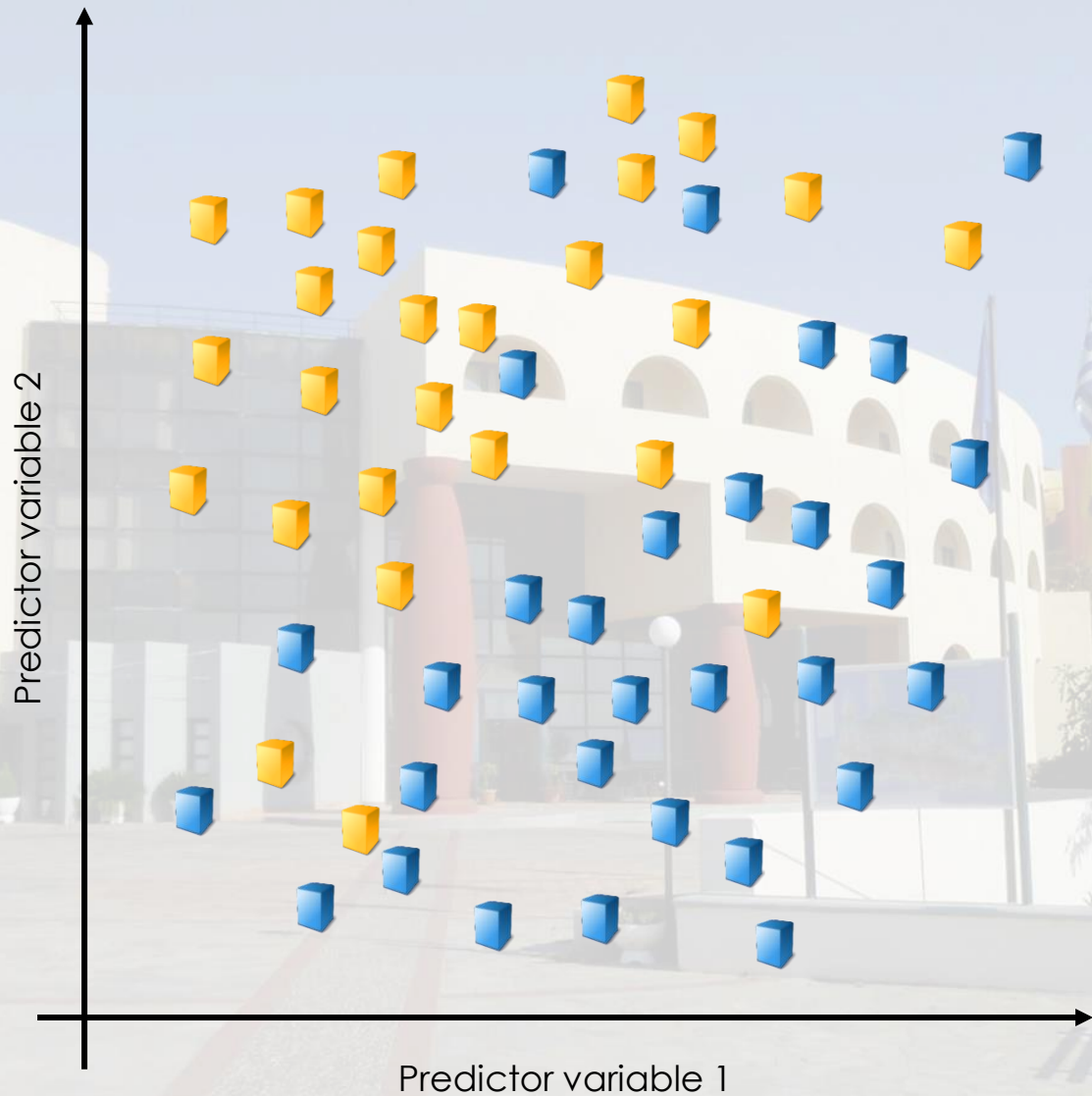
One must decide how to use the available labeled data in the construction of the classifier or regressor. **Can we use the same data for both training and testing?**

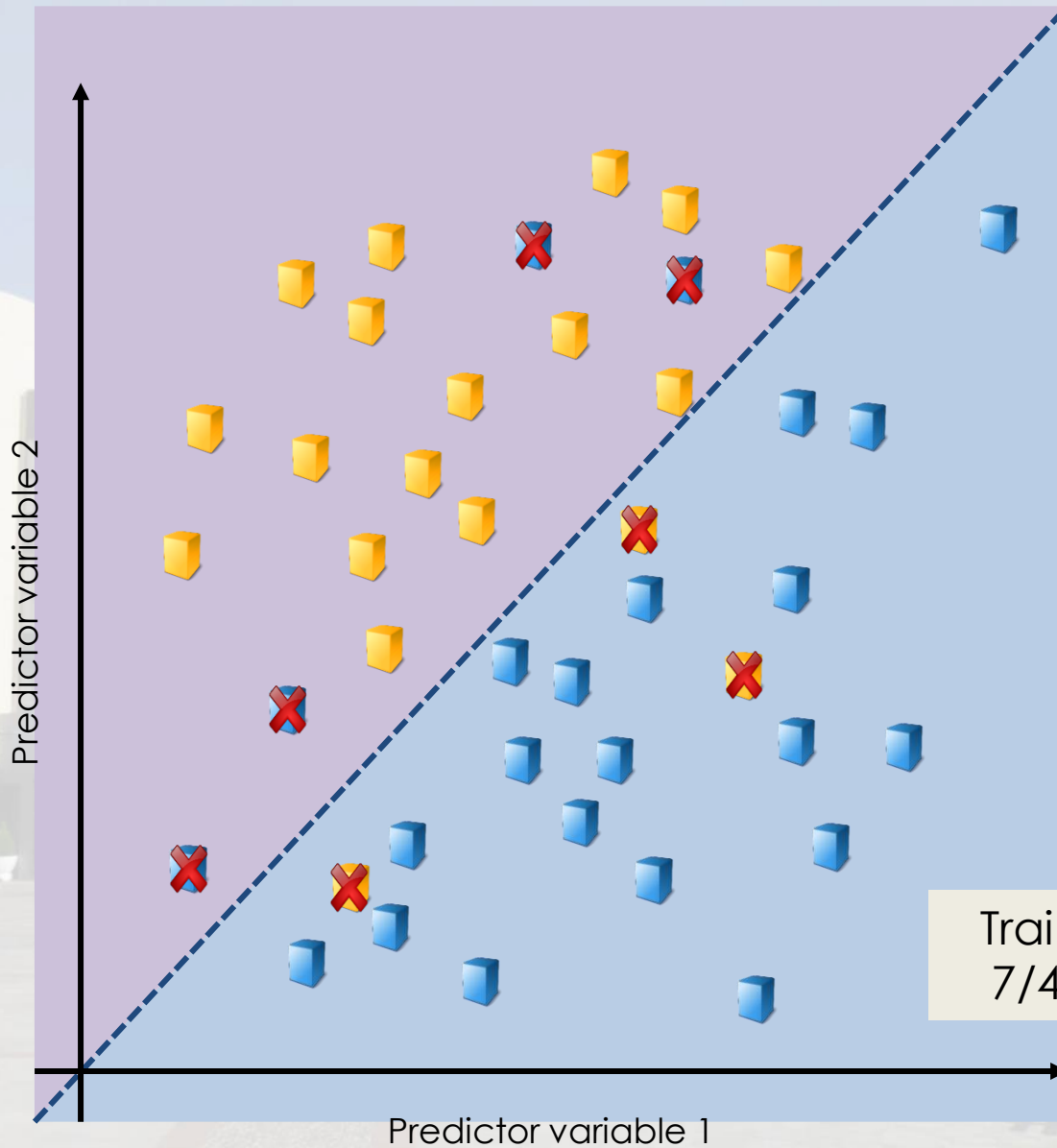
**Answer: No** – the error estimated in the training phase ("resubstitution error") is optimistic: this is the phenomenon called **over-training**

During training, the model attempts to learn the features of the joint distribution  $p(x,y)$  of data  $x$  and labels (or values, in regression)  $y$

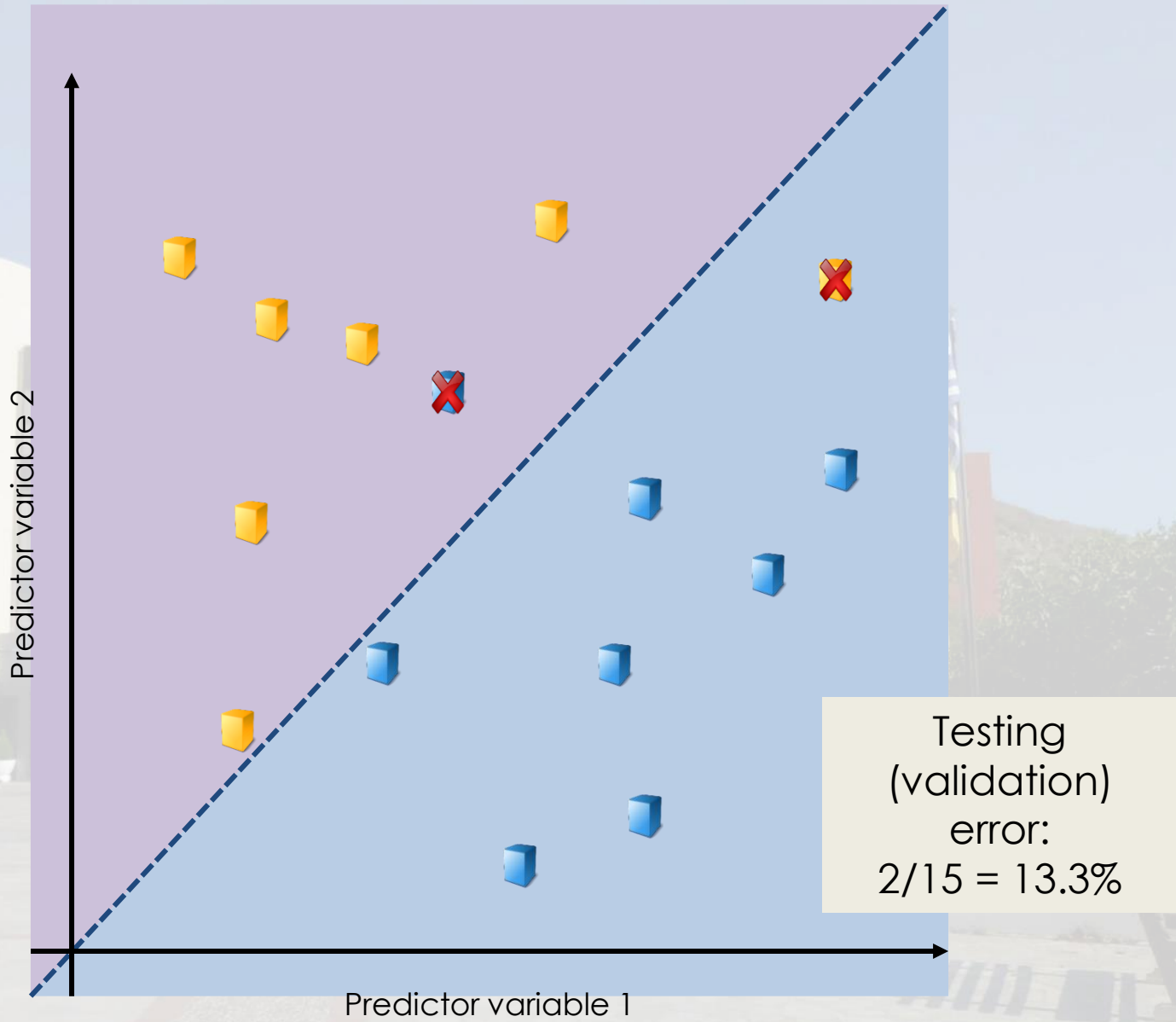
But **training data carries noise with it** – and this component will be learned by a flexible model too, deteriorating the accuracy on data not seen in the training phase

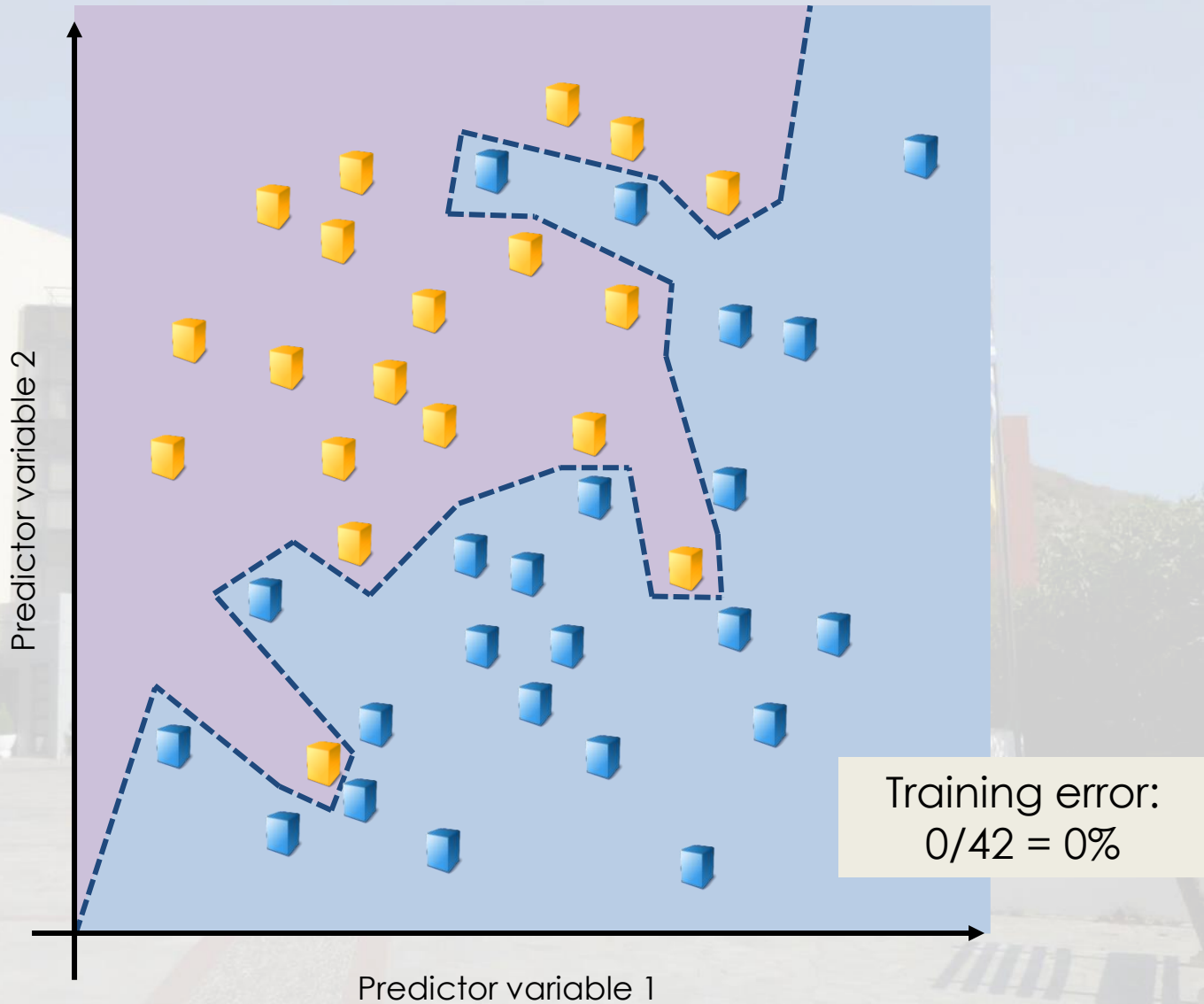


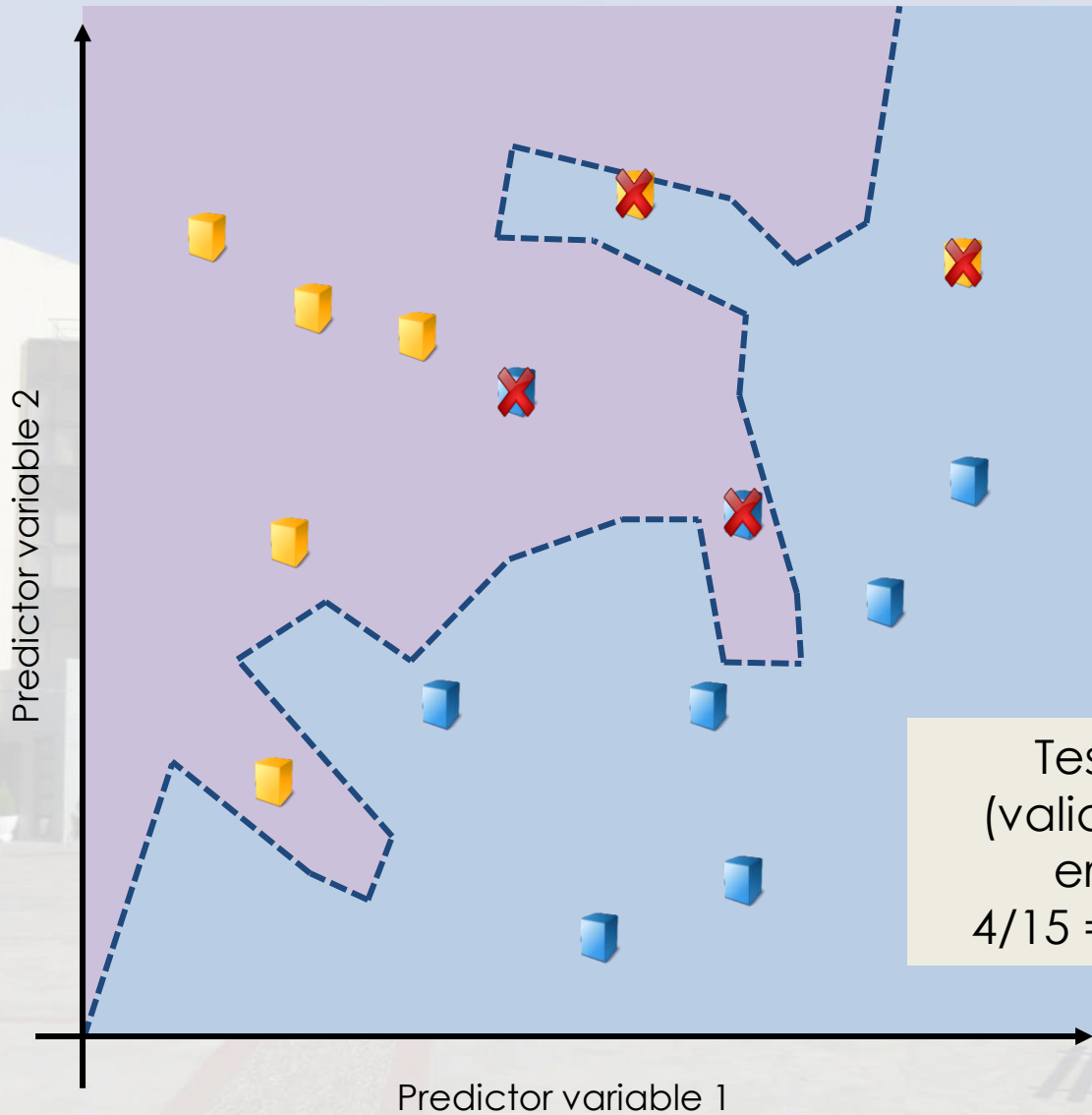












# Training vs Testing

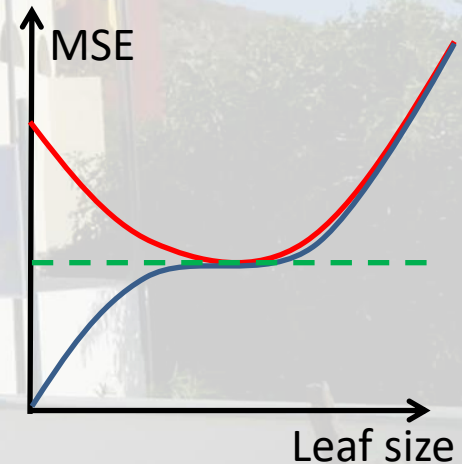
So we split the labelled data into **training and testing** subsets.

Let us consider a decision tree where we vary the leaf size from 1 (perfect classification at training stage) to  $N$  (training size, no tree). As the size of leaves increases we go from a flexible, high-variance tree to a robust, but high-bias one.

We observe that the training error increases with leaf size. This is a general property: the training error decreases with the model complexity

The test error instead **decreases** with leaf size until an optimal value is reached, which is close to the Bayes error.

(The above behaviour is not universal; for e.g. ensemble learners the test error also increases monotonically with leaf size)

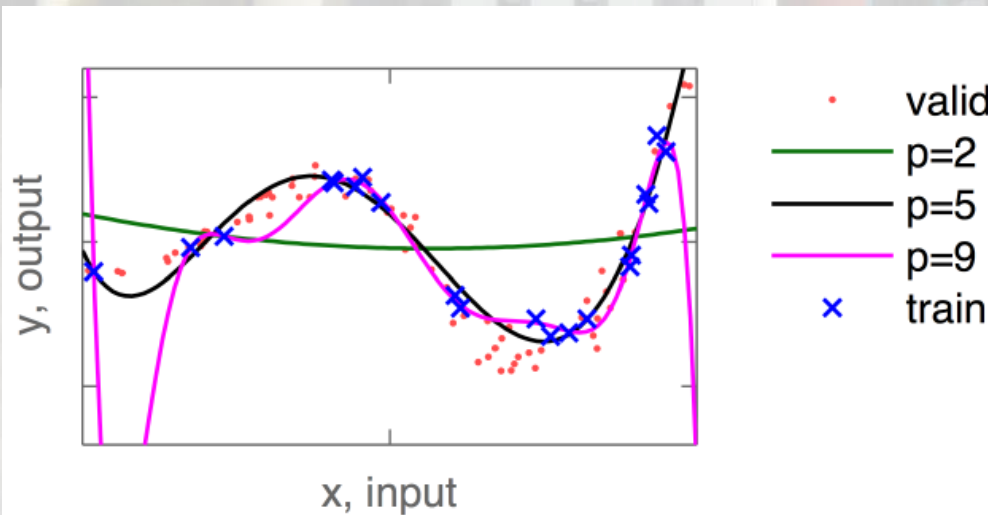


# One fitting example

Here we have training data (blue crosses) and a few polynomial models that try to capture their variation ( $y$ ) by minimizing the mean square error

Validation data (red points) also shown

The problem cannot be "solved" by training data alone, as a more complex model would always seem to work better than a simpler one



A graph of the MSE vs model complexity is all that is needed to see what model complexity is appropriate. But **we use test data for that!**

# Training, Validation, Testing

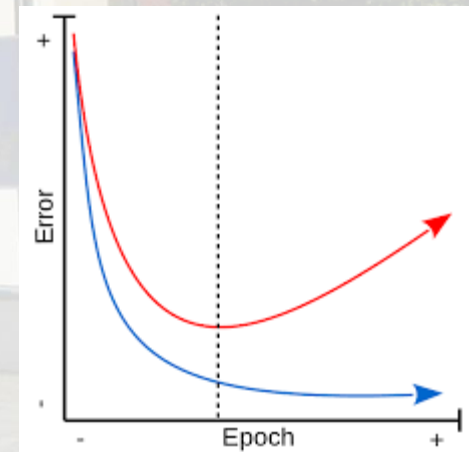
In order to construct and study a classifier built with a supervised algorithm – e.g. a BDT, one needs labelled data from the two classes.

Be given a sample of  $N_S$  signal events and  $N_B$  background events, one usually separates these sets in three parts:

**Training set:** events used to build the classifier. The algorithm employs them to estimate the prior densities of S and B, or directly the likelihood ratio or a monotonous function of it

**Validation set:** this is used to understand whether the training was too aggressive (overfitting), and to tune the algorithm parameters for best results

**Test set:** this sample is **totally independent from the former two**, and it is used to **obtain a unbiased estimate of the final performance** of the model, previously learned, validated and optimized.



The quality of the generalization can be further studied by more advanced partitions and resampling techniques. A common one is **k-fold cross validation** (see below).

# Leave-one-out Cross Validation

Data is costly! So, **often it is impractical to keep large holdout samples** for validation.

Imagine you want to optimize a hyperparameter  $s$  affecting the precision of your model. How to proceed?

You can come up with the idea of removing just one event from the set. Train the learner on  $N-1$  events, and test it on the  $N^{\text{th}}$  one:

$$\text{MSE}_N = (y_N - y^*)^2$$

That is a **very rough estimate!** But we can iterate on all  $N$  and take the mean:

$$\langle \text{MSE} \rangle = 1/N \sum_i (\text{MSE}_i)$$

You are using almost all data for training, so the returned answer is stable and has low variance; plus, the method is deterministic.

LOOCV is good, but can be **very CPU consuming** for large samples → use it only for very small data sizes.

# k-Fold Cross Validation

Leave-one-out cross validation can be generalized by leaving out  $1/k^{\text{th}}$  of the data

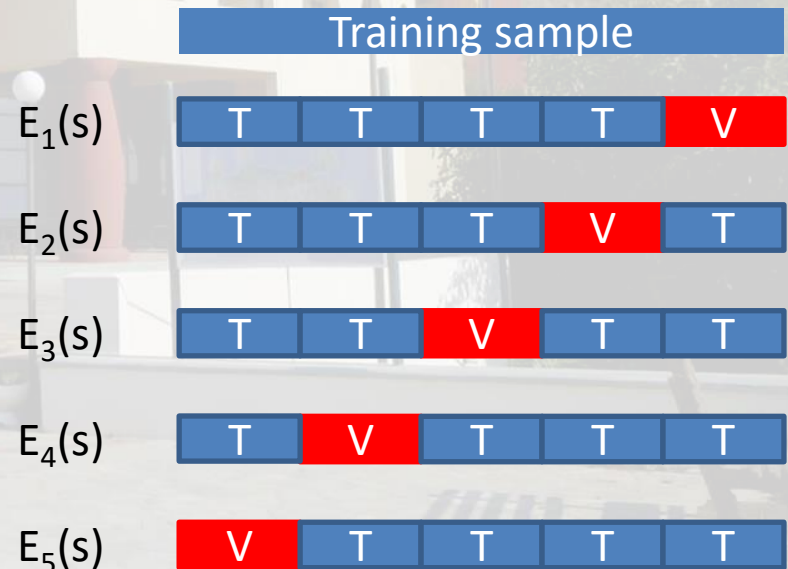
If  $k$  is  $>4$ , the training accuracy is not affected significantly, and we still get a reasonable estimate of the uncertainty.

## Recipe:

- Divide training data into  $N$  equally sized subsets ( $N=5-10$  is typical)
- For each  $k=1\dots N$ , train a classifier, or fit data with  $k$ -th sample as hold-out; apply to  $k$ -th subset and obtain error (or loss)  $E_k(s)$
- Obtain CV error by averaging:

$$\text{CVE}(s) = 1/N \text{ Sum}(E_k(s))$$

- Pick  $s$  such that CVE is minimum



Narsky advises  $N=10$  for most problems.



# LOSS MINIMIZATION



# Gradient Descent

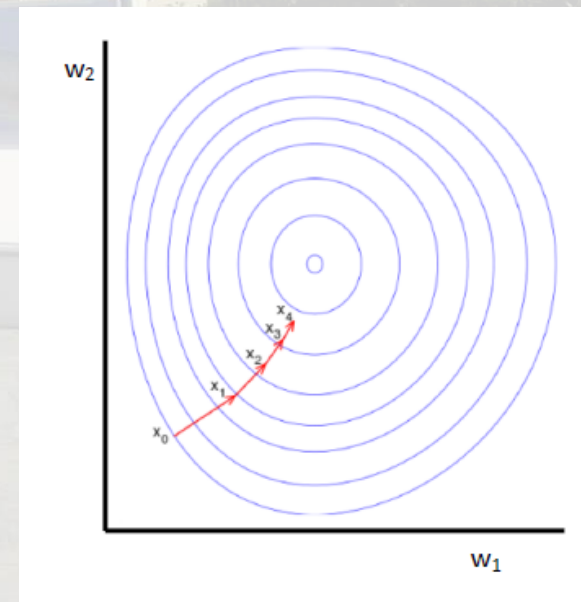
To minimize  $-\log L$  and find optimal value of model parameters, in the absence of an analytical description, we "descend" toward the minimum by approximating the shortest route with local information:

- 1) find gradient of  $L$  w.r.t. parameters  $w$ :  $\frac{\partial L(w)}{\partial w}$
- 2) update parameters:

$$w' \leftarrow w - \eta \frac{\partial L(w)}{\partial w}$$

and iterate.

Success depends on how fast you descend, moduled by "learning rate"  $\eta$ .



# Stochastic Gradient Descent

Computing the gradient over the whole training set at each step is sub-optimal:

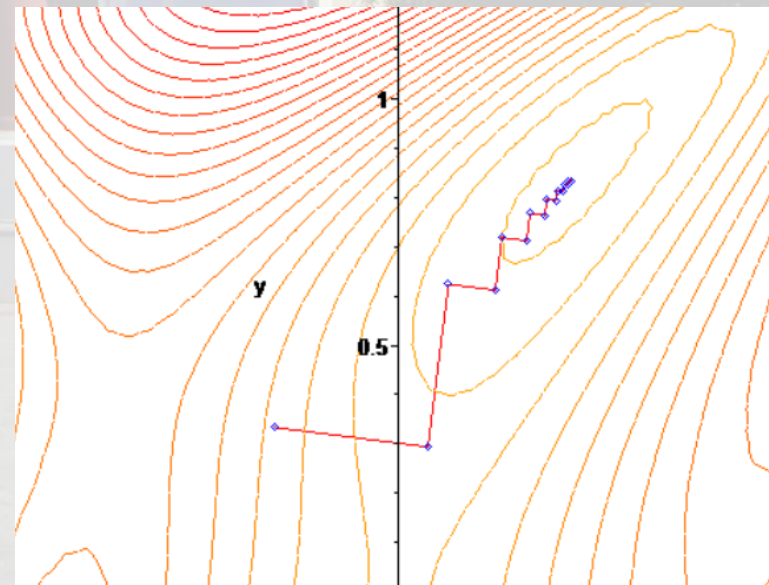
- CPU-intensive (must pass all dataset)
- large memory use, intractable if too large datasets
- does not allow updates on-the-fly (adding data online)

Also, it can become ineffective, as risk of getting stuck in local minima is large in multi-D

Most modern deep ML methods employ "stochastic" techniques to find the optimal working point / parameter values

This relies on the possibility to **decompose the loss function into the sum of per-example losses**.

SGD updates parameters on a per-event basis → objective function becomes noisy, but this has merits (can jump out of local minima)



# Mini-batch SGD

One may improve on per-event SGD updating by computing the gradient over small batches of training data

- get the best of both worlds:
- fast
  - no memory issues
  - scales well with data size
  - still can jump out of local minima
  - noise averages out a bit

Recipe:

$$w \leftarrow w - \eta \cdot \nabla_w L(w; x^{i:i+M}, y^{i:i+M})$$

# Advanced descent strategies

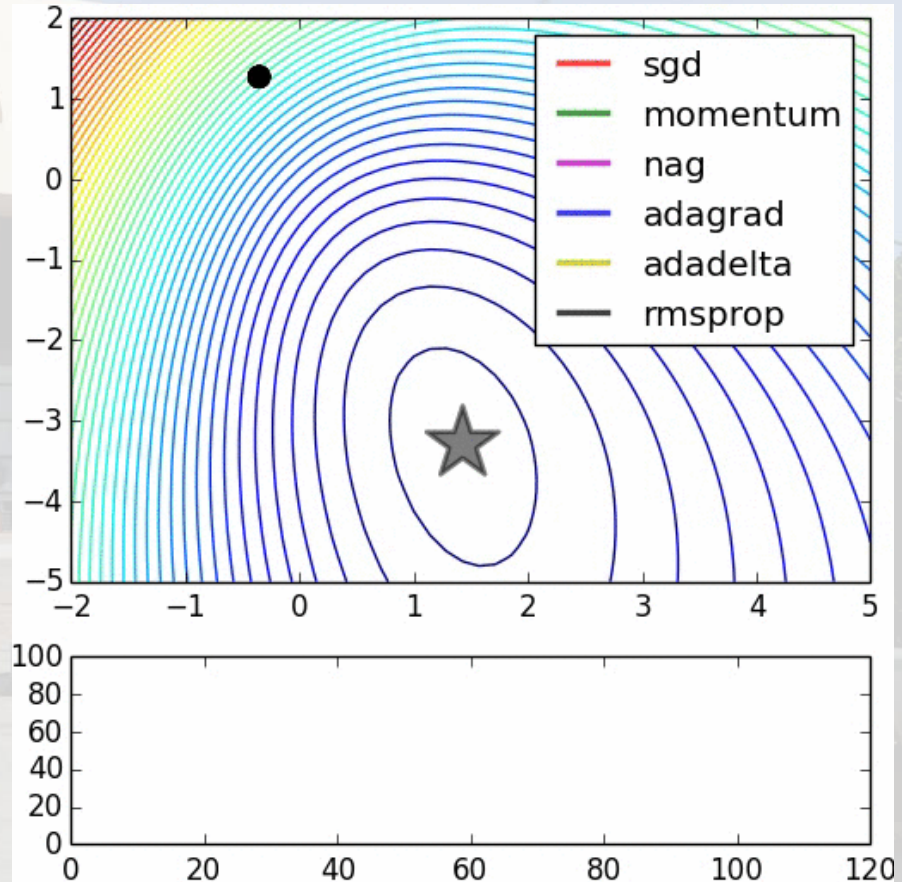
Finding the real, absolute minimum of a function with many parameters in a multi-D space can be *very* tricky, and take a lot of time

A number of variants of mini-batch SGD exist on the market:

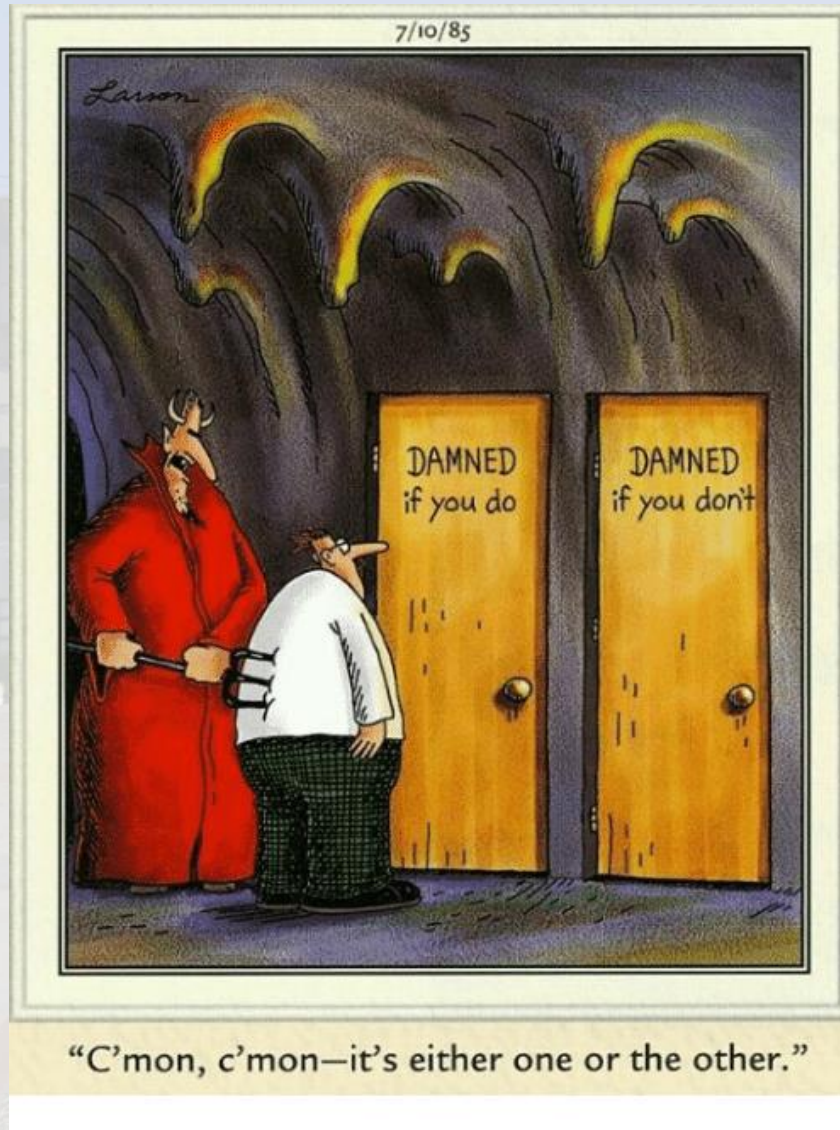
- momentum descent
- ADA gradient
- ADAM
- Nesterov accelerated gradient (NAG)
- and others

Their performance depends on the specific problem, the data size, sparsity, etcetera

Take-home bit: don't just pick the first off the shelf. Experiment with different methods, try to understand what is best for your case



# DECISION TREES



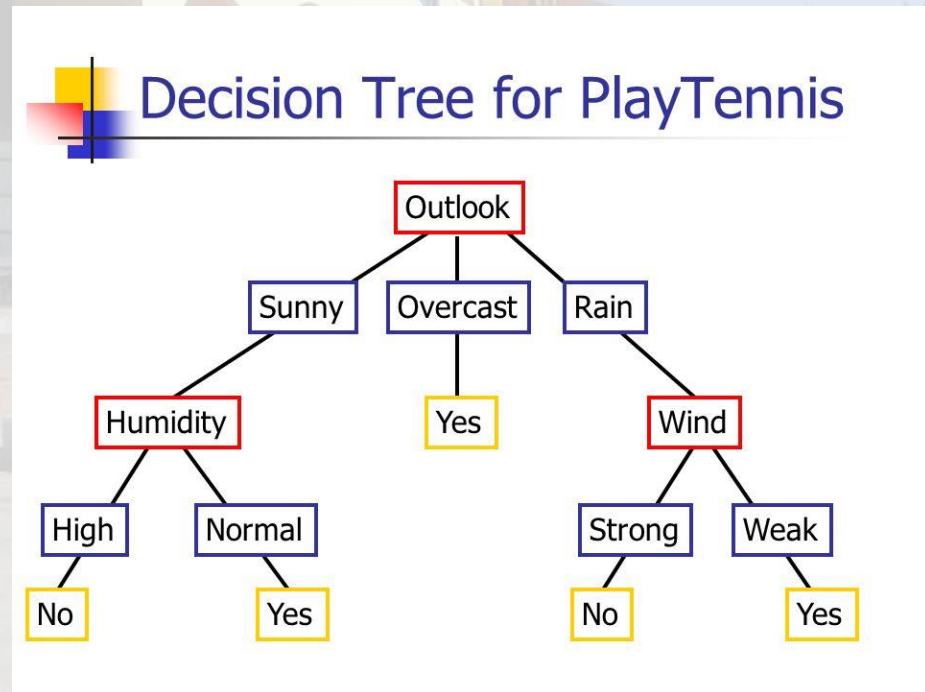
# Decision Trees

A "decision tree" is a tree constructed by "leaves" that are rules to split the data in the different classes, based on the data features.

If each event to be classified has variables  $x_1, x_2, x_3, x_4, \dots$ , I may create a tree by posing conditions on each variable, in a chain

A decision tree is not generally restricted to two possible decisions, but the most simple problems lend themselves to this form

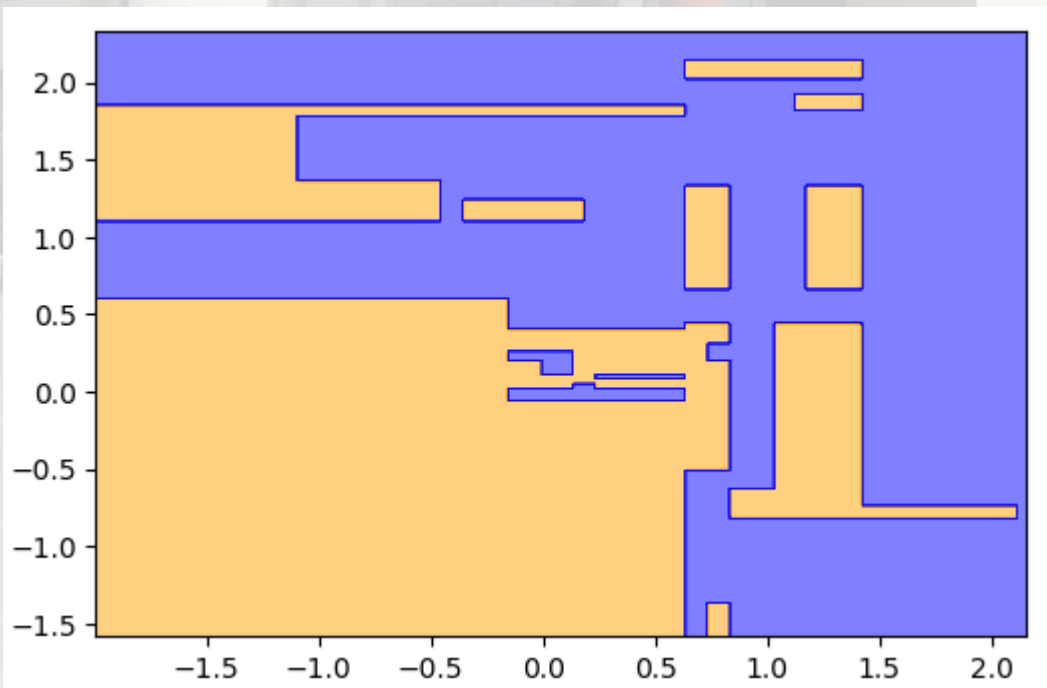
What the tree does for you is to **partition the multi-D space describing the possible states of an observation** (right: atmospheric weather used to decide whether to play Tennis) **in hypercubes** where the decision **optimizes some quantity of our interest** (in this case the satisfaction of playing)



# Two-Variable Example

The graph shows how the decision space of two variables could be partitioned by a decision tree with a large number of "splits".

The tree operates "linear cuts" (such as  $x < x^*$  or similar). Yet, due to the branching nature of the structure, very **complex decision boundaries may be created**



Due to their simplicity, and the absence of variable transformations, **DTs were among the first MVA algorithms adopted for HEP S/B discrimination problems.**



# Training a tree

Splits are obtained by conditions on a single feature ( $j$ ) of the data at a time,

$$x^{(j)} < > t^*$$

The probability of a class at each split is evaluated by the number of examples of that class in the partitioned set  $N_m$  from training data,

$$p(c_i) = N_{im}/N_m$$

We take the  $N_m$  events of the parent node  $m$  and for each split criterion  $\theta = (x^{(j)}, t_m^*)$  we get two subsets  $Q_{\text{left}}, Q_{\text{right}}$  of size  $n_{\text{left}}, n_{\text{right}}$ . We then compute the function

$$G(Q, \theta) = \frac{n_{\text{left}}}{N_m} H(Q_{\text{left}}(\theta)) + \frac{n_{\text{right}}}{N_m} H(Q_{\text{right}}(\theta))$$

$H()$  is an **impurity measure** (see below). We may now compute the **best split**:

$$\theta^* = \arg \min_{\theta} G(Q, \theta)$$

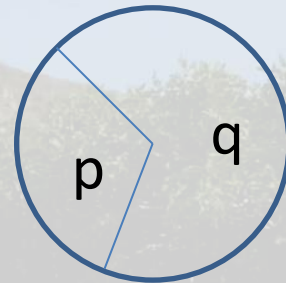
(elsewhere one equivalently finds this described as "maximizing the impurity gain", defining  $\Delta I = I_0 - I_L - I_R$ , and choosing  $\max \Delta I$ )

# Three measures of node impurity

To grow effective trees, we need **nodes to be as pure as possible**. On the other hand we also need nodes to not have too small probability, or we will be **overfitting the training data**.

The two criteria are conflicting for classes that overlap. So we need to first of all **define a measure of impurity** of each node.

It is common to define this as a symmetrical function  $I(t)=\phi(p,q)$ , when  $p, q$  are the relative frequencies of the two classes in node  $t$ , and with  $\phi(0,1) = \phi(1,0) = 0$ , and  $\phi(\frac{1}{2},\frac{1}{2}) = \frac{1}{2}$



If we just look at the classification error for one class as a measure of impurity, then we have

$$\phi(p,q) = 1 - \max(p,q).$$

This obliges the above definitions, but being linear it does not lend itself as an attractive way to optimize the tree construction.

A better (and non-linear) rule is the cross-entropy, defined as

$$\phi(p,q) = -p \log_2 p - q \log_2 q$$

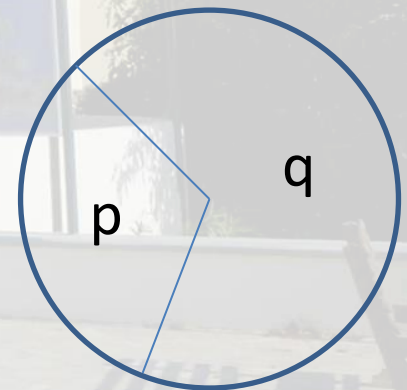
The third impurity measure is the Gini impurity index (see next slide).

# The Gini diversity index

The Gini index or Gini coefficient is a statistical measure of distribution developed by the Italian statistician Corrado Gini in 1912. It is very well known as a gauge of economic inequality, but it has much broader applications.

If a population includes two classes of elements, with relative frequencies  $p$  and  $q$  (i.e. defined so that  $p+q=1$ ), the Gini index is the symmetric function  $\phi(p,q) = 1-p^2-q^2$

As before,  $\phi(0,1) = \phi(1,0) = 0$  indicates that the population is pure (only contains elements of one type), while  $\phi(\frac{1}{2},\frac{1}{2})=\frac{1}{2}$  indicates maximum entropy.

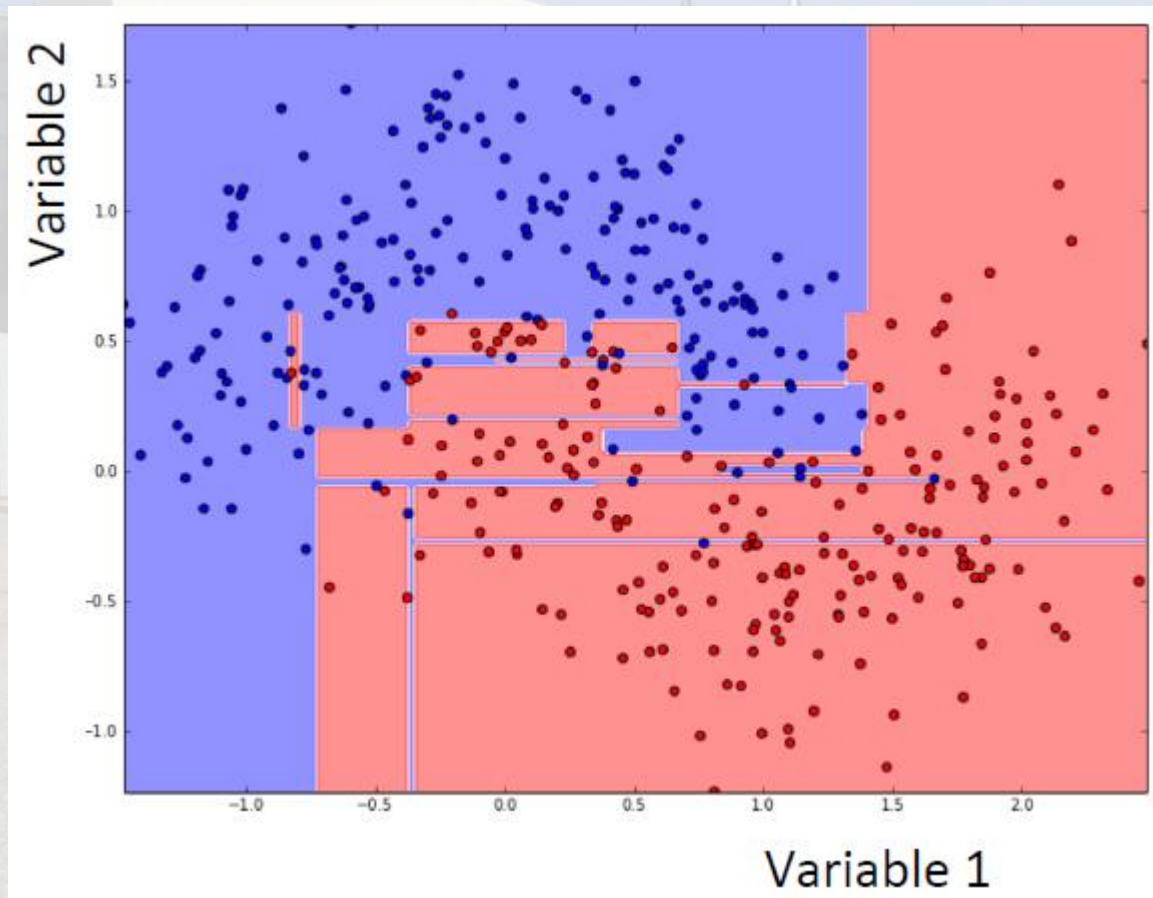


Clearly an equivalent definition is  $\phi(p,q) = 2pq$

# DT: the perfect classifier?

If you allow your tree to grow indefinitely, it will continue to split impure nodes, ending up classifying events in "pure" leaves

Of course this has huge variance, and in fact it is a blatant example of overfitting the training data



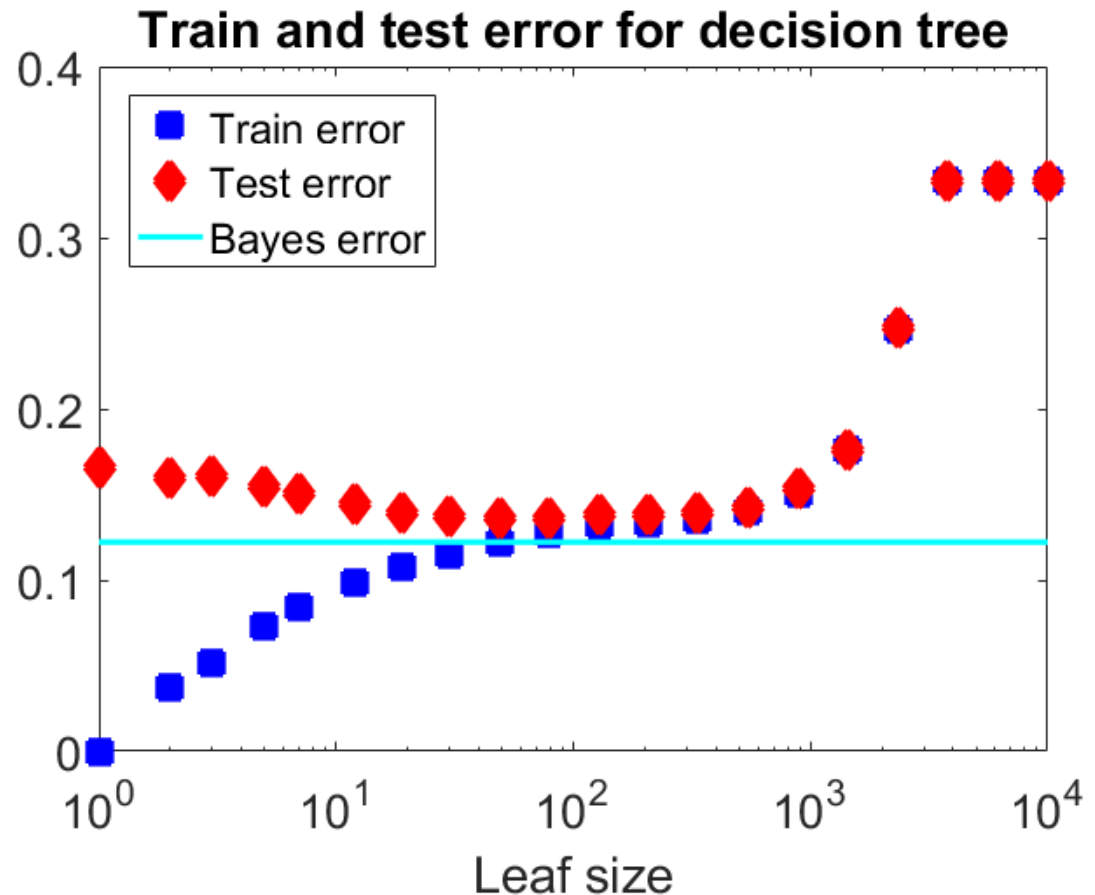
# When to stop?

As the leaf size decreases, leaves increasingly contain only events of one class

But this is true only for training data!

Validation is crucial to decide where to stop

The graph on the right, showing classification error versus leaf size, is quite typical of decision trees



# Pros and cons

- Decision trees are an **attractive choice** for applications where performance is not the culprit:
  - they are extremely simple to interpret (in low D!)
  - they demand no preprocessing (scaling, standardization) and handle categorical data easily
  - they work well regardless of number of dimensions of feature space
- But **DTs have shortcomings, too**:
  - overfitting is under no control → a careful validation is required
  - they are not stable WRT perturbations of the input data: change the training by a little, and the tree can grow very differently
  - The local optimization of tree splits does not guarantee reaching a global optimum (a NP complete problem)

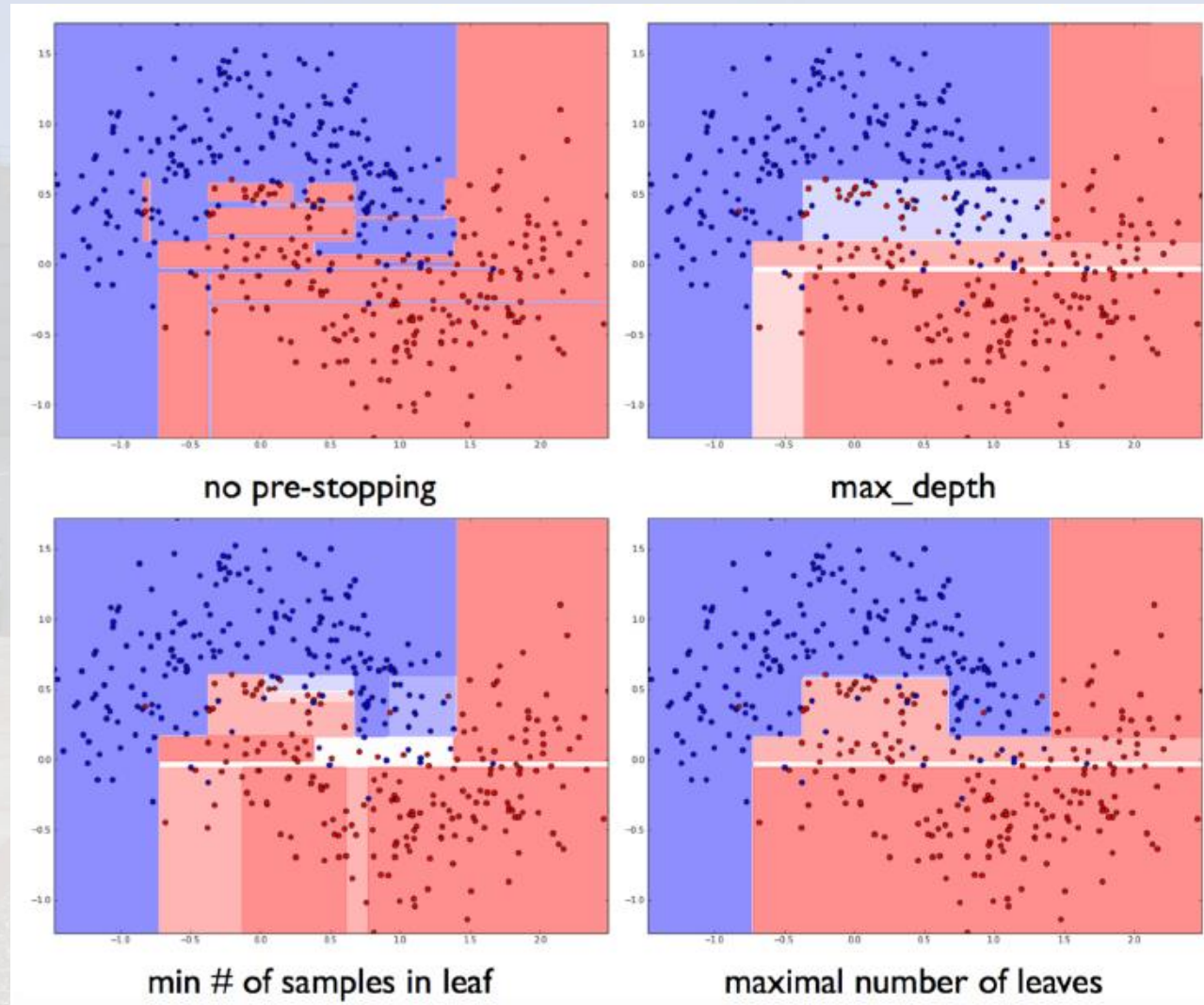
There are ways to overcome the above shortcomings. They rely on **early stopping**, and **ensemble methods**.

# Early stopping criteria

To mitigate the overfitting, one can adopt a fixed rule, such as:

- set a minimum number of events per leaf
- set a max number of leaves
- set a max tree depth

These recipes all work, although it is not easy to decide what is best for each case



# Pruning

Pruning means what it means: cut branches, replacing each with the node at their root. It is **a form of regularization**: we reduce the model complexity, hoping that the variance gain is not offset by a bias loss.

One may treat pruning by defining a risk function for each node  $t$  as the classification error of the node weighted by its probability,

$$r(t) = p(t)\varepsilon(t)$$

and for a tree as the sum over all leaves of the tree:

$$r(T) = \sum_{t \in L(t)} r(t)$$

When using training data,  $r(T)$  will tend to zero; one can however try to penalize the overtraining by adding to  $r(T)$  a term proportional to the number of leaves of the tree.



# Ensemble methods

Here we take the case of decision trees to illustrate the power of ensemble methods, which are however **applicable to any supervised learning tool**

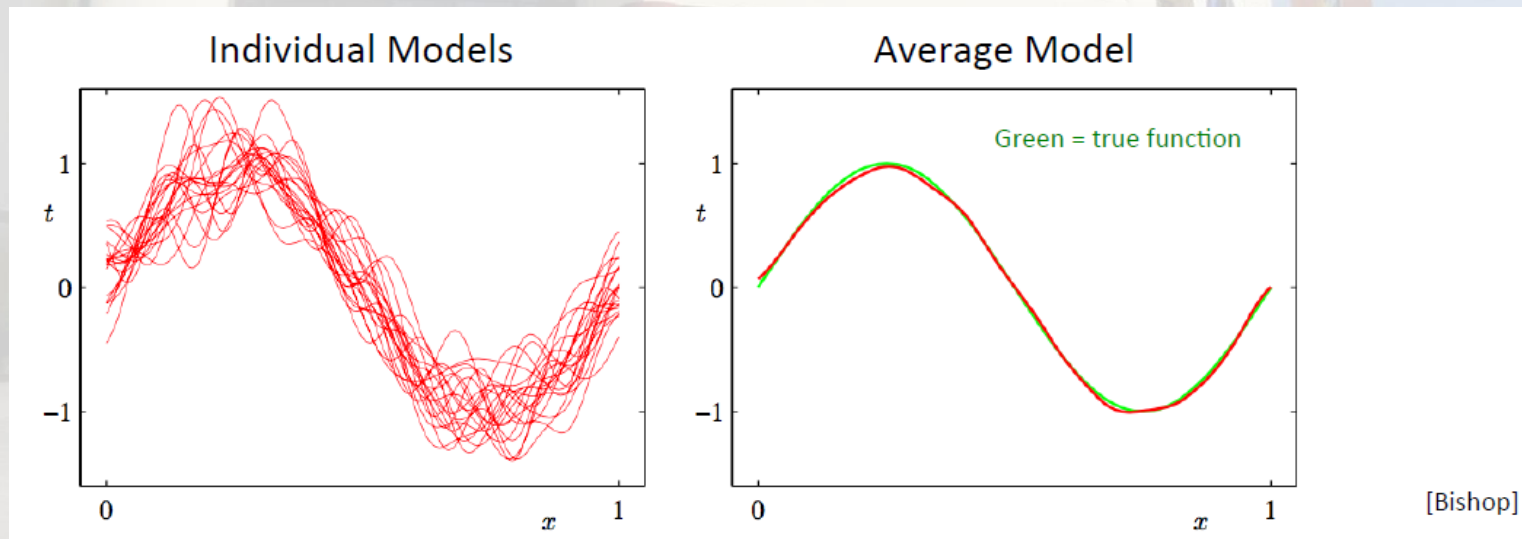
The idea is that we can **reduce the variance of a prediction without increasing the bias**, a "holy grail" in statistics. How can we do that?

By training slightly different models and **taking a majority vote** (for classification; for regression one would average the scores).

- The bias does not increase because the **result behaves similarly to any one of the inputs**: the average ensemble performance is equal to the average performance of its members.
- The **variance does decrease** because fluctuations and noisy predictions are averaged out: a spurious pattern picked up by one model will be damped in the pool

# Combining weak learners

A combination of the prediction of several weak learners (small correlation with target value) with high variance can be a powerful model!



For decision trees the benefit is also the control of overfitting, as the "perfect classification" issue of growing pure leaves is solved automatically

# Bagging and Boosting

**Bagging** ("Bootstrap aggregating") is a very effective ensemble technique employing bootstrap to leverage the replica of training sets

The training dataset is sampled with replacement, and every time a new classifier is trained with the resulting set

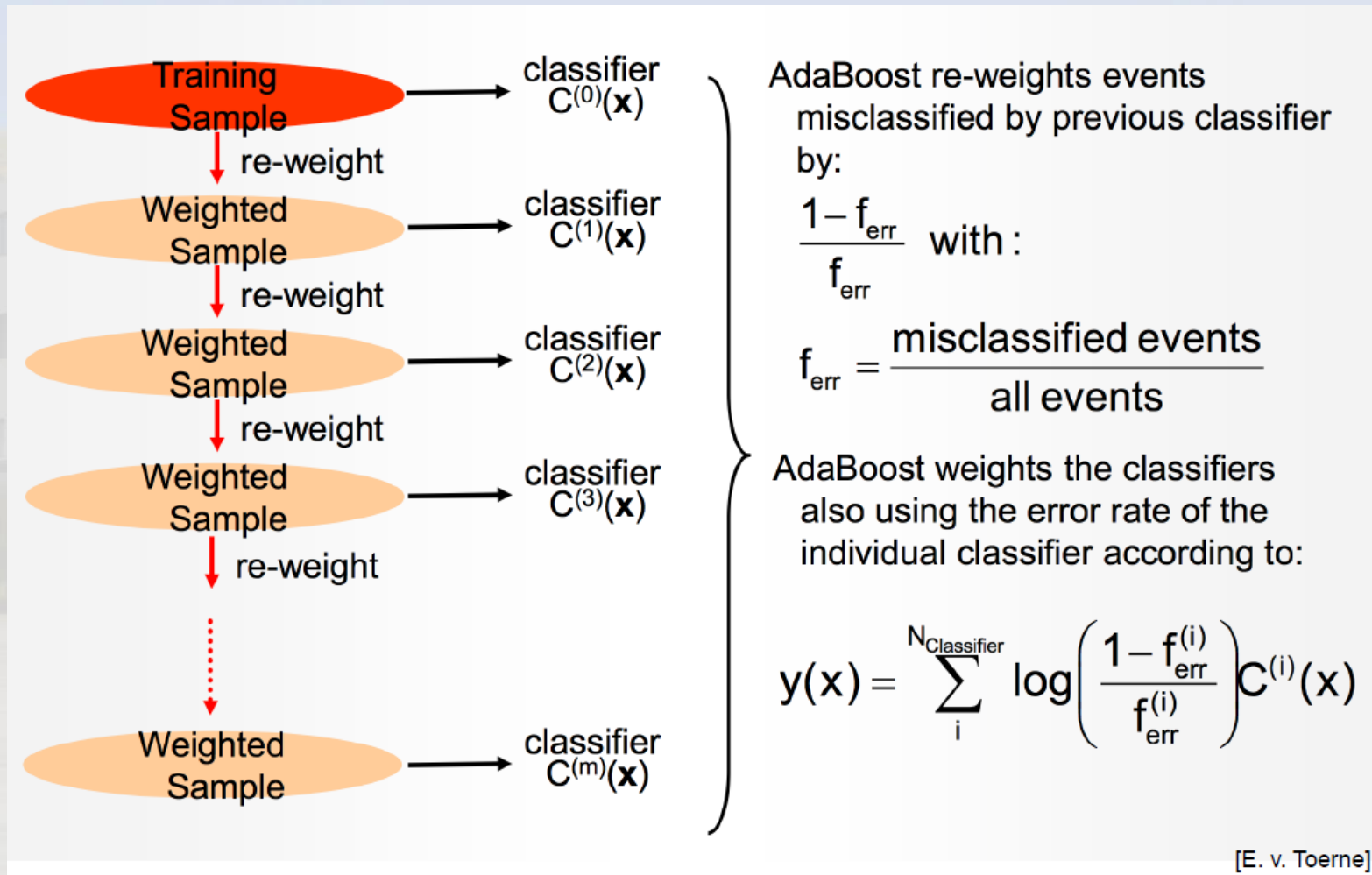
The prediction is obtained by a **majority vote** of the pool of classifiers, or by an average (in case of regression)

- The **Random Forest algorithm** uses bagging to stabilize the response (see below)

The idea of **Boosting** is instead to **train a sequence of models**, each of which gives more weight to events not classified correctly by the previous ones.

- At the heart of boosted decision trees

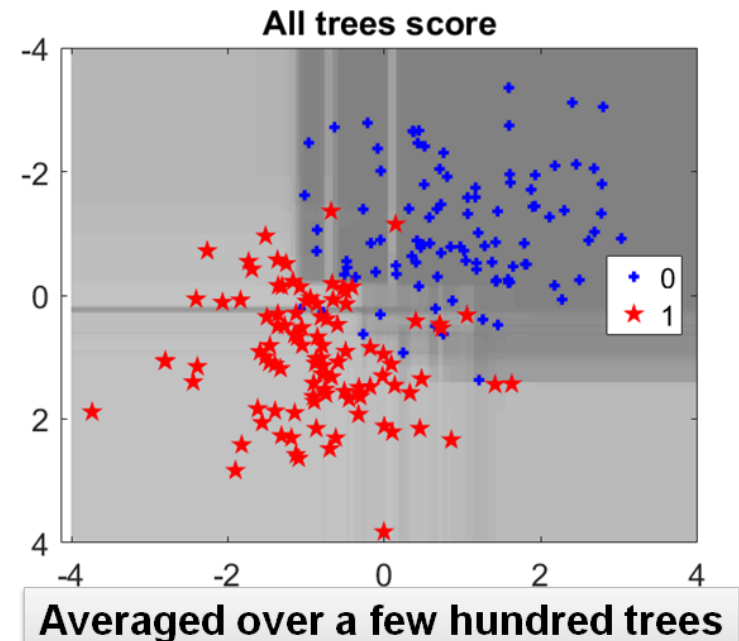
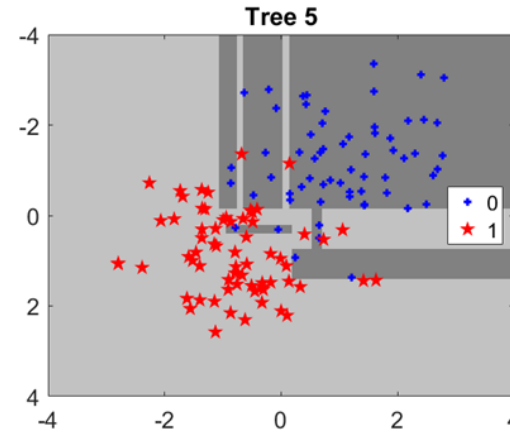
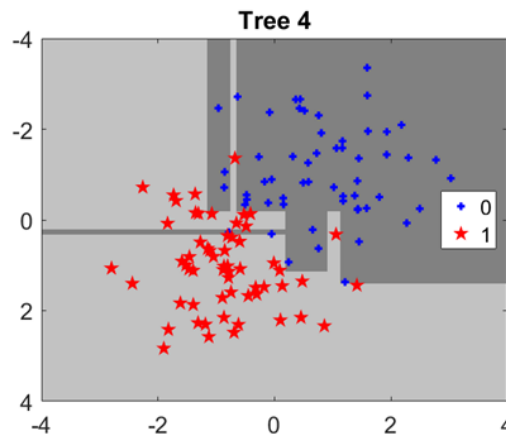
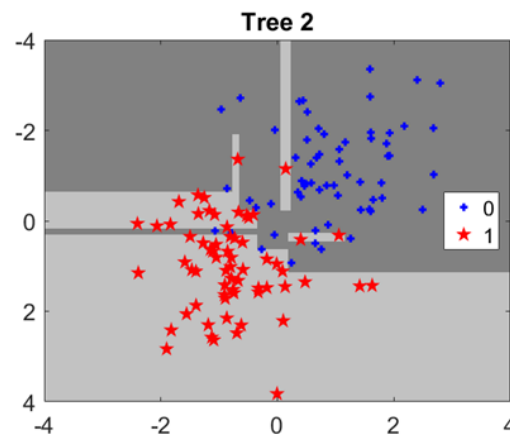
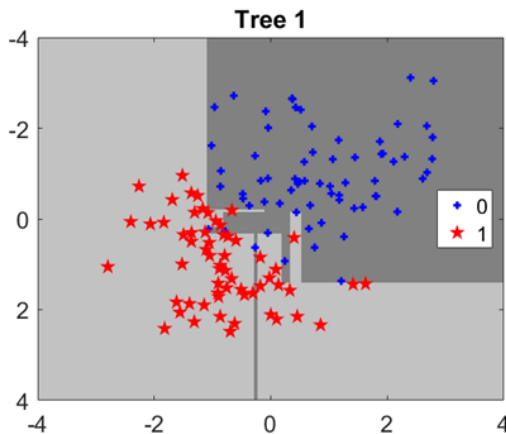
# The ADABOOST algorithm



Similar schemes are used by other algorithms (Gradient Boosting, XGBoost, ...)

# Example: averaging trees

Trees grown on bootstrapped training data learn different decision boundaries; the averaging does better than any of the inputs



# Random Forests

The Random Forest algorithm was first proposed by Breiman (2001), but is based on a 1995 idea of Tim Kan Ho.

RF employs two ensemble techniques. The first is **bagging of the training sample**, to grow a forest of different trees based on different training data. The second is the **subsampling of the feature space**.

If I **choose a subset of the variables (e.g.  $x_1, x_3, x_7$ )** to create a split in a node of a decision tree, and another subset ( $x_2, x_4, x_5, x_7$ ) to create a different one, there will be events that get classified in a different way by the two nodes.

Often there is a **dominant variables** that is used to decide the split, offsetting the power of the subdominant ones. RF avoids that problem by reducing the correlation of different trees.

RF grows trees where at each node a subset (typically of size  $D^{0.5}$ , where  $D$  is the dimensionality of the feature space) of the features is used to find the best split.

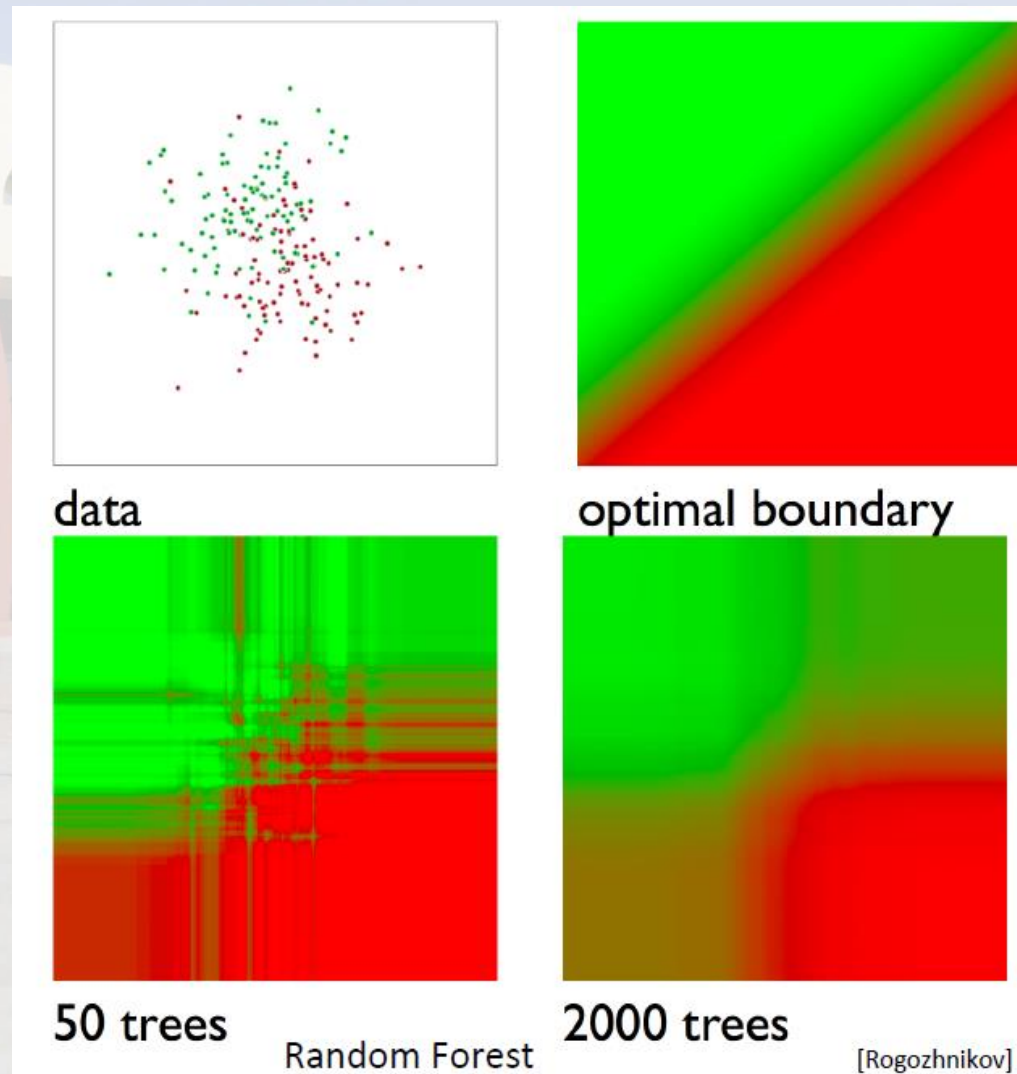
# Ensembles of trees: RF

Tree ensembles (like the Random Forest algorithm) have a number of attractive properties

- they usually **do not overfit**
- **they are powerful learners**

In addition they retain the **advantages of DTs:**

- they are simple to understand and interpret
- easy to train
- They work equally well with continuous as well as categorical data types
- no need to pre-process the data (e.g. invariant to standardization)

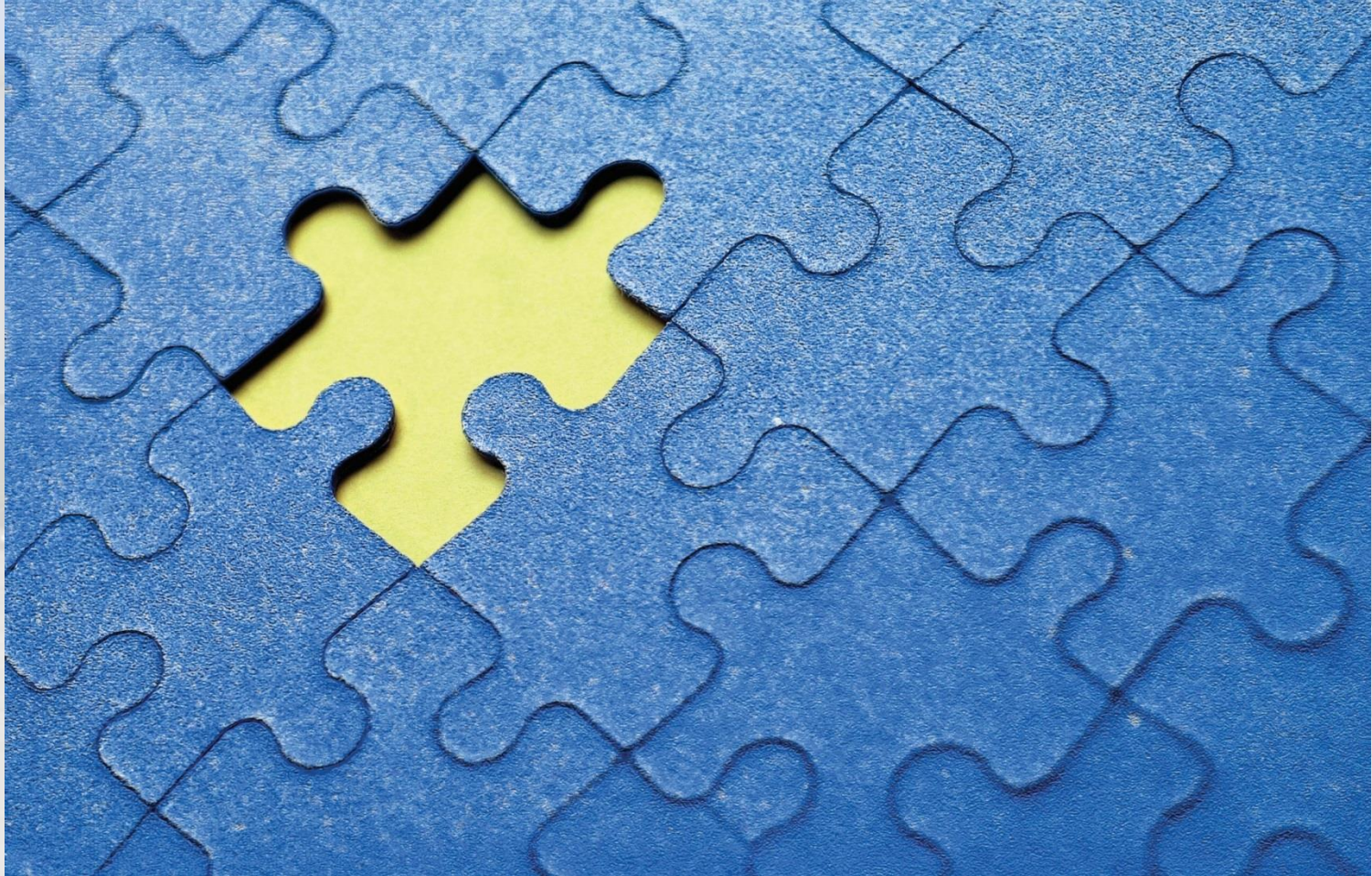


# Fun with Gradient Boosting

- See  
[http://arogozhnikov.github.io/2016/07/05/gradient\\_boosting\\_playground.html](http://arogozhnikov.github.io/2016/07/05/gradient_boosting_playground.html)



# LECTURE 2 CONCLUSIONS



# Conclusions for lecture 2

- Classification is a rich subject, and solutions depend on the specific needs of the problem
  - metrics for optimality vary a lot
- The loss function contains the recipe to give you the answer you want – pay attention to put it together (adding regularization where useful)
- Overfitting / overtraining must be avoided by careful testing; optimization handled with independent dataset. Apply cross validation when data is scarce,  $k=5-10$  typically good
- Random Forests / boosted trees are a very flexible, interpretable, powerful learner. Often hard to beat

# Lecture 3


## Neural networks



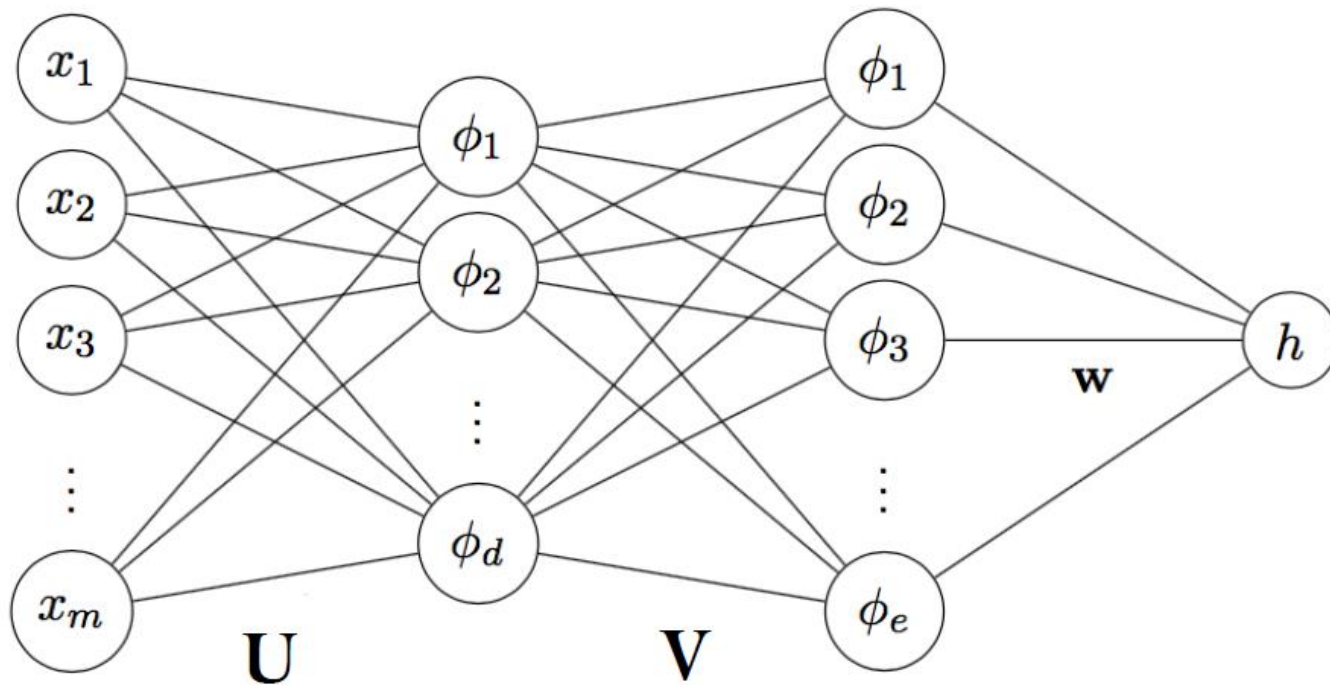
Higgs challenge  **the HiggsML challenge**  
May to September 2014

When **High Energy Physics** meets **Machine Learning**

info to participate and compete : <https://www.kaggle.com/c/higgs-boson>

# NEURAL NETWORKS



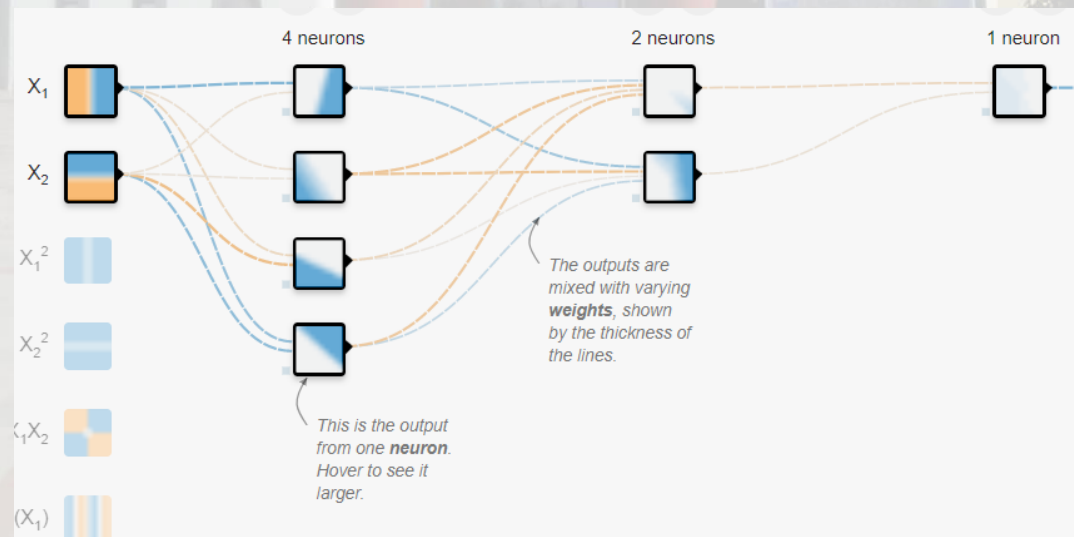
# Neural Networks - introduction

An artificial neural network (ANN) is a program that simulates the behaviour of a series of neurons and their connections

ANNs are **capable of producing very flexible functions of the feature space variables**

At the heart of the ANN there is an architecture of nodes organized in layers. Every "neuron" of a layer receives inputs from **some of** (or all) the neurons of the previous layer

Neural networks are extremely powerful tools for supervised learning tasks, such as classification and regression.



# Looking inside

Each neuron may emit a strong or weak signal, in response to the combined stimulus coming from its inputs → **activation functions** parametrize the response signal.

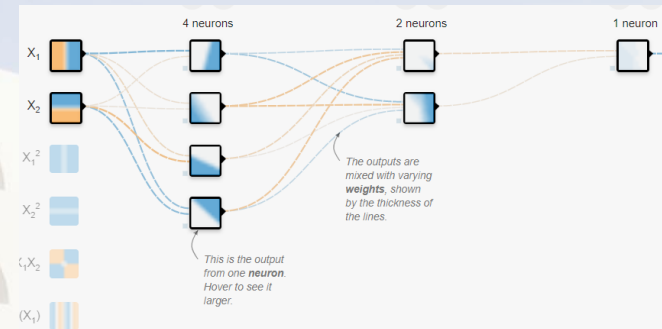
The signal is transmitted to the neurons connected to it in the next layer.

Mathematically, the behaviour of every neuron is described by two parameters (a bias and a weight). The training phase of the ANN (learning) consists in **finding parameter values** which **minimize a loss function**

For binary classification problems, the loss function may be simply the **fraction of misclassified events**. From that one can construct a **binary cross entropy**.

The crucial step in the minimization phase is "**back-propagation**".

Training events are used to compute the loss function. During back-propagation the contribution of each neuron (with associated weight and bias) to the loss is computed. This way **one may estimate how the loss would change if those parameters were changed**. The iteration of the procedure allows to obtain optimal values, with different convergence strategies possible (e.g. "**gradient descent**").



# The Perceptron

The perceptron is the simplest NN

The idea is to try to create a mathematical model of a single neuron, as a "node" which receives several inputs, sums them, and gets "activated" if the sum surpasses a fixed threshold

The perceptron task is to select between two classes based in the inputs it receives. The inputs are combined linearly:

$$u = w^T x + b$$

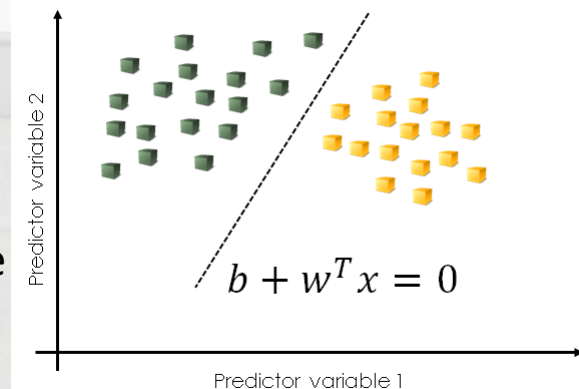
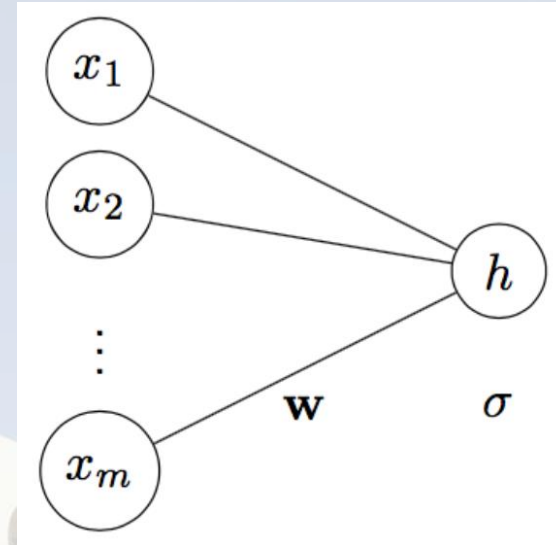
Above,  $x$  is a vector of inputs, and  $w$  is a "weight vector",  $b$  is a constant bias. The output is calculated as

$$o(u) = \begin{cases} +1 & \text{if } u \geq 0 \\ -1 & \text{if } u < 0 \end{cases}$$

The predicted class here depends on the sign of  $u$ . As  $w$  and  $b$  define a hyperplane in the feature space:

$$w^T x + b = 0,$$

by adjusting their values one can achieve ideal classification if the classes are linearly separable



# Learning $w$ and $b$

Suppose we have training data  $\{x_i\}$ ,  $i=1\dots N$  for the two classes, labeled as such:

$$\begin{aligned} y_i &= +1 && \text{for } i \text{ in } C_1 \\ y_i &= -1 && \text{for } i \text{ in } C_2 \end{aligned}$$

To simplify the math, we include  $b$  in the weight vector, adding a 0<sup>th</sup> component to  $x=[1, x_1, \dots, x_m]$  and  $w=\{b, w_1, \dots, w_m\}$ .

If for an event  $i$  we write  $u_i = w^T x_i$ , then (due to how we defined  $y$ )  $u_i y_i$  is  $>0$  ( $<0$ ) if the event is classified correctly (incorrectly). We can then write an error function, if we define  $\mathbf{M}(w)$  the set of misclassified events:

$$e(w) = - \sum_{i \in \mathbf{M}} w^T x_i y_i$$

We can **minimize this function to find the optimal weights**. This is done by iteratively stepping in the right direction:

$$w(k+1) = w(k) - \nabla e(w(k)) = \begin{cases} w(k) & \text{if } i \notin M \\ w(k) + x_i y_i & \text{if } i \in M \end{cases}$$

This way, the weights get adjusted to reduce the misclassification error.

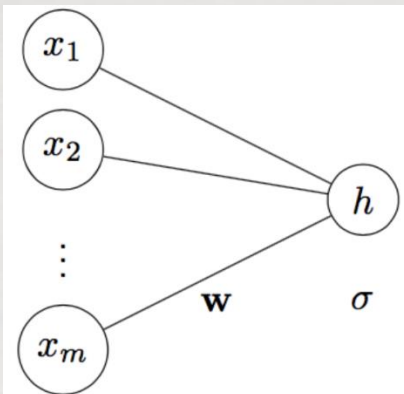
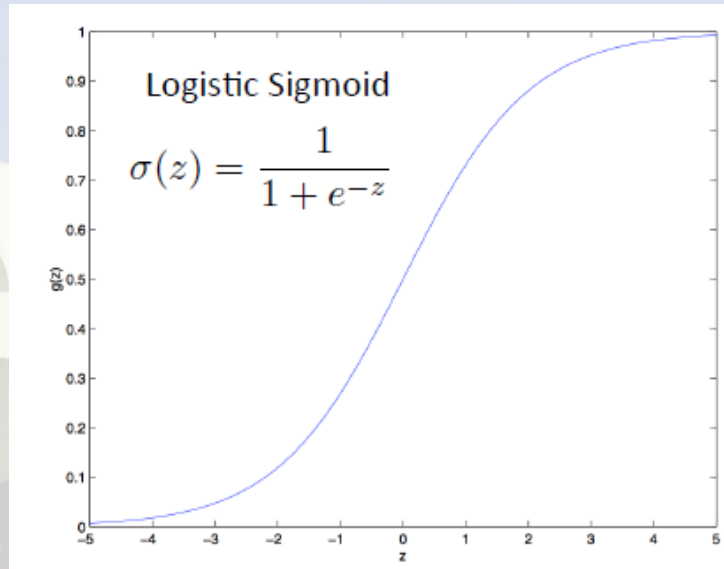


# The smooth output: logistic sigmoid

In neural networks, rather than a discontinuous score as in the perceptron, the response of a neuron is modeled by a continuous, differentiable function

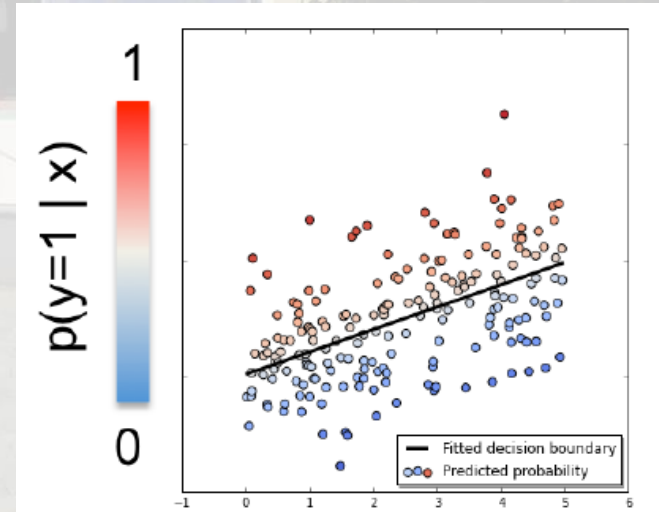
The simplest of these is the **sigmoid**.

In logistic regression, one assigns a **probability to events based on the distance from the boundary**, using the **logistic sigmoid** function:



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

$$p(y = 1 | \mathbf{x}) = \sigma(h(\mathbf{x}, \mathbf{w})) = \frac{1}{1 + e^{-\mathbf{w}^T \mathbf{x}}}$$



# Meaning of the sigmoid

If we express the posterior probability of  $x$  to be in class  $C_1$  as a sigmoid,

$$p(C_1|x) = \frac{p(x|C_1)p(C_1)}{p(x|C_1 \cup C_2)} = \sigma(u) = \frac{1}{1 + e^{-u}}$$

we can compute the inverse of  $\sigma$  as

$$u = \log \frac{\sigma}{1 - \sigma} = \log \frac{p(C_1|x)}{p(C_2|x)}$$

This is called **logit** function, and corresponds to the log of the relative posterior odds.

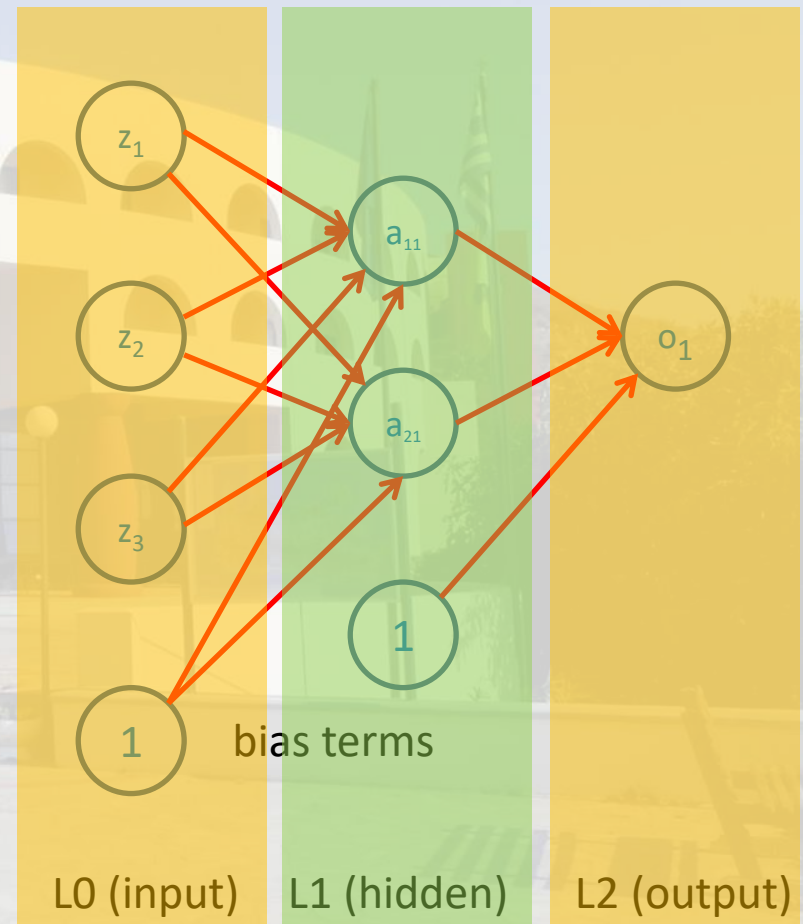
# The feed-forward neural network

We can put together these elements to create a **non-linear function of the inputs**, which can learn much more complex separation boundaries than a hyperplane

A feed-forward NN is composed of nodes connected by forward links. There is  $\geq 1$  hidden layer, and one output layer

- for binary classification, all you need is one node in it

The nodes need not all be connected, but **there must be no circular reference** – the value of a node must be a deterministic function of what comes before



# Calculation of the function

The FFNN is a complex, differentiable function of the inputs, and it offers a simple solution to the problem of optimizing its parameters.

For a formal treatment let us define:

- $z_i$  be the inputs to node  $i$  ( $i=1,\dots,Z$ , where  $Z$  is the number of inputs for that node;  $z_0=1$ ). For layer 0 (input layer),  $z_i=x_i$  is the vector of inputs.  
[When we need to specify it, we add an index  $j$  for the node and an index  $m$  for the layer, so  $z_{ijm}$  is the  $i^{\text{th}}$  input to the  $j^{\text{th}}$  node of the  $m^{\text{th}}$  layer]
- $w_{ijm}$  be the vector of weights for that node (with  $w_{0jm}=b_{jm}$  the bias term).

The rule is that at each node the inputs are summed with a linear weighting, to obtain the activation of that node,  $a$  ( $a_{jm}$  when needed):

$$a = \sum_{i=0}^Z w_i z_i$$

$i=1\dots Z$  inputs for a node  
 $j=1\dots J$  nodes of a layer  
 $m=1\dots M$  layers

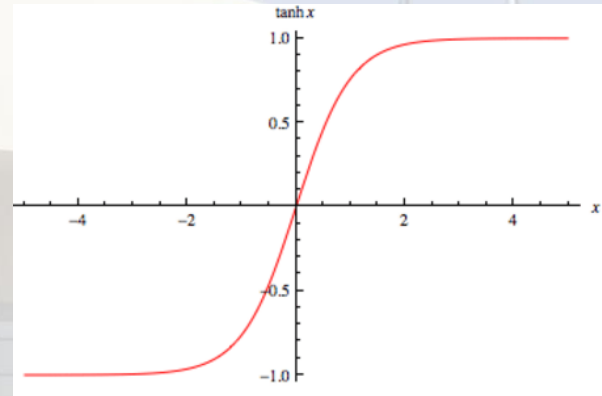
# Adding nonlinearity

Nonlinearity is introduced by choosing a suitable continuous differentiable function of  $a$  to model the behaviour of the node. A common choice for inner layers is

$$h(a) = \tanh(a)$$

whose derivative is

$$h'(a) = 1 - h(a)^2$$



For the output layer, as we saw, the **sigmoid** is the common choice for binary classification (one output). For  $K$  classes, one uses the **soft-max generalization**, which retains the interpretability of outputs as posterior probabilities:

$$o_k = \sigma(a_k) = \frac{e^{a_k}}{\sum_{i=1}^K e^{a_i}}$$

# Calculation/2

Let us consider a 3/2/1 architecture, and compute activations in the hidden layer as

$$\text{(for each node } j\text{)} \quad a_{j1} = \sum_{i=0}^3 w_{ij1} x_i$$

The output layer receives as inputs the sum of  $j=1, j=2$  activation functions

$$z_j = h(a_{j1}) = \tanh a_{j1}$$

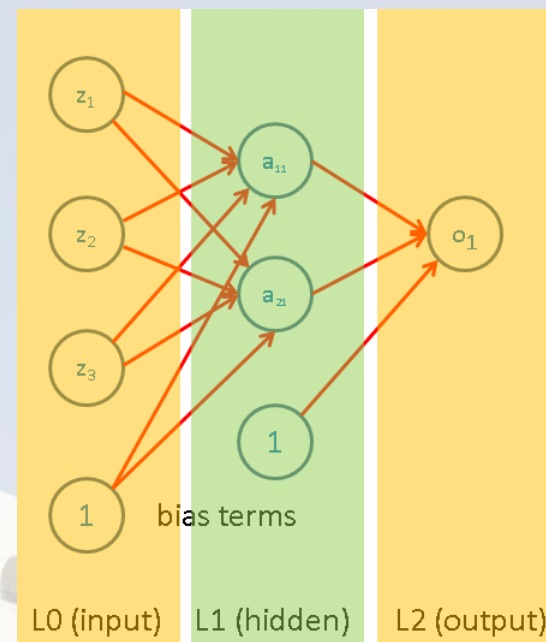
The activation in the output layer is then

$$a = \sum_{i=0}^2 w_{i12} z_i \quad \text{(node 1 of layer 2)}$$

and the output is the activation function (a sigmoid) of the above:

$$o = \sigma(a)$$

$$= \left\{ 1 + \exp \left[ -w_{012} - \sum_{j=1}^2 w_{j12} \tanh \left( w_{0j1} + \sum_{i=1}^3 w_{ij1} x_i \right) \right] \right\}^{-1}$$



# Backpropagation: where the magic happens

Consider the two-class case, with binary output  $y_{12}=1/0$  for signal and background, and the 3/2/1 2-layer network of the previous slides. If  $a(x,w)$  is the activation on the output node, the output is

$$o(x, w) = \frac{1}{1 + e^{-a(x,w)}}$$

This means that  $p(C_1 | x) = o(x,w)$ , and  $p(C_2 | x) = 1 - o(x,w)$  so we can write syntetically

$$p(y | x) = o(x,w)^y [1 - o(x,w)]^{1-y}$$

We have a differentiable model of the class probabilities, so we may write the  $-\log(L)$  for a training dataset  $\{x_{(1)} \dots x_{(N)}\}$  as

$$e(w) = - \sum_{n=1}^N [y_n \log o_n(x_n, w) + (1 - y_n) \log(1 - o_n(x_n, w))]$$

This is called (binary) cross entropy; it is the most commonly used loss for classification.

# Finding the weights

To find the optimal weights, we need to **minimize the BCE loss**. We are helped by noting that the loss is decomposable in per-event contributions  $e_n$ , which are a function of the weights. We need to only find the gradient of the error function with respect to each weight:

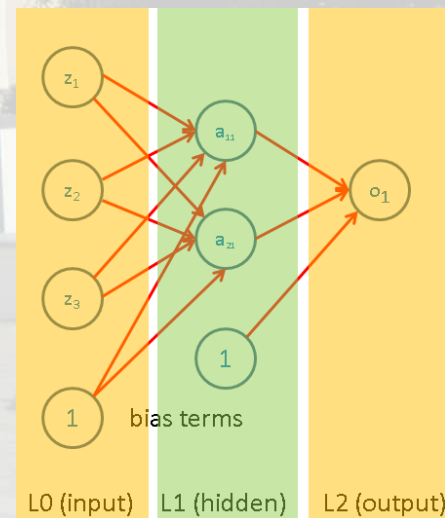
$$e_n(w) = -\{y_n \log o_n(x_n, w) + (1 - y_n) \log[1 - o_n(x_n, w)]\}$$

We take the derivative of  $e_n$  WRT the weight for the  $i^{\text{th}}$  input to node  $j$  in layer  $m$ ,  $w_{ijm}$ :

$$\frac{\partial e_n}{\partial w_{ijm}} = \frac{\partial e_n}{\partial a_{jm}} \frac{\partial a_{jm}}{\partial w_{ijm}} = e_{njm} z_{ijm}$$

where  $e_{njm} = \frac{\partial e_n}{\partial a_{jm}}$  and we have used the activation

$$a_{jm} = \sum_{i=0}^{n_{m-1}} w_{ijm} z_{ijm}$$





# Finding the weights/2

We work our way from the output layer M to the previous ones. While for the output layer the error caused by event n is  $e_{n1M} = o_n - y_n$ , for nodes at layer  $m=M-1$  we should write

$$e_{njm} = \frac{\partial e_n}{\partial a_{jm}} = \sum_{q=1}^{n_{m+1}} \frac{\partial e_n}{\partial a_{q,m+1}} \frac{\partial a_{q,m+1}}{\partial a_{jm}} = \sum_{q=1}^{n_{m+1}} e_{nq,m+1} \frac{\partial a_{q,m+1}}{\partial a_{jm}}$$

(At layer  $m+1=M$  there is only one node, but this formula works for any layer)

To evaluate derivatives we proceed thus:

$$\begin{aligned} \frac{\partial a_{q,m+1}}{\partial a_{jm}} &= \\ &= \sum_{i=1}^{n_m} w_{iq,m+1} \frac{\partial z_{iq,m+1}}{\partial a_{jm}} = \sum_{i=1}^{n_m} w_{iq,m+1} \frac{\partial h(a_{im})}{\partial a_{jm}} = w_{iq,m+1} h'(a_{jm}) \end{aligned}$$

Remember:

$$a_{jm} = \sum_{i=0}^{n_{m-1}} w_{ijm} z_{ijm}$$

Hence we find

$$e_{njm} = h'(a_{jm}) \sum_{q=1}^{n_{m+1}} w_{qj,m+1} e_{nq,m+1}$$

# Finding the weights/3

The formulas of the previous slide allow us to work our way back recursively through the NN, using the chain rule.

For tanh() activation in inner layers the error contribution of event n due to weights in node j of layer m is written

$$e_{njm} = [1 - h(a_{jm})^2] \sum_{q=1}^{n_{m+1}} w_{qj,m+1} e_{nq,m+1}$$

The weight updating process is iterative, and modulated by a learning rate  $\eta$ :

$$w^{(k+1)} = w^k - \eta \nabla e(w^k)$$

One may choose to use the whole training set to evaluate the gradient (batch learning), or to update weights at each new event evaluation (online learning). They have different applications and properties.

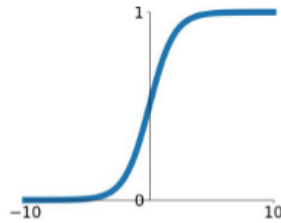
Online learning is better fit to jump out of minima, but the learning may take longer.

# Choices of Activation function

We want it non-linear, otherwise hidden layers do nothing; it must be monotonic to ensure convergence of the optimization problem; and smooth. Often also preferable to have rapidly changing for input close to zero, slowly changing for large input

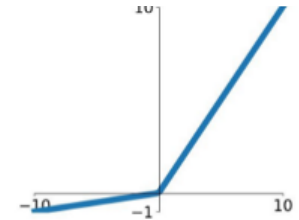
## Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



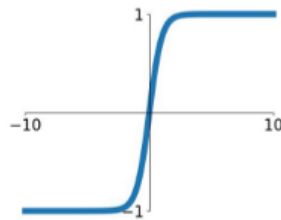
## Leaky ReLU

$$\max(0.1x, x)$$



## tanh

$$\tanh(x)$$

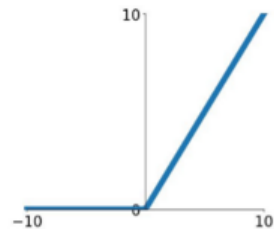


## Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

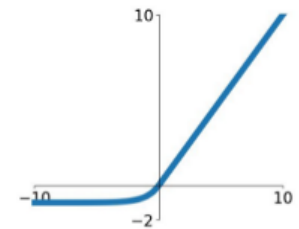
## ReLU

$$\max(0, x)$$



## ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



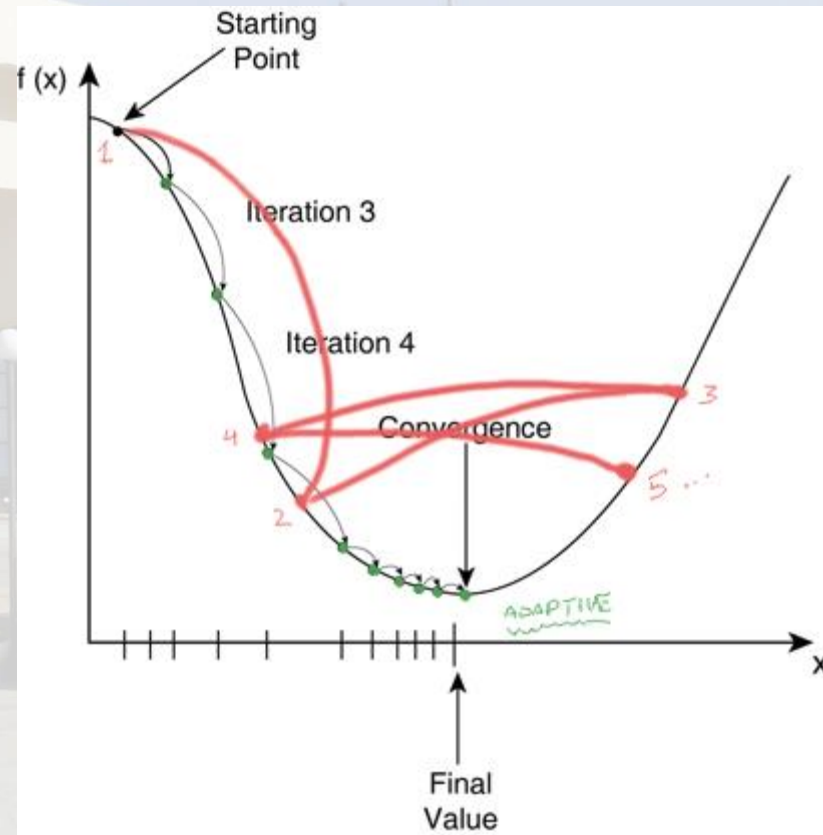
# Learning rate

Above we mentioned the learning rate – the parameter  $\eta$  controlling how fast the parameters of the learner are updated

In a NN the weights, on which depend the strength of the response of an activated node, are adjoined by back-propagation

**For NNs  $\eta$  is one of the crucial parameters in the search of optimality**

Advanced techniques have been devised to overcome the difficulty. These include slowly decreasing  $\eta$ , scheduled modulations in  $\eta$ , momentum, etcetera.



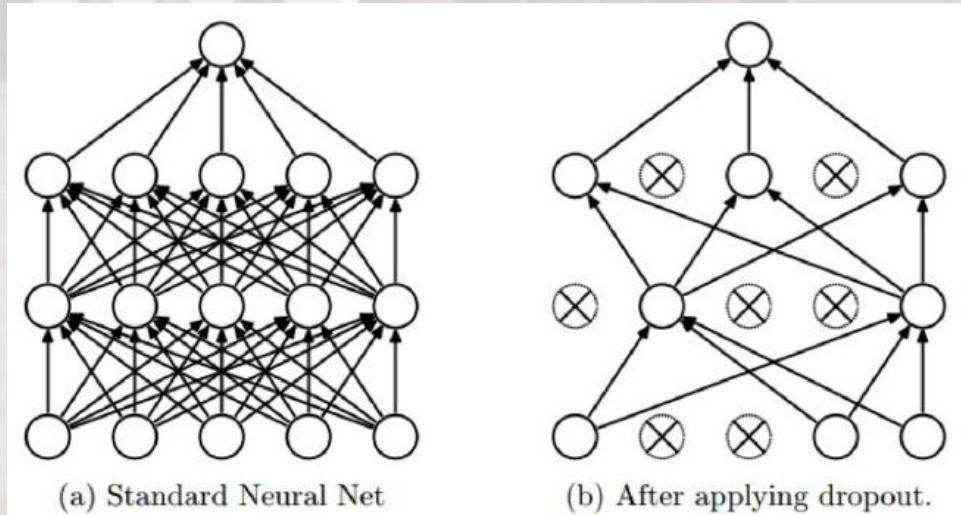
# Regularization

We have already encountered the concept in general. In ANNs, regularization is similarly applied by **adding to the loss a penalty term**

- L1 loss:
- L2 loss: (AKA "weight decay")

A different method is called "dropout": during training, a random set of nodes is removed at each pass.

- prevents collective effects conditioning the training



# Playing with NNs



# Let us play a little

A wonderful web page allows us to fiddle with the architecture and hyperparameters of a NN, training it to solve simple 2D problems.

<https://playground.tensorflow.org/>

Tinker With a **Neural Network** Right Here in Your Browser.  
Don't Worry, You Can't Break It. We Promise.

Epoch: 000,000 | Learning rate: 0.03 | Activation: Tanh | Regularization: None | Regularization rate: 0 | Problem type: Classification

DATA: Which dataset do you want to use?

FEATURES: Which properties do you want to feed in?

2 HIDDEN LAYERS

4 neurons | 2 neurons

OUTPUT: Test loss 0.569 | Training loss 0.548

Ratio of training to test data: 50%

Noise: 0

Batch size: 10

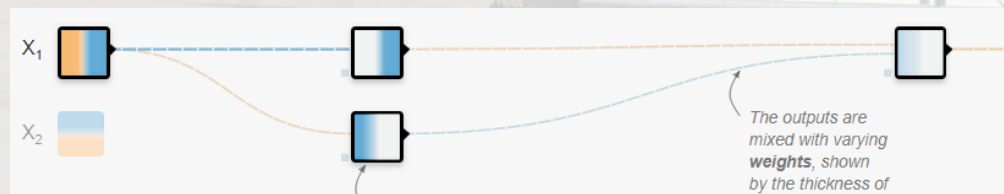
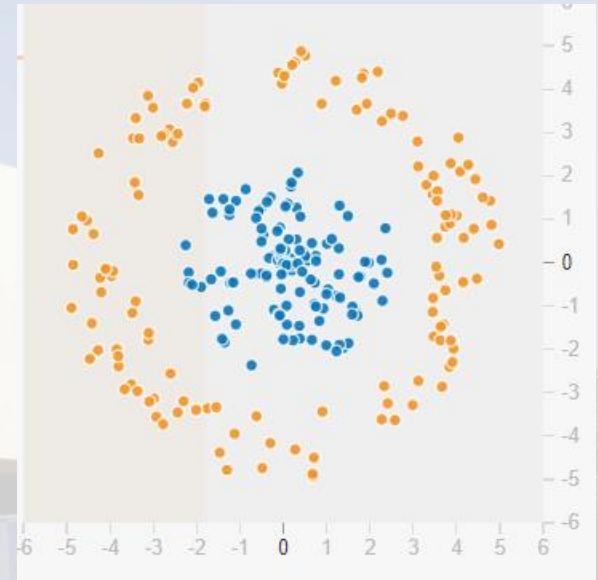
*The outputs are mixed with varying weights, shown by the thickness of the lines.*

*This is the output from one neuron. Hover to see it.*

# Exercises with simple NNs

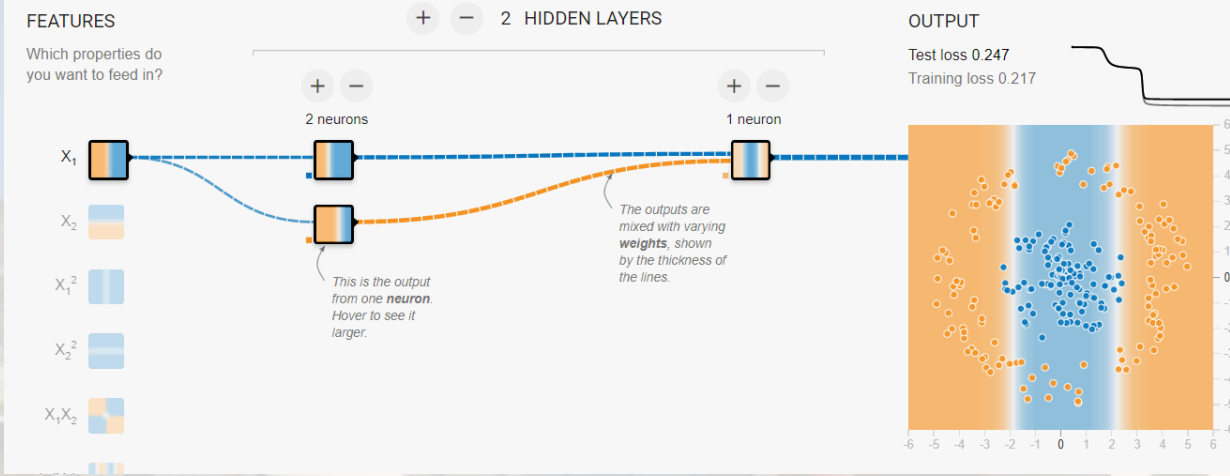
1) We take the first proposed dataset in the web site

What will be able to do if we use only one input (e.g. the first one from top) ?



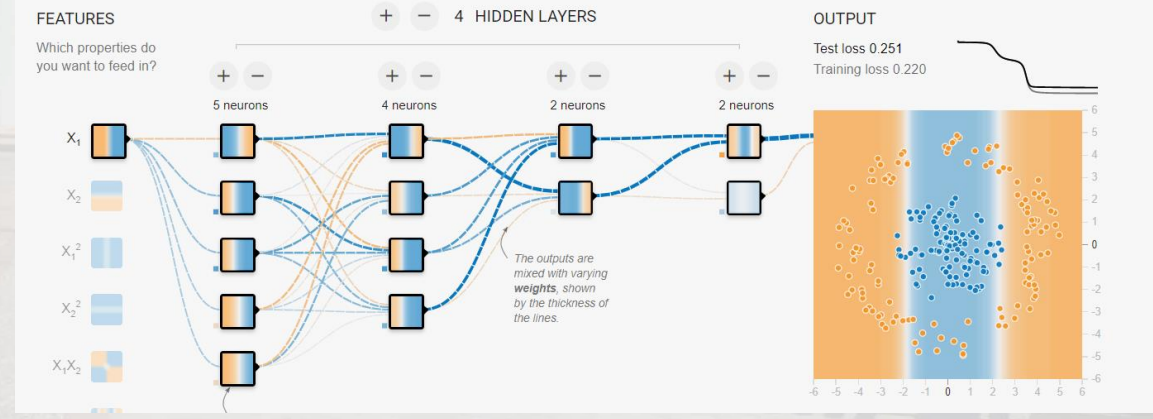


Epoch 000,541    Learning rate 0.03    Activation Tanh    Regularization None    Regularization rate 0    Problem type Classification



Left: the simplest possible NN quickly converges to the optimal result

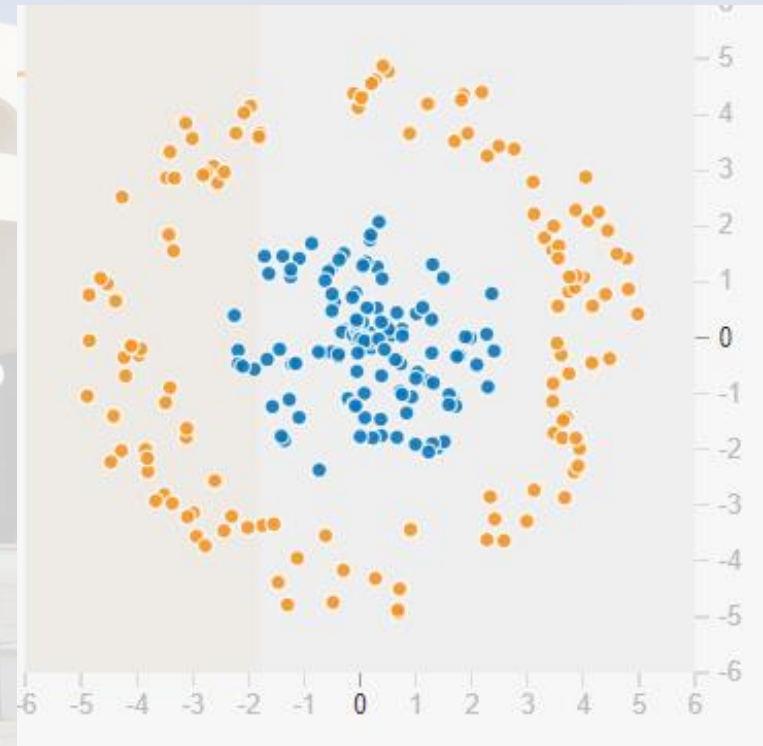
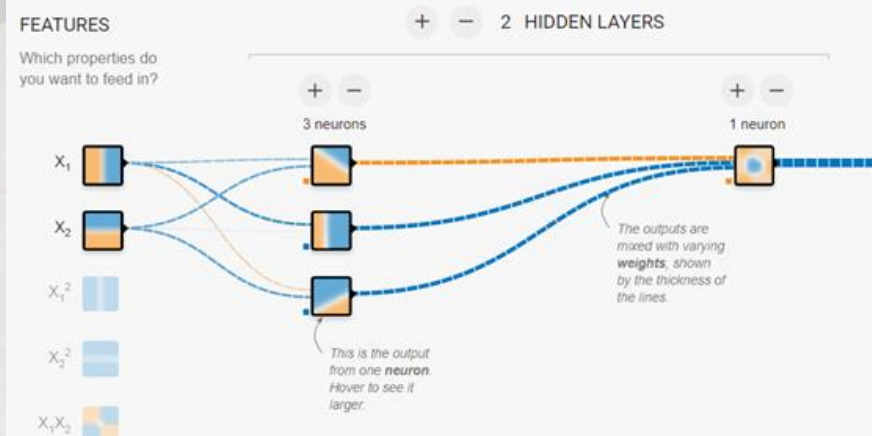
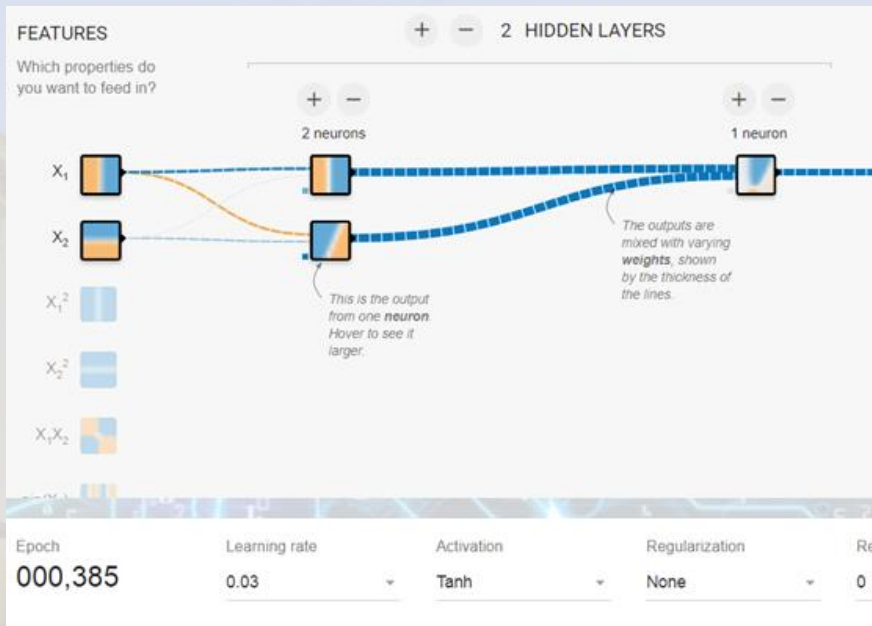
Epoch 000,238    Learning rate 0.03    Activation Tanh    Regularization None    Regularization rate 0    Problem type Classification



Right: a much more complex network does not perform any better, but takes much longer and requires tuning

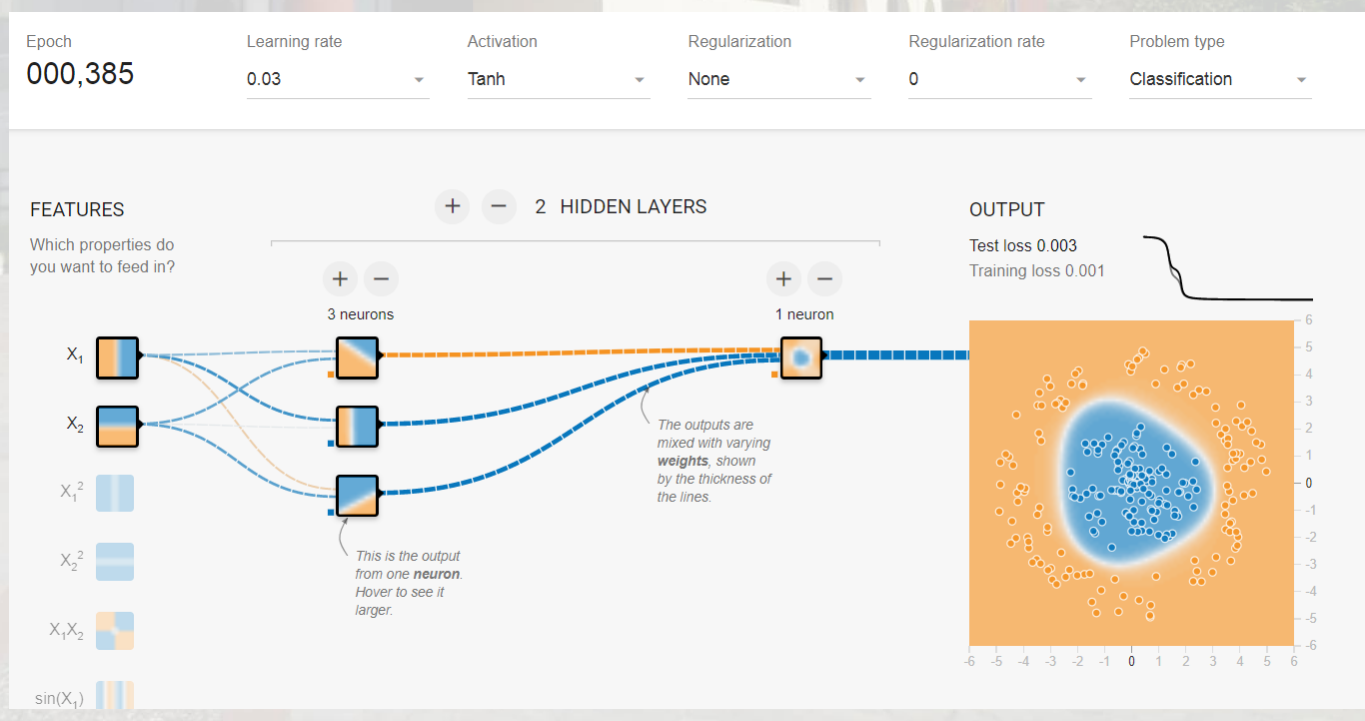
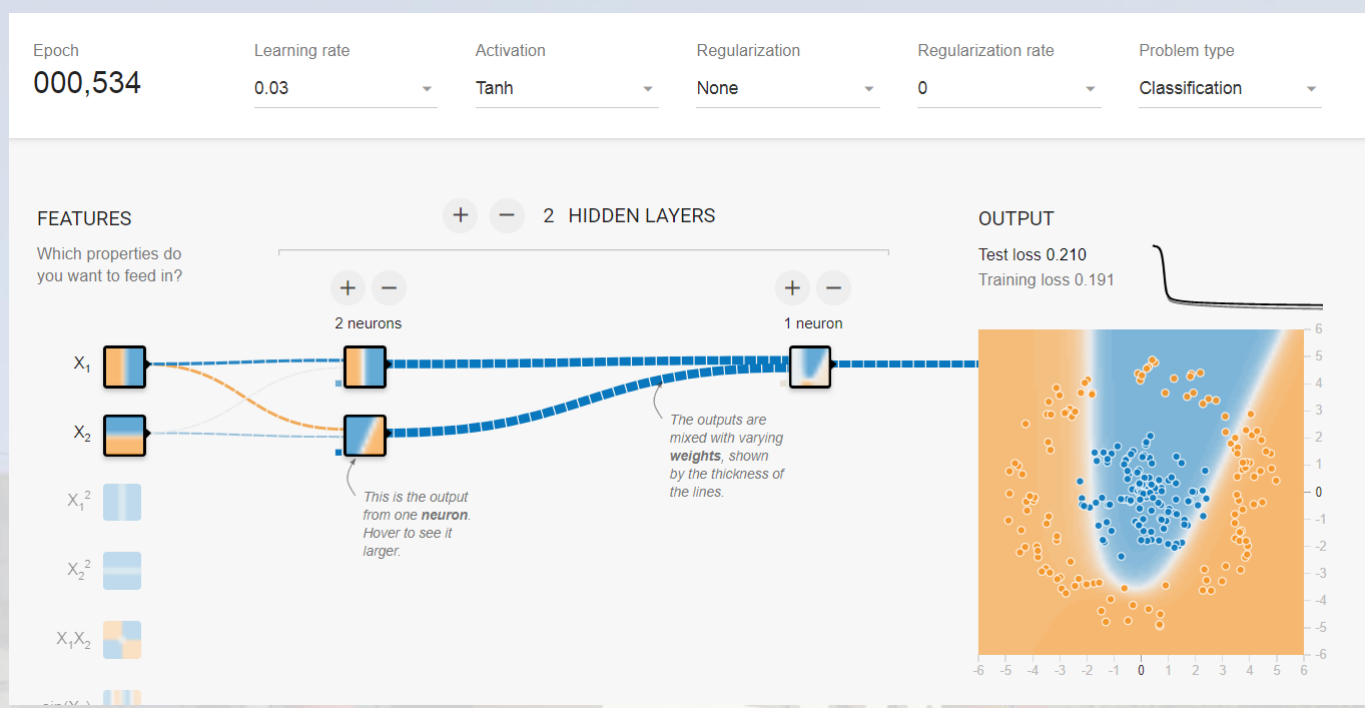
→ Answer: you can't do better than about 0.25, independently of the chosen architecture / hyperparameters

## 2) What if we use both the x and y inputs? **What changes** when we go from a 2-nodes hidden layer to one with more ?



Top: two nodes in the hidden layer. The test loss is 0.21

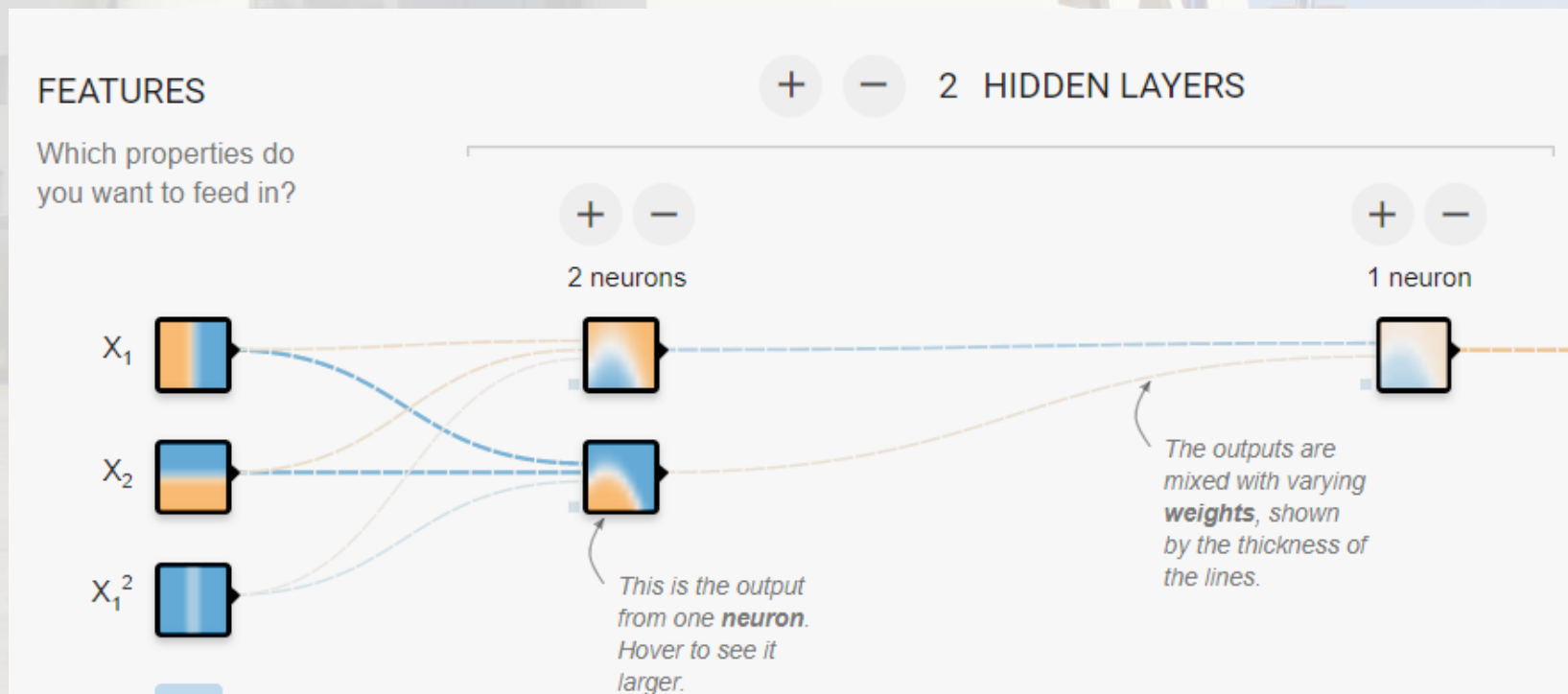
Bottom: three nodes vastly outperform the 2-node NN; the classification becomes perfect in this case



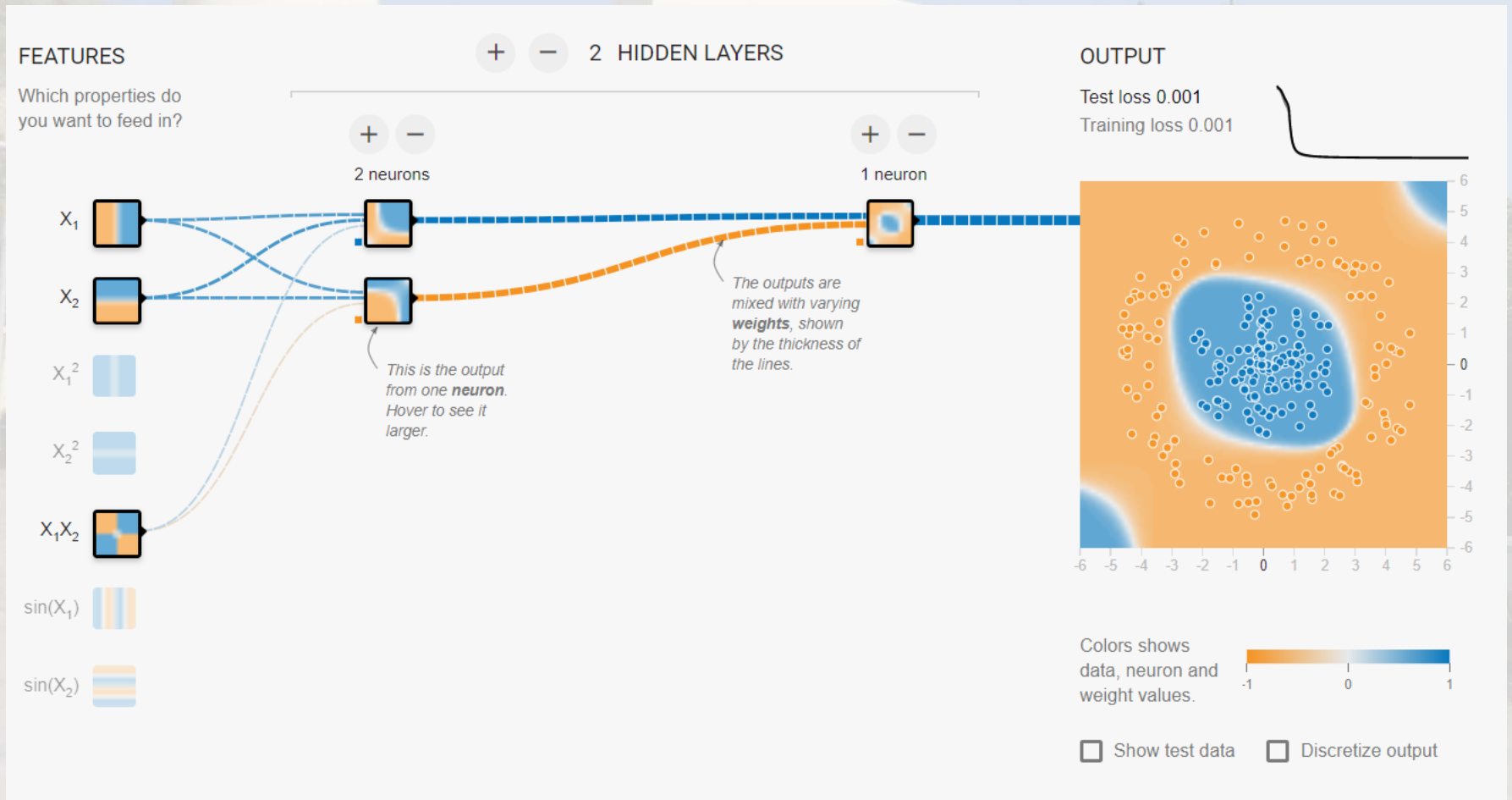
2b) What happens if we add more inputs, but keep just two nodes in the hidden layer?

→ A 3/2/1 architecture, and let's use a tanh activation, with 0.03 learning rate.

**What test loss can you get ? What inputs did you pick?**

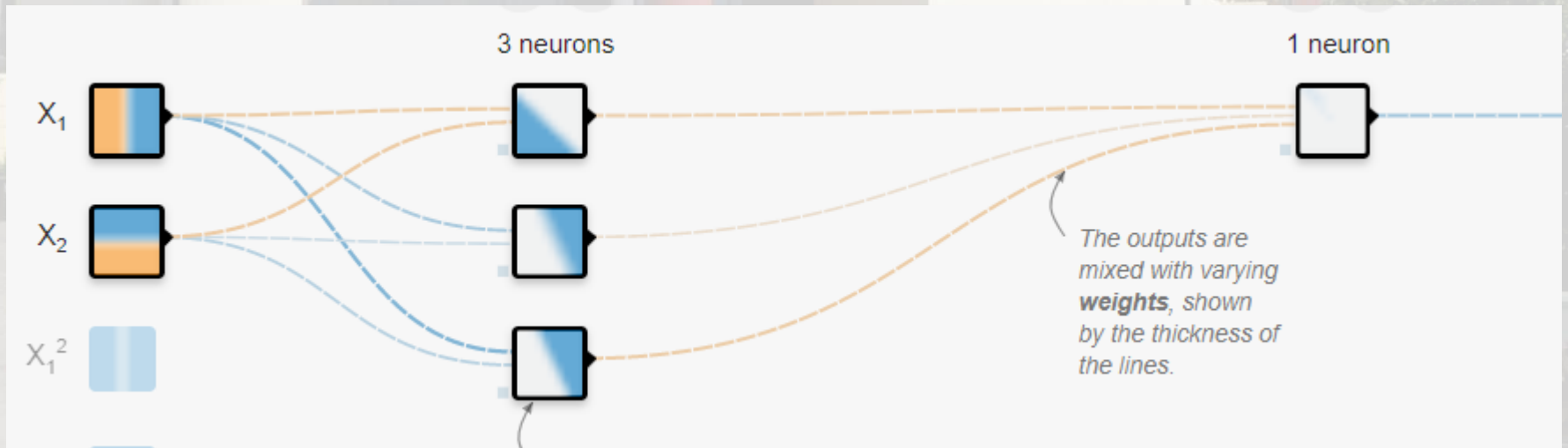


# With 3 inputs the classification becomes perfect even with only two nodes in the hidden layer. Why?



3) Now let's try using two inputs and three nodes in the HL, but change from tanh to relu the activation function.

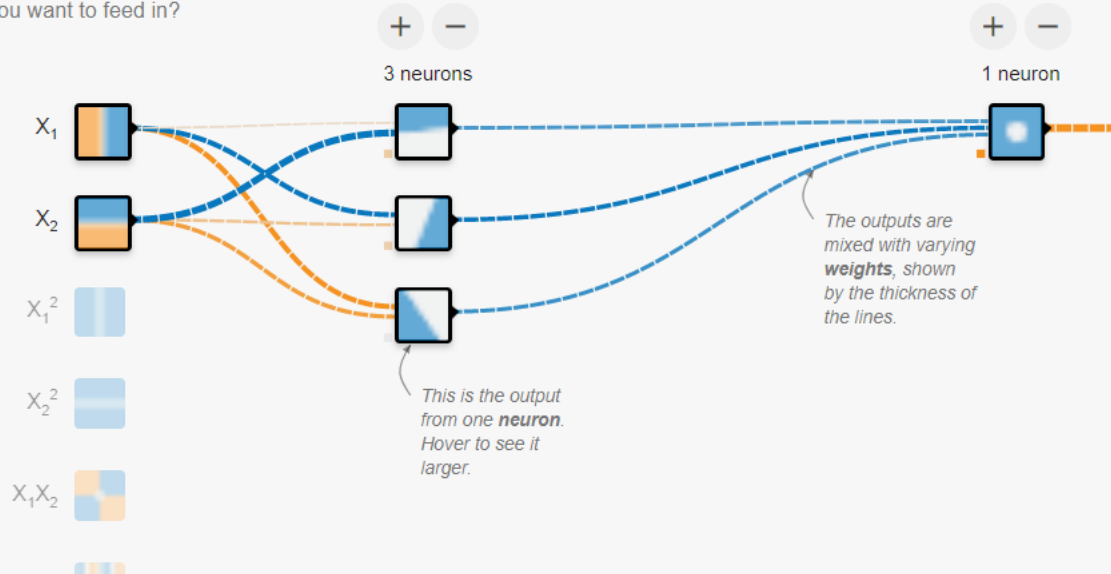
What do we expect will change?



Epoch **000,506**    Learning rate **0.03**    Activation **ReLU**    Regularization **None**    Regularization rate **0**    Problem type **Classification**

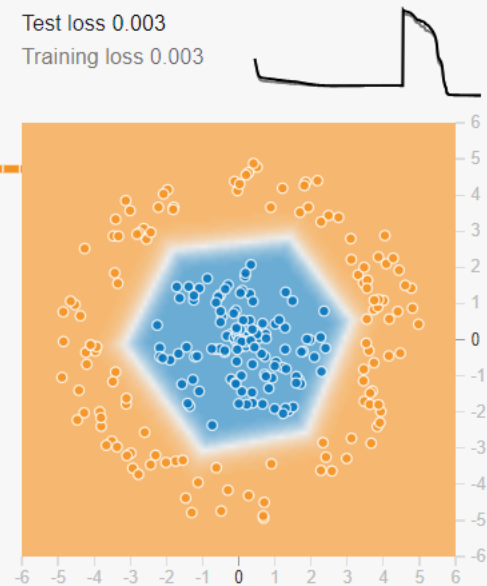
### FEATURES

Which properties do you want to feed in?



### OUTPUT

Test loss 0.003  
Training loss 0.003

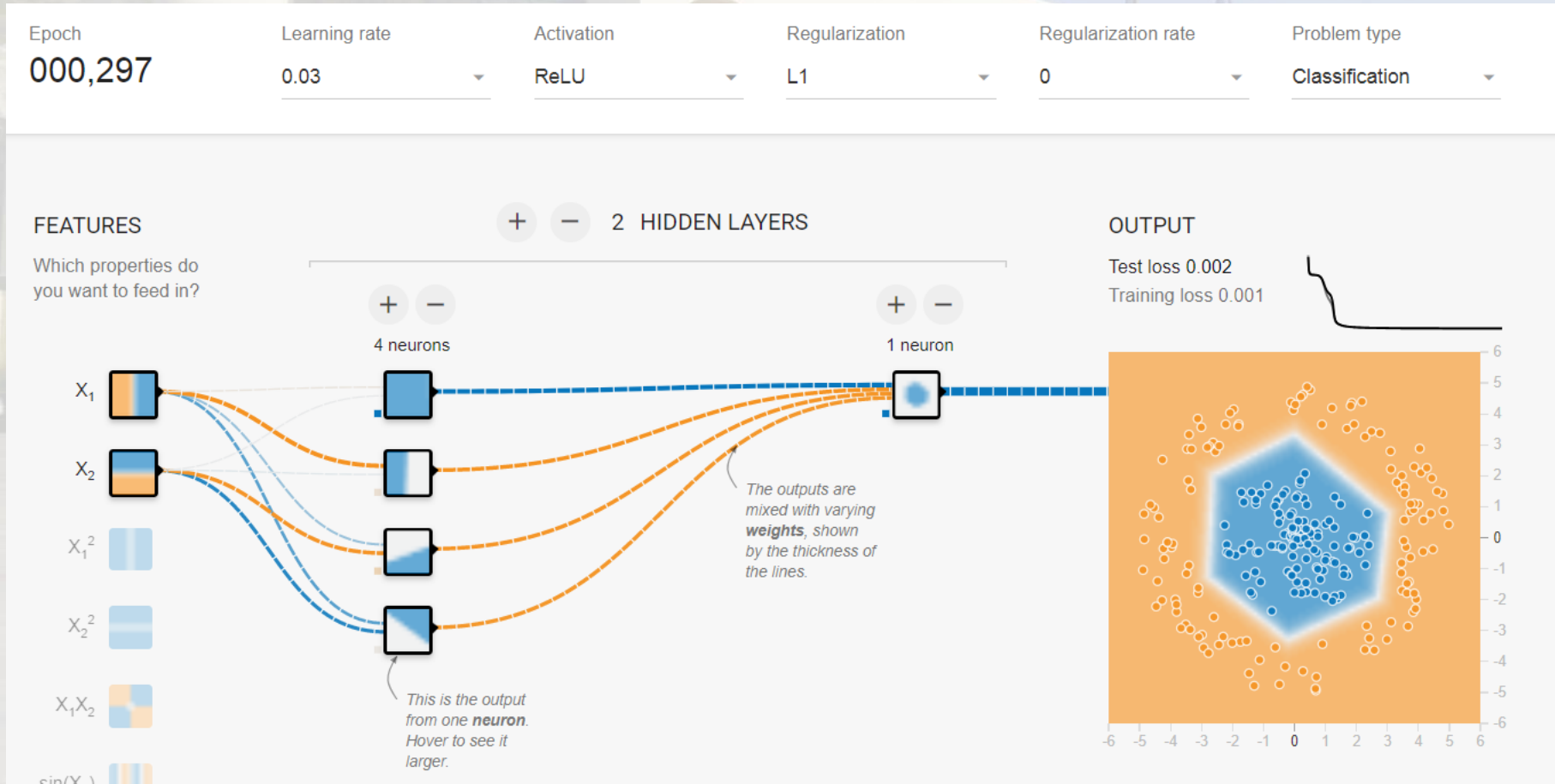


We get a more spikey decision boundary, an effect of the sharp nature of "relu"

Question: why 6 sides? How many edges do you expect in the decision boundary if you use 4 neurons?

# 8? Not necessarily!...

## Why?

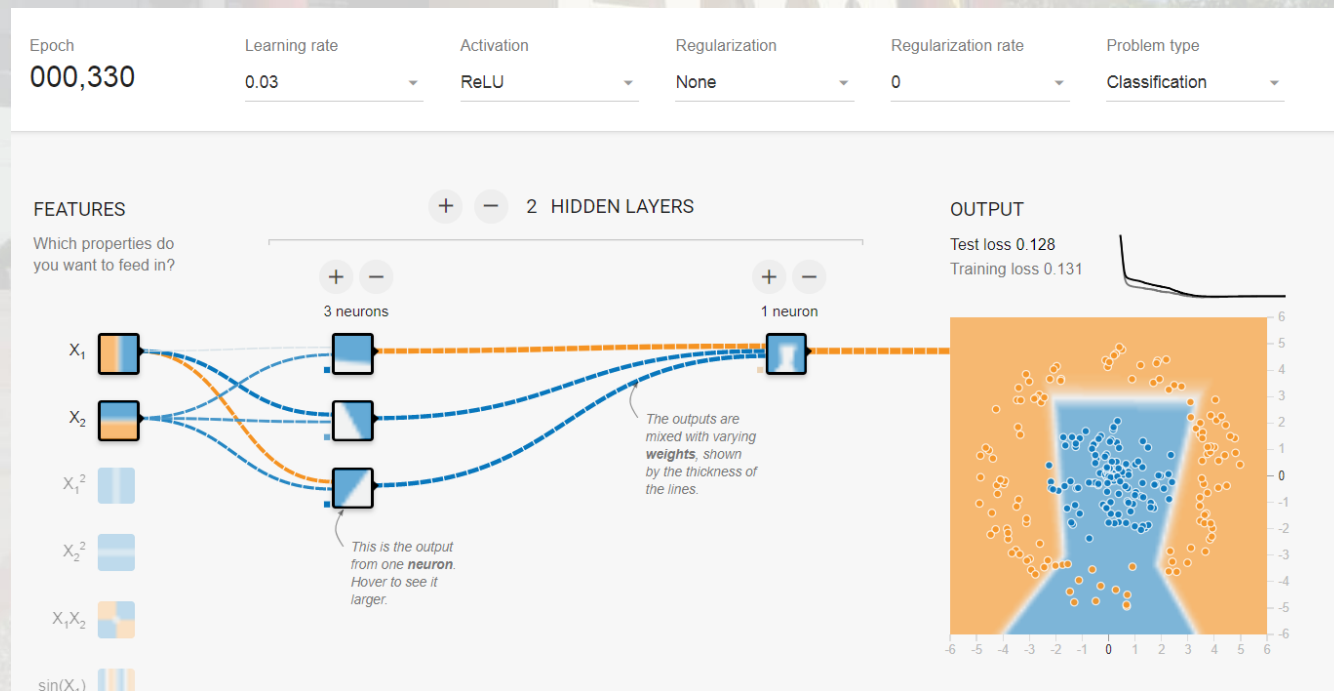




Here is a pathological case:

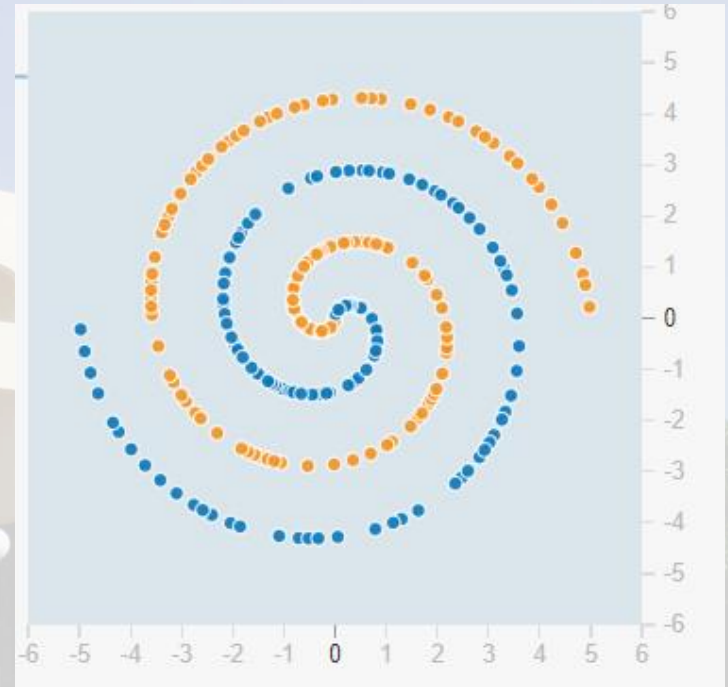
we seem to fail to obtain the wanted result from these three neurons!  
Maybe we can intervene to drift away from the local minimum?

The application allows you to do this manually, by clicking on the link connecting the first hidden node from the top to the output one, modifying the weight



Let us go back to the **tanh** activation

And take the 4th dataset:



This problem is much more complex. We can start with only two inputs  $x, y$

Try this at home. What result do you get ? Do you need many layers? How long does the training requires to converge ?

Using only  $x,y$  inputs it is harder to get to a good result...

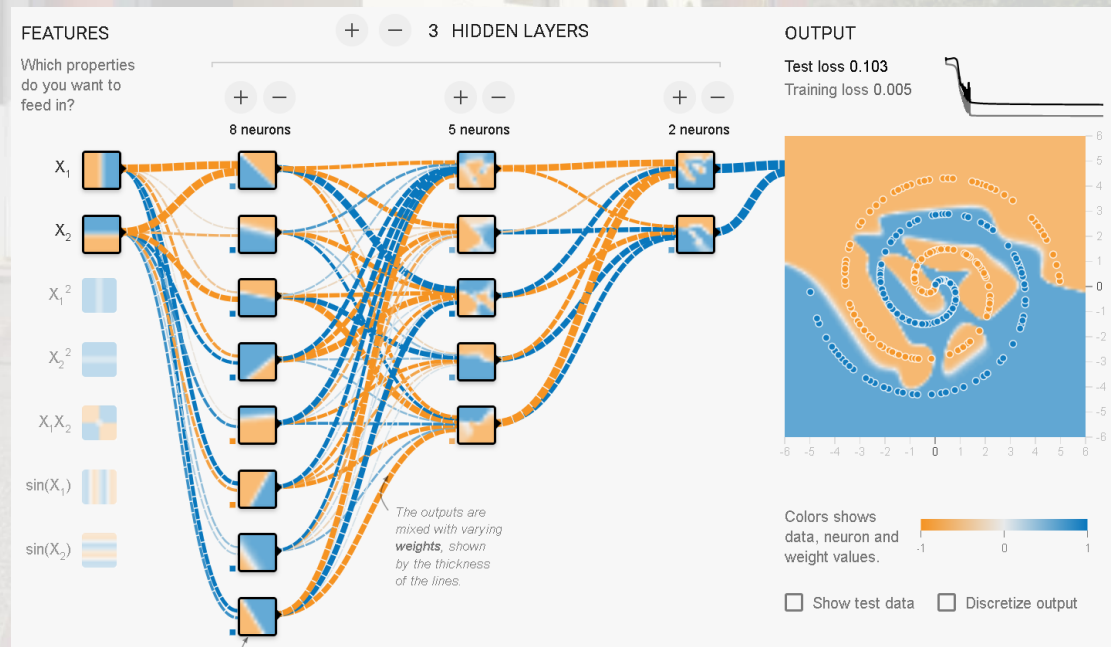
But why? After all,  $\{x,y\}$  are a complete set – a sufficient statistic!

By sweating over this simple problem you will come to learn that in order to give the NN the needed flexibility to learn the correct non-linear combinations of the given features, you need to train for a much longer number of epochs.

Note that a NN can perfectly well emulate, with appropriate training, whatever combination of the inputs, without your need to input it to it.

But this points clearly to a conclusion:

A bit of feature engineering is worth many long hours spent training a very flexible network!



# Final Challenge

Try this at home.

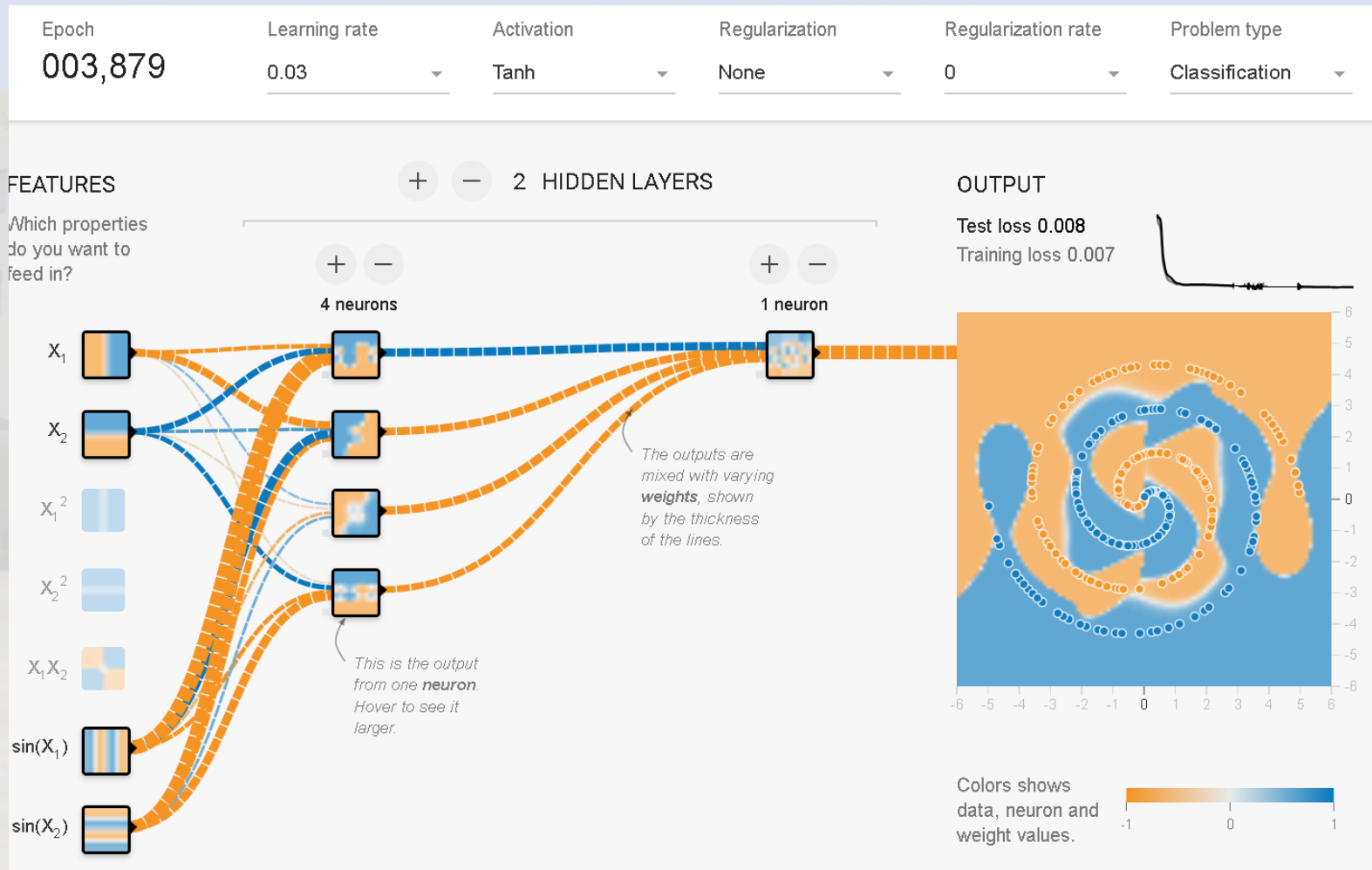
Take the fourth dataset, and use a tanh activation. Use == 4 inputs of your choice, 2 hidden layers, and a maximum total of 6 nodes in the hidden layers

**What test loss do you get ?**

(Hint: you should get a loss close to zero)

**Extra: if you can get a good score, can you do it with one less node?**

# My solution



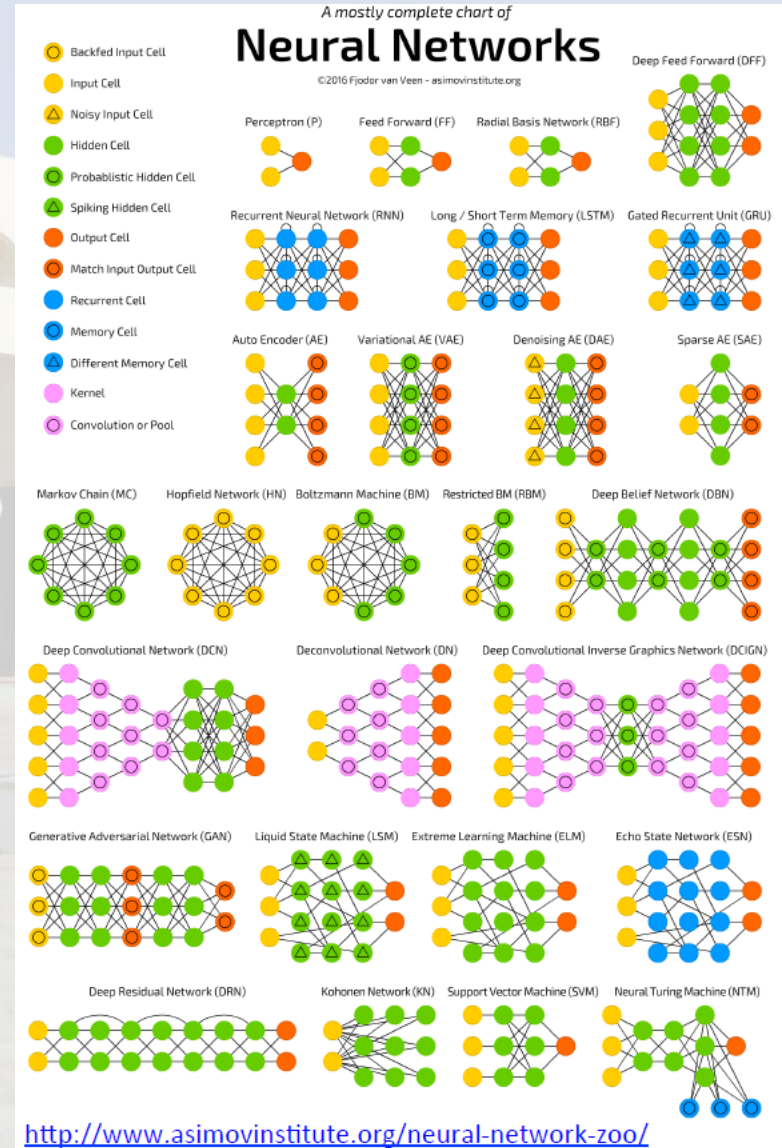
# ADVANCED TECHNIQUES



# NNs everywhere

The ascent of deep learning in the XXI century has brought to the design and development of many specialized architectures optimized to different tasks

We can give only a peek, as it is more fruitful to make sure you have gotten the gist of the basic concepts



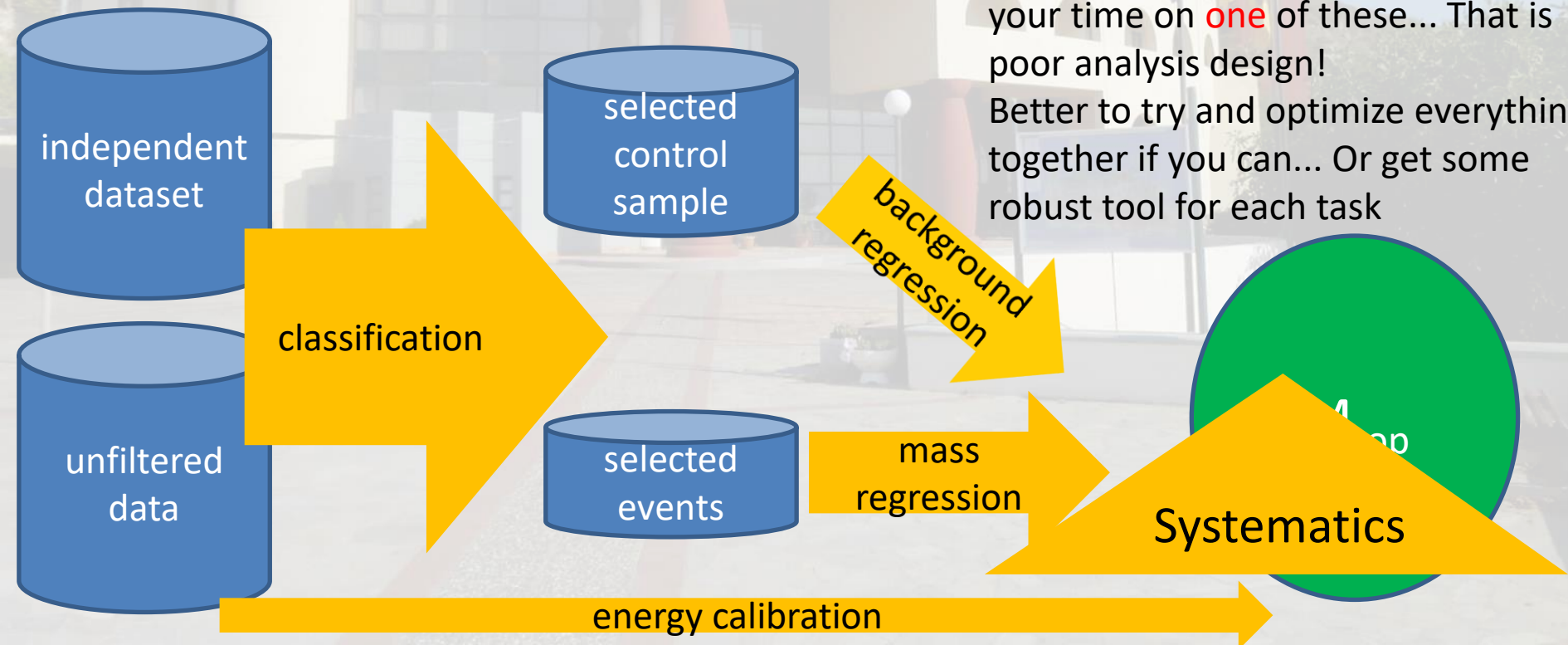
# The multi-body problem

The more one works on a NN, the better results can be achieved (up to a limit given by the NP lemma, which is however not usually achieved)

So is the message "take a problem and work at it very hard"? Not necessarily...

Take e.g. a HEP analysis where e.g. we want to measure the mass of a particle (e.g. top quark). There are multi-variate problems everywhere:

It makes little sense to spend all your time on **one** of these... That is poor analysis design!  
Better to try and optimize everything together if you can... Or get some robust tool for each task



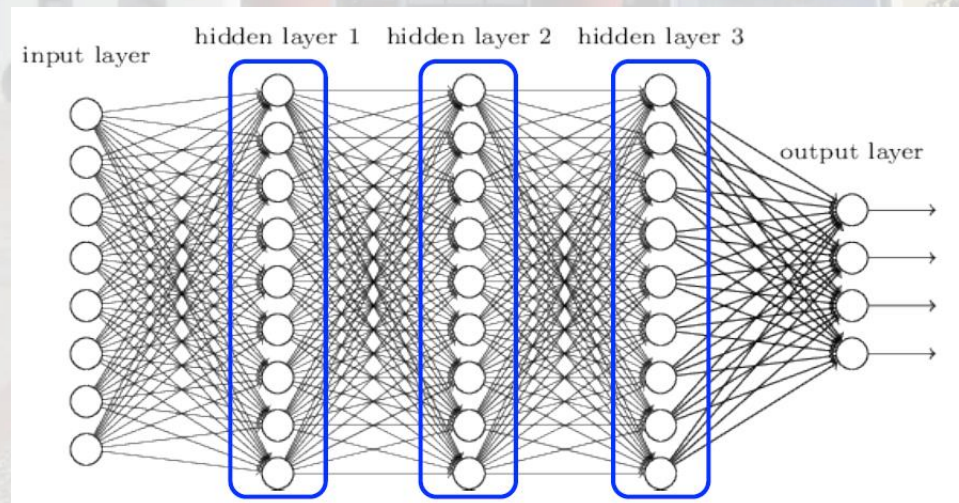


# Deep neural networks

**What is "deep"?** Arguable, but already a network with  $>2$  hidden layers can be enormously complex

DNNs are appropriate for very complex data. While **a single hidden layer should suffice to produce arbitrarily complex functions**, to do so the number of neurons has to grow exponentially  $\rightarrow$  better to increase the number of layers, essentially factorizing the learning process

DNNs can be powerful but are usually difficult to train



# Convolutional neural networks

CNNs are a specialized form of DNNs

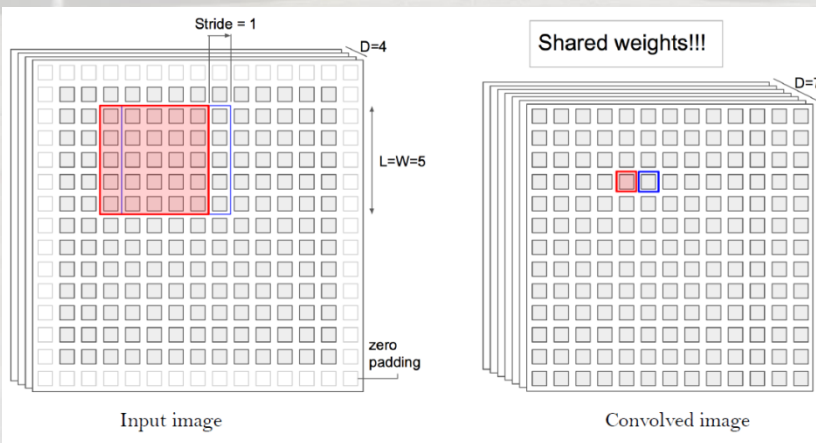
A very important commercial application is image recognition → used to drive cars, recognize faces, interpret content



# Convolutional neural networks

A convolution can be applied to reduce the dimensionality of the input (e.g. a high-res image), retaining the important information for later more effective processing

A number of "filters" can be used to reduce the input data



1 <sub>x1</sub>	1 <sub>x0</sub>	1 <sub>x1</sub>	0	0
0 <sub>x0</sub>	1 <sub>x1</sub>	1 <sub>x0</sub>	1	0
0 <sub>x1</sub>	0 <sub>x0</sub>	1 <sub>x1</sub>	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved Feature

# Example of 3x3 filter

original



filter (3 x 3)

0	0	0
0	1	0
0	0	0

identity



# Example of a blurring 5x5 filter

original



filter (5 x 5)

0	0	0	0	0
0	1	1	1	0
0	1	1	1	0
0	1	1	1	0
0	0	0	0	0

blur



# A 5x5 sharpening filter

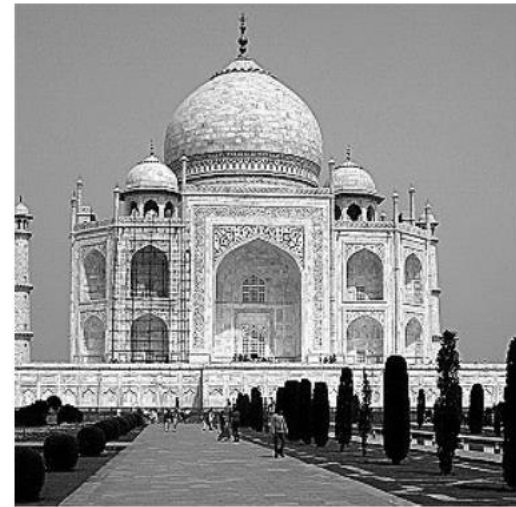
original



filter (5 x 5)

0	0	0	0	0
0	0	-1	0	0
0	-1	5	-1	0
0	0	-1	0	0
0	0	0	0	0

sharpen



# Example of a 3x3 edge detector

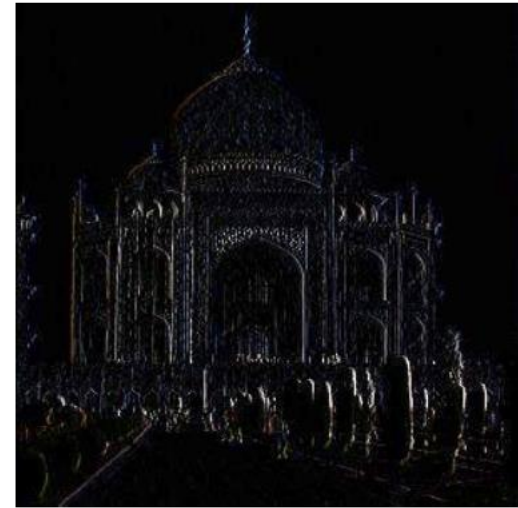
original



filter (3 x 3)

	0	0	0	
	-1	1	0	
	0	0	0	

vertical edge detector



# A 3x3 all-edge detector

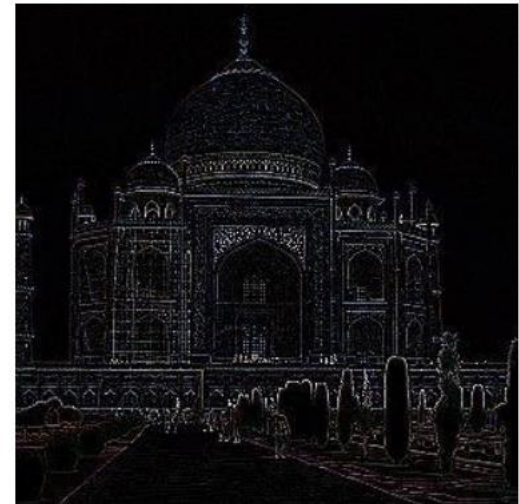
original



filter (3 x 3)

0	1	0	
1	-4	1	
0	1	0	

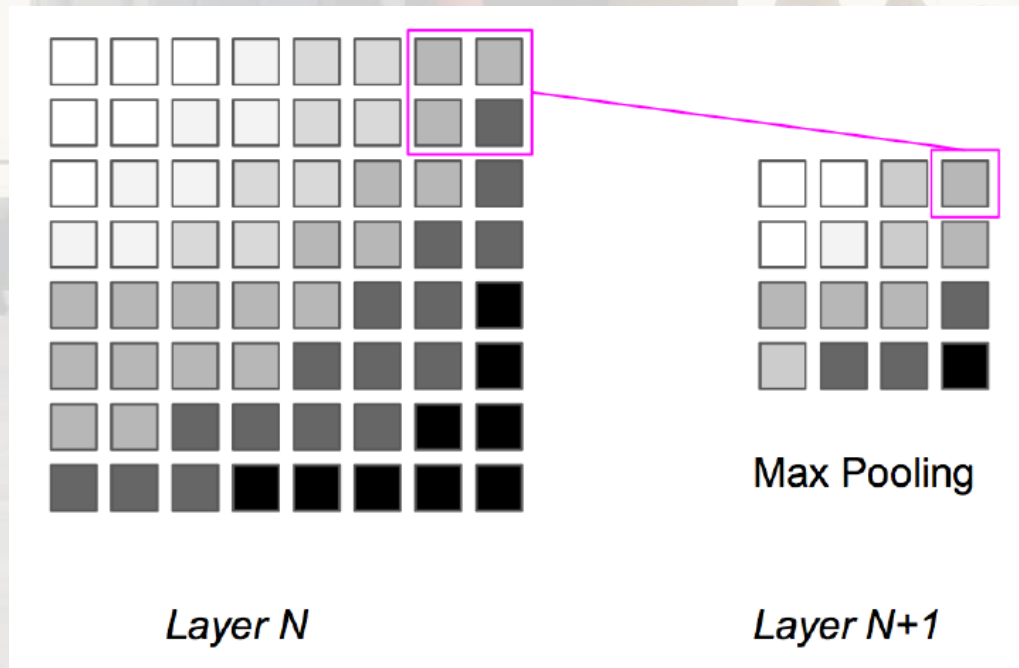
all edge detector



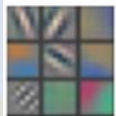


# Max pooling

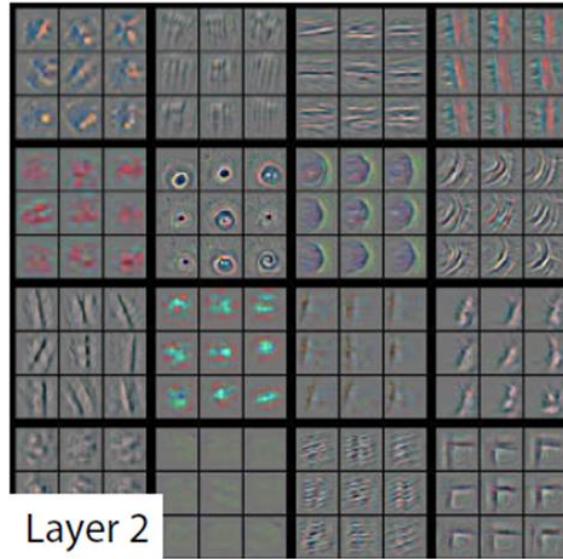
A different reduction of dimensionality is achieved by the method of "max pooling", which retains the most interesting information from the inputs, producing in the output a more compact map of the image



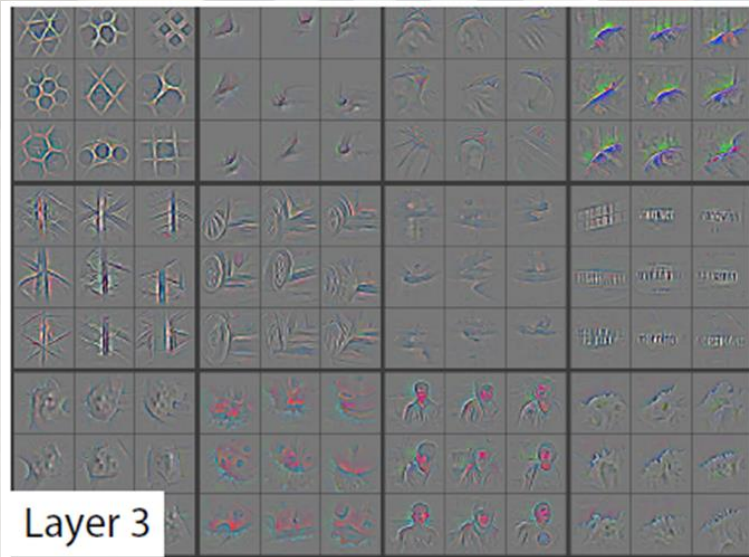
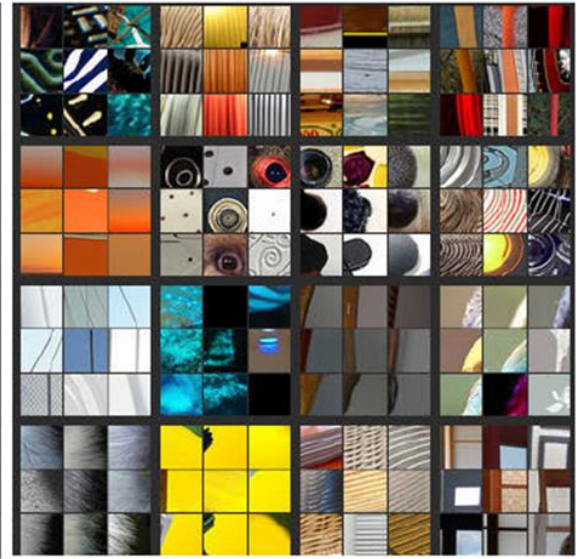
# Feature detection



Layer 1



Layer 2



Layer 3



# PRACTICAL TIPS



# Step 0: γνῶθι σεαυτόν

First of all, you should understand the specific needs of your problem. Name it!, e.g.

- Classification? Multi-class classification? Regression? Clustering? Density estimation? Hypothesis test? Goodness of fit? Optimization? ...
- Is it supervised or not supervised?
- If classification, do you need to estimate densities or can you directly create a discriminator?
- What dimensionality do your data have? High/low/can be reduced/cannot ...
- Are your data tall, wide, do they miss entries... ? Does it look like you need to work on preprocessing / data augmentation?
- Do you aim for a robust result or a performant one?

# Step 1: choosing what fits

- Want something simple? kNN may do very well for low-D (or if you can reduce D)
- Want insight in algorithm choices? Prefer decision trees
- Need high performance, aren't scared of complex optimization? A (D)NN can be your best friend (for a long time 😊)

In general, you should know that there is no free lunch (Wolpert, 1996)! It was shown that **there is no a priori method that outperforms others if no prior specification of the problem is given.**

[This is analogue to the issue "what GOF measure is best?" → there is no answer, it depends on the specific density]

This is why many different algorithms exist, and more are coming in...

But some general empirical observations have been made

# Empirical analysis

A survey of 179 methods (not including DNNs) was made testing them on 121 datasets

→ RF was the best performer in 84% of cases (see

<http://jmlr.csail.mit.edu/papers/volume15/delgado14a/delgado14a.pdf>)

Kaggle competitions also allow to draw some conclusions (M.Kagan):

- When high-level features informative of the system are present, winners are often RF
- When you have lots of unstructured, low-level information per event, DNN outperform all others
- CNNs typically work best in image classification, RNN excel in text/speech recognition

# Random bits

- Check what others have done in similar problems, even outside your domain – study the literature if you at all can!
- **Try simple things first** – they may be all you need (and they might even be best)
- Don't avoid preprocessing! Study your data to see if there are degeneracies that allow you to **augment your training set**
- Always **set up a robust validation scheme**; divide your data accordingly; do not use validation data for testing.
- Check for overtraining using cross-validation, but don't forget to avoid undertraining!
- Use CV also to **tune all the hyperparameters** that may affect your results

# CONCLUSIONS





# Conclusions

I do hope these lectures have brought you a bit closer to the world of Machine Learning

- or at least that I have not bored you to death, if you knew everything already!

As with any field in rapid development, you do not need to become all-knowledgeable before you can become a practitioner: on the contrary! The best advice I can give you is - Jump in wherever you see fit and start swimming!

You will be surprised to see how fun it is to play with these tools – not to mention the fun you may have by coding your own methods (although clearly that's not everybody's definition of "fun"...) )

# Some take-away bits

- Don't look for complex solutions when simple ones work well
  - Hastie: often kNN performs best !
  - Useful to understand easy tools before you can exploit hard ones
- As powerful as individual tools are, they aren't the answer to the question "what is best"
  - the mastery of the data analyzer is to **optimally combine** the proper ingredients to achieve their task, and then **add the killing bit** that is only useful in the particular application at hand
- In NN design, **the loss function is where the money is**
  - improvements in the inputs have also large impact in results (see tutorials)
  - smart scanning for absolute minimum is important
  - attempts to improve on already reasonably flexible designs likely to not give as big gains



**Thank you!**