University of Pisa
Computer Science

Machine Learning

Based on Prof. Alessio Micheli's lectures

February 20, 2020

Alessandro Cudazzo

Giulia Volpi

ACADEMIC YEAR 2019/2020

Notes

These notes are intended only as support for the study of the subject and as slides side notes. They do not cover the whole program but only a part, to compensate for the missing parts, we recommend the use of slides and books provided by Prof. Micheli for each topic. We hope these notes will help future students and if you find any mistakes or want to help extend these notes, feel free to do so in the GitHub repo.

What you will find here: Introduction to ML, Linear Model and K-NN, Neural Networks, Validation, Statistical Learning Theory (STL).

What's missing: Concept Learning, SVM, Bias Variance, Deep Learning (CNN, Depp, Rand), SOM, Bayes Learning, RNN, SDL.

Contents

1 Introduction

> The problem of learning is arguably at the very core of the problem of intelligence, both biological and artificial
>
> *Poggio, Shelton, AI Magazine 1999*

1.1 What is ML

The field of **machine learning** is concerned with the question of how to construct computer programs that automatically improve with experience. In recent years many successful machine learning applications have been developed, ranging from data-mining programs that learn to detect fraudulent credit card transactions, to information-filtering systems that learn users' reading preferences, to autonomous vehicles that learn to drive on public highways. At the same time, there have been important advances in the theory and algorithms that form the foundations of this field. Machine learning draws on concepts and results from many fields, including statistics, artificial intelligence, philosophy, information theory, biology, cognitive science, computational complexity, and control theory

Machine Learning has emerged as an area of research combining the aims of *creating computers that could learn* (AI - Build adaptive/personalized intelligent systems) and new powerful *adaptive/statistical tools* with rigorous foundation in computational science for data analysis.

So **Learning** is the major challenge and a strategic way to provide intelligence into the systems.

Definition 1.1. A computer program is said to **learn** from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

The ML studies and proposes methods to build (infer) a model (dependencies / functions / hypotheses) from examples of observed data

- that fits the know examples
- able to generalize, with reasonable accuracy for new data (according to verifiable results, under statistical and computational conditions and criteria)
- Considering the expressiveness and algorithmic complexity of the models and learning algorithms

E.g. Inferring general functions from know data:

- Handwriting Recognition: x (Data from pen motion) and $f(x)$ (Letter of the alphabet)
- Face recognition: x (Bitmap picture of person's face) $f(x)$ (Name of the person)

So we have some opportunity (if useful) and awareness (needs and limits):

Utility of predictive models: (in the following cases):

- no (or poor) theory (or knowledge to explain the phenomenon)
- uncertain, noisy or incomplete data (which hinder formalization of solutions)

Requests:

- source of training experience (representative data)
- tolerance on the precision of results

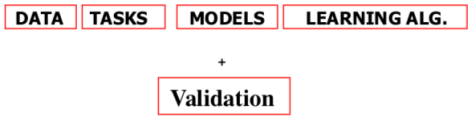


Figure 1.1: Components of a ML system

1.2 Overview: Components of a ML System

Main components of a ML system, Framework as a guide to the key design choices:

1.3 Data

The data represent the available facts (experience). Representation problem: to capture the structure of the analyzed objects. **Type**: Flat, Structured, ...

We will generally use flat data:
E.g. **Flat DATA** (attribute-value language): fixed-size vectors of properties (features), single table of tuple (measurements of the objects), Attributes can be discrete or continuous

Fruits	Weight	Cost $	Color	Bio
Fruit 1 (lemon)	2.1	0.5	y	1
Fruit 2 (apple)	3.5	0.6	r	?

→ Attributes (discrete/continuous)

- Categorical/continuous, missing data,...
- Preprocessing: e.g. Variable scaling, encoding*, selection...

Of course Data type can be different, usually there is a Data Preprocessing step, e.g. Variable scaling, encoding, selection... For this step, see in Data Mining: *Data Understanding* and *Data Preparation*.

Example and terminologies
Let's see a example with some Medical records in the flat case:

Patients	Age	Smoke	Sex	Lab Test
Pat 1	101	0.8	M	1
Pat 2	30	0.0	F	?

→ Attributes (discrete/continuous)

- Each row ($\mathbf{x}$, vector): example, pattern, instance, sample,...
- Dimension of data set: number of examples l
- Dimension (of $\mathbf{x}$): number of features n
- If we will index the features/inputs/variables by j : variable x_j is (typically) the j-th feature/property/attribute/element of $\mathbf{x}$.

but may be to simplify we need to use subscript index for other meanings

- $\mathbf{x}_p$ is (typically) the p-th pattern/example/raw ($\mathbf{x}$ bold is a vector)
- x_{pj} for example can be the attribute j of the pattern p

DATA Encoding
For the **Flat case** we can have **Numerical encoding for categories** e.g.:

- 0/1 (or –1/+1) for 2 classes
- 1,2,3... Warning: grade of similarity (1 vs 2 or 3): useful for "order categorical" variables (e.g small, medium, large)
- 1-of-k (or 1-hot) encoding: useful for symbols

A	1	0	0
B	0	1	0
C	0	0	1

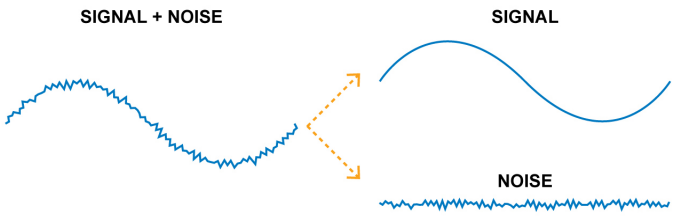
Useful both for input or output variables

This is useful because the scalar product between two vectors representing two coded symbols is zero. The vectors are orthogonal, this implies that there is no relation between the category symbols.

Another types of Data:
E.g. **Structures DATA**: Sequences (lists), trees, graphs, Multi-relational data (table) (in DB)
Examples: images, microarray, temporal data, strings of a language, DNA e proteins, hierarchical relationships, molecules, hyperlink connectivity in web pages, ...

Further terminologies

- **Noise**: addition of external factors to the stream of target information (signal); due to randomness in the measurements, not due to the underlying law: e.g. Gaussian noise



- **Outliers**: are unusual data values that are not consistent with most observations (e.g. due to abnormal measurements errors). Some operation can be perform: outlier detection and preprocessing (removal) or we have to use some Robust Modeling Methods.
- **Feature selection**: selection of a small number of informative features: it can provide an optimal representation for a learning problem

1.4 Task

The task defines the purpose of the application: Knowledge that we want to achieve? Which is the helpful nature of the result? What information are available?
There are two main categories of tasks:

- **Predictive** (Classification, Regression): function approximation (build a function from examples)
- **Descriptive** (Cluster Analysis, Association Rule): find subsets or groups of unclassified data

We can usually divide tasks in *Supervised Learning* and *Unsupervised Learning*:

Supervised Learning
Given a training example as $< input, output >=< \mathbf{x}, d >$ (**labeled example**) for an unknown function f,

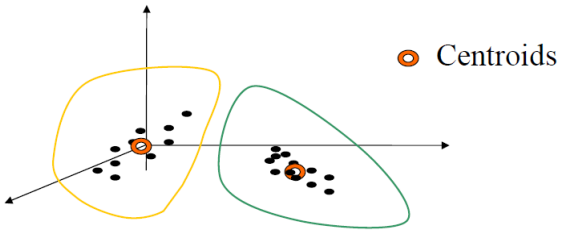


Figure 1.2: An Example of Clustering, Partition of data into clusters (subsets of "similar" data)

the aim is to find a *good* approximation to f (an **hypothesis** h that can used for prediction on unseen data **x'**).
Target d (or t or y) is given by the teacher according to a $f(\mathbf{x})$ (unknown function), d is usually a numerical/categorical label:

- *Classification*: discrete value outputs:

$$f(\mathbf{x}) \in \{1, 2, \dots, \} \text{ classes (discrete-valued function)}$$

- *Regression*: real continuous output values (approximate a real-valued target function)

Note: we can use the same model, we only have to change the domain of the output (Unified vision thanks to the formalism of func. approximation)

Unsupervised Learning
Given a training set of **unlabeled data** $< \mathbf{x} >$, the aim is to find a *natural groupings* in the set. Some taks are:

- Clustering, see figure 1.2
- Dimensionality reduction/ Visualization/Preprocessing
- Modeling the data density

1.4.1 Classification

(Supervised) Classification: Patterns (features vectors) are seen as members of a class and the goal is to assign the patterns observed classes (label). So, the aim is to find a *good* approximation to $f(\mathbf{x})$ (an **hypothesis** h) that can be used to return a prediction on unseen data **x'** and tell what class **x'** belongs to, see figure 1.3.

If the number of classes is 2, the approximation of $f(\mathbf{x})$ is a boolean function (binary classification, **concept learning**), instead if the number of classes is greater than 2, we talk about *multi-class problem* $(C_1, C_2, \dots, C_k)$.

The classification may be viewed as the allocation of the input space in decision regions (e.g. 0/1) and we want to find a linear separator on an instance space $\mathbf{x} = (x_1, x_2)$in $\mathbb{R}^2$ where $f(x) = 0/1$(or $-1/+1$).
In figure 1.4a we can see an hyperplane which separates the points and this is a hypothesis $h(x)$ (an approximation to $f(\mathbf{x})$) found in the space of hypotheses H (set of dichotomies induced by hyperplanes), see figure 1.5.

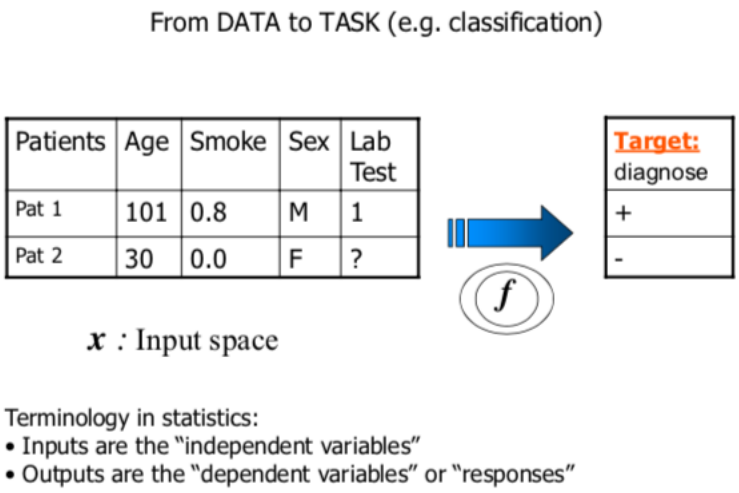
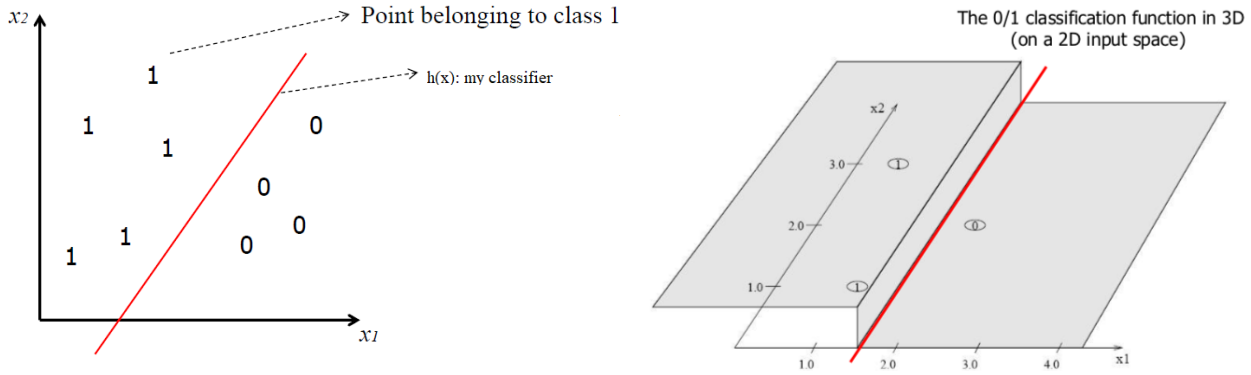


Figure 1.3: A classification task: we want to get an approximation to $f(\mathbf{x})$, we want to understand if future patients are positive or negative to a certain diagnosis



(a) Separating (hyper)plane : x.

(b) Geometrical 3D view: classifier

Figure 1.4: An example of classification, that use a linear separator on the instance space.

$$\boldsymbol{wx} + w_0 = w_1 x_1 + w_2 x_2 + w_0 = 0$$

$$h(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{wx} + w_0 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

or

$$h(\boldsymbol{x}) = sign(\boldsymbol{wx} + w_0)$$

Linear threshold unit (LTU)

Indicator functions

Figure 1.5: Hyperplanes classifier

1.4.2 Regression

Process of estimating of a real-value function on the basis of finite set of noisy samples (supervised task), we have known pairs $(x, f(x) + \text{random noise})$.
Goal Task: find a function that approximate the data in figure 1.6:

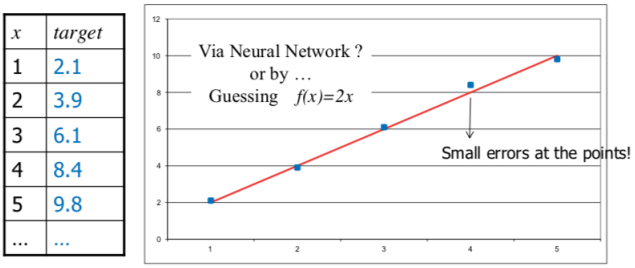
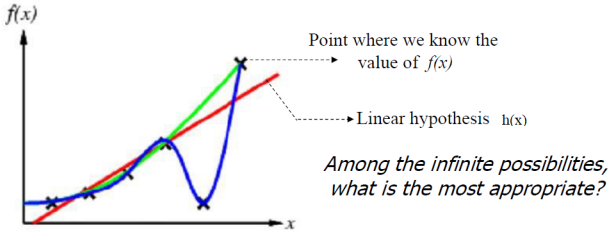


Figure 1.6: A example of regression

Regression: $x =$ variables (e.g real values), $(f(x) + \text{random noise})$ real values that we know, so we want to find a curve that best fit the data: *curve fitting*. An example of curve fitting is a **linear hypothesis**:



$h_w(x) = w_1 x + w_0 = 0.2x - 0.4$ but we have to choose the best one with some criteria.

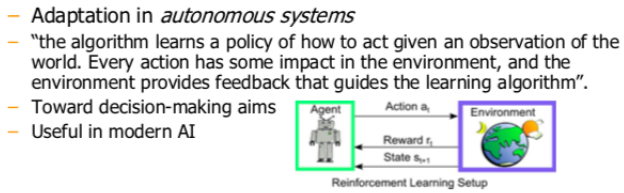
1.4.3 Other Tasks

There are other types that we will not discuss, like:

- **Dimensionality reduction** (in unsupervised Learning)

$$< x_1, x_2, \ldots, x_n > \rightarrow < x_1, x_2, \ldots, x_m > \text{ with } m < n$$

- **Reinforcement Learning**(learning with right/wrong critic): This is usually a supervised learning task with some police!



- **Semi-supervised learning**: combines both labeled and unlabeled examples to generate an appropriate function or classifier.

,

1.5 Models

The aim is to capture/describes the relationships among the data (on the basis of the task) by a "language". The "language" is related to the representation used to get knowledge. It defines the class of functions that the learning machine can implement (*hypothesis space*). E.g. set of functions $h(x, w)$, where w is the (abstract) parameter.

Here some definition:

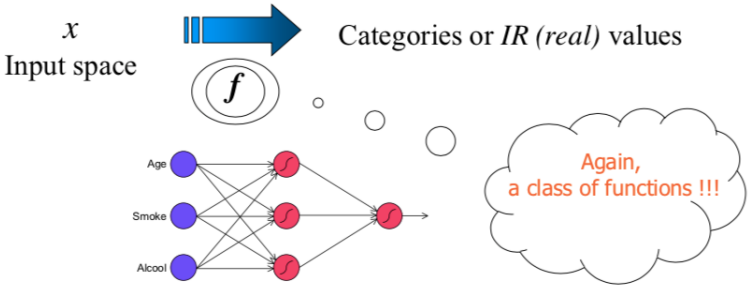
- **Training examples** (supervised learning):
  An example of the form $(x, f(x))$. x is usually a vector of features, $t = f(x)$ is called the target value.
- **Target function**: The true function f (that we don't have! we want to approximate it!)
- **Hypothesis**: A proposed function h believed to be similar to f. An expression in a given *language* that describes the relationships among data.
- **Hypotheses space (H)**: The space of all hypotheses that can, in principle be output by the learning algorithm (set of functions $h(x, w)$, where w is the parameter), this is a Hilbert Space.

Some models are shown here:

Just to have a preview of different *representation* of hypothesis
(because you already know the language of equations, logic, probability):

- **Linear models** (representation of H defines a continuously parameterized space of potential hypothesis);
  each assignment of w is a different hypothesis, e.g:
  - $h(\mathbf{x}) = sign(\mathbf{w}\mathbf{x} + w_0)$ — *binary classifier*
  - $h_w(x)=w_1x+w_0$ E.g. $h_w(x)= 2x+150$ — *simple linear regression*
- **Symbolic Rules**: (hypothesis space is based on discrete representations); different rules are possible , e.g:
  - *if* $(x_1=0)$ *and* $(x_2=1)$ *then* $h(\mathbf{x})=1$ — *binary classifier*
  - *else* $h(\mathbf{x})=0$
- **Probabilistic models**: estimate $p(x,y)$
- K Nearest neighbor regression: Predict mean y value of nearest neighbors (memory-based)

Another example: we will see a **neural networks**, beyond the neurobiological inspiration, as a computational model for the treatment of data, capable of approximating complex (non-linear) relationships between inputs and outputs.



1.5.1 Pardigms and methods (Languages for H)

- Symbolics and Rule-based (or discrete H)
  - Conjuction of literals, Decision trees (propositional rules)
  - Inductive grammars, Evolutionary algorithms, ...
  - Inductive Logic Programming (first order logic rules)
- Sub-symbolic (or continuous H)
  - Linear discriminant analysis, Multiple Linear Regression, LTU
  - Neural networks
  - Kernel methods (SVMs, gaussian kernels, spectral kernels, etc)
- Probabilistic/Generative
  - Traditional parametric models (density estimation,discriminant analysis,polynomial regression,...)
  - Graphical models: Bayesian networks,naive Bayes, PLSA, Markov models, Hidden Markov models, ...
- Instance-based
  - Nearest neighbor

1.5.2 How many models?

No Free Lunch Theorem : there is no universal "best" learning method (without any knowledge, for any problems,...): if an algorithm achieves superior results on some problems, it must pay with inferiority on other problems. In this sense there is no free lunch.

The course provide a set of models and the critical instrument to compare them.

NOTE: However, not all the models are equivalent: (instead of assuming a specific form for the target function (parametric models in classical Statistics))
The core of ML is on flexible approaches that can in principle approximate arbitrary functions (universal approximation property)

- Typical powerful models of ML: e.g. Neural Networks
- ...but "power is nothing without control"
- We need of inductive principia for the control of the complexity

1.6 Learning Algorithms

A Learning Algorithms is basing on data, task and model!
(Heuristic) search through the hypothesis space H of the **best hypothesis** (Typically searching for the h with the minimum "error", the best approximation to the (unknown) target function). Practically we want to find parameters w that give the best h.

H may not coincide with the set of all possible functions and the search can not "be exhaustive": it needs to make assumptions (we will see the role of *Inductive bias*), see figure 1.7.

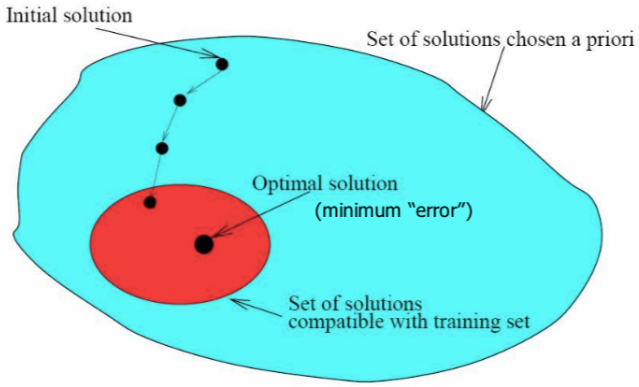


Figure 1.7: Search on the hypothesis space

Learning (terminologies): According to the different paradigms/contexts "learning" can be differently termed or have different acceptations:

- Inference (statistics)
- Inference: Abduction/Induction (logic)
- Adapting (biology, systems)
- Optimizing (mathematics)
- Training (e.g. Neural Networks)
- Function approximations (mathematics)

Can be more specifically found in other sub-fields:

- Regression analysis (statistics), curve fitting (math, CS), ...
- Or using other terminologies e.g. "Fitting a multivariate function"

1.7 Task + Model = Loss

We have to find a "**good**" approximation to f from examples. How to measure the quality of the approximation?

We want to measure the "distance" between $h(x)$ (output of the model for input $\mathbf{x}$) and d (according to a $f(\mathbf{x}) +$ random noise)

Minimization of errors in training, check of errors in test

We will use:

- *Loss function*: $L(h(\mathbf{x}), d)$ (high value means poor approximation)
- The Error (or Risk) is an expected value of the loss

$$\textit{Empirical Risk} : R_{emp} = \sum_{i=1}^{l} L(h(\mathbf{x}), d)$$

  e.g. a "sum" or mean of the loss over the set of samples.

We will change L for different tasks.

Common Tasks review
A possible classifications of common learning tasks specifying the (changing of the) nature of the loss function, output and hypothesis space.

- Survey of common learning tasks
- Nature of models (hypothesis spaces) for different class of tasks.

1.7.1 Loss Function example: Regression

In a regression task we want to predict a numerical value. We have an **Hypothesis Space** H that is a set of real-valued functions and the **Loss function** will measures the approximation accuracy/error between $h(x_i)$ and $d_i = f(x_i) + e$ (real value function + random error).

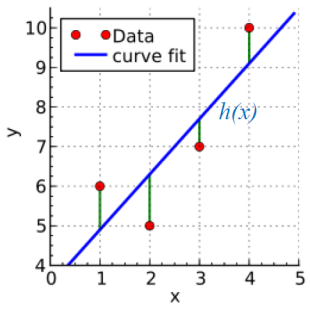
A common loss function for Regression (predicting a numerical value) is the *squared error*: $L(h(\mathbf{x}_i), d_i) = (d_i - h(\mathbf{x}_i))^2$ (squared error because we want the distance and a negative or positive value makes no different).

R_{emp}: the mean over the data set provide the *Mean Square Error* (MSE)

In the example we have $h(x)=w_1x+w_0$ as the blue line and in green the errors at the data points (x_i, y_i) (in red), where the target d_i for x_i is y_i in the example

The Mean Square Error (MSE) is the mean of the square of the green errors.

$$E(\mathbf{w}) = \frac{1}{l}\sum_{p=1}^{l}\left(y_p - h_w(\mathbf{x}_p)\right)^2$$



Note: this plot is take from internet, I used different colors before: blue for the points (data) and red for the line in my previous plot.
Also the *y* are the desidered (target d) values

1.7.2 Loss Function example: Classification

Classification of data into discrete classes, d_i can be e.g. 0/1 and $h(x_i)$ will predict 0/1. So our **Loss Function** will measures the classification error:

$$L(h(\mathbf{x}_i), d_i) = \begin{cases} 0 & if \;\; h(\mathbf{x}_i) = d_i \\ 1 & otherwise \end{cases}$$

R_{emp}: The mean over the data set provide the number/percentage of misclassified patterns

E.g. 20 out of 100 are misclassified $\rightarrow$ 20% errors, i.e. 80% of accuracy

1.7.3 Loss Function example: Clustering and Vector Quantization

Goal: optimal partitioning of unknown distribution in x-space into regions (clusters) approximated by a cluster center or prototype. e.g see figure 1.2

H: a set of vector quantizers $x \rightarrow c(x)$ (continuosspace $\rightarrow$ discrete space)

Loss Function: would be the squared error distortion:

$$L(h(\mathbf{x}_i)) = \langle (\mathbf{x}_i - h(\mathbf{x}_i)), (\mathbf{x}_i - h(\mathbf{x}_i)) \rangle \quad (\text{ inner product })$$

Proximity of the pattern to the centroid of its cluster

1.7.4 Loss Function example: Density estimation

Density estimation (generative, "parametric methods") from an assumed class of density

- **Output**: a density e.g. normal distribution with mean m and variance $sigma^2$: $p(x \mid m, sigma^2)$
- **H**: a set of densities (e.g. m and $sigma^2$ are the two unknown *parameters*)

- A *common* **loss function** for density estimation:

$$L(h(\boldsymbol{x}_i)) = -\ln(h(\boldsymbol{x}_i)) \longrightarrow \text{We'll see later}$$

- Related to "maximizing the (log) likelihood function". [not hear]
- E.g. $P(x_1, x_2, x_3, \ldots \mid m, sigma^2)$

1.8 GENERALIZATION

Learning: search for a good function in a function space from known data (typically minimizing an Error/Loss), but how we can say that this model is good for our task after the learning step?

A model is Good with respect to the **generalization error**: how accurately the model predicts over novel samples of data (Error/Loss measured over new data)

So *Generalization* is a crucial point of ML. We will see the difference between "Easy to use ML tools" and the "correct/good use of ML".

So we have two main phases:

- **Learning** phase **(training, fitting)**: build the model from know data – *training data* (and bias).
- **Predictive** phase **(test)**: apply to new example (we take the input **x** and we compute the response by the model): evaluation of the predictive hypothesis, i.e. of the **generalization capability**.

The Predictive phase correspond to the Deployment/Inference use of the ML built model.

⇒ Generalization is a crucial point of ML!!!

Evaluation of performances for ML systems = predictive accuracy
Estimated by the error computed on the (Hold out) Test Set

Accuracy/performance estimation is a critical aspect, we can do it by:

- theory to understand under what mathematical conditions a model is able to generalize (*Statistical Learning Theory* [Vapnik])
- Empirical (training, test) and cross-validation techniques

NB: The performance on training data provide an overoptimistic evaluation, so never test accuracy on training set!

A very important phase is **Validation**!

Some definition:

Inductive Learning Hypothesis: any hypothesis h found to approximate the target function (f) well over a sufficiently large set of training examples will also approximate the target function well over any other unobserved examples.

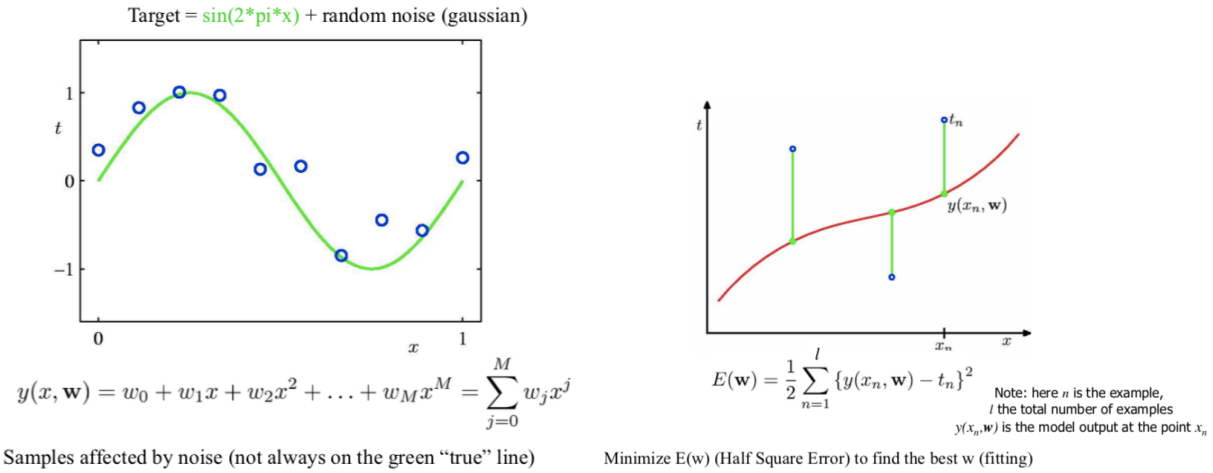
This is the fundamental assumption of inductive learning (Supervised learning) (e.g. in concept learning) and we will have much more to say about it.

Overfitting: It output a hypothesis $h(\cdot) \in H$ having true error ϵ and empirical error E, but There is another $h(\cdot)^{'} \in H$ having $E^{'} > E$ and $\epsilon^{'} < \epsilon$

1.8.1 Complexity on case of study

Let's take an example with a **parametric model** for regression: **Polynomial Curve Fitting Example**, with just one variable and we assume to know the target function, see figure 1.8a and as error function we will use the Sum-of-Squares Error, see figure 1.8b.

- The set of function is assumed as polynomials with degree M
- The complexity of the hypothesis increase with the degree M
- l = number of examples



(a) Polynomial Curve Fitting

(b) Sum-of-Squares Error Function

Figure 1.8: A simple parametric model for regression example

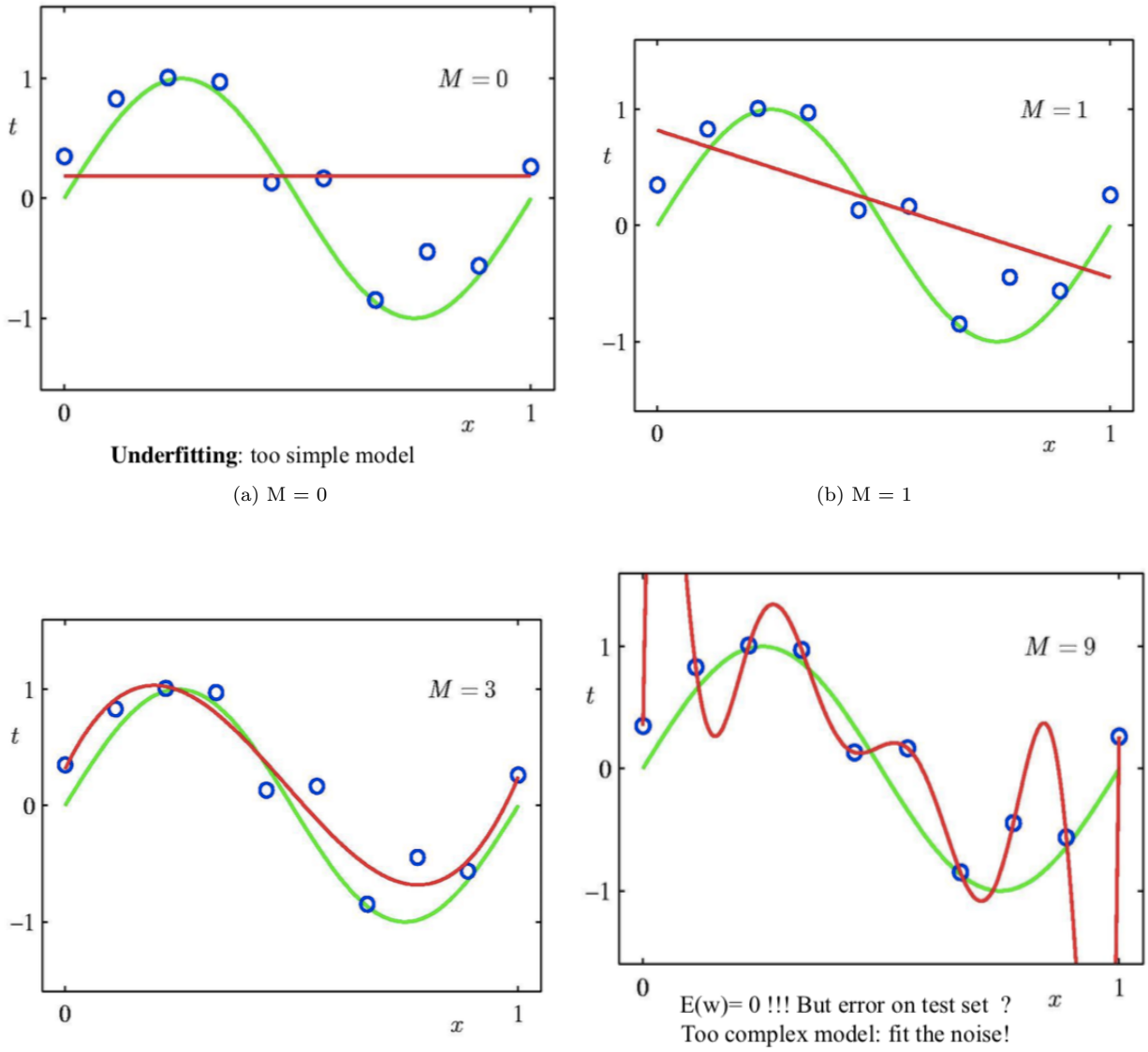
Warning: This is an artificial simplified task (unrealistic due to the use of just 1 input variable, the fact that we know the target function in advance, ...)

let's analyze how good is a model if we increase the complexity and how change if the number of data, made available in the training phase, increases.

Case $l = 10$:
Assume that we have some few data (figure 1.8a) and let's see how our model will fit the data if we increase the complexity of the model.

Let's see some cases:

- M = 0: in this case we are in **underfitting**, we have a very simple model that cannot fit the data. With M = 0 the model it's just the mean. See figure 1.9a.
- M = 1: Linear model, still a model model, dosen't not fit good the data. See figure 1.9b.
- M = 3, this time, the model seems very good, we have a low error. See figure 1.9c.
- M = 9: Too complex model, it's fit the noise too!, the error on the training test is $E(w) = 0$, but error on the test set? See figure 1.9d. Poor representation of the (green) true function (due to) **overfitting**).



(a) M = 0

(b) M = 1

(c) M = 3

(d) M = 9

Figure 1.9: The complexity of the hypothesis increase with the degree M

So let's plot Root-Mean-Square(RMS) Error and see how change with the complexity (M) of the model, where $w*$ are the polynomial coefficients:

$$E_{RMS} = \sqrt{2E(w^*)/l}$$

As we can see in figure 1.10 the error is 0 in the training test (overfitting) but the error is really high on the test set. the model is not able to generalize.

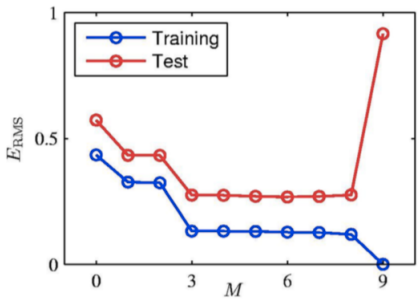
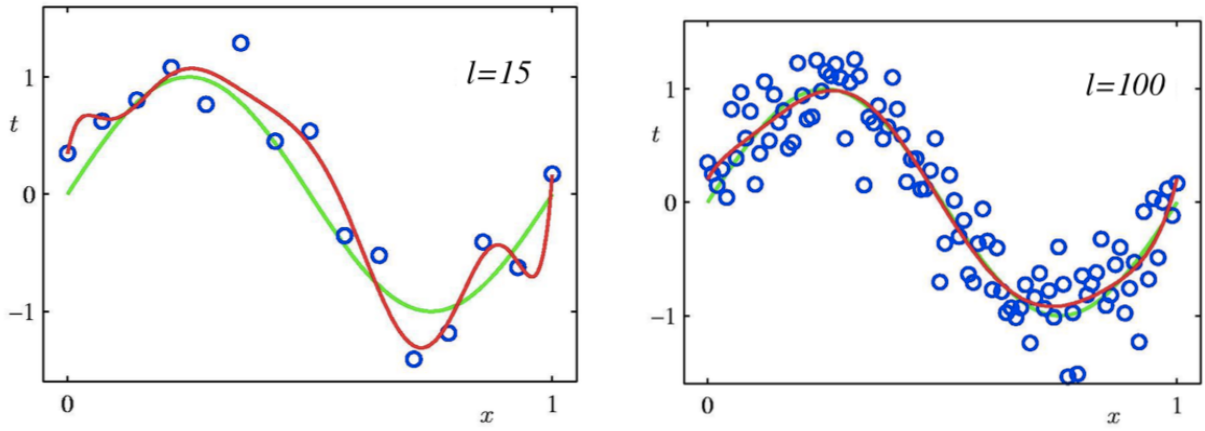


Figure 1.10: Caption

Case $l = 15$:
With a 9th order polynomial model ($M = 9$) and more data the model seems to get a better generalize for the future data, but still not good. See figure 1.11a.

Case $l = 100$:
With a 9th order polynomial model ($M = 9$) and more data the model seems to get a better generalize for the future data, but still not good. See figure 1.11b.



(a) l = 15

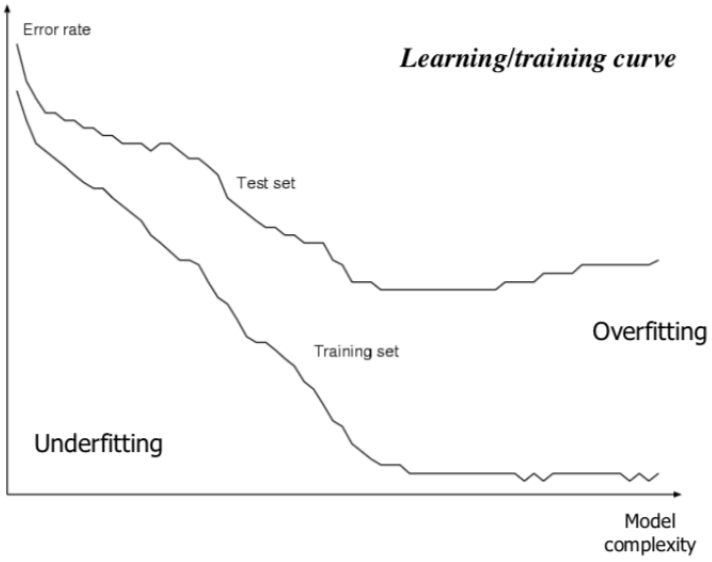
(b) l = 100

Figure 1.11: The complexity of the hypothesis with the degree 9 when the number of data change

So model complexity and the number of data are very important for the generalization capability of a model, let's put all together:

- The generalization capability (measured as a risk or test error) of a model with respect to the training error and avoid overfitting and underfitting zones
- The role of model complexity
- The role of the number of data

Typical behavior of learning



1.8.2 Toward Statistical Learning Theory (SLT)

Statistical Learning Theory (SLT) is a general theory relating such topics. In a (Simplified) Formal Setting:

1. we want to approximate an unknown function $f(\mathbf{x})$
2. minimize a risk function:
$$R = \int L(d, h(\mathbf{x}))dP(\mathbf{x}, d)$$
This is the **True error** over all the data (that we don't have). d is the value from the teacher and $P(x, d)$ the probability distribution. A loss (or cost) function could be for example: $L(h(\mathbf{x}), d) = (d - h(\mathbf{x}))^2$
3. then search h in H minimizing R

But we have only a finite data set: $TR = (\mathbf{x}_i, d_i)\ \forall i = 1, \ldots, l$. So we can't use R as minimize risk function. To search h, we will minimize the empirical risk (training error), finding the best values for the model free parameters:
$$R_{emp} = 1/l \sum_{i=1} l(d_i - h(\mathbf{x}_i))^2$$
Emprical Risk Minimixzation (ERM) Inductive Principle.

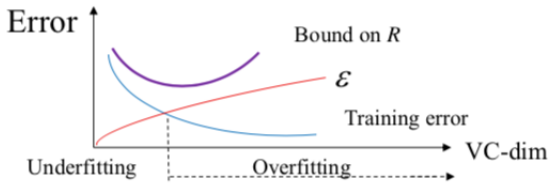
Vapnik-Chervonenkis-dim and SLT: a general theory
we can use R_{emp} to approximate R!

VC-dim: measure complexity of H (flexibility to fit data), e.g. Num. of parameters for polynomials, NN, ...

The Vapnik-Chervonenkis bound (VC-bound(s)) is in the form:
$$R \leq R_{emp} + \epsilon(1/l, VC, 1/\delta) \ , \ \text{with probability } 1 - \delta$$
where $R_{emp} + \epsilon$ is the **guaranteed risk** and ϵ is the **VC-*confidence***

- Higher l (data): lower R
- Higher VC-dim (fix l) : lower R_{emp} but R may increase (overfitting)

Structural Risk Minimization: find a trade-off Concept of control of the model complexity (flexibility): trade-off between model complexity and TR (training) accuracy (fitting).



Model selection: choose the model (H) with the better bound on the true risk

An example It is possible to derive an upper bound of the ideal error which is valid with probability $(1-\delta)$, delta being arbitrarily small, of the form:

- General: $R \leq R_{emp} + \varepsilon(1/l, VC, 1/\delta)$
- Example: $R \leq R_{emp} + \varepsilon(VC/l, -\ln(\delta/l))$
- There are different bounds formulations according to different classes of f, of tasks, etc.
- More in general, in other words (simplifying): we can make a good approximation of f from examples, provided we have a good number of data, and the complexity of the model is suitable for the task at hand.

1.8.3 Complexity control

SLT - Statistical Learning Theory:

- It allows formal framing of the problem of generalization and overfitting, providing analytic upper-bound to the risk R for the prediction over all the data, regardless to the type of learning algorithm or details of the model..
- The ML is well founded: the Learning risk can be analytically limited and only few concepts are fundamentals !
- SLT has allowed to leads to new models (e.g SVM) and other methods that directly consider the control of the complexity in the construction of the model
- bases one of the inductive principles on the control of the complexity
- explain the main difference with respect to supporting methods from CM (providing the techniques to perform fitting), apart from modelling aspects

But there are other open questions: what (other) principles are to found the control of the complexity? How to work in practice? How to measure the complexity (or fitting flexibility)? How find the best trade-off between fitting and model complexity?

1.9 VALIDATION

Evaluate generalization capabilities (of your $h(x)$). **Warning**: The performance on training data provide an overoptimistic evaluation

Evaluation of performances for ML systems = Predictive accuracy

Validation phase has two main aims:

- **Model selection**: estimating the performance (generalization error) of different learning models in order to choose the best one (to generalize).
  this includes search the best hyper-parameters of your model (e.g. polynomial order, ...).

It returns a model

- **Model assessment**: having chosen a final model, estimating/evaluating its prediction error/ risk (generalization error) on new test data (measure of the quality/performance of the ultimately chosen model).

It returns an estimation

Gold rule: Keep separation between goals and use separate data sets for this two operation

In an ideal world, for validation we should have:

- a large training set (to find the best hypothesis, see the theory)
- a large validation set for model selection
- a very large external unseen data test set

But with a finite and often small data set we will have just a estimation of the generalization performance. Let's see two basic techniques for validation: Simple hold-out (basic setting) and K-fold Cross Validation.

1.9.1 Hold out cross validation

Hold out: basic setting.
Partition data set D into training set (TR), validation or selection set (VL) and test set (TS). See figure 1.12

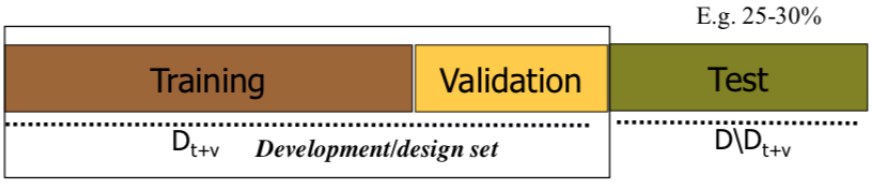


Figure 1.12: Hold out cross validation

- All the three sets are disjoint sets
- Training is used to run the training algorithm
- VL can be used to select the best model (e.g hyper-parameters tuning)
- Test set is not to be used for tuning/selecting the best h: it is only for model assessment

Figure 1.13: k-fold cross validation

1.9.2 Hold out and K-fold cross validation

When we have a small number of data, Hold out Cross Validation can make insufficient use of data. To solve this problem we can use another technique called: **K-fold Cross-Validation**:

- Split the data set D into k mutually exclusive subsets $D_1, D_2, \ldots, D_k$
- Train the learning algorithm on $\frac{D}{D_i}$ and test it on D_i repeat this phase k time and then we take the mean of the validation results.
- Can be applied for both VL or TS splitting
- It uses all the data for training and validation/testing

But this technique has some issues:

- How many folds? 3-fold, 5-fold , 10-fold, ..., 1-leave-out
- Often computationally very expensive (we have to perform training and validation phases k times)

Combinable with validation set, double-K-fold CV, ...

1.9.3 TR/VL/TS by a schema

:

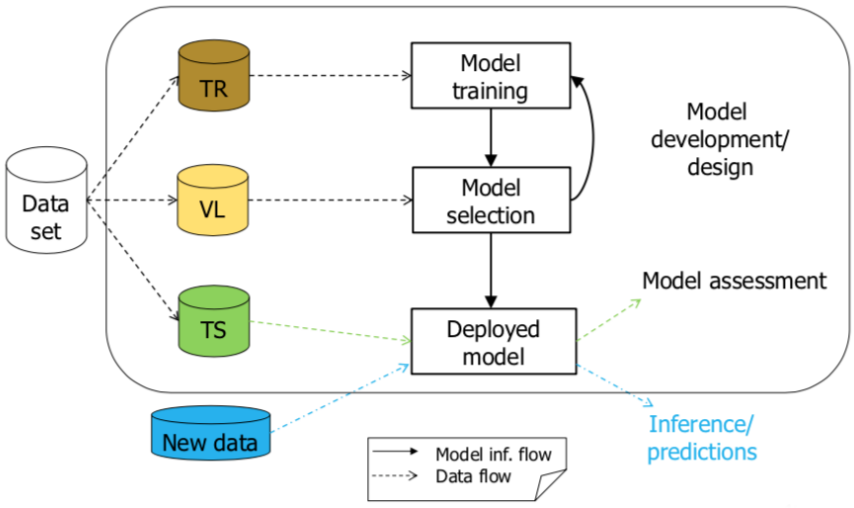


Figure 1.14: TR/VL/TS by a schema

1.9.4 Classification Accuracy

In the field of machine learning and specifically the problem of statistical classification, a confusion matrix, also known as an error matrix. A confusion matrix is a table that is often used to describe the performance of a classification model (or "classifier") on a set of test data for which the true values are known. It allows the visualization of the performance of an algorithm. See figure 1.15.

Actual \ Predicted	Positive	Negative
Positive	TP	FN
Negative	FP	TN

Figure 1.15: Confusion matrix

The number of correct and incorrect predictions are summarized with count values and broken down by each class. This is the key to the confusion matrix. The confusion matrix shows the ways in which your classification model is confused when it makes predictions. It gives us insight not only into the errors being made by a classifier but more importantly the types of errors that are being made.

Definition of the Terms:

- Positive (P) : Observation is positive (for example: is an apple).
- Negative (N) : Observation is not positive (for example: is not an apple).
- True Positive (TP) : Observation is positive, and is predicted to be positive.
- False Negative (FN) : Observation is positive, but is predicted negative.
- True Negative (TN) : Observation is negative, and is predicted to be negative.
- False Positive (FP) : Observation is negative, but is predicted positive.

Classification Rate/Accuracy is given by the relation:

$$\frac{TP + TN}{Total = TP + TN + FP + FN}$$

Two importat statistical measures are: *Specificity* and *Sensitivity*.

Specificity
Specificity (also called the true negative rate) measures the proportion of actual negatives that are correctly identified as such (e.g., the percentage of healthy people who are correctly identified as not having the condition).

$$Specificity = \frac{TN}{FP + TN} = 1 - FPR$$

where FPR (fall-out or false positive rate) $= FP/N = FP/(FP + TN)$

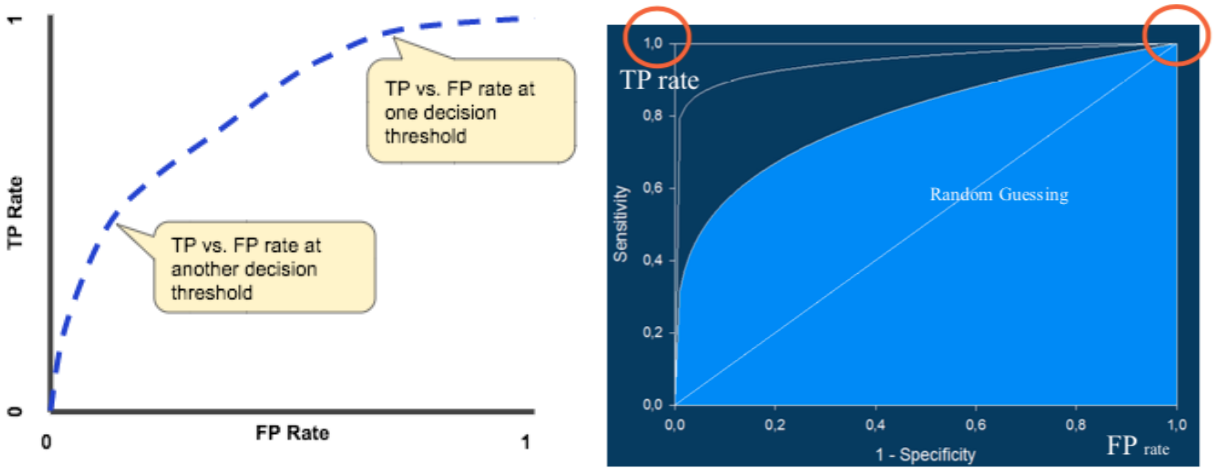
Sensitivity
Sensitivity (also called the true positive rate, the recall, or probability of detection[1] in some fields) measures the proportion of actual positives that are correctly identified as such (e.g., the percentage of sick people who are correctly identified as having the condition).

$$Sensitivity = \frac{TP}{TP + FN}$$

NOTE: for binary classif.: 50% correctly classified = "coin" (random guess) predictor. Of course could exists trivial classifier with unbalanced data (e.g. 99% of the data are positive).

ROC Curve
An ROC curve (receiver operating characteristic curve) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters: Sensitivity (True Positive Rate (TPR)) and False Positive Rate (FPR) (1 - Specificity).

Lowering the classification threshold classifies more items as positive, thus increasing both False Positives and True Positives. See figure 1.16a.



(a) AUC (Area under the ROC Curve) (b) AUC

Figure 1.16: ROC curve

AUC stands for "Area under the ROC Curve." and measures the entire two-dimensional area underneath the entire ROC curve (think integral calculus) from (0,0) to (1,1). See figure 1.16b.

AUC provides an aggregate measure of performance across all possible classification thresholds. One way of interpreting AUC is as the probability that the model ranks a random positive example more highly than a random negative example.
The diagonal corresponds to the worst classifier (random guessing). Better curves have higher AUC (Area Under the Curve). In 1.0 we have the perfect classification.

1.10 The Design Cycle

Data collection: adequately large and representative set of examples for training and test the system

Data representation: Often the most critical phase for an overall success.

- domain dependent, exploit prior knowledge of the application expert
- Feature selection
- Outliers detection
- Other preprocessing: variable scaling, missing data,..

Model choice:

- statement of the problem

- hypothesis formulation: You must know the limits of applicability of your model
- complexity control

Building of the model (core of ML): through the learning algorithm using the training data
Evaluation: performance = predictive accuracy

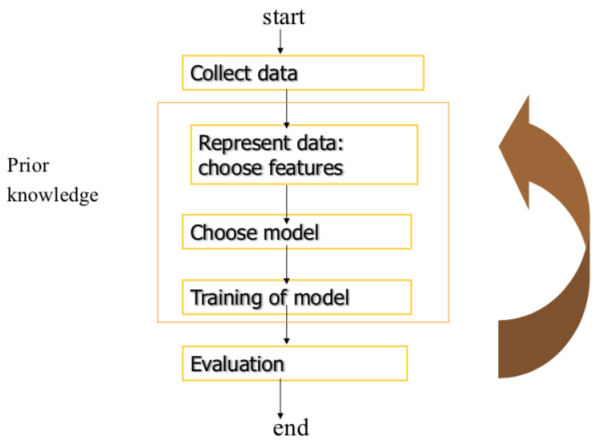


Figure 1.17: Design cycle

1.11 Misinterpretations

For every statistical models (including Data Mining (DM) applications)

Causality is (often) assumed and a set of data representative of the phenomena is needed.

- Not for unrelated variables and for random phenomena (lotteries)
- Uninformative input variables → poor modeling → Poor learning results

Causality cannot be inferred from data analysis alone:

- People in Florida are older(on av.) than in other US states.
- Florida climate causes people to live longer ?

May be there is a statistical dependencies for reasons outside the data

More specifically for ML:

- Powerful models (even for "garbage" data) → higher risk !
- Not-well validated results: the predicted outcome and the interpretation can be misleading.

2 Linear models

In this section we will going to analyze a linear model for classification and regression (Supervised Learning) both as task in function approximation.

This is a **Parametric model**: A learning model that summarizes data with a set of parameters of fixed size (independent of the number of training examples) is called a parametric model.

- **Given**: Training examples as $< input, output >=< x, d >$, we have labeled examples for some for some unknown function f. (For us f is known only at the given example points).
- **Find**: a good approximation to f (that can used for prediction on unseen data $\mathbf{x}'$

The target value $\mathbf{d}$ (or $\mathbf{t}$ or $\mathbf{y}$), given by the teacher according to $f(\mathbf{x})$, are numerical/ categorical label:

- Classification: $f(x)$ return the (assumed) correct class for x, where $f(x)$ is a discrete valued function.
- Regression: approximate a real-valued target function (in $\mathbb{R}$ o $\mathbb{R}^K$).

Here some Data notation in figure 2.1:

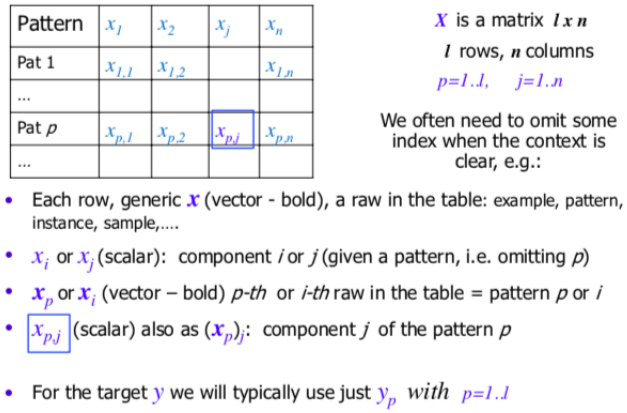


Figure 2.1: Data notation for this section

Now will we start to see linear models for Regression and classification where the H (hypothesis space is linear). Same model can be applied for both tasks and we will see how.

"Despite the great inroads made by modern nonparametric regression techniques, linear models remain important, and so we need to understand them well." (Hastie)

The linear model has been the mainstay of statistics and has a lot of studies and in many books (mathematics, statistics, numerical analysis, applicative fields, ML, ...). This model is also used/included in more complex models.

We will start with the simplest form: linear in the input variables.

2.1 Regression

Let's see first how we can formulate the learning problem as a **Least mean square (LMS)** problem and let's start to see it in a simplified setting: **univariate case**.

A regression task is the process of estimating of a real-value function on the basis of finite set of noisy samples, we have known pairs $(x, f(x) + \text{random noise})$ and we want to find an h (hypothesis) that fit the data. See figure 2.2.

We want to solve it (how to find $\mathbf{w}$) in a systematic way.

x	target
1	2.1
2	3.9
3	6.1
4	8.4
5	9.8
...	...

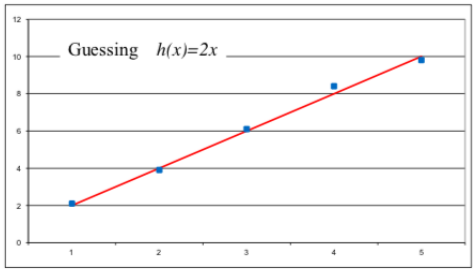


Figure 2.2: Regression Example

2.1.1 Univariate Linear Regression

Univariate case, simple linear regression:

- we start with 1 input variable x and 1 output variable y
- we assume a model $h_w(x)$ expressed as:

$$h_w(x) = out = w_1 x + w_0$$

where **w** are real-valued **coefficients/free parameters (weights)**.

The idea is **fitting** the data by a "straight line".

In this case we have Infinite hypothesis space (continuous w values) but we have nice solution from classical math (going back to Gauss/Legendre 1795!). Surprisingly we can "learn" by this basic tool and although simple it include many relevant concept of modern ML and it is a basis of evolved methods in the field.

⇒ **Learning via LMS**:
Learn: means find the **w** such that minimize error/empirical loss (bestdata fitting – on the training set with l examples).

- Given a set of l training example (x_p, y_p)
- find $h_w(x)$ in the form: $w_1 x + w_0$ (hence the values of w) that minimizes the expected loss on the training data.

As Loss function we will use the square of errors, Least (Mean) Square: find w to minimize the residual sum of squares: $argmin_w \|Error(\mathbf{w})\|$

$$Loss(h_{\mathbf{w}}) = E(\mathbf{w}) = \sum_{p=1}^{l} (y_p - h_{\mathbf{w}}(x_p))^2 = \sum_{p=1}^{l} (y_p - (w_1 x_p + w_0))^2$$

Where x_p is p-th input/patter/example, y_p the output for p, w free par., l num. of examples

Note:

- to have the mean, divide by l
- On the notation: Indeed for the univariate case, with 1 variable: $x_p = x_{p,l} = (x_p)_l$

The method of least squares is a standard approach to the approximate solution of over-determined systems, i.e., sets of equations in which there are more equations than unknowns.

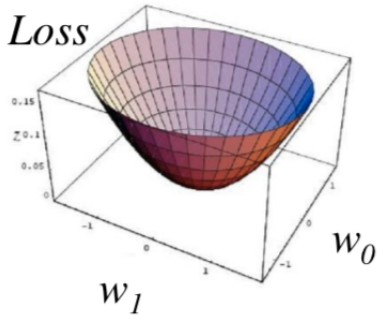
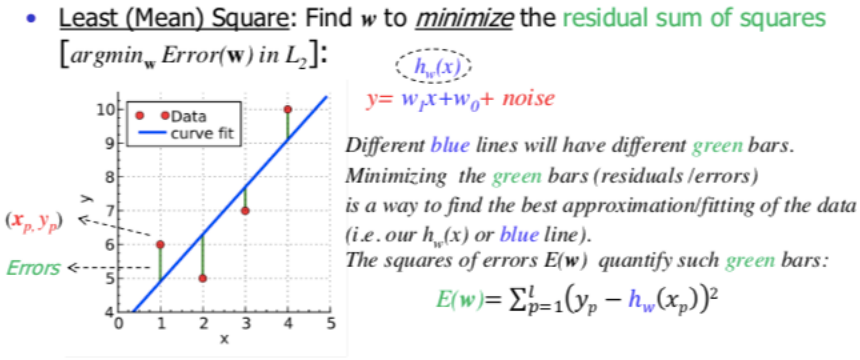


Figure 2.3: Loss function plot with two free parameters



⇒ **How to solve?**:
Remember: local minimum as stationary point: the gradient is null:

$$\frac{\partial E(\mathbf{w})}{\partial w_i} = 0 \ , \ i = 1, \ldots, \text{dim\_input} + 1 = 1, \ldots, n+1$$

For the simple Lin. Regr. (2 free parameters):

$$\frac{\partial E(\mathbf{w})}{\partial w_i} = 0 \ , \ \frac{\partial E(\mathbf{w_0})}{\partial w_1} = 0$$

We have a convex loss function, so there is no local minima and there is a closed form. The solution is:

$$\boxed{w_1 = \frac{\sum x_p y_p - \frac{1}{l}\sum x_p \sum y_p}{\sum x_p^2 - \frac{1}{l}(\sum x_p)^2}} = \frac{\text{Cov}[x,y]}{\text{Var}[x]}, \quad \boxed{w_0 = \overline{y} - w_1 \overline{x}}$$

$$\frac{1}{l}\sum_{p->l} y_p \qquad \frac{1}{l}\sum_{p->l} x_p$$

Let's compute the gradient for 1 (each) patter p:

$$\frac{\partial E(\mathbf{w})}{\partial w_i} = \frac{\partial (y - h_{\mathbf{w}}(x))^2}{\partial w_i} = 2(y - h_{\mathbf{w}}(x)) \frac{\partial (y - h_{\mathbf{w}}(x))}{\partial w_i} =$$
$$= 2(y - h_{\mathbf{w}(x)})) \frac{\partial (y - (w_1 x + w_0))}{\partial w_i}$$

and we will have:

$$\frac{\partial E(\mathbf{w})}{\partial w_0} = -2(y - h_{\mathbf{w}}(x)) \ , \frac{\partial E(\mathbf{w})}{\partial w_1} = -2(y - h_{\mathbf{w}}(x))x$$

then we will sum up for l patterns (x_p, y_p):

$$\frac{\partial E(\mathbf{w})}{\partial w_0} = -2\sum_{p=1}^{l}(y_p - h_{\mathbf{w}}(x_p)) \ , \ \frac{\partial E(\mathbf{w})}{\partial w_1} = -2\sum_{p=1}^{l}(y_p - h_{\mathbf{w}}(x_p))x_p$$

2.1.2 Linear Regression with multidimensional inputs case

Assuming column vector $\mathbf{x}$, $\mathbf{w} \in \mathbb{R}^n$, number of data l:

$$\mathbf{w}^T\mathbf{x} + w_0 = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$$

Note:

- that often (in NN), as before, the transpose notation T in w^T is omitted
- w_0 is the intercept/threshold/bias/offset (has nothing to do with the inductive bias, bias is just a name here)

Often it is convenient to include the constant $x_0 = 1$ so that we can write:

$$\mathbf{w}^T\mathbf{x} = \mathbf{x}^T\mathbf{w} \ , \ \mathbf{x}^T = [1, x_1, x_2, \ldots, x_n] \text{ and } \mathbf{w^T} = [w_0, w_1, w_2, \ldots, w_n]$$

So, the "linear" model is now:

$$h(\mathbf{x}_p) = \mathbf{x}_p\mathbf{w} = \sum_{i=0}^{n} x_{p,i}w_i \ , \ w_i \text{ continuous (free) parameters "weights"}$$

2.2 Classification

We want now to use the same linear model (with LMS) presented for the regression task and use it for the classification task. The problem in Classification is shown in figure 2.4

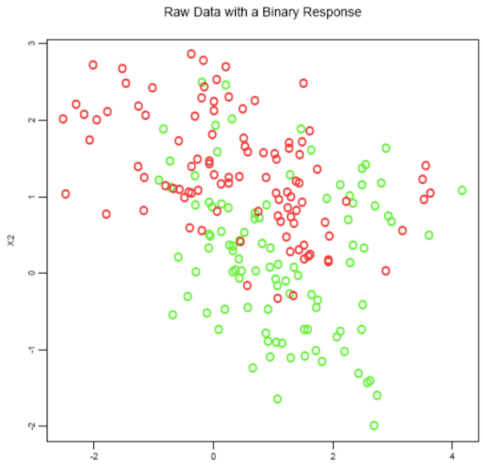


Figure 2.4: A classification problem (Data may be generated by gaussian distribution (for each class) with different means or by a mixture of different low variance gaussian distributions)

We have 200 points generated in $\mathbb{R}^2$ from an unknown distribution, 100 in each of two classes and we want to build a rule to predict the color of future points.

We reuse the linear model for the classification task:

The same models (used for regression) can be used for classification: categorical targets, e.g. $0/1$ or $-1/+1$

- In this case we use an hyperplane ($\mathbf{wx}$) assuming negative or positive values.
- We exploit such models to decide if a point $\mathbf{x}$ belong to positive or negative zone of the hyperplane (to classify it)
- So we want to set $\mathbf{w}$ (by learning) s.t. we get good classification accuracy

Geometrical view: hyperplane:

We define an hyperplaen: $\mathbf{w}^T\mathbf{x}$, see figure 2.5 where our hyperplane is:

$$\mathbf{w}^T\mathbf{x} = w_1x_1 + w_2x_2 + w_0 = 0$$

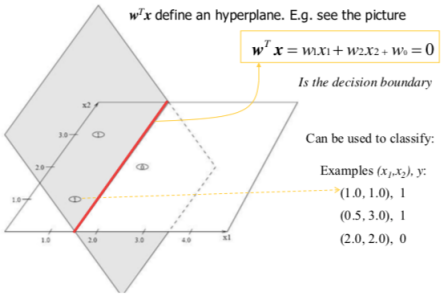


Figure 2.5: Geometrical view: hyperplane

Geometrical view: classifier:
In this case our hypothesis ($h(x)$) is called **Linear threshold unit (LTU)** and is defined as follow:

$$h(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{wx} + w_0 \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

or

$$h(\boldsymbol{x}) = sign(\boldsymbol{wx} + w_0)$$

Linear threshold unit (LTU)
Indicator functions

In case of w_0 included in $\mathbf{w}$:

$$h(\mathbf{x}_p) = sign(\mathbf{x}_p^T\mathbf{w}) = sign(\sum_{i=0}^{n} x_{p,i}w_i)$$

A Geometrical view of our classifier is shown in figur 2.6

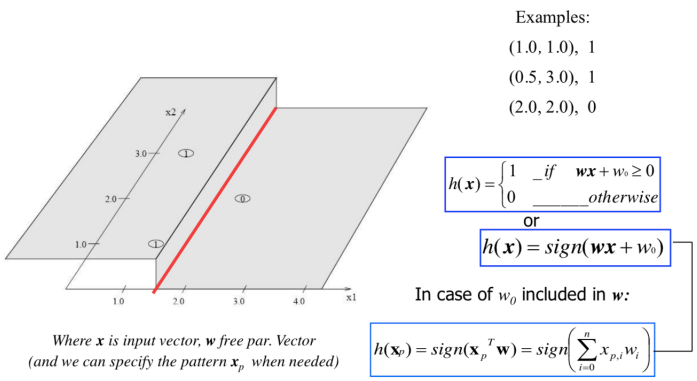
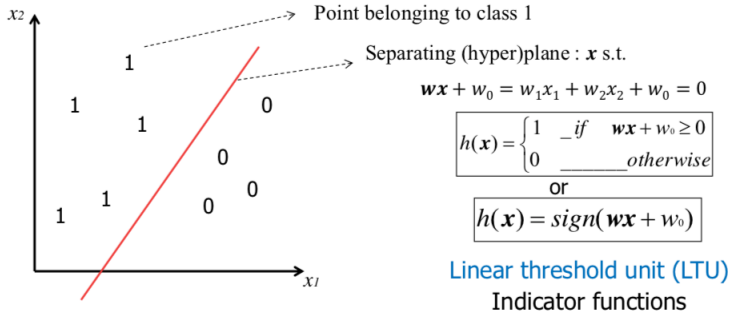


Figure 2.6: Geometrical view: hyperplane

Classification by linear decision boundary:
The classification may be viewed as the allocation of the input space in decision regions (e.g. 0/1) and the linear decision boundary can solve a linearly separable problem.

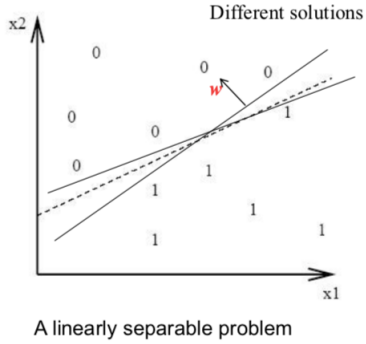
Example: linear separator on 2-dim instance space where our hypothesis space is composed by all the possible hyperplane (how the weight ($\mathbf{w}$) change).

$$\mathbf{x} = (x_1, x_2) \in \mathbb{R}^2 \ , \ f(x) = 0/1 \text{ (or -1/+1)}$$



In general the solution is not unique: there are many possible hyperplanes separating, let's see some hyperplane properties and how the hyperplane change with different $\mathbf{w}$ values.

Hyperplane properties:

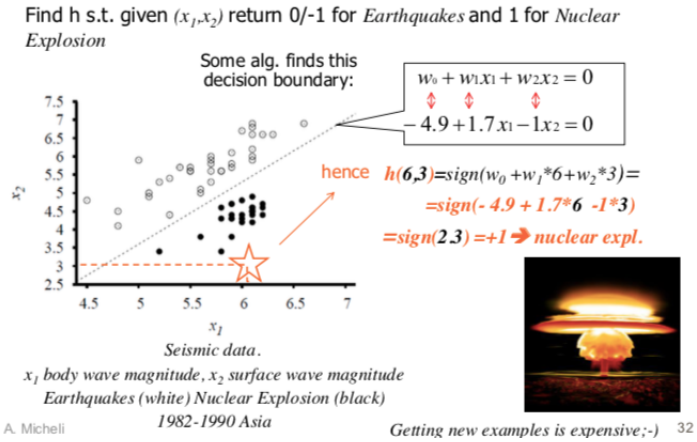


- If $w_0 = 0$ the line goes through the origin of the coordinate system.

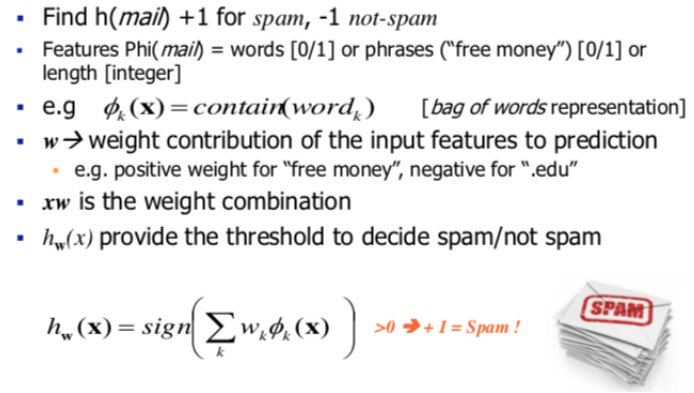
- If $n > 2 \rightarrow$ hyperplane
- Scaling freedom: the same hyperplane multiplying $\mathbf{w}$ by K
- $\mathbf{w}$ is a vector orthogonal to the hyperplane, given $\mathbf{x_a}, \mathbf{x_b}$ (belonging to the hyperplane):

$$\mathbf{x_a^T w} + w_0 = 0 \; ; \; \mathbf{x_a^T w} + w_0 = 0 \rightarrow \mathbf{w^T}(\mathbf{x_a} - \mathbf{x_b}) = 0 \rightarrow \text{orthogonal vectors}$$

Here two examples:
(AIMA) Classify a new data (x_1, x_2)



Spam



2.3 Learning Algorithms

A learning algorithm is divided into two types:

- **Eager**: Analyze the training data and construct an explicit hypothesis.
- **Lazy**: Store the training data and wait until a test data point is presented, then construct an ad hoc hypothesis to classify that one data point.

The linear model expected to construct an explict hypothesis from the training set, so the learning algorithm for the linear model is **Eager**.

We are going to introduce 2 *learning algorithms* for the regression and for the classification task using a linear model, both based on *LMS*.

- A direct approach based on **normal equation** solution
- An iterative approach based on **gradient descent**

The learning algorithm has

We start redefining the learning problem and the loss for them (for l data and multidimensional inputs).

The learning problem (classification tasks)
Given a set of l training example $(\mathbf{x}_i, y_i)$ and a loss function (measure) L, we want to **find** the weight vector $\mathbf{w}$ that minimizes the expected loss on the training data.

$$R_{emp} = 1/l \sum_{i=1}^{l} L(h(\mathbf{x}_i), y_i)$$

For classification: Using a piecewise constant (over $sign(\mathbf{w^T x})$) for the loss can make this a difficult problem because it is not continuous and differentiable. Assume we still use the least squares (as for the regression case).

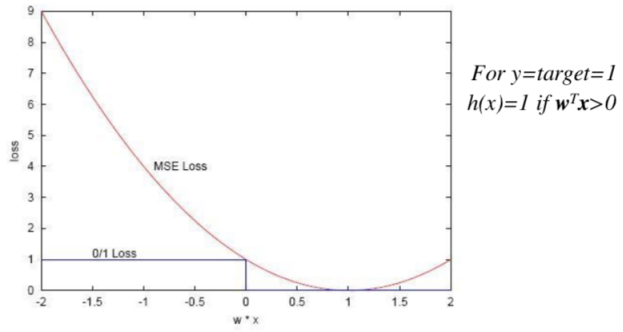


Figure 2.7: Both satisfies the minimization of error

Initially, we can make the optimization problem easier by replacing the original objective function **L** by a smooth, differentiable function. For example, consider the mean squared error. See figure 2.7.

Learning (a classifier) by Least Squares:

Given a set of l training example $(\mathbf{x}_i, y_i)$, **Find** optimal values for w (for fitting of Training data) by using the **least squares** that minimize the residual sum of squares:

$$E(\mathbf{w}) = 1/l \sum_{i=1}^{l} (y_i - \mathbf{x}_i^T \mathbf{w})^2 = \|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$$

Min error: if $y_i = 1$ then $x_i^T w \to 1$; if $y_i = 0$ then $x_i^T w \to 0$

In $E(\mathbf{W})$ we do not use $h(\mathbf{x})$, as for regression, but to hold a continuous differentiable loss (because $h(\mathbf{x}) = sign(\mathbf{wx})$ for classification) we will use $\mathbf{x}_i^T\mathbf{w}$
This is a *quadratic function* so the minimum always exists (but may be not unique).
(X is a matrix $\mathbf{l}\,\mathbf{x}\,\mathbf{n}$ with a row for each input vector $\mathbf{x}_i$).

2.3.1 Normal equation and direct approach solution

Differentiating $E(\mathbf{w})$ with respect to $\mathbf{w}$ we get the normal equation (point with gradient of E w.r.t $\mathbf{w} = 0$):

$$(\mathbf{X^T X})\mathbf{w} = \mathbf{X^T y}$$

Proof. Now we want to compute $\frac{\partial E(\mathbf{w})}{\mathbf{w}_j} = 0 \;\;,\;\; j = 0, \ldots, n$

$E(\mathbf{w}) = \sum_{i=1}^{l}(y_i - \sum_{t=0}^{n} w_t x_{i,t})^2 = \sum_{i=1}^{l}(\delta_i(\mathbf{w}))^2$ (l patterns)

$$\frac{\partial E(\mathbf{w})}{\mathbf{w}_j} = 2\sum_{i=1}^{l}\delta_i(\mathbf{w})\frac{\partial(\delta_i(\mathbf{w}))}{\partial \mathbf{w}_j} = 2\sum_{i=1}^{l}\delta_i(\mathbf{w})\frac{\partial(y_i - \sum_{t=0}^{n} w_t x_{i,t})}{\partial \mathbf{w}_j} =$$

$$= 2\sum_{i=1}^{l}\delta_i(\mathbf{w})(-x_{i,j}) = -2\sum_{i=1}^{l}x_{i,j}\delta_i(\mathbf{w}) = -2\sum_{i=1}^{l}x_{i,j}(y_i - \sum_{t=0}^{n} w_t x_{i,t}) = 0$$

$$\sum_{i=1}^{l}x_{i,j}y_i = \sum_{i=1}^{l}\sum_{t=1}^{n} w_t x_{i,t} x_{i,j} \;\;,\;\; j = 0, \ldots, n$$

$$\mathbf{X^T y} = (\mathbf{X^T X})\mathbf{w}$$

□

So, if $(\mathbf{X^T X})$ is not singular the unique solution is given by:

$$\mathbf{w} = (\mathbf{X^T X})^{-1}\mathbf{X^T y} = \mathbf{X}^{+}\mathbf{y}$$

Else the solution are infinite (satisfying the normal equation):

we can choose the min norm (w) solution.

note: ($\mathbf{X}^+$ is the Moore-Penrose pseudoinverse also if X is not invertible)

Direct approach by SVD:
The Singular Value Decomposition can be used for computing the pseudoinverse of a matrix:

$$\boldsymbol{X} = \boldsymbol{U}\underset{\text{diagonal}}{\boldsymbol{\Sigma}}\boldsymbol{V}^T \Rightarrow \boldsymbol{X}^+ = \boldsymbol{V}\boldsymbol{\Sigma}^+\boldsymbol{U}^T$$
($\boldsymbol{\Sigma}^+$: by replacing every nonzero entry by its reciprocal)

Moreover we can apply directly SVD to compute $\mathbf{w} = \mathbf{X}^{+}\mathbf{y}$ obtaining the minimal norm (on w) solution of least squares problem.

2.3.2 Gradient descent

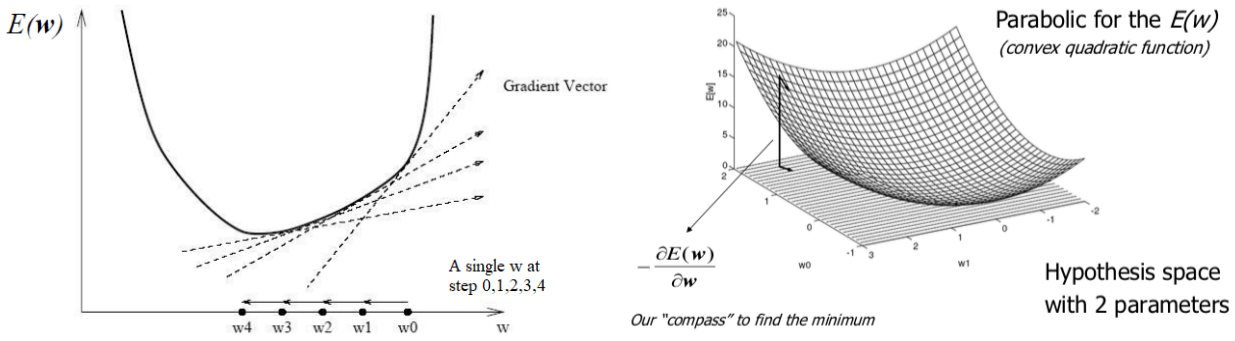
An iterative approach based on *gradient descent*.
Previous derivation suggest the line to construct an iterative algorithm based on :

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}_j} = -2\sum_{i=1}^{l}(y_i - \mathbf{x}_i^T\mathbf{w})(\mathbf{x}_i)_j$$

Where

- $\mathbf{x}_i$: i-th input pattern
- y_i: the output for p
- $\mathbf{w}$: free parameters
- l: numbers of examples
- $(\mathbf{x}_i)_j$: component j of pattern i

Gradient (ascent direction): we can move toward the minimum with a gradient descent (- gradient of $E(\mathbf{w})$).
Local search: begins with initial weight vector. Modifies it iteratively to decrease up to minimize the error function (steepest descent).



(a) Gradient descent

(b) H for 2 –w linear model

Figure 2.8: Gradient

Delta Rule
A training rule whose key idea behind the delta rule is to use gradient descent to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples.
The ≪movements≫ will be made iteratively according to:

$$\mathbf{w}_{new} = \mathbf{w}_{old} + eta \times \Delta\mathbf{w}$$

where eta is the step size, $0 < eta < 1$ or a value

learning rate (η) - eta =: speed/stability trade-off and is the step size: can be (gradually) decreased to zero (guarantee convergence, avoiding oscillation around the min.): many variants will be introduced later.

In figure 2.9 is shown how the gradient descent change with different learning rate and figure 2.11 is shown how the learning curve can change.

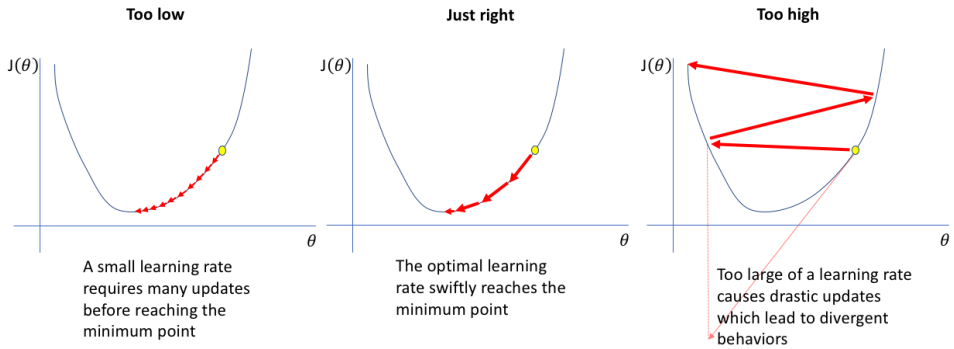
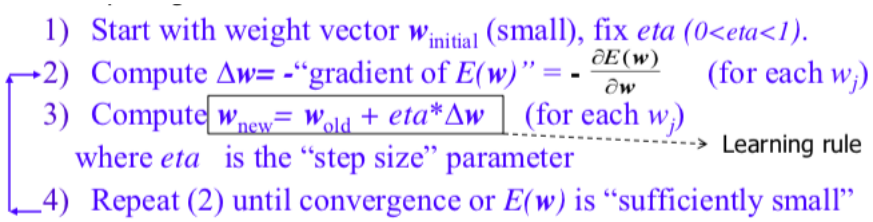


Figure 2.9: Learning rate

Gradient descent algorithm



In the stardard case we use $\Delta w/l$: least mean squares (when you divide by l). It's good to normalize the value of the graident, otherwise could be too large and could lead to oscillations (bad learning curve)

There are two versions of the algorithm:

- For **batch version** the gradient is the sum over all the l patterns (using p):

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}_i} = -2\sum_{p=1}^{l}(y_p - \mathbf{x}_p^T\mathbf{w})x_{p,i}$$

  So, to calculate the gradient of the cost function, we need to sum the cost of each pattern. If we have 3 million patterns, we have to loop through 3 million times or use the dot product.
  Note that:

  - $x_{p,i}$ is the component i of pattern p
  - 2 is constant that can be ignored to develop the algorithm
  - we typically use *LMS* (use $1/l$ in front of the sum)
  - provide a more "precise" evaluation of the gradient over a set of l data
  - We upgrade the weights after this sum

- For the **on-line/stochastic version** we upgrade the weights with the error that is computed for each pattern.

$$\frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}_i} = -2(y_p - \mathbf{x}_p^T\mathbf{w})x_{p,i}$$

  In SGD weights are updated upon examining each training example so, we only use 1 example for each learning step. Because it's using only one example at a time, its path to the minima more random than that of the batch gradient. Usually, before for-looping, you need to randomly shuffle the training examples.

- We will see intermediate cases later (as **mini-batch**): Mini-batch gradient descent uses n data points (instead of 1 sample in SGD) at each iteration.

Here an example showing how the trajectory change with different gradient descent algorithm:

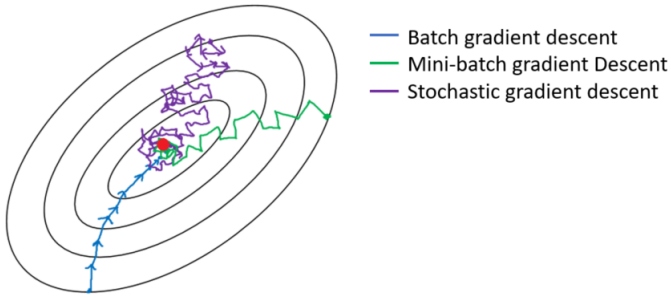


Figure 2.10: Gradient descent variants trajectory towards minimum

Learning curve examples: the curve can change with different gradient algorithm and different value of the learning rate:

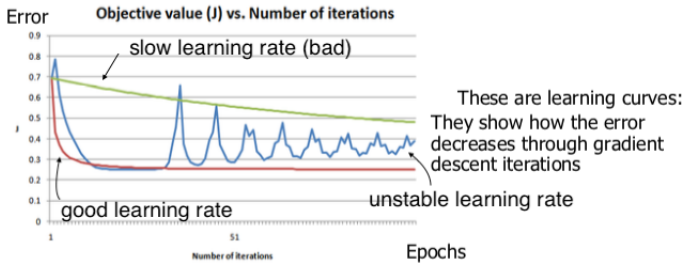


Figure 2.11: Learning curve examples

Gradient descent as Error correction rule

$$\Delta w_j = \sum_{p=1}^{l} (y_p - \mathbf{x}_p^T \mathbf{w})(\mathbf{x}_p)_j$$

This is an "error correction" rule (Widrow-Hoff) that change the w proportionally to the error (target-output):
For example:

- $(target - output) = err = 0 \rightarrow$ no correction
- $(input_j > 0)\ if\ err\ +$ (output is too low) :

$$increase\ w_j \rightarrow\ increase\ output \rightarrow less\ err$$

- $(input_j > 0)\ if\ err\ -$ (output is too high) :

$$decrease\ w_j \rightarrow\ reduce\ output \rightarrow less\ err$$

- $(input_j < 0)\ if\ err\ +$ (output is too low) :

$$decrease\ w_j \rightarrow\ increase\ output \rightarrow less\ err$$

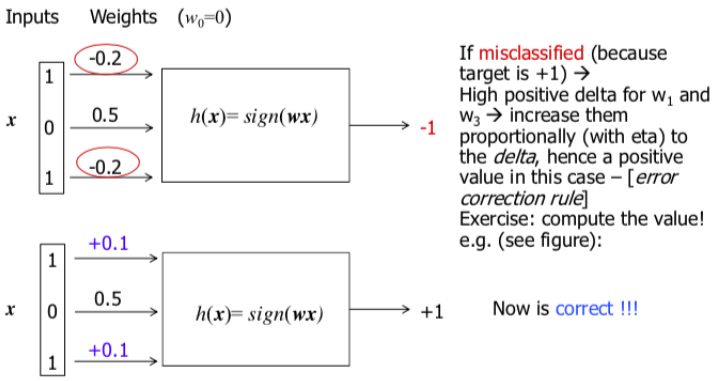
- $(input_j < 0)\ if\ err\ -$ (output is too high) :

$$increase\ w_j \rightarrow\ reduce\ output \rightarrow less\ err$$

It allow us to search through an *infinite hypothesis* space and it can be easily always applied for *continues* H and differentiable loss.

NOTE: Is the gradient descent efficient? Many improvement are possible: Newton and quasi-newton methods, Conjugate Gradient, ...

Here and example of Delta-W as Error Correction Learning rule:



Summarizing

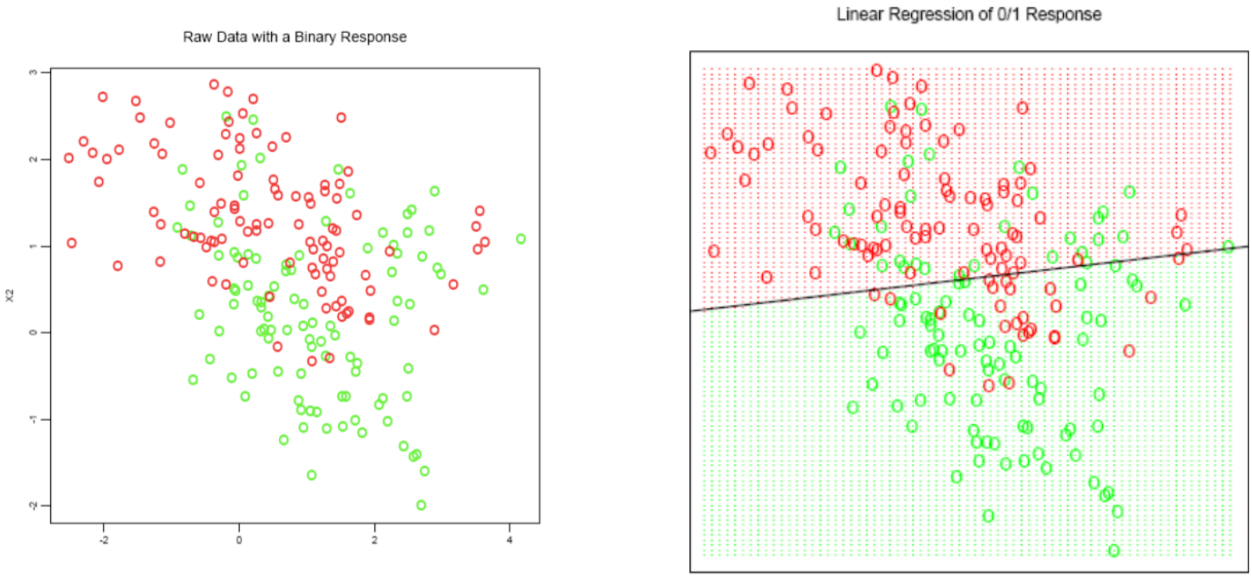
- Model trained (on tr set) with LS (LMS) on **wx** by the simple gradient descent algorithm used for linear regression
- Model used for classification applying the threshold function $h(\mathbf{x}) = sign(\mathbf{wx})$
- The error can be computed as classification error or number of misclassified patterns (not only by the Mean Square Error)

$$L(h(\mathbf{x}_p), d_p) = \begin{cases} 0 & \_if \quad h(\mathbf{x}_p) = d_p \\ 1 & \_\_\_\_\_otherwise \end{cases} \qquad mean\_err = \frac{1}{l} \underbrace{\sum_{i=1}^{l} L(h(\mathbf{x}_i), d_i)}_{num\_err}$$

- ACCURACY = mean of correctly classified = $(l - num\_err)/l$

2.4 Linear model on a classification problem

Let's analyze the problem shown in figure 2.12a problem and let's apply the linear model on the training data set.



(a) Raw data

(b) After LMS algorithm

Figure 2.12: Solution of a classification example with LMS

A classification example in two dimensions. The classes are coded as a binary variable. In figure 2.12b we can see the result of the linear model: The line is the decision boundary defined by $\mathbf{x^T w} = 0.5$.

$$h(\mathbf{x}) = \begin{cases} 1 & if \ \ \mathbf{x^T w} > 0.5 \\ 0 & otherwise \end{cases}$$

The decision boundary is $\{\mathbf{x} \mid \mathbf{x^T w} = 0.5\}$ is linear (and seems to make many errors on the training data). Is it true?

Good or bad approximation: Possible scenarios (we know the true target function!):

- Scenario 1: The data in each class are generated from a Gaussian distribution with uncorrelated components, same variances, and different means.

  In this case ⇒ the linear regression rule (by LS) is almost optimal (is the best one can do). The region of overlap is inevitable (due to errors in the input data).

- Scenario 2: The data in each class are generated from a mixture of 10 gaussians in each class.

  In this case ⇒ the linear model is far too rigid: next models for it!

2.5 Linear regression (in statistics)

In statistics, **linear regression** is used for two things:

- to construct a simple formula that will predict a value or values for a variable given the value of another variable.
- to test whether and how a given variable is related to another variable or variables.

Statistical Parametric models: note that, in general,"**linear**" does not refer to this straight line, but rather to the way in which the regression coefficients occur in the regression equation. (See next for "linear basis expansion")

NOTE: Least squares corresponds to the maximum likelihood criterion if the experimental errors have a normal distribution.

2.6 Linear model (in ML): Inductive Bias (alla Mitchell)

In the Linear model we have two Bias:

- **Language bias**: the H is a set of linear functions (may be very restrictive and rigid)
- **Search bias**: ordered search guided by the Least Squares minimization goal: for instance we could prefer a different method to obtain a restriction on the values of parameters, achieving a different solutions with other properties...

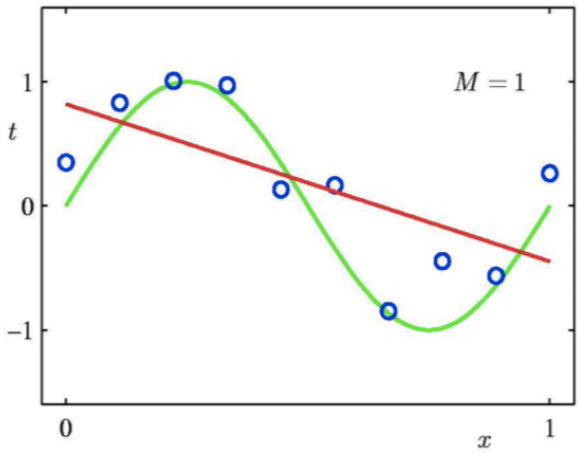
It show that even for a "simple" model there are many possibilities. We still need a principled approach! (see theory of ML)...

2.7 Limitations

This linear model has some limitation: It's a **rigid model**.

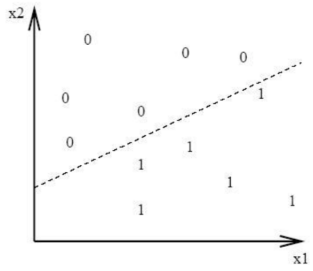
■ **Regression tasks for non linear problems**

In this case we have a very poor solution, a linear model can't fit a non linear problem! we have to find a solution for it!

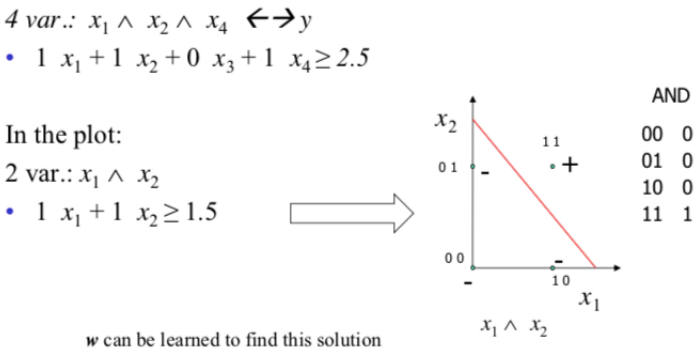


■Classification tasks

- In geometry, two set of points in a two-dimensional plot are **linearly separable** when the two sets of points can be completely separated by a single line.
- In general, two groups are linearly separable in $n-dimensional$ space if they can be separated by an $(n-1)-dimensional$ hyperplane.
- The linear decision boundary can provide exact solutions only for linearly separable sets of points.

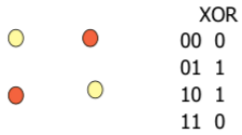


Example: Conjunctions
We can represent conjunctions by the linear models:



Example: 3 point
Given 3 points, we can find a separation plane only if they are not aligned, otherwise no (3 aligned points with 0 in the middle and others 1):

Example: 4 points
Given 4 points we can't always find a separtion plane (we can find a labeling such that the linear classifier fails to be perfect), e.g xor:



2.8 A generalization

The linear model is a very rigid model, here we will show a basis expansion. This allow to get more **Flexibility**.

■**Regression task**
Basis transformation(**linear basis expansion**) :

$$h_w(\mathbf{x}) = \sum_{k=0}^{K} w_k \phi_k(\mathbf{x})$$

Augment the input vector with additional variables ($K > n$) which are transformations of $\mathbf{x}$ according to a function $\phi_k : R^n \to R$. ϕ can be every function.

E.g:

- Polynomial representation of x: $\phi(x) = x_j^2$ or $\phi(x) = x_j x_i$, or other functions
- Non-linear transformation of single inputs: $\phi(x) = log(x_j)$, $\phi(x) = root(x_j)$, ...
- Non-linear transformation of multiple input: $\phi(x) = \|x\|$
- Splines, ...

This generalization is called: **dictionary** approaches and the model is linear in the parameters (also in phi, not in x!), so we can use the same learning algorithm as before!

Example:

- 1-dim x : $\phi_j(x) = x^j$ ($1 - dim$ polynomial regression ($K = M$)):

$$h(\mathbf{x}) = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

- any other, $\phi(\mathbf{x}) = \phi([x_1, x_2, x_3])$:

$$h(\mathbf{x}) = w_1 x_1 + w_2 x_2 + w_3 log(x_2) + w_4 log(x_3) + w_5 (x_2 x_3) + w_0$$

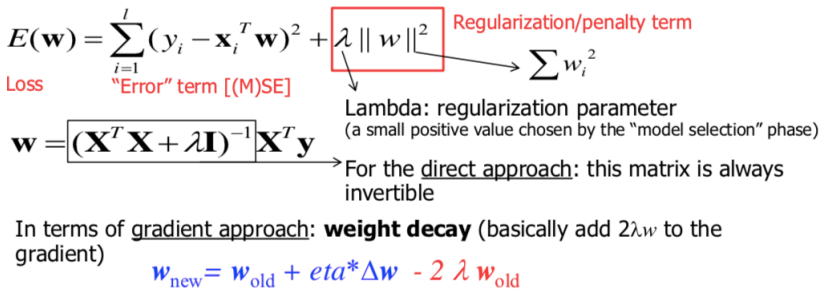
For *classification task* we add sign before the sum.

- **PROS**: Can model more complicated relationships (than linear). It is more expressive with more flexibility.
- **CONS**: With *large* basis of functions, we easily risk *overfitting*, hence we require methods for *controlling the complexity* of the model. Furthermore:
  - Phi are fixed before observing training data, it's hard to decide which ϕ will be good for our data! (versus adaptive /non-linear in parameters e.g. in Neural Network)
  - *Curse of dimensionality* (the volume of the problem space increases so fast that the available data become sparse).

2.9 Improvements

2.9.1 How to control model complexity? Regularization

■Ridge regression (Tikhonov regularization)
smoothed model → e.g. lower variance



Lambda (λ) regularization parameter:

$$0 \leq \lambda < 1$$

Possible to add constraints to the sum of value of $|w_j|$ favouring "sparse" models e.g. with less terms due to weights $w_j = 0$ (i.e. less complex solutions): ridge regression [or lasso, et al. (by different norms)]

- The penalty term *penalizes high value of the weights* and tends to drive all the weights to smaller values (some weights values can go even to zero).
- It *implements* a control of the model complexity
- This leads to a model with *less VC-Dim.*
- λ values can rule the underfitting/overfitting cases.

Tikhonov regularization can be applied for every function and allow us to control the VC-dim. It's better to have a big H space and then control the search bias with λ.

■Example of control of model complexity:
Let's suppose to have a regression task and we will use this $h(\mathbf{x})$:

$$h(\mathbf{x}) = w_0 + w_1 x + w_2 x^2 + \cdots + w_M x^M = \sum_{j=0}^{M} w_j x^j$$

$$x : \phi_j(x) = x^j (1 - dim \text{ polynomial regression } (M = 9))$$

Let's see how we can control the complexity of the model:

- in figure 2.13a we can see the case where $\lambda = 0$ and there is no regularization. In this case how $h(\mathbf{x})$ is in overfitting and the error $E(w) = 0$ on the training set. This model is too complex and is able to fit the noise too. (we have to control the complexity of the model in this case
- in figure 2.13b we use a small positive lambda ($\lambda = 0.0000000152$. The model has been regularized it and the VC-dim has been reduced too. This model seems to works much better.
- in figure 2.13c we use a very high lambda, close to 1 ($ln\lambda = 0$) and the model goes in underfitting.

From the three previous cases we understand that we need a trade-off and we need to find the right lamda value that does not make us go underfitting or overfitting.

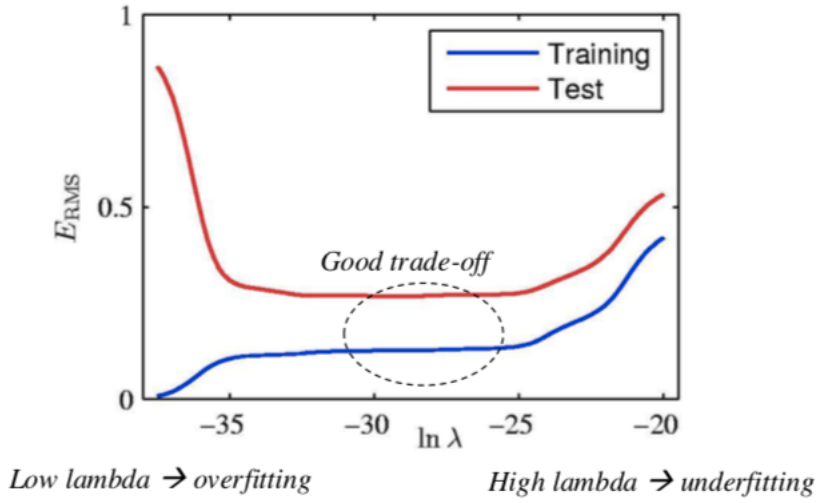


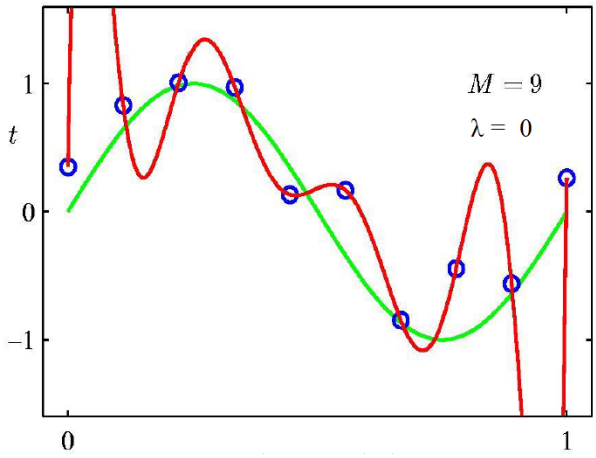
Figure 2.14: Regularization: E_{RMS} vs $ln\lambda$

In figure 2.14 we can see how the E_{RMS} change with different value of λ and figure 2.15 shows how the value of weights change with different value of λ.

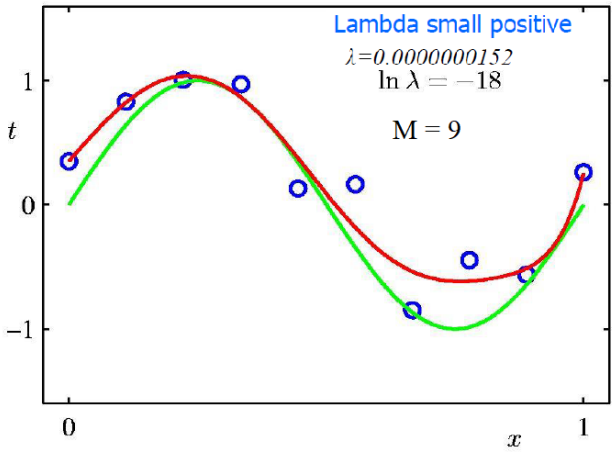
0 lambda ↖ ↗ Very high lambda

	$\ln\lambda = -\infty$	$\ln\lambda = -18$	$\ln\lambda = 0$
$w_0^\star$	0.35	0.35	0.13
$w_1^\star$	232.37	4.74	-0.05
$w_2^\star$	-5321.83	-0.77	-0.06
$w_3^\star$	48568.31	-31.97	-0.05
$w_4^\star$	-231639.30	-3.89	-0.03
$w_5^\star$	640042.26	55.28	-0.02
$w_6^\star$	-1061800.52	41.32	-0.01
$w_7^\star$	1042400.18	-45.95	-0.00
$w_8^\star$	-557682.99	-91.53	0.00
$w_9^\star$	125201.43	72.68	0.01

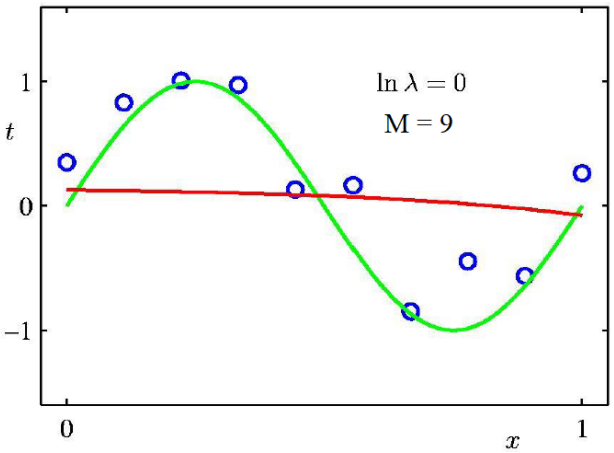
Figure 2.15: Polynomial Coefficients



(a) Lambda = 0



(b) $ln\lambda = -18$: lambda small positive



(c) $ln\lambda = 0$: very high lambda, close to 1

Figure 2.13: 9th Order Polynomial with control of complexity

2.9.2 Other Regularization Technology for Linear Models

There is other technology used for regularization that we will introduce in later sections:

- Ridge regression ($\|\cdot\|_2$)
- Lasso ($\|\cdot\|_1$)
- Elastic nets (use both $\|\cdot\|_1$ and $\|\cdot\|_2$)

The L2 norm penalizes the square value of the weight and tends to drive all the weights to smaller values. On the other hand, the L1 norm penalizes the absolute value of the weight and tends to drive some weights to exactly zero (while allowing some weights to be large) → toward feature selection!

NOTE: Unfortunately $\|\cdot\|_1$ (absolute value) introduce a non differentiable so the loss needs other approaches.

2.9.3 Others Improvements

- Inputs with added noise (data augmentation)

- Derived inputs: a small number of new variables is used in place of the x inputs, which are a linear combination of x.
  - *Principal Component Regression*
  - *Partial Least Squares*

2.10 Multi-class task

There is two very simple approaches for multi-class:

- **OVA (one-vs-all)**: a discriminant function for each class, built on top of real-valued binary classifiers:
  - train K different binary classifiers, each one trained to distinguish the examples in a single class from the examples in all remaining classes.
  - to classify a new example, the K classifiers are run, and the classifier which outputs the largest (most positive) value is chosen.

$$h(\mathbf{x}) = \arg\max_i h_i(\mathbf{x})$$

  e.g. Class 1-of-K rep: red, green, blue → (0,0,1), (0,1,0), (1,0,0). → solve 3 linear models.

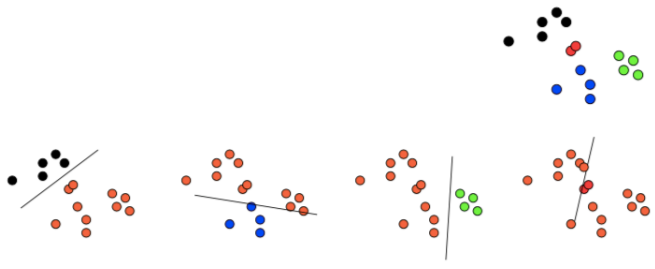
- **AVA (all-vs-all = one-versus-one)**: Each classifier separates a pair of classes. Build $K(K-1)$ classifiers K-way multiclass problem, one classifier to distinguish each pair of classes i and j.
  - to classify a new example, all the classifiers are run, and the winner is the one with the max sum of outputs versus all the other classes OR the class with most votes

$$h(\mathbf{x}) = \arg\max_i \left(\sum_j h_{ij}(\mathbf{x})\right)$$

  - training data set for each classifier is much smaller

OVA, AVA suffers from ambiguities in that some regions of its input space may receive the same number of votes.

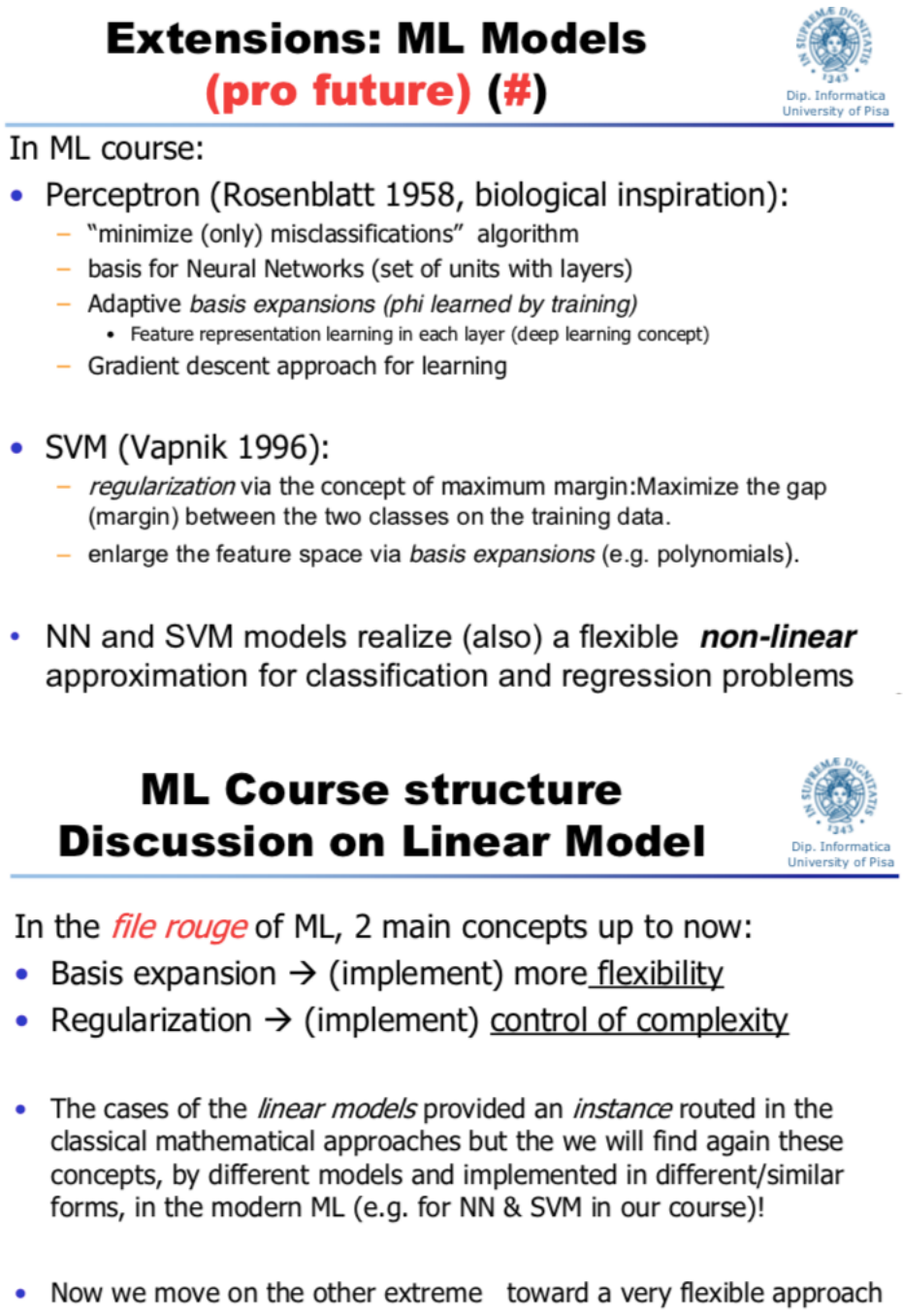
Criticism: The problem can become not easy



- Masking: classes can be masked by others (for high K)
- A wide array of more sophisticated approaches for multiclass classification exists ...
- Some models can deal directly with multi-output

2.11 Other learner models for classification

- Linear Discriminant Analysis (also multi-class)
- Logistic regression
  $P(y|\mathbf{x})$ starting from modeling the class density as a know density



3 K-Nearest Neighbor Model

In the previous section we analyzed a quite rigid model, now we will move to the K-Nearest Neighbor (k-nn) model, a very flexible approach.

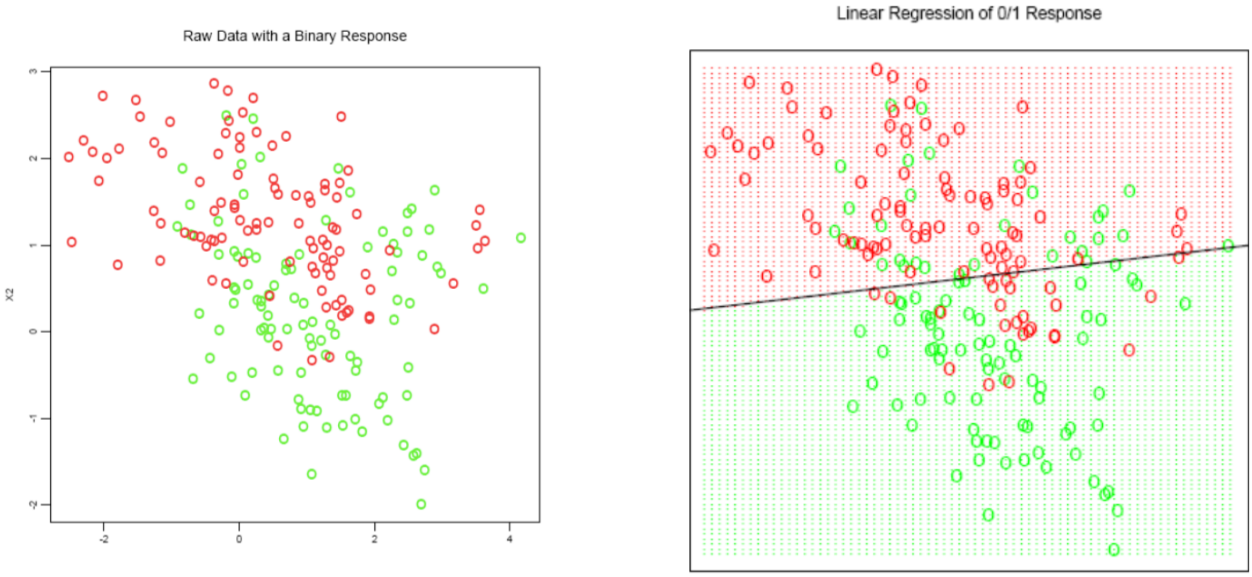
The k-nearest neighbors algorithm (k-NN) is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether k-NN is used for classification or regression. More in details:

K-nn is a **Lazy** learning algorithm, it store the training data and wait until a test data point is presented, then construct an ad hoc hypothesis to classify that one data point.

- It makes **local estimations** (by locally constant functions) and has not global hypothesis for all the instances, no model to be fit
- we need to memorize the input examples

This method is in also called **distance-based methods**, memory based or instance-based.

Let take once again the example used for classification, we have seen that applying the linear model seems to make many errors on the training data. let's see what happens when applying a more flexible method like the **K-nn**.



(a) Raw data

(b) After LMS algorithm

Figure 3.1: Solution of a classification example with LMS

Let's start to analyze the 1-kk case.

3.1 1-Nearest Neighbor Algorithm

The algorithm for the 1-nn case is:

1. Simply store the training data $< \mathbf{x}_j, y_j >,\ j = 1, \ldots, l$
2. given an input $\mathbf{x}$ (with dimension n)

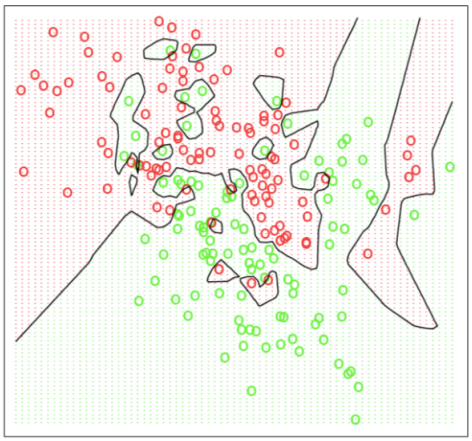


Figure 3.2: 1-Nearest Neighbor Classifier

3. now we want to find the nearest training example $\mathbf{x}_i$:
   find the index i s.t. we have the min $d(\mathbf{x}, \mathbf{x_i}) \rightarrow i(\mathbf{x}) = \arg\min_j d(\mathbf{x}, \mathbf{x_j})$

   e.g. Euclidian distance: $d(\mathbf{x}, \mathbf{x_j}) = \sqrt{\sum_{t=1}^{n}(x_t - x_{jt})^2} = \|\mathbf{x} - \mathbf{x}_j\|$ (pattern $\mathbf{x_j}$, component t)

4. then output y_i

By applying the algorithm to the problem above we can see how this new model works. The result of the 1-nn algorithm is shown in fig. 3.2

We can immediately notice that:

- the model is very flexible
- there is No misclassifications in TR data: 0 training error. What about the test data?
- Decision boundaries is **not linear**: it is quite **irregular**
- May be unnecessary noisy

3.2 K-Nearest Neighbors Algorithm

The algorithm is the same for the 1-nn, here we will extend it to the k-nn case. The output depends on whether k-NN is used for classification or regression:

- **Regression Case**: take the average.

$$avg_k(\mathbf{x}) = 1/k \sum_{\mathbf{x}_i \in N_k(\mathbf{x})} y_i$$

  where $N_k(\mathbf{x})$ is a neighborhood of $\mathbf{x}$ that contains exactly k neighbors (closest patterns according to d): k-nearest neighborhood: **K-nn**.

- **Classification Case**: If there is a clear dominance of one of the classes in the neighborhood of an observation $\mathbf{x}$, then it is likely that the observation itself would belong to that class, too. Thus the classification rule is the **majority voting** among the members of $N_k(\mathbf{x})$:

$$h(\mathbf{x}) = \begin{cases} 1 & if \ \ avg_k(\mathbf{x}) > 0.5 \\ 0 & otherwise \end{cases} \quad \text{for targets } y \text{ in } \{0, 1\}$$

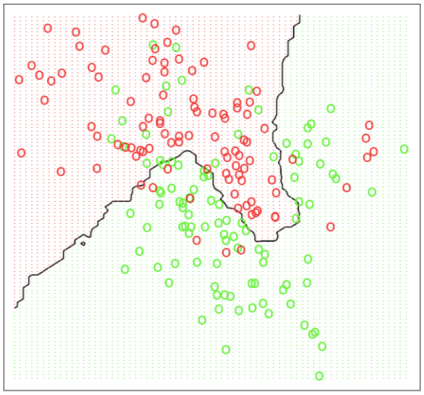


Figure 3.4: 15-Nearest Neighbor Classifier

So, a natural way to classify a new point is to have a look at its neighbors. Let's see two example:

■ **1-nn vs 5-nn example**
with two different value of k we can have different classifications (fig. 3.3:

- 1-nn return $+$ for $\mathbf{x}_q$
- 5-nn return $-$ for $\mathbf{x}_q$

Smoothing over a set of neighbors for noisy data.

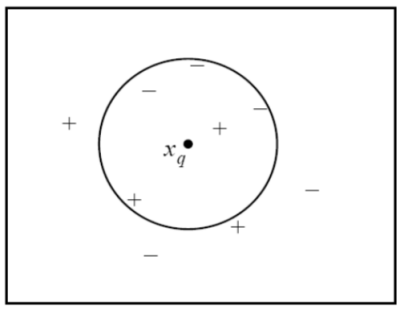


Figure 3.3: 1-nn vs 5-nn example

■ **15-nn example**
We can see the result of 15-nn algorithm. Where the predicted class is hence chosen by majority vote amongst the 15-nearest neighbors.

- Still very flexible
- Some misclassifications in training data
- Decision boundaries is not linear: it is still quite, although less, irregular
- Decision boundary adapts to the local densities of the classes

■ **Voronoi diagram**
The Voronoi diagram is implicitly used by K-NN. Each cell consisting of all points closer to $\mathbf{x}$ than to any other patterns. The segments of the Voronoi diagram are all the points in the plane that are equidistant to two patterns. See fig. 3.5

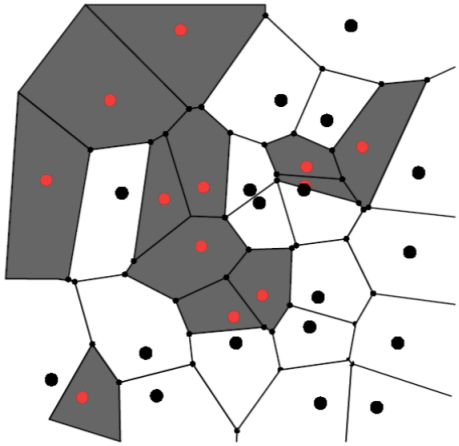


Figure 3.5: Voronoi diagram

3.3 K-nn for multi-class

The algorithm is the same used for classification but return the class most common amongst its k nearest neighbors

$$h(\mathbf{x}) = \arg\max_{v} \sum_{\mathbf{x}_i \in N_k(\mathbf{x})} \mathbf{l}_{v,y_i}$$

$$\mathbf{l}_{v,y_i} = \begin{cases} 1 & if \;\; v = y_i \\ 0 & otherwise \end{cases}$$

3.4 K-nn variants: Weighted distance

It can be useful to weight the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones.

$$h(\mathbf{x}) = \arg\max_{v} \sum_{\mathbf{x}_i \in N_k(\mathbf{x})} \mathbf{l}_{v,y_i} \bullet \frac{1}{d(\mathbf{x}, \mathbf{x}_i)^2}$$

$$\mathbf{l}_{v,y_i} = \begin{cases} 1 & if \;\; v = y_i \\ 0 & otherwise \end{cases}$$

If $d = 0$ for a i, return y_i

3.5 K-nn versus Linear model

Now we have two different models, they are two extremes of the machine learning panorama, let's to compare them:

- The linear model is a rigid model (low variance) and the K-nn is a flexible model (high variance):
  - In K-nn with small k, few points can change the decision boundary
  - We may pay a price for this flexibility
- Eager versus lazy
- Parametric versus instance-based
- global hypothesis for all the instances versus local estimations function (all computation is deferred until classification/regression request)

Given this two scenario:

- **Scenario 1**: The training data in each class were generated from bivariate Gaussian distributions with uncorrelated components and different means.
- **Scenario 2**: The training data in each class came from a mixture of 10 low- variance Gaussian distributions, with individual means themselves distributed as Gaussian.

The linear decision boundary from least squares is very smooth, and apparently stable to fit, see fig 3.1b. It does appear to rely heavily on the assumption that a linear decision boundary is appropriate. In language we will develop shortly, it has low variance and potentially high bias. On the other hand, the k-nearest-neighbor procedures do not appear to rely on any stringent assumptions about the underlying data, and can adapt to any situation, see fig. 3.2 and fig. 3.4. However, any particular subregion of the decision boundary depends on a handful of input points and their particular positions, and is thus wiggly and unstable—high variance and low bias.

Each method has its own situations for which it works best.

■ **How many parameters for LM and K-nn**

Usually a linear regression model uses $n+1$ parameters (n.b. $\mathbf{x} \in \mathbb{R}^n$) to describe the global function that will fit the training data.
It appears that k-nearest-neighbor fits have a single parameter, the number of neighbors k, compared to the $n+1$ parameters in least-squares fits. Although this is the case, we will see that the effective number of parameters of k-nearest neighbors is l/k (l is the number of data, in *Hastie* book $l = N$) and is generally bigger than $n+1$, and decreases with increasing k.

To get an idea of why, note that if the neighborhoods were nonoverlapping, there would be N/k neighborhoods and we would fit one parameter (a mean) in each neighborhood.

The error on the training data should be approximately an increasing function of k, and will always be 0 for k = 1. It is also clear that we cannot use sum-of-squared errors on the training set as a criterion for picking k, since we would always pick k = 1.

■ **LS linear vs K-nn with various k values**

Figure 3.6 shows the results of classifying 10,000 new observations generated from the model and we can observe the misclassification curves for the simulation example used fig 3.1b, fig. 3.2 and fig. 3.4. We compare the results for least squares and those for k-nearest neighbors for a range of values of k.
Note: how we move from underfitting to overfitting moving the values of K (i.e. the rate l/k): more flexibility allows to find the best result if we control "complexity" by K:

- **Underfitting**: $l/k = 1\ ,\ k = l$, in this case the number of nearest-neighbor is equivalent to the number of the data l.
- **Overfitting**: $l/k = l\ ,\ k = 1$, in this case every sample is nearest-neighbor.

3.6 Bayes Optimal Classifier and K-NN

Bayes optimal solution (called Bayes classifier):

Let's suppose that we know the density $P(x, y)$ we can classifiy to the most probable class, usig the conditional (discrete) distribution as:

$$\text{output the class } v \text{ s.t. is max } P(v|\mathbf{x}) \quad (v\ in\ \{C_1, C_2, C_3, \ldots, C_k\})$$

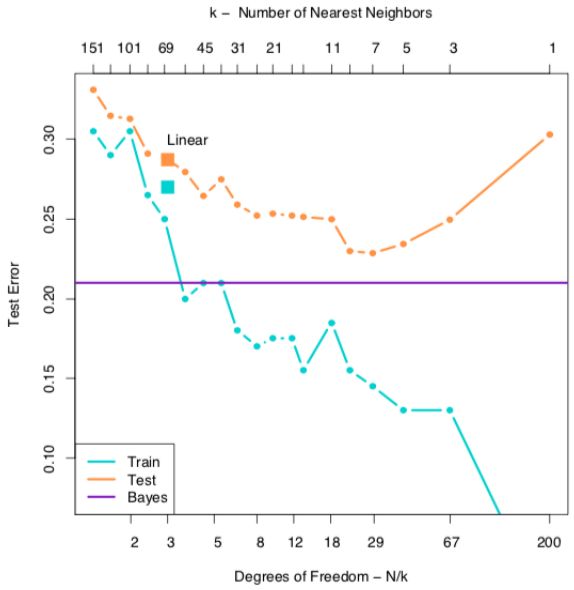
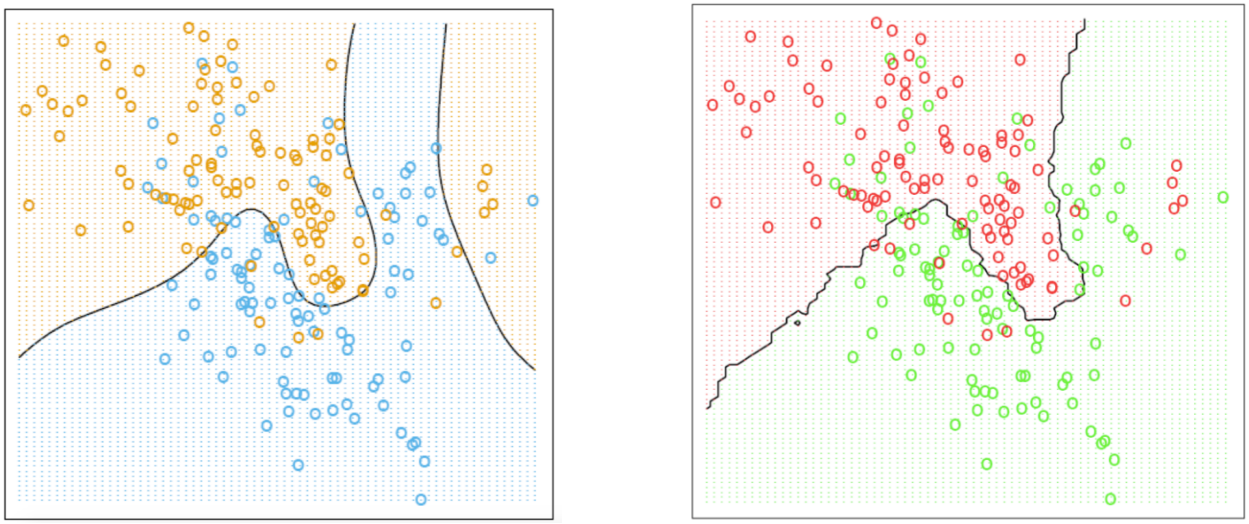


Figure 3.6: A single training sample of size 200 was used, and a test sample of size 10,000. The orange curves are test and the blue are training error for k-nearest-neighbor classification. The results for linear regression are the bigger orange and blue squares at three degrees of freedom. The purple line is the optimal Bayes error rate.

the Bayes rate is the error rate of the (optimal) Bayes classifier: the minimum achievable error rate given the distribution of the data (assuming that the generating density is known)



(a) Bayes Optimal Classifier

(b) 15-nearest neighbors

Figure 3.7: the k-nn classifier directly approximates the solution of a bayes optimal classifier

In figure 3.7a we can see the optimal Bayes decision boundary for the simulation example. Since the generating density is known for each class, this boundary can be calculated exactly.

Note on K-NN: we see that the k-nn classifier directly approximates this solution (a majority vote in a nearest neighborhood amounts to exactly this), except that conditional probability at a point is relaxed to

conditional probability within a neighborhood of a point, and probabilities are estimated by training-sample proportions.

3.7 Inductive Bias of k-nn

- The assumed distance tell us which is the most similar examples
- The classification is assumed similar to the classification of the neighbors according to the assumed metric

3.8 Criticism and Limitations

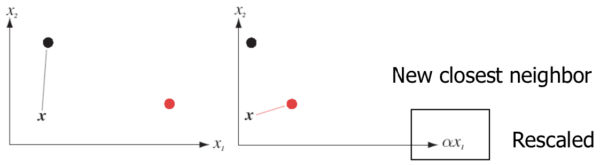
The K-nn models offer little interpretation:

- Subjectivity of interpretation
- Dependence on the metric

This model has some criticism and limitations, let's see some of them.

3.8.1 Scale changes and other metrics

- Variable scaling can have an high impact (i.e. k-nn is fragile even with respect to basic preprocessing)



  Domain knowledge dependent choice. If they should contribute equally:

  - Pay attention to disparity in the ranges of each variables
  - rescale data to equalize inputs ranges ($\rightarrow$ change the metric)! (E.g. mean zero and variance 1 normalization)

- Symbolic data require "ad-hoc" metrics (e.g. Hamming distance between two strings of equal length is the number of positions for which the corresponding symbols are different)

3.8.2 Computational cost

Note that K-nn makes the local approximation to the target function for each new example to be predicted: The computational cost is deferred to the prediction phase!

Moreover: high retrieval cost: computationally intensive for each new input: computing the distances from the test sample to all stored vectors, so the cost in time is proportional to the number of stored patterns and of course there is an high cost in term of space.

Some Optimization:

- "ad-hoc" proximity search algorithms to optimize
- E.g. by indexing the patterns

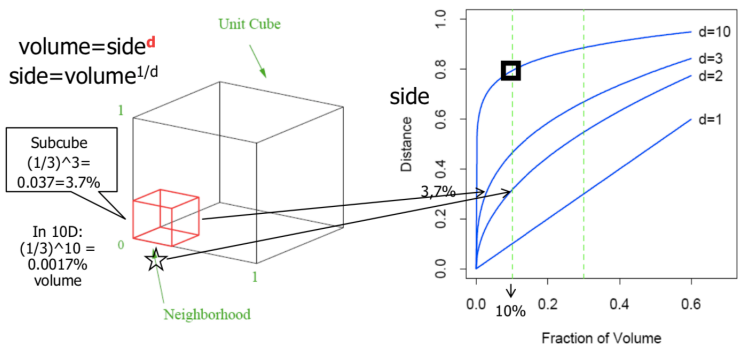


Figure 3.8: The curse of dimensionality is well illustrated by a subcubical neighborhood for uniform data in a unit cube. The figure on the right shows the side-length ($r^{1/d}$) of the subcube needed to capture a fraction r of the volume of the data, for different dimensions d. In ten dimensions we need to cover 80% of the range of each coordinate to capture 10% of the data. (since $0.1^{1/10} = 0.8$), while in 2D 30% (0.316 sidelenght) was sufficient.

3.8.3 Curse of Dimensionality

We have examined two learning techniques for prediction so far: the stable but biased linear model and the less stable but apparently less biased class of k-nearest-neighbor estimates. It would seem that with a reasonably large set of training data, we could always approximate the theoretically optimal conditional expectation by k-nearest-neighbor averaging, since we should be able to find a fairly large neighborhood of observations close to any x and average them. This approach and our intuition breaks down in high dimensions, and the phenomenon is commonly referred to as the **curse of dimensionality** (Bellman, 1961). There are many manifestations of this problem, and we will examine a few here.

1. **It is hard to find nearby points in high dimensions**:
   K-nearest neighbors can fail in high dimensions, because it becomes difficult to gather K observations close to a target (query) point $\mathbf{x}_q$:

   - near neighborhoods tend to be spatially large, and estimates are not longer local
   - Reducing the spatial size of the neighborhood means reducing K, and the variance of the estimate increases ($\rightarrow$ overfitting).

   Consider the nearest-neighbor procedure for inputs uniformly distributed in a p-dimensional unit hypercube, as in Figure 3.8. Suppose we send out a hypercubical neighborhood about a target point to capture a fraction r of the observations. Since this corresponds to a fraction r of the unit volume, the expected edge length will be $ep(r) = r^{1/d}$. In ten dimensions $e_{10}(0.01) = 0.63$ and $e_{10}(0.1) = 0.80$, while the entire range for each input is only 1.0. So to capture 1% or 10% of the data to form a local average, we must cover 63% or 80% of the range of each input variable. Such neighborhoods are no longer "local." Reducing r dramatically does not help much either, since the fewer observations we average, the higher is the variance of our fit.

   Further explanation: Image that to have K data you need 10% of the volume. How much sidelength do you need? From 30% to 80% moving from 2D to 10D.

   On the other side:

   - 1D: with 0.3 sidelength we take 30% of data volume
   - 2D: with 0.3 sidelength we take 10% of data volume (the red square)
   - 3D: with 0.3 sidelength we take 3.7% of data volume (the red cube)
   - 10D: with 0.3 sidelength we take 0.0017% of data volume.

This sidelength is not sufficient to have K data, unless we use small K (which can lead to overfitting)

2. **Low sampling density for high-dim data**
   Sampling density is proportional to $l^{1/d}$ (l data, $d = dim$).

   Another manifestation of the curse is that the sampling density is proportional to $l^{1/d}$, where d is the dimension of the input space and l is the number of samples. Thus, if $l = 100$ represents a dense sample for a single input problem and 100 points are sufficient to estimate a function in $\mathbb{R}^1$, then $l = 100^{10}$ is the sample size required for the same sampling density with 10 inputs (to achieve similar accuracy in $\mathbb{R}^{10}$ (10-dim input). Thus in high dimensions all feasible training samples sparsely populate the input space.

3. **Irrelevant features: The Curse of Noisy** if the target depends on only few of many features in x (e.g. 2 out 20), we could retrieve a "similar pattern" with the similarity dominated by the large number of irrelevant features. It grows with the dimensionality.

4. **All sample points are close to an edge of the sample (From *Hastie*** Another consequence of the sparse sampling in high dimensions is that all sample points are close to an edge of the sample. Consider N data points uniformly distributed in a d-dimensional unit ball centered at the origin. Suppose we consider a nearest-neighbor estimate at the origin. The median distance from the origin to the closest data point is given by the expression:

$$d(d, l) = \left(1 - \frac{1^{1/l}}{2}\right)^{1/d}$$

   A more complicated expression exists for the mean distance to the closest point. For $l = 500$, $d = 10$, $d(d, l) \approx 0.52$, more than halfway to the boundary. Hence most data points are closer to the boundary of the sample space than to any other data point. The reason that this presents a problem is that prediction is much more difficult near the edges of the training sample. One must extrapolate from neighboring sample points rather than interpolate between them.

3.9 An improvement

Irrelevant features:
We may weights features according to their relevance

- Stretching the axes along some dimension
- Weights can be searched by a (expensive) model selection approach or other approaches ...

Feature selection approaches: eliminates some variables → reduce input dimension

3.10 other local models in ML

- Kernel smoothers
- Local linear regression
- Prototype methods
- Case-basedreasoning

3.11 Summary: K-nn design choices

- The distance metric to measure the closeness between patterns (e.g. Euclidian, Hamming, Manhattan distance..., weights on input features). Often this is the key for a successful applications!
- K (number of neighbors: control underfitting/overfitting)

Often necessary to select:

- A subset of data (set of prototypes): e.g by clustering
- A subset of features

Various approaches to deal with the issues mentioned above.

K-nn in the course: some general lessons

- Too low variance is poor (rigid linear models), too high variance is dangerous (K-nn)

- Other concepts presented todays that are interesting in general:
  - **Smoothing** techniques (in K-nn by increasing K)
  - **Curse of dimensionality** (the volume of the problem space increases so fast that the available data become sparse)
  - Again an **instance of the Statistical Learning Theory** bound plot: see the misclassification plot moving K
  - **Inductive Bias** for K-nn (metric based issue, which is underestimated in many books)

ML Course structure And now?

- K-nn does not build a "learned model", not regularity/knowledge extraction/synthesis
  - It does not match the "learning" objective (that can forget the examples after the model is built)

- In the next lectures, we will look to model
  - Compact as the LTU model (all the knowledge in few parameters)
  - More flexible than the linear model (flexible as the K-nn)
    - with a suitable support to the control of the complexity

4 Neural Networks

In this chapter we describe a class of learning methods that was developed separately in different fields-statistics and artificial intelligence—based on essentially identical models. The central idea is to extract linear combinations of the inputs as derived features, and then model the target as a nonlinear function of these features. The result is a powerful learning method, with widespread applications in many fields.

Neural network learning methods provide a robust approach to approximating real-valued, discrete-valued, and vector-valued target functions. For certain types of problems, such as learning to interpret complex real-world sensor data, artificial neural networks are among the most effective learning methods currently known. For example, the BACKPROPAGATION algorithm described in this chapter has proven surprisingly successful in many practical problems such as learning to recognize handwritten characters (LeCun et al. 1989), learning to recognize spoken words (Lang et al. 1990), and learning to recognize faces (Cottrell 1990). One survey of practical applications is provided by Rumelhart et al. (1994).

4.1 Biological Motivation

The study of artificial neural networks (ANNs) has been inspired in part by the observation that biological learning systems are built of very complex webs of interconnected neurons. In rough analogy, artificial neural networks are built out of a densely interconnected set of simple units, where each unit takes a number of real-valued inputs (possibly the outputs of other units) and produces a single real-valued output (which may become the input to many other units).

To develop a feel for this analogy, let us consider a few facts from neuro-biology. The human brain, for example, is estimated to contain a densely inter- connected network of approximately 10^{11} neurons, each connected, on average, to 10^4 others. Neuron activity is typically excited or inhibited through connections to other neurons. The fastest neuron switching times are known to be on the order of 10^{-3} seconds–quite slow compared to computer switching speeds of 10^{-10} seconds. Yet humans are able to make surprisingly complex decisions, surprisingly quickly.

For example,it requires approximately 10^{-1} seconds to visually recognize your mother. Notice the sequence of neuron firings that can take place during this 10^{-1} second interval cannot possibly be longer than a few hundred steps, given the switching speed of single neurons. This observation has led many to speculate that the information-processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons. One motivation for ANN systems is to capture this kind of highly parallel computation based on distributed representations. Most ANN software runs on sequential machines emulating distributed processes, although faster versions of the algorithms have also been implemented on highly parallel machines and on specialized hardware designed specifically for ANN applications.

While ANNs are loosely motivated by biological neural systems, there are many complexities to biological neural systems that are not modeled by ANNs, and many features of the ANNs we discuss here are known to be inconsistent with biological systems. For example, we consider here ANNs whose individual units output a single constant value, whereas biological neurons output a complex time series of spikes

4.2 Appropriate problems for NN Learning

ANN learning is well-suited to problems in which the training data corresponds to noisy, complex sensor data, such as inputs from cameras and microphones. it is also applicable to problems for which more symbolic representations are often used, such as the decision tree learning tasks. ANN is appropriate for problems with the following characteristics:

- *Instances are represented by many attribute-value pairs.* The target function to be learned is defined over instances that can be described by a vector of predefined features, such as the pixel values. These input attributes may be highly correlated or independent of one another. Input values can be any real values.

- *The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes*

- *The training examples may contain errors.* ANN learning methods are quite robust to noise in the training data.
- *Long training times are acceptable.* Network training algorithms typically require longer training times than, say, decision tree learning algorithms. Training times can range from a few seconds to many hours, depending on factors such as the number of weights in the network, the number of training examples considered, and the settings of various learning algorithm parameters.
- *Fast evaluation of the learned target function may be required.* Although ANN learning times are relatively long, evaluating the learned network, in order to apply it to a subsequent instance, is typically very fast.
- *The ability of humans to understand the learned target function is not important.* The weights learned by neural networks are often difficult for humans to interpret. Learned neural networks are less easily communicated to humans than learned rules.

4.3 Artificial Neuron: processing unit

A neural network is composed of several neurons. Let's look at the single unit in detail.

- Neuron, node or **unit**
- *Inputs*: from extern source or other units, typically are real numbers
- *Input Connections*: **weights** $\boldsymbol{w}$. Free parameters that can be modified by learning (synaptic strength).

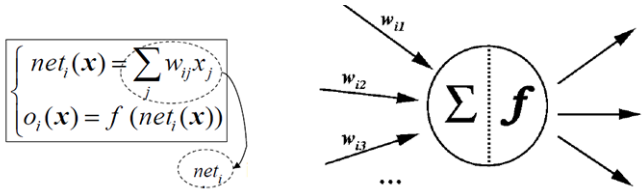


Figure 4.1: Processing unit

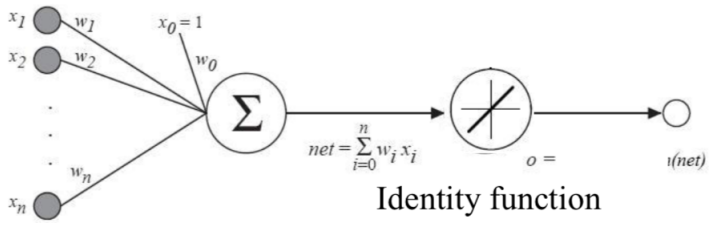
- The weighted sum net_i is called the **net input** to unit i
- Note that w_{ij} refers to the weight of the unit i, in other words from unit j to unit i (not the other way around, this is a standard notation).
- The function f is the unit's **activation function** (for example linear, LTU, ...).
- Output $o_i(\mathbf{x})$.

4.4 Neuron: Three activation functions

Changing the activation function we can have different neural networks.

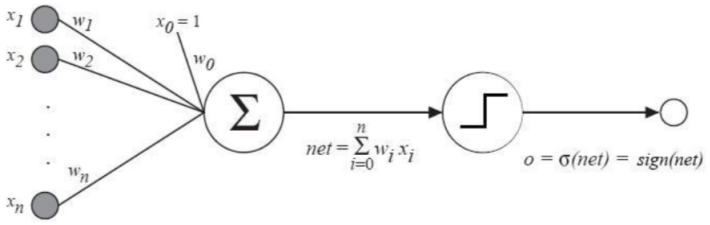
- **Linear activation function**

$$o = net = h(\mathbf{x}) = \sum_i w_i x_i$$



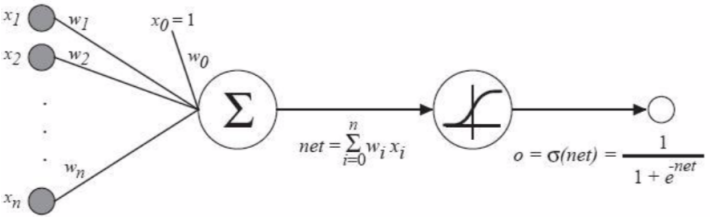
- **Perceptron** (see LTU) Threshold activation function

$$o = \sigma(net) = sign(net)$$



- **Logistic function**: sigmoid

$$o = \sigma(net) = \frac{1}{1 - e^{-net}}$$



4.5 Rosenblatt's Perceptron

In the formative years of neural networks (1943–1958), several researchers stand out for their pioneering contributions:

- McCulloch and Pitts (1943) for introducing the idea of neural networks as computing machines.
- Hebb(1949) for postulating the first rule for self-organized learning.
- Rosenblatt (1958) for proposing the perceptron as the first model for learning with a teacher (i.e., supervised learning).

Rosenblatt's perceptron is built around the McCulloch and Pitts model of a neuron (Biologically inspired model) and this is a model of historical importance for the ML, with different approaches since the early 60s. This neural networks system si base on the unit called a *perceptron* (Rosenblatt, 1958) illustrated in Figure 4.2, it is suitable for automated learning (there is a learning algorithm) and it has well-defined computational properties (Convergence theorem).

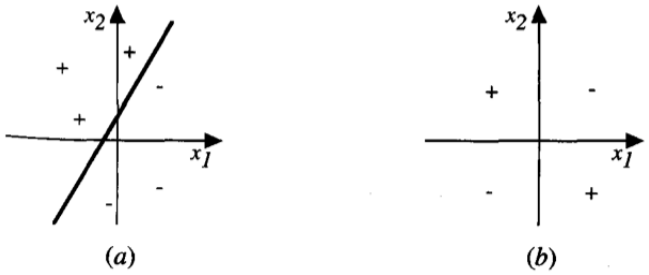


Figure 4.3: The decision surface represented by a two-input perceptron. (a) A set of training examples and the decision surface of a perceptron that classifies them correctly. (b) A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line). x l and x2 are the Perceptron inputs. Positive examples are indicated by "+", negative by "-".

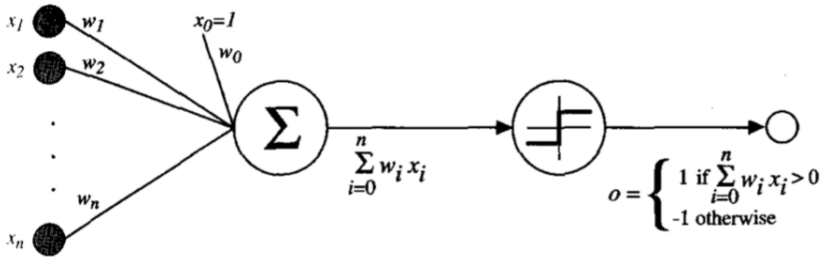


Figure 4.2: Perceptron

The *perceptron* is the simplest form of a neural network used for the classifi- cation of patterns said to be linearly separable (i.e., patterns that lie on opposite sides of a hyperplane).
It takes a vector of real-valued inputs, calculates a linear combination of these inputs, then outputs a 1 if the result is greater than some threshold and -1 otherwise. More precisely, given inputs x_l through x_n, the output $o(x_1, ..., x_n)$ computed by the perceptron is

$$o(x_1, \ldots, x_n) = \begin{cases} 1 \text{ if } w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n > 0 \\ -1 \text{ otherwise} \end{cases}$$

where each w_i is a real-valued constant, or **weight**, that determines the contribution of input x_i to the perceptron output (for brevity $o(\mathbf{x}) = sgn(\mathbf{wx})$). Notice the quantity $(-w_O)$ is a threshold that the weighted combination of inputs $w_l x_l + ... + w_n x_n$ must surpass in order for the perceptron to output a 1.

4.5.1 Representational Power of Perceptron

We can view the perceptron as representing a hyperplane decision surface in the n-dimensional space of instances (i.e., points). The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side, as illustrated in Figure 4.3.

The equation for this decision hyperplane is $wx = 0$. Of course, some sets of positive and negative examples cannot be separated by any hyperplane. Those that can be separated are called *linearly separable* sets of examples.

A single perceptron can be used to represent many boolean functions. For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights $w_0 = -8$, and $w_l = w_2 = .5$. This perceptron can be made to represent the OR function instead by altering the threshold to $w_0 = -.3$, see Figure 4.4.

A single perceptron can be used to represent many boolean functions (for example AND, OR, NAND and NOR boolean functions). The ability of perceptrons to represent AND, OR, NAND, and NOR is important because every boolean function can be represented by some network of interconnected units based on these primitives. In fact, every boolean function can be represented by some network of perceptrons only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage. One way is to represent the boolean function in disjunctive normal form (i.e., as the disjunction (OR) of a set of conjunctions (ANDs) of the inputs and their negations). Note that the input to an AND perceptron can be negated simply by changing the sign of the corresponding input weight. Because networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units

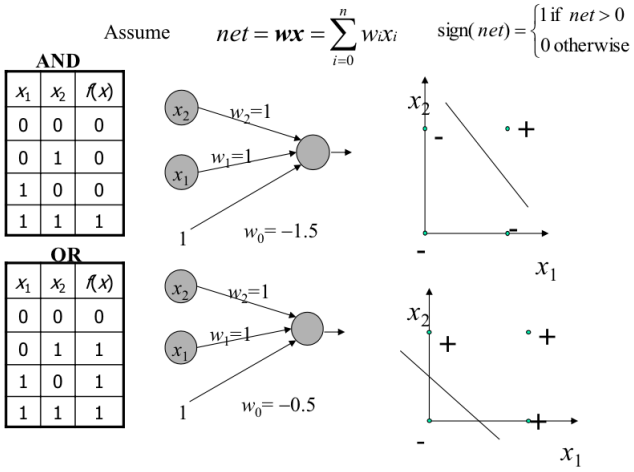


Figure 4.4: AND, OR Boolean functions

Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function.

Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if $x_l \neq x_2$. Note the set is linearly nonseparable.

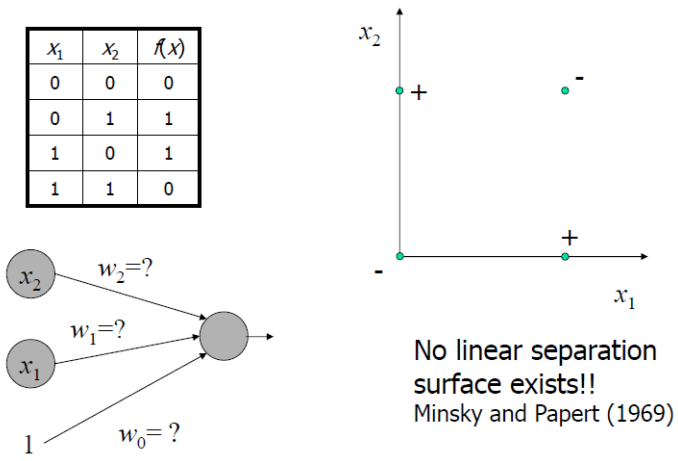


Figure 4.5: Exclusive OR (XOR)

To represent the XOR we can use a *two layers network*, where h_1, h_2 are **hidden layers**.

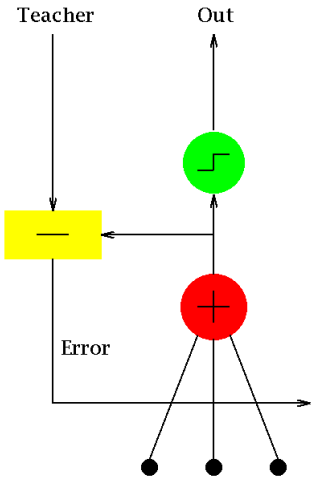


Figure 4.7: Learning inside a single layer ADALINE

$$x_1 \oplus x_2 = x_1 \cdot \overline{x_2} + \overline{x_1} \cdot x_2 \qquad \text{let}: \quad h_1 = x_1 \cdot x_2 \quad (\text{dot} = \text{AND})$$
$$h_2 = x_1 + x_2 \quad (+ = \text{OR})$$
$$\text{then we have}: \; x_1 \oplus x_2 = \overline{h_1} \cdot h_2$$

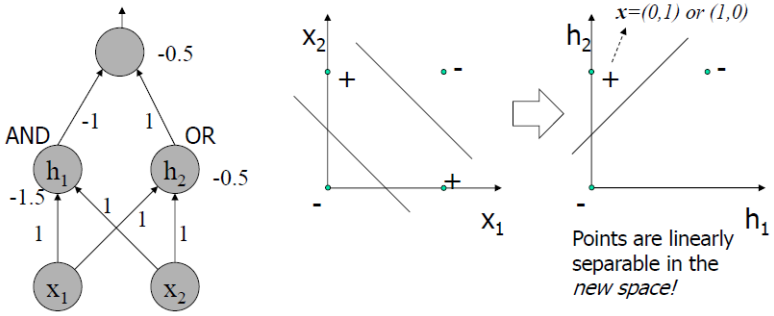


Figure 4.6: XOR by a Two Layers Network

4.6 Learning for one unit model

Two kinds of method for learning

- **ADALINE (Adaptive Linear Neuron)** is an early single-layer artificial neural network (Widrow, Hoff). It is based on the McCulloch–Pitts neuron. It consists of a weight, a bias and a summation function. See Figure 4.7.

  This net is linear, LMS direct solution and gradient descent solution. Adeline infact is in also know as Delta rule, Least mean square or Widrow- Hoff.

  - Regression tasks: See the LMS algorithm
  - For classification: See the LTU and LMS algorithm of a previous lectures (on linear model)
  - An approach that we will generalize to MLP

- **Perceptron**: (Rosenblatt): non-linear (with hard limiter or Threshold activation function)

  - Only classification: can rappresent a linear decision boundaries (see LTU) of only linearly separable problems and is always able to learn it: convergence theorem

The difference between Adaline and the standard (McCulloch–Pitts) perceptron is that in the learning phase, the weights are adjusted according to the weighted sum of the inputs (the net). In the standard perceptron, the net is passed to the activation (transfer) function and the function's output is used for adjusting the weights.

4.6.1 The Perceptron Learning Algorithm

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct output (± 1) for each of the given training examples:

- find $\mathbf{w}$ such that $sign(\mathbf{wx}) = d$
- On-line algorithm: a step can be made for each input pattern

1. Initialize the weights (either to zero or to a small random value)
2. pick a learning rate η (this is a number between 0 and 1)
3. until stopping condition is satisfied (e.g. weights don't change):

For each training pattern $(\boldsymbol{x}, d)$ $(d = +1 \text{ or } -1)$:

- compute output activation $out = sign(\boldsymbol{w}\,\boldsymbol{x})$ [+1,-1]
- If $out = d$, don't change weights I.e. "minimize (only) misclassifications"
- If $out \neq d$, update the weights:
  - $\boldsymbol{w}_{new} = \boldsymbol{w}_{old} + \eta\ d\,\boldsymbol{x}$ add $+\eta\,\boldsymbol{x}$ if $\boldsymbol{wx} <= 0$ and $d=+1$; $-\eta\,\boldsymbol{x}$ if $\boldsymbol{wx} > 0$ and $d=-1$

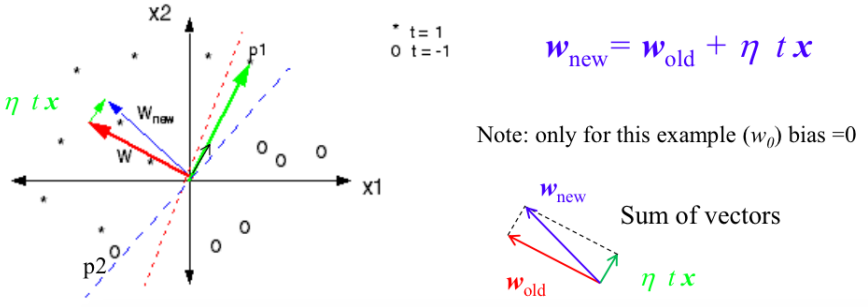
Or (in a different form)

- $\boldsymbol{w}_{new} = \boldsymbol{w}_{old} + 1/2\,\eta\ (d\text{-}out)\,\boldsymbol{x}$

Δw in LMS was different due to sign() in out (here)

■ **Geometrical view**

An example: Before updating the weight $\mathbf{w}$, we note that both $p1$ and $p2$ are incorrectly classified (red dashed line is decision boundary). Suppose we choose p1 to update the weights as in picture below. $p1$ has target value $t = 1$, so that $\mathbf{w}$ is moved a small amount in the direction of $p1$. The new boundary (blue dashed line) is better than before, (and w_new closer to $p1$). Exercise: what happens if the training pattern $p2$ is chosen ? same thing!



■ **Perceptron Learning Algorithm and Delta Rule**

The Perceptron Learning rule can be posed also into delta rule form (using +1,-1 encoding for the two classes) to show similarity and differences with Delta/ Widrow- Hoff/Adaline/LMS rule.

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \eta(d - out)\mathbf{x}$$

This is an *error-correction* rule (already seen in LMS section) that change the $\mathbf{w}$ proportionally to the error ($\delta = d - out$). Weights are modified at each step as happens in **delta rule**, which revises the weight w associated with input x according to the rule ($\delta = d - out$) where d is the target output for the current training example, *out* is the output generated by the perceptron and *eta* is a positive constant called the *learning rate*. The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (for example 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases.

Note: In terms of "neurons": the adjustment made to a synaptic weight is proportional to the product of error signal and the input signal that excite the synapse

4.6.2 Perceptron Convergence Theorem

The Perceptron can represent linear decision boundaries (see LTU) and can solve linearly separable problem. Can it always *learn* the solution?
Yes! The Perceptron with the *"Perceptron Learning Algorithm"* is always able to learn what it is able to represent: *Perceptron Convergence Theorem.*

Theorem: *The perceptron is guaranteed to converge (classifying correctly all the input patterns) in a finite number of steps if the problem is linearly separable.*

⇒**preliminaries**:
Let's suppose to have a Training Set: $(\mathbf{x}_i, d_i)$ where $d_i = +1$ or -1. Now, we focus only on the *Positive patterns* with $d_i = +1$. If we have two classes that are linearly separable we will have:

$$\text{Linearly separable } \rightarrow \exists\ \mathbf{w}^* \text{ solution s.t. } d_i(\mathbf{w}^*\mathbf{x}_i) \geq \alpha$$

$$\alpha = \min_i d_i(\mathbf{w}^*\mathbf{x}_i) > 0$$

Hence, $\mathbf{w}^*(d_i\mathbf{x}_i) \geq \alpha$
Defining $x_i^{'} = (d_i\mathbf{x}_i)$, then $\mathbf{w}^*$ is a solution iff $\mathbf{w}^*$ is a solution of $(x_i^{'}, +1)$

(if) $\mathbf{w}^*$ solve $\rightarrow d_i(\mathbf{w}^*\mathbf{x}_i) \geq \alpha \rightarrow \mathbf{w}^*(d_i\mathbf{x}_i) \geq \alpha \rightarrow (\mathbf{w}^*\mathbf{x}_i) > \alpha \rightarrow \mathbf{w}^*$ is a solution of $(x_i^{'}, +1)$
(only if) $\mathbf{w}^*$ is a solution of $(x_i^{'}, +1) \rightarrow (\mathbf{w}^* d_i\mathbf{x}_i) \geq \alpha \rightarrow d_i(\mathbf{w}^*\mathbf{x}_i) \geq \alpha \rightarrow \mathbf{w}^*$ solve for $\mathbf{x}_i$

Now let's assume that at step 0: $\mathbf{w}(\mathbf{0}) = 0$ and $\eta = 1, \beta = \max_i \|\mathbf{x}_i\|^2$
After q errors (all false negative, the perceptron incorrectly classifies it:

$$\mathbf{w}(q) = \sum_{j=1}^{q} \mathbf{x}_{i_j}\ ,\ \text{because } \mathbf{w}(j) = \mathbf{w}(j-1) + \mathbf{x}_{i_j}$$

Note: $\mathbf{w}_{new} = \mathbf{w}_{old} + \eta(d)\mathbf{x}$ but all d are $+1$ here, we are considering only positive patterns.

⇒**Proof**:
we can find lower and upper bound to $\|\mathbf{w}\|\,|$ as a function of q^2 steps (lower bound) and q steps (upper bound) $\rightarrow$ we can find q s.t. the algorithm converges.

- Lower bound on $\|\mathbf{w}(q)\|$:

$$\mathbf{w}^*\mathbf{w}(q) = \mathbf{w}^* \sum_{j=1}^{q} \mathbf{x}_{i_j} \geq q\alpha\ \ ,\ \ \alpha = \min_i(\mathbf{w}^*\mathbf{x}_i)$$

We make use of an inequality known as the Cauchy–Schwarz inequality:

$$(\mathbf{wv})^2 \leq \left\|\mathbf{w}^2\right\| \left\|\mathbf{v}^2\right\| \text{ (Cauchy–Schwarz)}$$

So we can write:

$$\|\mathbf{w}^*\|^2 \|\mathbf{w}(q)\|^2 \geq (\mathbf{w}^*\mathbf{w}(q))^2 \geq (q\alpha)^2$$

the lower bound is:

$$\|\mathbf{w}(q)\|^2 \geq \frac{(q\alpha)^2}{\|\mathbf{w}^*\|^2}$$

- Upper Bound on $\|\mathbf{w}(q)\|$:

we know that $\|\mathbf{a}+\mathbf{b}\|^2 = \|a\|^2 + 2\mathbf{ab} + \|b\|^2$, so we can write:

$$\|\mathbf{w}(q)\|^2 = \left\|\mathbf{w}(q-1) + \mathbf{x}_{i_q}\right\|^2 = \|\mathbf{w}(q-1)\|^2 + 2\mathbf{w}(q-1)\mathbf{x}_{i_q} + \left\|\mathbf{x}_{i_q}\right\|^2$$

But for the q-th error, $\mathbf{w}(q-1)\mathbf{x}_{i_q} \leq 0$. We therefore deduce that:

$$\|\mathbf{w}(q)\|^2 \leq \|\mathbf{w}(q-1)\|^2 + \left\|\mathbf{x}_{i_q}\right\|^2$$

invoking the assumed initial condition $\mathbf{w}(0) = 0$, we get the upper bound:

$$\|\mathbf{w}(q)\|^2 \leq \sum_{j=1}^{q} \left\|\mathbf{x}_{i_j}\right\|^2 \leq q\beta \ , \ \beta = \max_i \|\mathbf{x}_i\|^2$$

Let's put all together:

$$q\beta \geq \|\mathbf{w}(q)\|^2 \geq \frac{(q\alpha)^2}{\|\mathbf{w}^*\|^2}$$

$$q\beta \geq q^2\alpha^{'}$$

$$\beta \geq q\alpha^{'}$$

$$q \leq \frac{\beta}{\alpha^{'}}$$

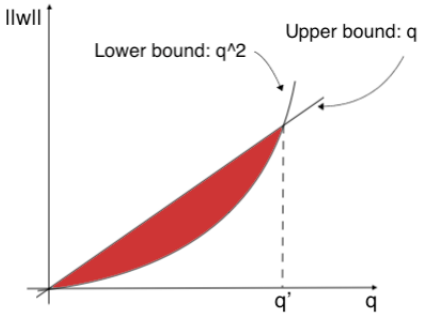


Figure 4.8: Lower and Upper bound

We have thus proved that for $\eta = 1$ and $\mathbf{w}(0) = 0$, and given that a solution vector $\mathbf{w}^*$ exists, the rule for adapting the synaptic weights of the perceptron must terminate after at most q interations. Surprisingly, this statement, proved for positive patterns, also holds for negative patterns.

4.6.3 Differences between "Perc. Learning Alg." and LMS Alg.

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \eta(d - out)\mathbf{x}$$

- LMS rule derived without threshold activation functions: minimize the error of the linear unit (using directly $\mathbf{wx}$)
- Hence, for training:

  $$\text{LMS has } \delta = (d - \mathbf{wx}) \text{ and PLA has } \delta = (d - sign(\mathbf{wx}))$$

  Of course the model trained with LMS can still be used for classification applying the threshold function $h(x) = sign(\mathbf{wx})$ (**LTU**)
- LMS not necessarily minimize the number of TR example misclassified by LTU, see Fig 4.9.

  $$E(\mathbf{w}) = \sum_i (y_i - \mathbf{x}_i^T \mathbf{w})^2$$

  It changes the weights not only for the misclassified patterns.

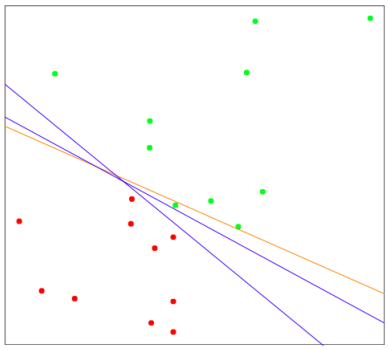


Figure 4.9: A toy example with two classes separable by a hyperplane. The orange line is the least squares solution, which misclassifies one of the training points. Also shown are two blue separating hyperplanes found by the perceptron learning algorithm with different random starts.

- PLA if training examples are linearly separable, it always converges (after a finite number of iterations) to a perfect classifier, otherwise it not.
- LMS-(gradient-descent) converges *asymptotically*, regardless of whether the training data are linearly separable (hence even if they are not linearly separable).

Note:

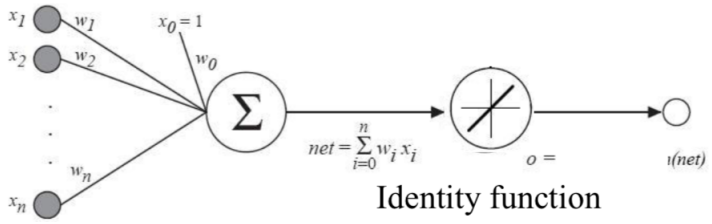
- Delta rule for linear unit $\rightarrow$ it minimize the Meand Square (MS) loss.
- "Perc. Learning Alg" $\rightarrow$ it has been posed also into delta rule form (using +1,-1 encoding for the two classes) to show similarity and differences.

4.7 Activation functions

In artificial neural networks, the activation function of a node defines the output of that node given an input or set of inputs. However, only nonlinear activation functions allow such networks to compute nontrivial problems. This function is also called the transfer function.

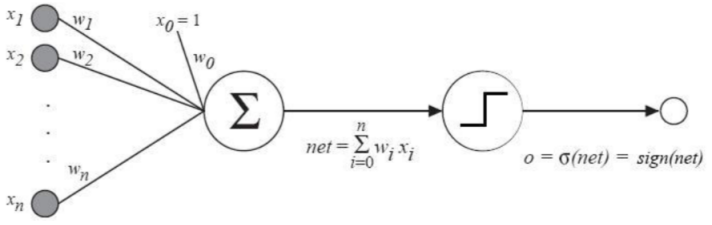
4.7.1 Linear function

$$o = net = h(\mathbf{x}) = \sum_i w_i x_i$$



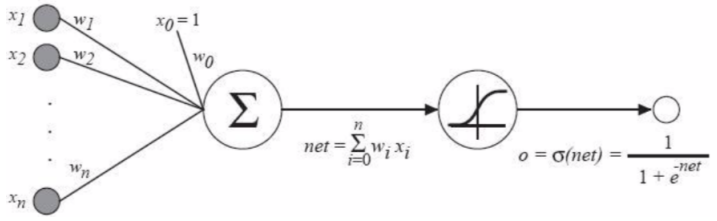
4.7.2 Threshold (or step) function (Perceptron/LTU)

$$o = \sigma(net) = sign(net)$$



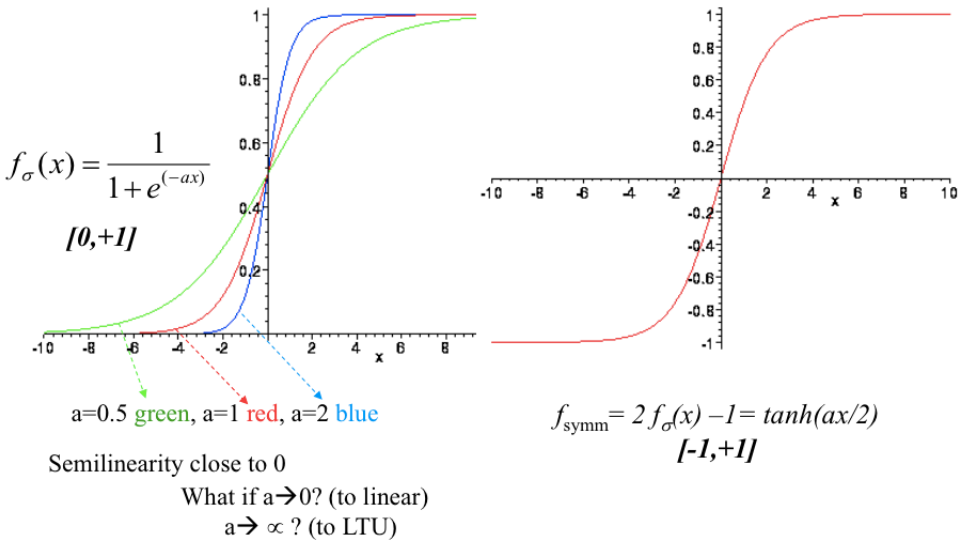
4.7.3 Sigmoidal logistic Function

A non-linear squashing function like the sigmoidal logistic Function: assumes a continuous range of values in bounded interval [0,1].



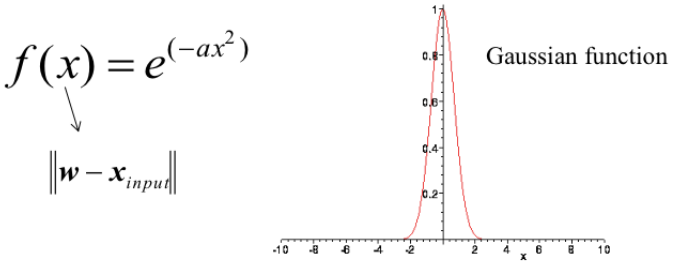
$$f_\sigma(x) = \frac{1}{1 - e^{-ax}}$$

The sigmoidal-logistic function has the property to be a smoothed differentiable threshold function. a is the slope parameter of the sigmoid function.



4.7.4 Other Activation Functions

■ **Radial Basis Functions**
Radial Basis Functions → RBF networks



■ **Softmax**
see next lectures (multi-output case)

■ **Stochastic neurons**
output is +1 with probability $P(net)$ or –1 with $1 - P(net)$ → Boltzmann machines and other models rooted in statistical mechanics.

■ **Tanh-like**
Tanh-like (piecewise linear approximation) for efficient computation. n steepness (slope) of the function Rounding operator (floor) Shift operation in binary rep.

$$f(x) = sign(x)\left[1 + \frac{1}{2^{\lfloor 2^n|x| \rfloor}}\left(\frac{2^n|x| - \lfloor 2^n|x| \rfloor}{2} - 1\right)\right]$$

■ **ReLU (Rectified Linear Unit)**
Rectifier → ReLU (Rectified Linear Unit): see Deep Learning section. It became a default choice for Deep models, so it (and its variants) deserves more attention discussing Deep Learning.

$$f(x) = \begin{cases} 0 & for \;\; x < 0 \\ x & for \;\; x \geq 0 \end{cases}$$

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

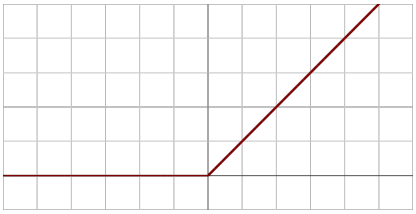


Figure 4.10: Relu activation function

■ **Softplus (smooth approximation)**

$$f(x) = ln(1 + e^x)$$

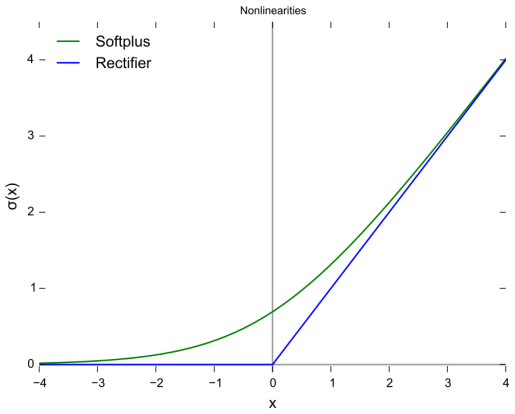


Figure 4.11: Softplus (smooth approximation)

4.7.5 Activation functions: derivatives

- The derivative of the identity function is 1.
- The derivative of the step function is not defined which is exactly why it isn't used.
- Sigmoids: for asimmetric and simmetric case we have ($a = 1$):

$$\frac{df_\sigma(x)}{dx} = f_\sigma(x)(1 - f_\sigma(x)) \qquad \frac{df_{tanh}(x)}{dx} = 1 - f_{tanh}^2$$

  this is true also with any a.

4.8 Least Mean Square with Sigmoids

The sigmoidal-logistic function has the property to be a smoothed differentiable threshold function. Hence, we can derive a Least (Mean) Square algorithm by computing the gradient of the loss function as for the linear units (also for a classifier).

From $o(\mathbf{x}) = \mathbf{xw}$ to $o(\mathbf{x}) = f_\sigma(\mathbf{xw})$ where f_σ is a logistic function and we want to find $\mathbf{w}$ to minimize the residual sum of squares:

$$E(\mathbf{w}) = \sum_p (d_p - o(\mathbf{x}_p))^2 = \sum_p (d_p - f_\sigma(\mathbf{x_p^T w}))^2$$

■ **LMS with 1 non-linear unit**:
The algorithm is always the same, gradient descent algorithm (batch or online), for the Delta rule we have to compute the gradient of the loss.

$$\frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}_i} \ , \qquad o(\mathbf{x}_p) = out(\mathbf{x}_p) = f_\sigma(\mathbf{x}\mathbf{w})$$

Now we want to compute $\frac{\partial E(\mathbf{w})}{\mathbf{w}_j}$:

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}_j} = \frac{\sum_p \partial (d_p - f_\sigma(\mathbf{x}_p^T \mathbf{w}))^2}{\partial \mathbf{w}_j} \ , \ net = \mathbf{x}_p^T \mathbf{w}$$

$$\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}_j} = \frac{\partial E(\mathbf{w})}{\partial net}\frac{\partial net}{\partial \mathbf{w}_j} = \frac{\sum_p \partial (d_p - f_\sigma(net))^2}{\partial \mathbf{w}_j}\frac{\partial net}{\partial \mathbf{w}_j} \ , \ out = f_\sigma(net)$$

$$\frac{\partial net}{\partial \mathbf{w}_j} = \frac{\partial \sum_{t=0}^{n} w_t x_{p,t}}{\partial \mathbf{w}_j} = (x_p)_j$$

$$\frac{\partial E(\mathbf{w})}{\partial net} = \frac{\partial E(\mathbf{w})}{\partial out}\frac{\partial out}{\partial net} = \frac{\sum_p \partial (d_p - out)^2}{\partial out} f_\sigma^{'}(net) =$$

$$= 2\sum_p (d_p - out)\frac{\partial (d_p - out)}{\partial out} f_\sigma^{'}(net) = -2\sum_p (d_p - out) f_\sigma^{'}(net)$$

by put all together we can find:

$$\frac{\partial E_p(\mathbf{w})}{\partial \mathbf{w}_i} = -2\sum_p (d_p - f_\sigma(net)) f_\sigma^{'}(net)(x_p)_j = -2\sum_p (x_p)_j \delta_p$$

■ **Gradient descent algorithm**:
The same as for linear unit using the new delta rule (batch/on-line):

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \eta \delta_p \mathbf{x}_p \ , \ \delta_p = (d_p - f_\sigma(net)) f_\sigma^{'}(net) \quad \text{(Pattern p)}$$

The unique Unit has not index here.

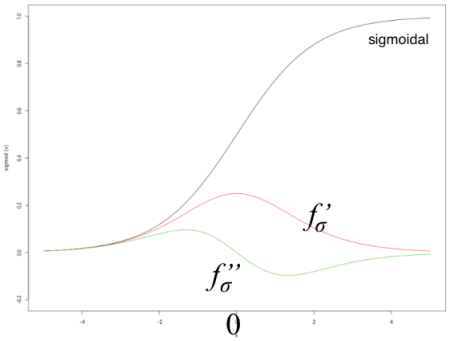


Figure 4.12: new delta rule

Note:

- Again we have an error correction rule, but the new δ_p it depends on the difference and on the derivative of the sigmoid function.
- the parameters a of the sigmoid function can affect the step of gradient descent.
- Max of $f^{'}$ for input close to 0 ($\sim$ linear unit).

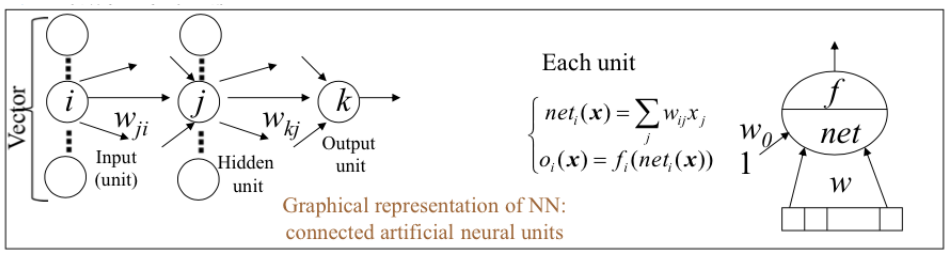
- Minimum of f' is for saturated cases: better start with small weights to avoid saturation, and hence very slow changes of w, at the beginning and use the regularization to lower the weights.
- Insight: also a bridge between the two algorithm: toward no corrections for correct output also for LMS algorithm.

4.9 Multi-Layer Perceptron (MLP)-NN

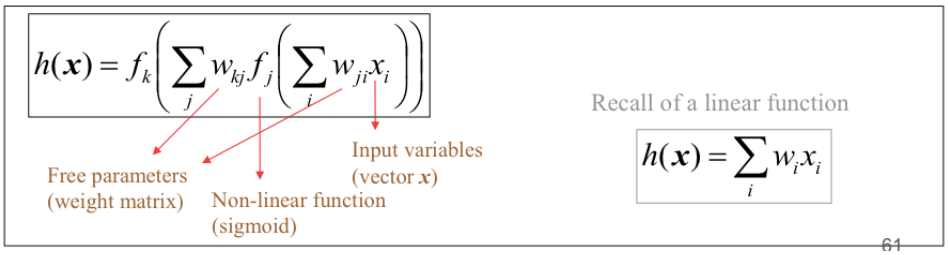
All the ingredients are ready and we can start to talk about Neural Networks. Neural Networks by two views of Multi-Layer Perceptron (MLP)-NN:

E.g. Standard feedforward NN (with one hidden layer):

1. A network of interconnected units:
the architectural graph of a multiplayer perceptron with one hidden layers and an output layer. To set the stage for a description of the multilayer percep-tron in its general form, the network shown here is fully connected. This means that a neu-ron in any layer of the network is connected to all the neurons (nodes) in the previous layer (except the input ones). Signal flow through the network progresses in a forward direction, from left to right and on a layer-by-layer basis.



2. A flexible function $h(x)$ (as nested non-linear functions)



■ **Neural Networks: components**
NN model traditionally presented by the type of:

- *UNIT*: net, activation functions
- *ARCHITECTURE*: number of units, topology (e.g. also number of layers)
- *LEARNING ALGORITHM*

The architecture of a NN defines the topology of the connections among the units. E.g **Feedforward** and **Recurrent** neural networks.

4.9.1 MLP Standard feedforward NN

The two-layer feedforward neural network described in Equation h(x) corresponds to the well-know MLP (Multi Layer Perceptron) architecture:

direction: input → output

- the units are connected by weighted links and they are organized in the form of layers.
- the input layer is simply the source of the input $\mathbf{x}$ that projects onto the hidden layer of units.
- the hidden layer projects onto the output layer (feedforward computation of a two-layer network).

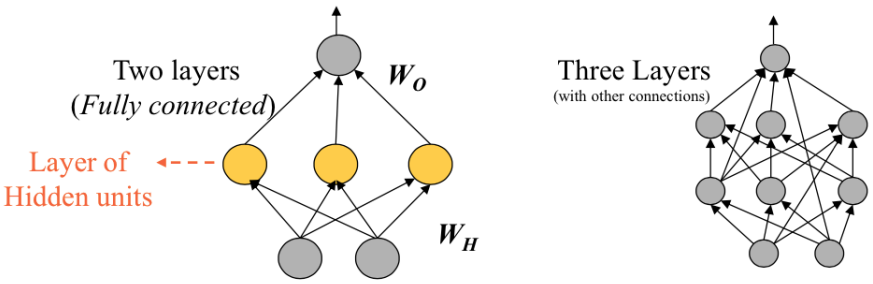


Figure 4.13: A feedforward NN on the lef and a Three Layers with other connections NN on the right

4.9.2 Recurrent neural networks

Recurrent neural networks: A different category of architecture, based on the addition of feedback loops connections in the network topology:

- The presence of self-loop connections provides the network with dynamical properties, letting a memory of the past computations in the model.
- This allows us to extend the representation capability of the model to the processing of sequences (and structured data).

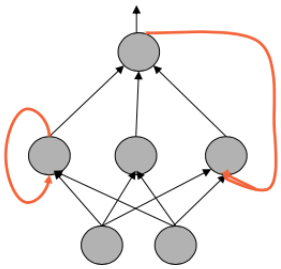


Figure 4.14: Recurrent neural networks

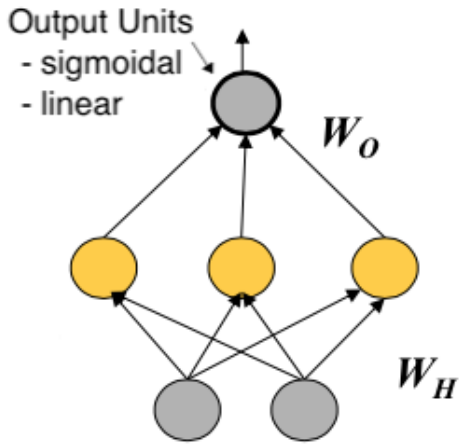
4.9.3 Flexibility of Neural Network model

There are some important questions about neural networks:

1. Why NN is a flexible model? why work well? Can be interpreted as linear basis expansion (LBE)?
2. Is the flexibility theoretically grounded?

■ **Neural Network flexibility**
Hypothesis space of a Neural Network: continuous space of all the functions that can be represented by assigning the weight values of the given architecture.

- Note that, depending on the class of values produced by the network output units, discrete or continuous, the model can deal, respectively, with classification (sigmoidal output f.) or regression (linear output f.) tasks.



(a) Neural Network

$$h(\boldsymbol{x}) = f_k\left(\sum_j w_{kj} f_j\left(\sum_i w_{ji} x_i\right)\right)$$

(b) $h(\mathbf{x})$

Figure 4.15: A Neural Network can solve regression and classification task.

- Also multi-regression or multi-classes classifier can be obtained by using multiple output units

■ **Neural Network as a function**

$$h(\boldsymbol{x}) = f_k\left(\sum_j w_{kj} f_j\left(\sum_i w_{ji} x_i\right)\right)$$

- This is the function computed by a two-layer feedforward neural network
- Units and architecture just a graphical representation (of the data flow process)
- each $f_j(\sum_i w_{ji} x_i)$, can be seen as computed by an independent processing element (unit)

Indeed, NN is a function non linear in the parameters w.

■ **Neural Network as a dictionary approach**
We have seen in the Least Mean Square the **linear basis expansion (LBE)**, in that case the phi was preselected and fiexed and does not change with trainig data:

$$\text{linear with respect to parameters w: } \sum_j w_j \phi_j(\mathbf{x})$$

But in NN we have an **Adaptive(flexible)** basis functions approach, the basis functions themselves are adapted to data (by fitting of w in phi):

$$h(\mathbf{x}) = f_k\left(\sum_j w_{kj}\phi_j(\mathbf{x}, \mathbf{w})\right) \quad , \quad \phi_j(\mathbf{x}, \mathbf{w}) = f_j\left(\sum_i w_{ji} x_i\right)$$

Note that we fix the same type of basis functions for all the terms in the basis expansion (given by the activation function).

$h(x)$ as (nonlinear function of weighted) sums of nonlinearly transformed linear models plus the important enhancement of adaptivity

■ Hidden Layer Relevance and Interpretation
Each basis function (**hidden unit**) compute an a **new** nonlinear derived features, **adaptively** by learning, according to the training data. I.e. the parameters of the basis function w are learned from data by learning.

$$\phi_j(\mathbf{x}, \mathbf{w}) = f_j\left(\sum_i w_{ji} x_i\right)$$

In other words:

- The representational capacity of the model is related to the presence of a hidden layer of units, with the use of non-linear activation function, that transforms the input pattern into the **internal representation** of the network.
- The learning process can define a **suitable internal representation**, also visible as new hidden features of data, allowing the model to **extract from data** the higher-order statistic that are relevant to approximate the target function.

$$h(\mathbf{x}) = f_k\left(\sum_j w_{kj} f_j\left(\sum_i w_{ji} x_i\right)\right)$$

Non-linear units are essential, MLP with linear units = 1 layer NN!

Anyway for the learning algorithm this introduce an issue:

Non-linear w.r.t. to $w \rightarrow$ non linear optimization problem.

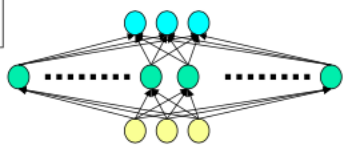
The hidden neurons act as feature detectors; as such, they play a critical role in the operation of a multilayer perceptron. As the learning process progresses across the multilayer perceptron, the hidden neurons begin to gradually "discover" the salient features that characterize the training data. They do so by performing a nonlinear transformation on the input data into a new space called the feature space. In this new space, the classes of interest in a pattern-classification task, for example, may be more easily separated from each other than could be the case in the original input data space. Indeed, it is the formation of this feature space through supervised learning that distinguishes the multilayer perceptron from Rosenblatt's perceptron.

■ Universal approximation
Many early results (Cybenko 1989, Hornik et al. 1993, etc...)

- A single hidden-layer network (with logistic activation functions) can approximate (arbitrarily well) every continuous (on hyper cubes) function (provided enough units in the hidden layer) [e.g. image it generalizes approximation by finite Fourier series]
- A MLP network can approximate (arbitrarily well) every input- output mapping (provided enough units in the hidden layers)

Given ε, ∃ h(x) s.t. $|\mathbf{f(x)-h(x)}|<\varepsilon$
∀ x in the hypercube

$$h(\boldsymbol{x}) = \underset{\text{linear}}{f_k}\left(\sum_j w_{kj} f_j\left(\sum_i w_{ji} x_i\right)\right)$$



Existence theorem! This not provide the algorithm nor the number of units

After this fundamental result (MLP is able to represent any function), two issues will deserve our attention:

- How to learn by NN
- How to decide a NN architecture

4.9.4 NN Expressive power

The **expressive power** of NN is strongly influenced by two aspects: the number of units and their configuration (architecture).

The number of units can be related to the discussion of the VC-dimension of the model. Specifically, the network capabilities are influenced by the number of parameters, that is proportional to the number of units, and further studies report also the dependencies on their value sizes, E.g.

Weights = 0 → minimal Vc-dim
Small weights→linear part of the activation function
Higher weights values→more complex model

Number of parameters (fully-connected neural network MLP):
#input-unit * #hidden-unit * #output-unit

■ **How many layers?**
A look ahead (toward **deep learning**):

- The Universal Approximation theorem is a fundamental contribution
- It show that 1 hidden layer is sufficient in general, but it does not assure that a "small number" of units could be sufficient (it does not provide a limit on such number)
- It is possible to find boundaries on such number (for many f. families)
- But also to find "no flattening" results (on efficiency, not expressivity): cases for which the implementation by a single hidden layer would require an exponential number of units (w.r.t n input dim.), or non-zero weights, while more layers can help (it can help for the number of units/weights and/or for learning such approximation).
  But is it easy to optimize (training) a MLP with many layers? we will see later in deep learning.

4.9.5 The Backpropagation Algorithm

How to learn the weights of a NN?
We want a *learning algorithm* that allows adapting the free-parameters of the model in order to obtain the best approximation of the target function.
As we saw, in the ML framework this is often realized in terms of minimization of an error (or loss) function on the training data set.

The problem is the same we have seen for the other models (repetita):

- **Given** a set of l training examples $(\mathbf{x}_p, d_p)$ and a loss function (measure) L
- **Find** the weight vector $\mathbf{w}$ that minimizes the expected loss on the training data:

$$E(\mathbf{w}) = R_{emp} = \frac{1}{l}\sum_{p=1}^{l} L(h(\mathbf{x}_p), d_p)$$

For the *perceptron* we have the *Perceptron learning algorithm (delta rule)*, now we have to define a **Learning Algorithm for the network (MLP)**.

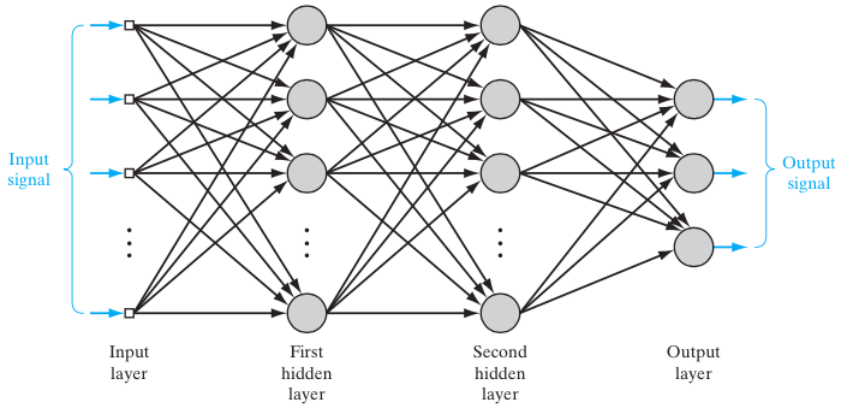


Figure 4.16: Architectural graph of a multilayer perceptron with two hidden layers.

■ **Credit assignment problem:**
When studying learning algorithms for distributed systems, exemplified by the multilayer perceptron of Figure 4.16, it is instructive to pay attention to the notion of credit assignment. Basically, the credit-assignment problem is the problem of assigning credit or blame for overall outcomes to each of the internal decisions made by the hidden computational units of the distributed learning system, recognizing that those decisions are responsible for the overall outcomes in the first place. In a multilayer perceptron using error-correlation learning, the credit-assignment problem arises because the operation of each hidden neuron and of each output neuron in the network is important to the network's correct overall action on a learning task of interest. That is, in order to solve the prescribed task, the network must assign certain forms of behavior to all of its neurons through a specification of the error-correction learning algorithm. With this background, consider the multilayer perceptron depicted in Fig. 4.16. Since each output neuron is visible to the outside world, it is possible to supply a desired response to guide the behavior of such a neuron. Thus, as far as output neurons are concerned, it is a straightforward matter to adjust the synaptic weights of each output neuron in accordance with the error-correction algorithm. But how do we assign credit or blame for the action of the hidden neurons when the error-correction learning algorithm is used to adjust the respective synaptic weights of these neurons? The answer to this fundamental question requires more detailed attention than in the case of output neurons. In what follows in this chapter, we show that the back-propagation algorithm, basic to the training of a multilayer perceptron, solves the credit-assignment problem in an elegant manner. Supposed too difficult by Minsky-Papert (1969), while faced by many researchers (see historical notes) with

the back-propagation algorithm popularized by Rumelhart, Hinton, Williams in the PDP book (1986)

So, the main problem is the Credit assignment problem, we can use a Gradient descend approach which minimizes a loss function, it can be extended to MLP with the back-propagation algorithm that will be able to change **the hidden layer weights**.

(Repetita) Non-linear with respect to $w \rightarrow$ non linear optimization problem

■ **The loading problem**
Loading problem: "loading" a given set of tr data into the free parameters of the Neural Networks

- given a network and a set of examples
- answer yes/no: is there a set of weights so that the network will be consistent with the examples?

The loading problem is NP-complete (Judd, 1990), this implies that hard problems cannot be "solved".

In practice networks can be trained in a reasonable amount of time (for example with the use of the back-propagation algorithm) although optimal solution is not guaranteed.

■ **Intro to Backpropgation**

Gradient descend approach, minimizing a loss function, can be extended to MLP. We need a differentiable loss function, differentiable activation functions and a network to follow the information flow.

So we want to **find w by computing the gradient of the loss function**

$$E(\mathbf{w}) = \sum_p (d_p - h(\mathbf{x}_p))^2$$

Here some useful definitions for backpropagation and future topics:

- **Batch**: you divide the dataset into a number of *batches*
- **Batch size**: total number of training examples present in a single batch
- **Epoch**: one epoch is when an entire dataset is passed forward and backward through the neural network only once (an entire cycle of training pattern presentation)
- **Iterations**: Is the number of batches needed to complete one epoch

Some nice properties (also for programming):

- easy because of the compositional form of the model
- keep track only of quantities local to each unit (by local variables) $\rightarrow$ modularity of the units is preserved

■ **The Back-propagation based learning algorithm**

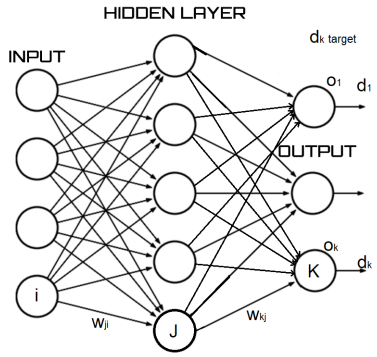


Figure 4.17: Feed forward fully connected neural network (MLP)

- Training Set = $\{(x^1, d^1), ..., (x^i, d^i)\}$
- $E_{tot} = \sum_p E_p$ where $E_p = \frac{1}{2}\sum_k (d_k - o_k)^2$ (for pattern p)
- Find the $\mathbf{w}$ parameters such that $min\, E_{tot}$
- by Least Mean Square
- Descent on the surface E on the basis of the gradient

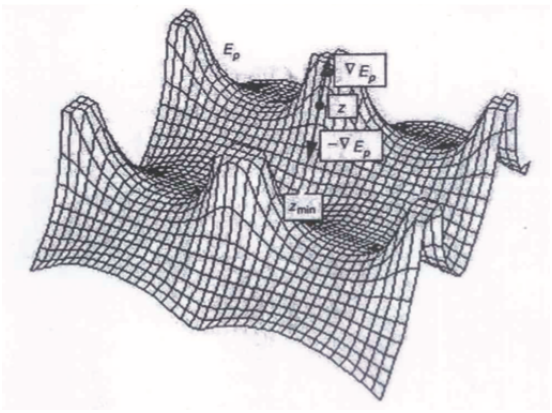
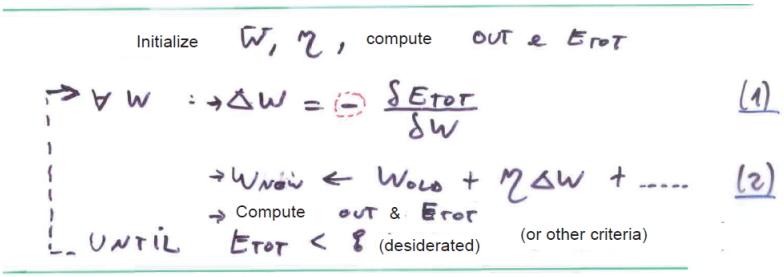


Figure 4.18: Example of E in the space of two weights (w_1 and w_2); the local gradient is shown in the point Z. Along the direction of (-gradient) we can reach the Z_{min} point.

Back-propagation As mentioned, this backpropagation is based on the *iterative gradient descent algorithm*: Loss function (Error E) → gradient computation → update the weights W



1. Gradient computation: $\frac{\partial E}{\partial w}$
2. Upgrade rule for w, there many: Standard back-propagation, quick-prop, R-prop, ...

We focus on step (1) - Gradient computation

$$\Delta w = -\frac{\partial E_{tot}}{\partial w} = -\sum_p \frac{\partial E_p}{\partial w} = \sum_p \Delta_p w$$

Where p refers to the p-th pattern feeding in input the neural network.

For the generic unit t:

$$\Delta_p w_{ti} = -\frac{\partial E_p}{\partial w_{ti}} = -\frac{\partial E_p}{\partial net_t} \cdot \frac{\partial net_t}{\partial w_{ti}} = \delta_t \cdot o_i$$

Where, assuming for pattern p, δ_t is the delta of unit t and o_i is the input i from a generic unit (o_i) to the unit t, since

$$\begin{cases} net_t = \sum_j w_{tj} o_j \\ o_t = f_t(net_t) \end{cases} \Rightarrow \frac{\partial \sum_j w_{tj} o_j}{\partial w_{ti}} = o_i$$

$o_j \rightarrow$ input j from a generic unit (o_j) to the unit t
$o_i \rightarrow$ all 0 except for $j = i$

Now we see the development of δ_t, the error term associated with unit t:

$$\delta_t = -\frac{\partial E_p}{\partial net_t} = -\frac{\partial E_p}{\partial o_t} \cdot \frac{\partial o_t}{\partial net_t} = -\frac{\partial E_p}{\partial o_t} \cdot f_t'(net_t)$$

Where $-\frac{\partial E_p}{\partial o_t}$ changes according to the type of unit:

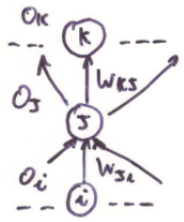


Figure 4.19: Use the network graph to localize the computation (here by indices)

- **Out unit K**

$$-\frac{\partial E_p}{\partial o_k} = -\frac{\partial \frac{1}{2}\sum_R (d_R - o_R)^2}{\partial o_k} = (d_k - o_k) \qquad (internal\ index\ R)$$
$$\Rightarrow \delta_k = (d_k - o_k) \cdot f_k'(net_k)$$

- **Hidden unit J**

$$-\frac{\partial E_p}{\partial o_j} = \sum_k -\frac{\partial E_p}{\partial net_k} \cdot \frac{\partial net_k}{\partial o_j} = \sum_k \delta_k \cdot w_{kj}$$

$$\text{Since} \begin{cases} -\frac{\partial E_p}{\partial net_k} = \delta_k \quad \text{(already computed)} \\ \frac{\partial net_k}{\partial o_j} = \frac{\partial \sum_R w_{kR} o_R}{\partial o_j} = w_{kj} \end{cases}$$

$$\Rightarrow \delta_j = (\sum_k \delta_k w_{kj}) \cdot f_j'(net_j)$$

It expresses the variation of E with respect to all the output unit o_k.
Each o_k (and net_k) depends on o_j, hence we introduce a sum over k.

Summary We derived, for a for a generic unit t, with input from o_i

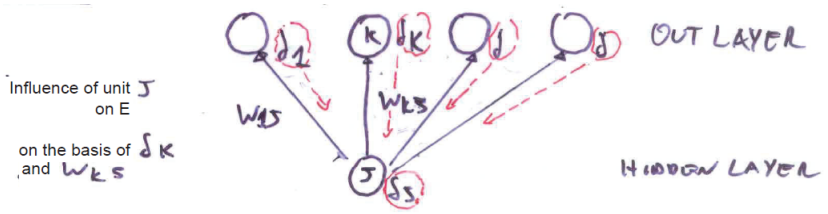
$$\Delta_p w_{ti} = \delta_t o_i$$

- IF $t = k$ (output unit)

$$\delta_k = (d_k - o_k) f_k^{'}(net_k)$$

- IF $t = j$ (hidden unit)

$$\delta_j = \left(\sum_k \delta_k w_{kj} \right) f_j^{'}(net_j)$$



Retropropagation of delta (δ) values from the output layer to obtain the error signal for the hidden layers. It can be applied (generalized) to m hidden layers, in other words, deltas come not only from the output layer but also from any generic layer above the current one (a generic upper layer k). More in general the

deltas come from any units to which this unit is connected to.
This will be foundamental for recurrent NN training, for deep learning, etc. Hence, for each pattern p:

$$w_{ti_{new}} = w_{ti_{old}} + \delta_t o_i$$

Where

- δ_t: delta a that level for t
- o_i: input to the unit t from i through the connections w_{ti}

Reference: Parallel Distributed Processing, Vol.1, by D. E. Rumelhart, J. L. McClelland, MIT Press : Chapter 8 by Rumelhart, Hinton, Williams (1986): *"Learning internal representations by error propagation"*, pages: 322-328.

4.9.6 Inductive Bias

What is the inductive bias by which Backpropagation generalizer beyond the observed data? It is difficult to characterize precisely the inductive bias of Backpropagation learning, because it depends on the interplay between the gradient descent search and the way in which the weight space spans the space of representable functions. However, one can roughly characterize it as smooth interpolation between data points.

So, Neural Network with backprogation learning algorithm:

- Generally related to the **smoothness** properties of functions we're going to search in the hypothesis space (A very common assumption in ML). E.g a locally limited value of the first derivative.

Non-smooth function is generally relative to a random number generator and generalization cannot be achieved.

4.10 Heuristic for the Backpropagation Algorithm

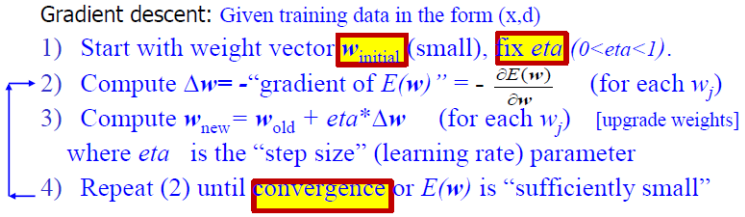
General problems:

- The model is often over-parametrized
- Optimization problem is not convex and potentially unstable

It is often said that the design of a neural network using the back-propagation algorithm is more of an art than a science, in the sense that many of the factors involved in the design are the results of one's own personal experience. There is some truth in this statement. Nevertheless, there are methods that will significantly improve the back- propagation algorithm's performance, as described here. Some of the factors are:

- **Hyperparameters**: Starting values, on-line/batch, learning rate
- Multiple Minima
- Stopping criteria
- Overfitting and regularization
- **Hyperparameters**: Number of hidden units
- Input scaling/ Output representation
- ...

In the previous section we saw that the back-propagation algorithm is essentially the gradient descent algorithm applied to neural networks. The back-propagation is used as a path through the weight space, the path depends on: the data (assumed already given), the neural network model, the starting point (initial weight values) and the final point (stopping rule). This allow to define a control for the search over the Hypothesis space.



But now $\Delta w = -\frac{\partial E(\mathbf{w})}{\partial \mathbf{w}}$ were obtained through back-propagation derivation/algorithm for any weights in the network. At step 2 to compute the Error we first apply inputs to the network computing a forward phase up to the output computation – then we start the backward phase to retro-propagate the deltas for the gradient.

In this section we discuss few hyperparameters (values that you have to set-up to run the training) of the algorithm, like those highlighted above.

4.10.1 Starting values ($w_{initial}$)

★ Initialize weights by random values near zero. Avoid all zero, high values or all equals: these can hamper the training. (for example in *range* [-0.7,+0.7] for standardized data)

Other ways:

- Considering the Fan-in (the number of inputs to a (hidden) unit), for example $range * 2/fan - in$:

  - Not if fan-in is too large
  - Not for output units (else delta start close to zero!)
- Other heuristics... (orthogonal matrix, ...)
- Very popular among practitioners: Glorot, Bengio AISTATS 2010: 0 for biases and w random from Uniform distribution in $[-1/a, +1/a]$ with $a = sqrt(fan - in)$ or (later results) a depending also on fan-out ...

Note (practical for the first part of the project): a very small NN (very few units) can have a stronger dependencies w.r.t. initialization.

4.10.2 Multiple Minima

The loss function is not *convex*, so we have many *local minima*.
But is global minima needed?

- Indeed in ML we don't need neither local or global minima on R_{emp} as we are searching the min of R (that we cannot compute).
  - Often we stop in a point that has not null gradient and hence also neither a local or global minima for the empirical (training) error.
- The NN build a variable size hp space $\rightarrow$ it increases VC-dim during training $\rightarrow$ the training error decreases toward zero not because we reach a global minimum but because the NN becomes too complex (high VC-dim).
  Where **VC-dim** measures the complexity of the hp space (= flexibility to fit data).
- Stopping before this condition of *overtraining* (and hence overfitting) is needed (independently from the issue on local/global minima).

Multiple Minima are not a big issue:

Multiple Minima are not a big issue because a "good" local minima is often sufficient, you can always see the final training error and check what is happening.

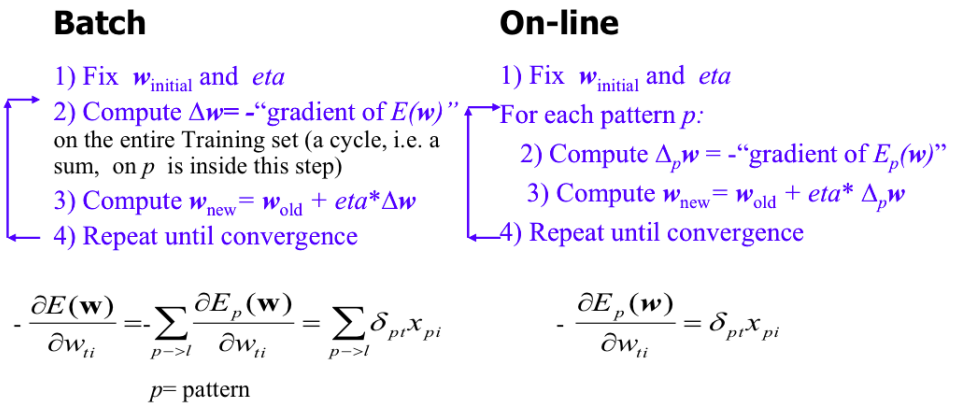
The result depends on starting weight values. so the starting weights values help us to find a *good* local minima.

★ Try a number of random starting configurations (for example 10 or more training runs or trials).

- Take the **mean results** (mean of errors) and look to variance to *evaluate* your model and then, if you like to have only 1 response :
  - you can choose the solution giving lowest (penalized) error (or the median)
  - You can take advantage of different end point by an average (committee) response: mean of the outputs/voting

4.10.3 On-line/Batch

Two versions of *Gradient descent algorithm*.
Steps 2 and 3 change to express an inner or external cycle on the patterns:



- For **Batch** version: sum up all the gradients of each pattern over an epoch and then update the weights (Δw after each "epoch" of l patterns).
- **Stochastic/on-line** version: upgrade $\mathbf{w}$ for each pattern p (by $\Delta_p w$) without wait the total sum over l:
  - can be faster, but need smaller η (*learning rate*): make progress with each examples it looks at.
  - since the gradient for a single data point can be considered a noisy approximation to the overall gradient, this is also called **stochastic** (noisy) gradient descent. *Zig-zag (stochastic) descent.*
- Another variation exists: **Stochastic Gradient Descent (SGD)** that use *minibatch* (few training examples) for each step of gradient descent.

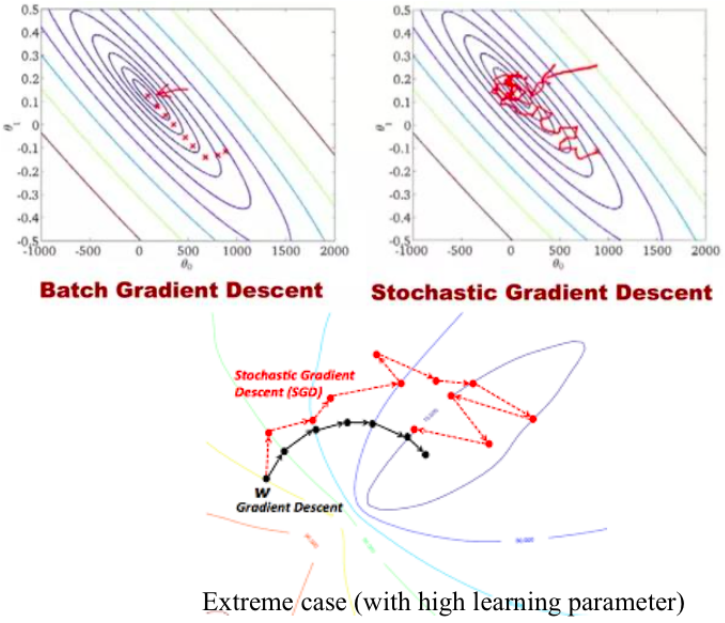
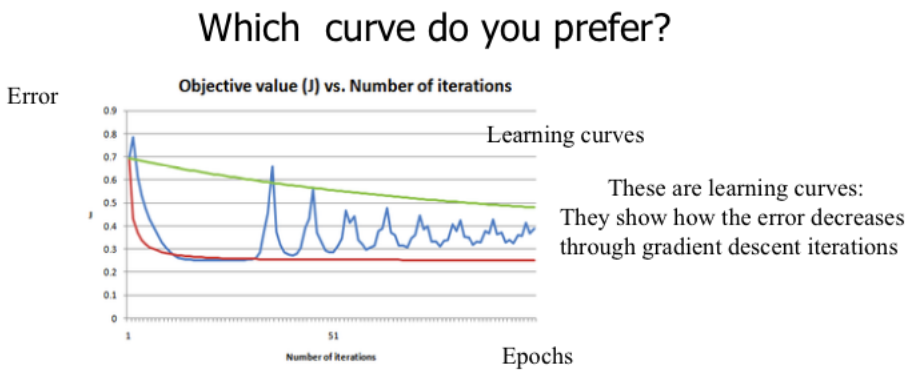


Figure 4.20: Plots stochastics vs batch (with high learning parameter))

Look to the Learning Curve



4.10.4 Mini-batch SGD

Mini-batch, with mb examples, to estimate the gradient (mb is the *batch size*).

$$1 < mb < l$$

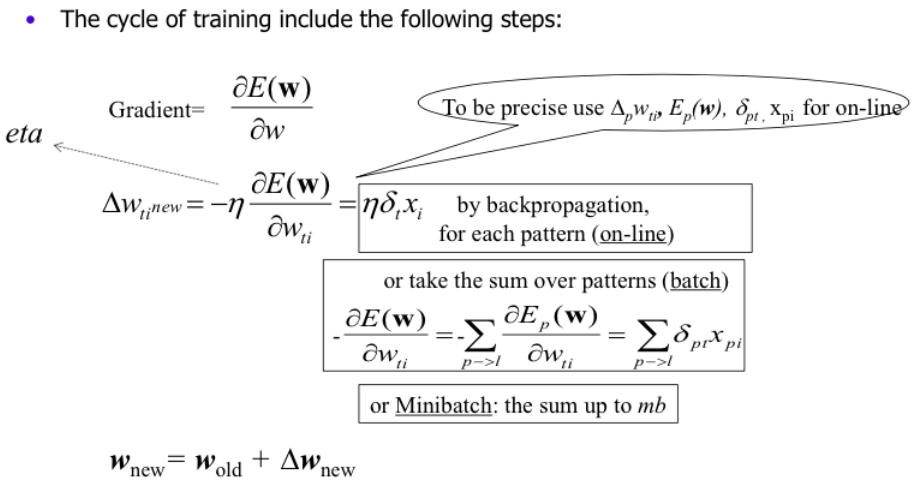
- $mb = 1 \rightarrow$ *Stochastic/on-line*
  - cannot exploit parallel processing of examples
  - may be unstable
  - can have a regularization effect (adding noise to the learning process), this could be a good side effect.
- $mb = l \rightarrow$ *Batch (full training set)*
  - more accurate gradient estimation
  - but slower! (wait all the data)

Minibatch stochastic method (SGD):

- Use random sampling to implement the minibatch (unbiased estimation). By *shuffling* (at the beginning for all the TR data or for each mini batch)
- With very large TR set or online streaming $\rightarrow$ even no epochs!
- Along with the advantage of gradient descent (linear cost wrt examples) SGD with mb is a key factor in extending learning to very large data sets (over millions of examples) because mb (and hence the cost of each SGD update) does not depend on l : typically just mini-batch and (again) no epochs repetition

4.10.5 Learning rate (η)

PREMISE:



Learning rate (η):

High versus low eta values: fast versus stable

- **Batch**: more accurate estimation of gradient $\rightarrow$ **higher** η (fast).
- **On-line**: use **smaller value of** η (stable). This approach can make the training faster and can help avoiding local minima:
  - Different order of pattern in different epochs should be used
  - ★ Usually random order (shuffling as already discussed)

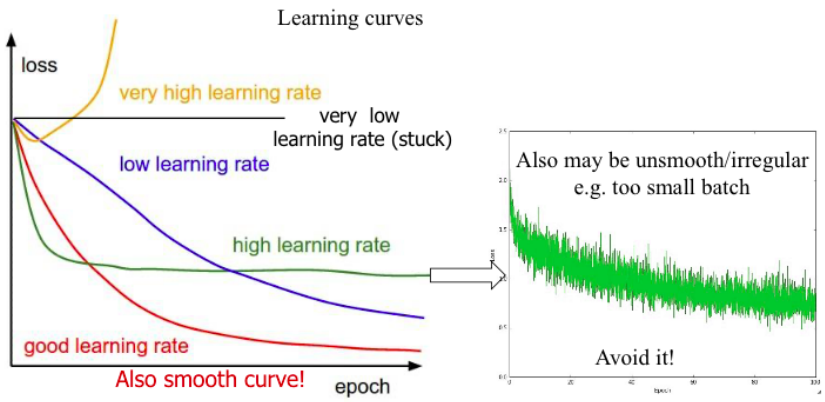
  Some better approaches:

1. ★ Momentum
2. Variable learning rate (initially high then decrease)
3. It can be varied depending on the layer (higher for deep layers) or fan-in (higher for few input connections)
4. Adaptive learning rates (change during training and for each w) $\rightarrow$ more in Deep Learning.

some possible values fore $\eta = [0.01, 0.5]$ in Least Mean Square.

Practical hints on η values

- Too large eta $\rightarrow$ instability, SGD can even increase errors
- Too small eta $\rightarrow$ not only slow, can even become permanently stuck with high training (TR) error
- **Learning curves**, plotting errors during training, allow you to check the behavior in the early phases of the model design for your task at hand (preliminary trials).
  ★ Please check the Learning Curve training a NN!



Here just looking to the optimization "failures" (on the TR set), not the generalization errors

- Of course the absolute value for training error depend also on the model capacity (see later for the number of units, regularization etc.)
- Useful to consider to use of the mean(average) of the gradients over the epoch: helps to have a "uniform" approach with respect to the number of input data ($Sum\ of\ gradients/l$).

(!) – Repetita from "Gradient descent algorithm": ***dividing by l correspond to have the Least Mean Squares***

NOTE: 0<eta<1 when we use LMS

(!) – *Note that it is equivalent to have:* ***eta/l***

– ***Off course /mb for mini batch***

  - Take care that on-line can be more unstable!
  - Note that if you use the "mean" of gradient for the epochs (batch), using on-line training will require much smaller eta to be comparable!!!! (or on the other way an higher eta for the batch case).

Learning rate and mini batch

Using mini-batch the gradient does not decrease to zero close to a minimum (as the exact gradient can do), hence fixed learning rate should be avoided.

For instance *decay linearly* eta for each step until iteration τ, using $\alpha = (step\ s)/\tau$, then stop and use fix (small) η:

$$\eta_s = (1 - \alpha)\eta_0 + \alpha\eta_\tau$$

When using the linear schedule, the parameters to choose are η_0, η_τ, and τ. Usually τ may be set to the number of iterations required to make a few hundred passes through the training set. Usually η_τ should be set to roughly 1 percent the value of η_0. The main question is how to set η_0. If it is too large, the learning curve will show violent oscillations, with the cost function often increasing significantly. Gentle oscillations are fine, especially if training with a stochastic cost function, such as the cost function arising from the use of dropout. If the learning rate is too low, learning proceeds slowly, and if the initial learning rate is too low, learning may become stuck with a high cost value.

Typically, the optimal initial learning rate, in terms of total training time and the final cost value, is higher than the learning rate that yields the best performance after the first 100 iterations or so. Therefore, it is usually best to monitor the first several iterations and use a learning rate that is higher than the best-performing learning rate at this time, but not so high that it causes severe instability. (preliminary trials or grid-search).

Adaptive Learning Rates

- Use separate eta for each parameter
- Automatically adapt during training
- Possibly avoiding/reducing the fine tuning phase via hyperparametrs selection

E.g.

- AdaGrad
- RMSProp (becoming quite popular for DL, e.g. caffe SW)
- Adam (also popular, often robust with the default value)
- ... continuously evolving

Combined with moment etc. Popular for DL: see DL book 8.5 if interested.

4.10.6 Momentum

★ Backpropagation + Momentum

Remember that the *movements* will be made iteratively according to

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \eta \cdot \Delta\mathbf{w}$$

That can be rewritten as

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \Delta\mathbf{w}_{new}$$

where

- **Without momentum**

$$\Delta w_{new} = -\eta \frac{\partial E(\mathbf{w})}{\partial w}$$

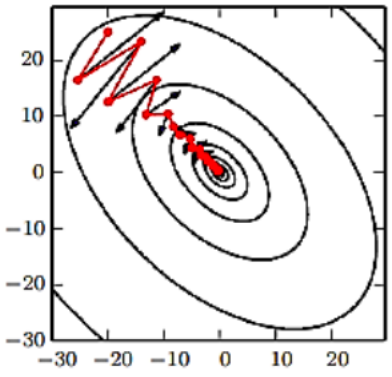
- **With momentum**

$$\Delta w_{new} = -\eta \frac{\partial E(\mathbf{w})}{\partial w} + \alpha \Delta w_{old}$$

  After you can save Δw_{new} for the next step ($\Delta w_{old} = \Delta w_{new}$).

Momentum parameter α s.t. $0 < \alpha < 1$, e.g 0.5 - 0.9

- Effect on a canyon of the error surface (# poor conditioning of the Hessian matrix).
- In **red** the path with the momentum, lengthwise traversing the valley toward the minimum
- Instead of zig-zag along the canyon walls (following the **black** directions given by the gradient)



- Faster in plateaus: same sign of gradient for consecutive iterations → momentum increases the delta

- Damps oscillations: stabilizing effect in directions that oscillate in sign
- *Inertia effect*: Allow us to use higher η $(0.2 - 0.9)$
- Momentum born with **batch** learning, and it is commonly assumed that helps more in batch mode
- In the **on-line** version can be used resorting to a moving average of the past gradients concept to smooth out the stochastic gradient samples

  [e.g. see it in the form grad= alfa grad +(1-alfa) grad_now]

  - It smooths the gradient over different examples (a different meaning w.r.t. the batch version that uses the same batch of examples)
  - You can use it multiplying by eta (as usual) for the DeltatW
  - See if really needed (as any other heuristic)
- For **Minibatch**: moving average of past gradients over the progressive minibatchs (that have different examples, not the step before for the same examples as for batch).

■ **Nesterov Momentum**

Evaluate the gradient after the momentum is applied

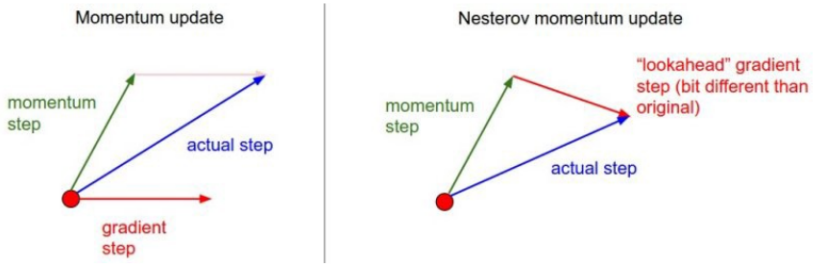
- First apply the momentum ($\underline{\mathbf{w}} = \mathbf{w}_{old} + \alpha \Delta \mathbf{w}_{old}$)
- Evaluate the *new gradient* on this interim point (with such $\underline{\mathbf{w}}$)
- Compute and apply the DeltaW as before (summing the momentum and the *new gradient*)

So, the new delta rule is:

$$\underline{\mathbf{w}} = \mathbf{w}_{old} + \alpha \Delta \mathbf{w}_{old}$$

$$\Delta w_{new} = -\eta \frac{\partial E(\underline{\mathbf{w}})}{\partial w} + \alpha \Delta w_{old}$$

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \Delta \mathbf{w}_{new}$$



(left)The old way. Instead of going towards the gradient step, sometimes the movement is towards a different direction thus wasting time (right) Nesterov momentum calculates the step to be taken in future and takes the corrective action

Shown to improve the rate of convergence for the batch mode (not for the stochastic case!). Popular in SW libraries such is Caffe (DL software)

4.10.7 optimizers

- As any iterative approaches…. it is generally efficient (linear in the number of free parameters) but depends on the number of training data and on the number of epochs
  - Second-order techniques, or other from CM? See next slide!
- Lots of other heuristics approaches (in the NN literature):
  - *Quick-prop*: goto to the minima of the local estimated convex function
  - *R-prop* does not use the value of the gradient (it can vanish for deep layers) but the sign of gradient
  - Deep Learning need special care (see later)

Second order methods

Use the second order derivatives (Hessian matrix): more information about the curvature of the objective function → better descent (*Newton's methods)*

- But the Hessian can be too large (i.e. computationally expensive), and see the next "saddle points" slide

In the form of *approximate* second order methods we find:

- *Approx. Newton's methods* (also *Gauss-Newton* and *Levenberg-Marquardt* approximation)
- Hessian free approaches (without computing H and H^{-1})
  - *Conjugate gradients*
    - can be nice for faster converge! (see e.g. in "Numerical Recipes in C").
  - *Quasi-Newton methods: BFGS* (Broyden–Fletcher–Goldfarb–Shanno Algorithm)
- …

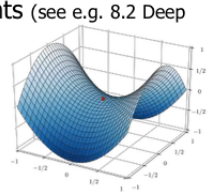
- References: Haykin book, DL book

- Possible to try them within a CM/ML prj? Yes, especially with ad hoc variations for NNs

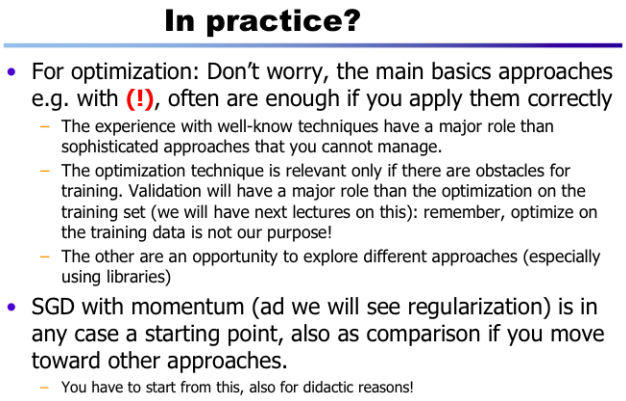
Means technical aspect; Not needed at the beginning

Saddle points #tech

- More intriguing discussion on saddle points (see e.g. 8.2 Deep Learning book)
- i.e. Gradient (close to) zero but not a local minima



- Saddle points can grow exponentially (wrt to local minima) in high dimensional spaces
- This can explain why (standard) *second-order methods* (looking to Hessian, e.g. *Newton's method*) have not succeed in replacing gradient descent for NN training
- Instead, Gradient descent empirically shows to escape saddle points […]
- But the length of the training path can be high for many other reason not related to local minima or saddle points (small gradient values), such as time to "circumnavigate" local maxima and so on…
- But such cases are not frequent! Why? Research is on-going…! *Typically, we are able to find a low value of the loss quickly enough to be useful!* And especially useful to generalize! (we don't need a minimum at all, see previous slides on multiple/global minima!).



4.10.8 Stopping criteria

There are many approaches:

- The basic is the used loss ($loss < k$); the best if you know the **tolerance of data (expert knowledge)**.
- Max (instead of mean) tolerance on data.
- Classification: Number of misclassified (error rate).
- No more relevant weight changes/ near zero gradient (Euclidian norm of gradient $< \epsilon$). No more significant error decreasing for a epoch (e.g. less than 0.1%).
  NOTE: may be premature (e.g. if eta is too small)! It can be applied observing k epochs.
- ★ In any case stop after an excessive number of epochs.
- Not necessarily stop with very low training error...we may be in overfitting (see next). We want a model that is able to generalize the data, so we have to do Validation!

4.10.9 Overfitting and regularization

■Stopping at the minimum?
Typically we don't want the global minimizer of $R_{emp}(w)$, as this is likely to be an overfitting solution (difference with respect to optimization (CM) methods).

The control of complexity it is our main aim to achieve the best generalization capability.
For instance we need to add some *regularization*:

- this is achieved directly through a *penalty term*, or indirectly by *early stopping*.
- Plus model selection (cross-validation) on empirical data to find the trade-off.

■Overfitting in NN
A neural network that is designed to generalize well will produce a correct input–output mapping even when the input is slightly different from the examples used to train the network, as illustrated in the figure. When, however, a neural network learns too many input–output examples, the network may end up memorizing the training data. It may do so by finding a feature (due to noise, for example) that is present in the training data, but not true of the underlying function that is to be modeled. Such a phenomenon is referred to as **overfitting** or **overtraining**. When the network is overtrained, it loses the ability to generalize between similar input–output patterns.

- Start learning with small weights (symmetry breaking).
- The mapping input-output is nearly linear:

  - number of effective free parameters (and VC-dim) nearly as in perceptron

- As optimization proceeds hidden units tend to *saturate*, *increasing the effective number of free parameters* (and introduce non-linearities) and hence *increase VC-dim.*
- A variable-size hypothesis space.

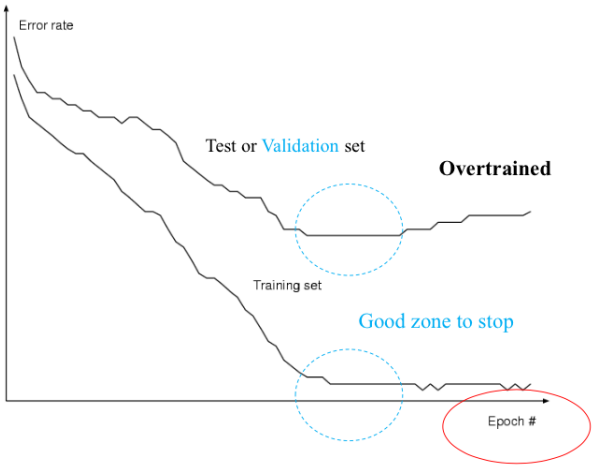


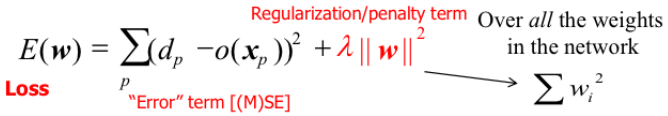
Figure 4.21: Typical behavior of backprop learning curve

So, how we can avoid the overfitting?

■Learning curve analysis
Recall the VC-bound plot! (theoretical ground). This is the learning curve plot for training and validation, same thing with different name. How to act?

- *Early stopping*: use a **validation set** for determining when to stop (at the validation error increasing ... quite vague!)
  - Use more than one epoch before estimating (patience).
  - Backtracking approaches.
  - Note that since effective number of parameters grows during the course of training, halting training correspond to *limit the effective complexity.*
- *Regularization* (on the loss : our main warhorse!)
- *Pruning methods* (see later)

★Regularization
Training → weight increases: we can optimize the loss considering the weight values.
A well know principled approach (related to **Tikhonov theory**):



- Effect: **weight decay** (basically add $2\lambda w$ to the gradient for each weight)

$$\boldsymbol{w}_{new} = \boldsymbol{w}_{old} + eta * \Delta \boldsymbol{w} - 2\lambda \boldsymbol{w}_{old}$$

- Chose the lambda parameters (generally very low value, e.g. 0.01): selected by the model selection phase (cross-validation)
- Note: if applied to linear model → it is the *ridge regression* *(see lect .on linear models)*
- More sophisticated penalty terms have been developed (e.g. for *weight elimination*: see Haykin)

Often we simplified calling equivalently (mean) Loss or Error or Risk the objective function. When we have regularization, as in the Tikhonov regularization, a penalty term in the objective function is better to clarify:

- We can use **Loss** name for the objective function $(MSE + Penalty)$ *for model training.*
- And **Error**/Risk for the "data term" (the (M)SE, depending on data)

$$\sum_p (d_p - o(\mathbf{x}_p))^2$$

to evaluate model error, because this is the measure useful to the user (using the model), and to be reported in the table or plots etc. (★).

■Regularization: details

- Note that often the bias w_0 is omitted from the regularizer (because its inclusion causes the results to be not independent from target shift/scaling) or it may be included but with its own regularization coefficient (see Bishop book, Hastie et al. book).
- Typically apply it in the batch version.
- For on-line/mini-batch take care of possible effects over many steps (patterns/examples): hence to compare with respect to batch version in a fair way do not force equal lambda but it would be better to use

$$\frac{\lambda \times mb}{l}$$

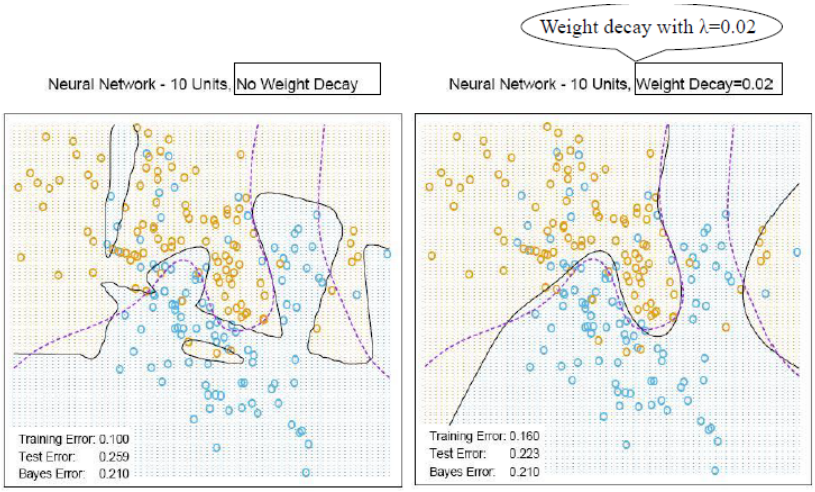
where l is the number of total patterns.

  - Of course if you choose lambda by model selection they will automatically select different lambda for on-line and batch (or any mini-batch)

- Other techniques: **Dropout** (see later).

Remember, regularization is for the control of the VC-dim, it has no relation with stability to the error of learning rate of the model.
So Regularization vs Learning Curve for the stopping criteria. Learning Curve is an empirical approach instead the regularization approach is not, the learning curve for validation will follow the learning curve for the training, we can stop with some epsilon.

■Effect of regularization



■Merging Weight decay + Momentum

- Merging Weight decay + Momentum ....

$$\Delta w_{ti} = -\eta \frac{\partial Loss(\boldsymbol{w})}{\partial w_{ti}} = \eta \delta_t x_i - \lambda w_{ti}$$ For the Regularization

$$\Delta w = -\eta \frac{\partial E(\boldsymbol{w})}{\partial w} + \alpha \Delta w_{old}$$?

We wrote for the Momentum, but what is E(w)?
Better a MSE error

So, does **eta** multiply lambda?
No, considering the Loss=Error+Penalty

$$\Delta w = \eta \delta x - \lambda w + \alpha \Delta w_{old}$$

$$\boldsymbol{w}_{\text{new}} = \boldsymbol{w}_{\text{old}} + \Delta \boldsymbol{w}$$

And now, does Δw include the lambda part**?** Often yes, e.g. Matlab
But we can also keep separation of hyperparameters writing:

$$\Delta w_{ti} = \eta \delta_t x_i + \alpha \Delta w_{old_{ti}}$$ Adding the lambda part directly to w update (separating it from the momentum memory)

$$w_{ti} = w_{ti} + \Delta w_{ti} - \lambda w_{ti}$$

So you can better control the different role of Eta, alfa and lambda

A further note: divide by *mb* or *l* only the gradient adding the sum over the patterns

4.10.10 Number of hidden units

- Related to the **control of complexity** issue,
- The number can be high if appropriate regularization is used.
- Related also to the input dimension and to the size of the TR data set.

In general, is a **model selection** issue:

- To few hidden units → underfitting
- To many hidden units → overfitting (can occur)

For example by cross-validation on a TR set (part for training) with validation set to select the model (number of units) + external test set to evaluate:

- Of course retrain the model for each different configuration!

There are two approaches:

- **Constructive approaches**: the learning algorithm decide the number starting with small network and then adding new units.
- **Pruning methods**: start with large networks and progressivelly elimnate weights or units.

■**Constructive approaches**
Incremental approach: algorithms that build a network starting with a minimal configuration and adds new units and connections during training.
Examples:

- For classification: Tower, Tiling, Upstart
- For both regression and classification: Cascade Correlation

An exemplificative and effective approach : the **Cascade Correlation** learning algorithm:

- Start with a minimum networks.
- Add units if they are needed until the error is low.

It learn both the network weights and network topology (number of units).

- automatic determination of network dimension and topology: CC allows to deal with hypothesis spaces of flexible size, since the number of hidden units is decided by the learning algorithm;
- training of a single unit for each step.

See Scott E. Fahlman and Christian Lebiere: The Cascade-Correlation Learning Architecture 1991 CMU-CS-90- 100 Carnegie Mellon University Also in NIPS 2, 1990, pages 524-532

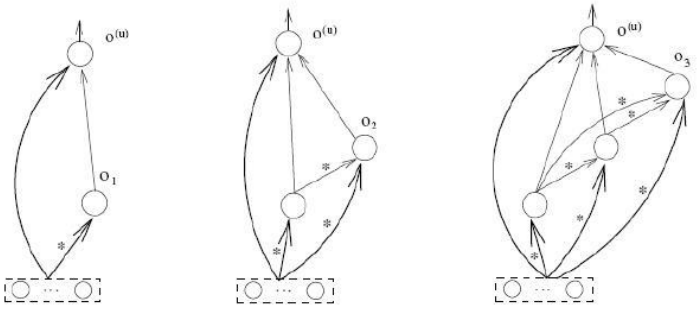
The CC algorithm

- **Starting** N0 network is a network without hidden units: Training of N0. Compute error for N0. If N0 cannot solve the problem go to N1.
- In the N1 network a hidden unit is added such that the ***correlation*** between the output of the unit and the residual error of network N0 is maximized (by training its weights, see next slide).
- After training, the weights of the new unit are **frozen** (they cannot be retrained in the next steps) and the remaining weights (output layer) are retrained.
- If the obtained network N1 cannot solve the problem, new hidden units are progressively **added** which are connected with all the inputs and previously installed hidden units.
- The process continues until the residual errors of the output layer satisfy a specified **stopping** criteria (e.g the errors are below a given threshold).

The method dynamically builds up a neural network and terminates once a sufficient number of hidden units has been found to solve the given problem.

Cascade of units

The evolution of a CC network with up to 3 hidden units



* = weights frozen after candidate training

- Specifically, the algorithm works interleaving the minimization of the total error function (LMS), e.g. by a simple backpropagation training of the output layer, and the **maximization** of the (non-normalized) correlation, i.e. the covariance, of the new inserted hidden (candidate) unit with the residual error:

$$S = \sum_k \left| \sum_p (O_p - mean_p(O))(E_{k,p} - mean_p(E_k)) \right|$$

$E_{k,p} = (o_{k,p} - d_{k,p})$

E: residual error with $o_{k,p}$ at the output layer

p: pattern. k: output unit

O (in eq. for S): candidate output

- **Exercise:** derivation (sketch on blackboard)
- Result: $\frac{\partial S}{\partial w_j} = \sum_k \operatorname{sgn} S_k \sum_p (E_{k,p} - mean_p(E_k)) f'(net_{h,p}) I_{j,p}$

h : candidate index (not needed)

- Weight update: why + dS/dw_j ?

CC Maximization of the correlation

$$S = \sum_{k}^{n} \left| \sum_{p}^{l} (O_p - mean_p(O))(E_{k,p} - mean_p(E_k)) \right|$$

In order to maximize S, we must compute $\frac{\delta S}{\delta w_j}$, the partial derivative of S with respect to each of the candidate unit's incoming weights, Wi. In a manner very similar to the derivation of the back-propagation rule in [Rumelhart, 1986], we can expand and differentiate the fonnula for S to get:

$$\frac{\delta S}{\delta w_j} = \sum_{k}^{n} sgn(S_k) \sum_{p}^{l} (E_{k,p} - mean_p(E_k)) f'(net_{h,p}) I_{j,p}$$

Derivation:

$$= \frac{\delta S}{\delta w_j} = \frac{\delta \sum_k^n \left| \sum_p^l (O_p - mean_p(O))(E_{k,p} - mean_p(E_k)) \right|}{\delta w_j} =$$

$$\sum_{k}^{n} sgn(S_k) \frac{\sum_p^l \delta[(O_p - mean_p(O))(E_{k,p} - mean_p(E_k))]}{\delta w_j}$$

the internal component of $\sum_p^l$ can be rewritten as:

$$\frac{\delta[(O_p - mean_p(O))(E_{k,p} - mean_p(E_k))]}{\delta w_j} = (E_{k,p} - mean_p(E_k)) \frac{\delta(O_p - mean_p(O))}{\delta net_p} \frac{\delta net_p}{\delta w_j} =$$

$$= (E_{k,p} - mean_p(E_k)) \frac{\delta(O_p - mean_p(O))}{\delta O_p} \frac{\delta O_p}{\delta net_p} \frac{\delta net_p}{\delta w_j}$$

we know that the average $mean_p(O)$ is independent of p so:

$$\frac{\delta(O_p - mean_p(O))}{\delta O_p} = 1 \ , \ \frac{\delta O_p}{\delta net_p} = f'(net_{h,p}) \ \text{ (where } h \text{ is the candidate index)}$$

and we can rewrite:

$$\frac{\delta net_p}{\delta w_j} = I_{j,p} \ \text{ input candidate from unit j and pattern p}$$

by bringing it all together we found the solution:

$$\frac{\delta S}{\delta w_j} = \sum_{k}^{n} sgn(S_k) \sum_{p}^{l} (E_{k,p} - mean_p(E_k)) f'(net_{h,p}) I_{j,p}$$

Of course, the delta is positive in this case (compared to the case of minimizing the error with backprogration) so we are doing gradient ascent (gradient descent is for the minimization).

Notes on CC

- The role of hidden units is to reduce the residual output error
  - It solve a specific sub-problem
  - It became a permanent "feature-detector"
- POOL: Typically, since the maximization of the correlation is obtained using a gradient ascent technique on a surface with several maxima, a pool of hidden units is trained and the best one selected. This help avoid local maxima.

- Greedy algorithms: easy to obtain converge, may also find a minimal number of units but it may lead into overfitting (need regularization).

Loss: Not necessarily "max S": e.g.,for regression, LMS in Prechelet 1997, Neural Networks Vol. 10 pp. 885-896

4.10.11 Input scaling / Output representation

■Input
Preprocessing can have large effect.
For different scale values, normalization via :

- *Standardizations* (often useful):
  For each feature we obtain zero mean and standard deviation 1: $v - mean/standard\ dev$.
- *Rescaling*: [0,1] range; $v - min/(max - min)$

Remember that

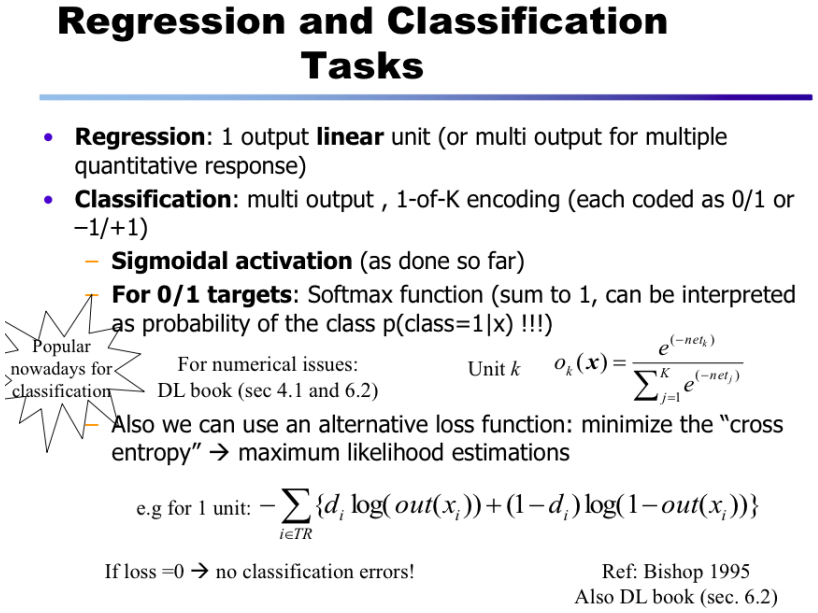
- *Categorical inputs*:
  A → 100, B → 010, C → 001
- *Missing data*: 0 does not mean "no input" if 0 is in the input range!

■Output
Classification
1-of-K with multioutput

- Sigmoid → choose the threshold to assign the class.
- Rejection zone.
- 1-of-K: the highest value
  - Often symmetric logistic learn faster (see Haykin)
- 0.9 can be used instead of 1 (as d value) to avoid asymptotical convergence (-0.9 for –1 or 0.1 for 0)

Of course, target range must be in the output range of units.

Regression
Output **linear** unit(s).



Open questions:

- Reflect on the role of the different hyper-parameters
- Connect to the theory (also later after next lectures)
  - E.g. relate the optimization of the loss with penalty term to the Statistical Learning Theory (VC-bound in the introduction and next lectures) to understand the role of lambda in controlling under/overfitting cases.

Hence, which are the most relevant for your project?

5 Validation

The generalization performance of a learning method relates to its predic- tion capability on independent test data. Assessment of this performance is extremely important in practice, since it guides the choice of learning method or model, and gives us a measure of the quality of the ultimately chosen model. In this chapter we describe and illustrate the key methods for performance assessment, and show how they are used to select models.

5.1 Motivation (Mitchell)

In many cases it is important to evaluate the performance of learned hypotheses as precisely as possible. One reason is simply to understand whether to use the hypothesis. For instance, when learning from a limited-size database indicating the effectiveness of different medical treatments, it is important to understand as precisely as possible the accuracy of the learned hypotheses. A second reason is that evaluating hypotheses is an integral component of many learning methods.
Estimating the accuracy of a hypothesis is relatively straightforward when data is plentiful. However, when we must learn a hypothesis and estimate its future accuracy given only a limited set of data, two key difficulties arise:

- **Bias in the estimate**. First, the observed accuracy of the learned hypothesis over the training examples is often a poor estimator of its accuracy over future examples. Because the learned hypothesis was derived from these examples, they will typically provide an optimistically biased estimate of hypothesis accuracy over future examples. This is especially likely when the learner considers a very rich hypothesis space, enabling it to overfit the training examples. To obtain an unbiased estimate of future accuracy, we typically test the hypothesis on some set of test examples chosen independently of the training examples and the hypothesis.
- **Variance in the estimate**. Second, even if the hypothesis accuracy is mea- sured over an unbiased set of test examples independent of the training examples, the measured accuracy can still vary from the true accuracy, de- pending on the makeup of the particular set of test examples. The smaller the set of test examples, the greater the expected variance.

5.2 A premise: Bias-Variance

Bias-Variance decomposition provides an useful framework to understand the validation issue, showing how the estimation of a model performance is difficult

- Considering also the role of different training set realizations
- Showing again the need of a trade-off between fitting capability (bias) and model flexibility (variance) in different way

Under/Over-fitting in a Bias-Variance plot

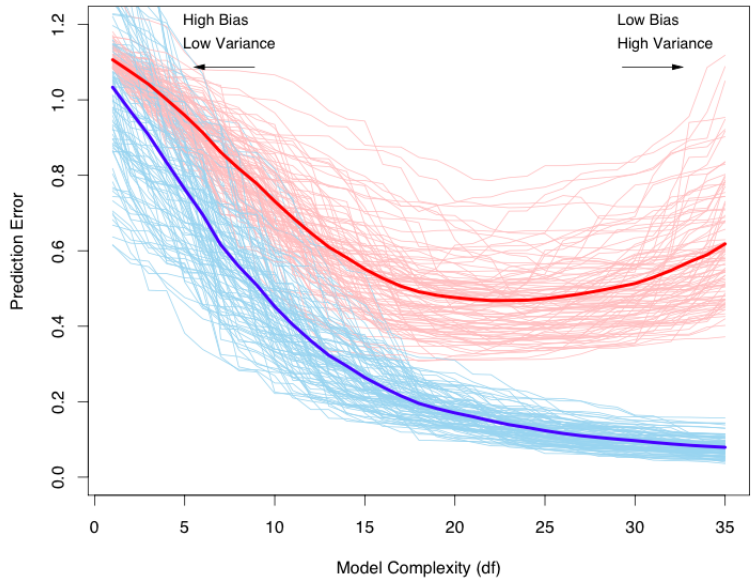


Figure 5.1: Behavior of test sample and training sample error as the model complexity is varied. The light blue curves show the training error $\overline{err}$, while the light red curves show the conditional test error Err_T for 100 training sets of size 50 each, as the model complexity is increased. The solid curves show the expected test error Err and the expected training error $E[\overline{err}]$.

In fig 5.1 illustrates the important issue in assessing the ability of a learning method to generalize. Consider first the case of a quantitative or interval scale response. We have a target variable Y, a vector of inputs X, and a prediction model $h(X)$ that has been estimated from a training set T. The loss function for measuring errors between Y and f(X) is denoted by $L(Y, h(X))$ (E.g. mean (square) error).

Unfortunately training error is not a good estimate of the test error, as seen in Figure 5.1. Training error consistently decreases with model complexity, typically dropping to zero if we increase the model complexity enough. However, a model with zero training error is overfit to the training data and will typically generalize poorly.

- we are looking for the best solution (minimal prediction – test error) searching a balancing between fitting (accuracy on training data) and model complexity
- Training set is not a good estimate of test error. Initially too many bias (under fitting, high TR error) than too high
- Assuming a tuning parameters θ (implicit or explicit) that varies the model complexity, we wish to find the value of θ that minimizes test error variance (low TR error but over fitting)
- Look for methods for estimating the expected error for a model

We can approximate the validation step:

- **analytically**
  - AIC, BIC (Akaike/Bayesian Information Criterion)
  - MDL (Minimum Description Length)
  - SRM: Structural Risk Minimization and VC-dimension
- **re-sampling**: efficient sample re-use
  - cross-validation: hold-out, K-fold CV, ...
  - bootstrap

In practice, often we can approximate the estimation on the data set by re-sampling.

5.3 Model Selection and Assessment

It is important to note that there are in fact two separate goals that we might have in mind:

- **Model selection**: estimating the performance of different learning models in order to choose the best one (to generalize) – this includes search the best hyper-parameters of your model (e.g. polynomial order, number of units in a NN, lambda, eta, ...).

It returns a model

- **Model assessment**: having chosen a final model (or a class of models), estimating/evaluating its prediction error/ risk (generalization error) on new test data (measure of the quality of the ultimately chosen model)

It returns an estimation value

So, usually the dataset is divided in three disjoint sets: a training set, a validation set, and a test set.

The **training set** is used to fit the models; **the validation set** (or selection set) is used to estimate prediction error for model selection and can be used to select the best models (e.g. hyper-parameters tuning);

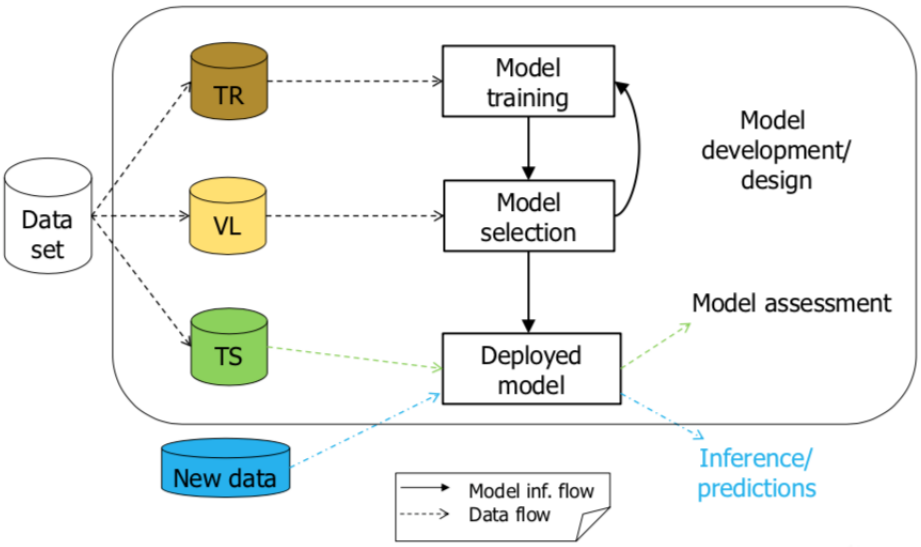
TR+VL sometimes are joitnly called development/design set
i.e. used to build the final model

The **test set** is used for assessment of the generalization error of the final chosen model. The test set should be kept in a "vault," and be brought out only at the end of the data analysis.

Suppose instead that we use the test-set repeatedly, what if test set is used in a (repeated) design cycle and choosing the model with smallest test-set error?

- We are making model selection and not reliable assessment (estimation of expected generalization error) and we need a new test set for the generalization error.
- In that case, used test set error provide an overoptimistic evaluation of the true test error ($\rightarrow$ we will see how easy is to obtain very high classification accuracy over random task even using the test set only implicitly)

TR/VL/TS by a simple schema



Gold rule: *Keep separation between goals and use separate data sets*

5.3.1 Counterexample

Suppose we have:

- 20-30 examples, 1000 random value input variables,
- random target 0/1
- We select 1 model with 1 input that guess (for chance) 99% on any tr and vl and ts set splitting (made after initial selection).

Perfect result (a model with accuracy 99%)? What is wrong? 99% is not good estimation of the test error (the right one is 50%).

Example:

#		25	26	27		
1	...	1	1	1	...	1
2	...	0	0	0	...	0
3	...	1	1	1	...	1
4	...	0	0	0	...	0
5	...	0	0	0	...	0
6	...	1	0	1	...	1
7	...	1	0	0	...	1

Training set is composed by Rows 1-5 and validation set by 6-7. If we choose the feature that share all the same value with the target column (25) we will get 100 of accuracy, but this is not true. So:

1. Estimation of error on TR and VL is NOT good risk estimation.
2. Using the entire dataset for feature/model selection prejudice the estimation (biased estimation, called subset selection bias). Test set was implicitly used at the beginning. Test set must be separated in advance, before making any model selection (even of the Feature selection) !

5.4 Cross-Validation

Generically we have used the term "cross validation" for:

- An approach to model selection based on direct estimation of prediction error by resampling (instead of analytical). **Choose hyperparmetrs estimating errors**.
- More specifically for implementations of estimation methods with resampling specifing how to divide data for both model selection and assessment. How to make partitions of the data set : e.g by K-fold CV.

5.4.1 Hold out

If we are in a data-rich situation, the best approach for both problems is to randomly divide the dataset into three parts: a training set, a validation set, and a test set.

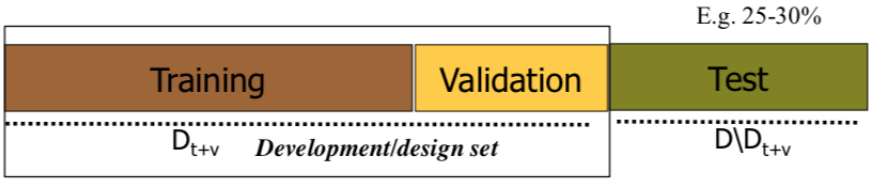


Figure 5.2: Caption

It is difficult to give a general rule on how to choose the number of observations in each of the three parts, as this depends on the signal-to-noise ratio (of the underlying function) in the data and the training sample size. A typical split might be 50% for training, and 25% each for validation and testing.

5.4.2 Hold out and K-fold cross validation

(a) k mutually exclusive subsets

Fold-1	Res1
Fold-2	Res2
Fold-3	Res3
....	...
TOTAL	**MEAN +/- SD**

(b) mean and standard deviation

Figure 5.3: k-fold cross validation

When we have a small number of data, Hold out Cross Validation can make insufficient use of data. To solve this problem we can use another technique called: **K-fold Cross-Validation**:

- Split the data set D into k mutually exclusive subsets $D_1, D_2, \ldots, D_k$
- Train the learning algorithm on $\frac{D}{D_i}$ and test it on D_i repeat this phase k time and then we take the mean of the validation results.
- Can be applied for both VL or TS splitting

- It uses all the data for training and validation/testing

But this technique has some issues:

- How many folds? 3-fold, 5-fold , 10-fold, ..., leave-one-out (when k = n)
- Often computationally very expensive (we have to perform training and validation phases k times)

Combinable with validation set, double-K-fold CV, ...

How many K?
It is interesting to wonder about what quantity K-fold cross-validation estimates. With K = N (Leave-one-out), the cross-validation estimator is approximately unbiased for the true (expected) prediction er- ror, but can have high variance because the N "training sets" are so similar to one another. The computational burden is also considerable, requiring N applications of the learning method. On the other hand, with K = 5 cross-validation has lower variance, we might guess that it estimates the expected error Err, since the training sets in each fold are quite different from the original training set.

5 or 10-fold CV are recommended as a good compromise (among bias and variance of estimation) with respect to Leave-one-out cross-validation (LOOCV) (Hastie et al. and the references therein)

5.5 An example of model selection and assessment

- Split data in TR and Test set (here simple hold-out or a K-fold CV)
- *[Model selection]* Use K-fold CV (internal) over set, obtaining TR e VL set in each folder, to find best hyper-parameters of your model (e.g. polynomial order, lambda of ridge regression, number units in a NN ...): Grid-search with many possible values of the hyper-parameters. For example a k-fold-CV for $\lambda = 0.1$, a CV for $\lambda = 0.01$, ... and then take the best λ (by the mean over the validation sets for each fold)
- Train on the whole TR set the final model
- *[Model assessment]* Evaluate it on the external test set

5.5.1 Lucky/Unlucky sampling

Can we avoid to be sensible to the particular partitioning of examples? (bias of the particular sample)

- Stratification procedure: stratification is the process of grouping members of the population into relatively
- Classification: for each (random) partition (TR and TS in hold-out/CV) each class is represented in approximately the same proportions as in the full data set
- Folder composition: check if order in data related to specific meaning
- Repeated hold-out method or CV: repeat splitting with different random sampling
  - E.g. repeat the CV 10 times an average the results to yield an overall estimation
  - Useful Especially for comparing different models !

  homogeneous subgroups before sampling.

5.5.2 Very few data (informal)

With very few data: difficult to say whether a sample is representative or not ....

- Stratification
- Avoid (or consider in evaluation):
  - Missing classes or features in training data
  - Special classes of data not sampled in TR or TS (again stratification)
  - Prior known outliers can affect the mean test results
  - On the opposite: selection of only "easy" cases
- blind test set can be misleading if it is
  - From a different distribution
  - Measured with different scale, different tolerance, etc.
  - Uncleaned, unpreprocessed, ...
  - Extrapolation (out of range data)

5.6 Searching hyperparameters

Parameters that are not directly learnt within estimators can be set by searching a hyperparameters space (of course by model selection on a validation set). Search best hyper-parameter values, two approach

- Exhaustive Grid Search: exhaustively generates candidates from a grid of parameter values.

TRIVIAL EXAMPLE:

Hyper-param.	Lambda 0.1	Lambda 0.01	Lambda 0.001
1 unit	Res1	Res4	Res7
10 units	Res2	Res5	Res8
100 units	**Res3**	Res6	Res9

Example: The best one is *Res3* → (Units=100, lambda=0.1) is the winner

AUTOMATIZE IT!!!
Parallelization is easy
(independence of trials)

- Randomized Parameter Optimization: J.Bergstra,Y.Bengio,(2012). "Random Search for Hyper-Parameter Optimization" J. Machine Learning Research 13: 281–305

Grid Search:
The cost of search can be high: Cartesian product between the sets of values for each hyperparameter.

Hyper-param.	Lambda 0.1	Lambda 0.01	Lambda 0.001
1 unit	Res1	Res4	Res7
10 units	Res2	Res5	Res8
100 units	**Res3**	Res6	Res9

$$\#(\text{values in the range})^{\#\text{hyperparameters}}$$

For example $3 * 3 = 3^2 = 9$, What with 5-6 hyperparameters, each with 10 possible different values? we will have a very big number of model to find and evaluate.

Can be useful to fix some hyperparameters values in a preliminary experimental phase (since they show to be not relevant)

Two (or more) levels of grid search:

- Apply a first coarse gird search to the (other) hyperparameters with a table on the combinations of all the possible (e.g. growing exponential) values too find good intervals (regions)
- Then a finer grid-search can be performed (over smaller interval and with selected hyperparameters)

5.7 Error Functions for Evaluation

- Measuring error (addenda to slides in the intro)
- Classification: see first lessons:
  - Accuracy (correct classification rate/error rate often in %), confusion matrix (specificity, sensitivity, ROC curve)
- Regression: residue $r_i=(y_i-o_i)$ y = *target*
  - MS error ($mean_i\,[r^2]$)→ RMS (root mean square) or S
  - Mean absolute error (mean of the absolute error |r|)
  - Max absolute error ($max_i\,[|r_i|]$)
  - R (correlation coefficient/index) (R^2 coeff. of determination):

  E.g. $R=sqrt(1-(S^2/S_y^2))$ where $S_y^2=mean_i\,[\,(y_i-mean_i\,[y])^2\,]$ *(variance of y)*
    - degree of linear dependence between the variables y e o in [0-1], where 1 is the best
  - Output versus targets plots
  - Various statistical tests and Statistical significance analysis

5.8 Bootstrap

Bootstrap is a random resampling with replacement and repeat it to form different subsets (vd or ts)

6 SLT

6.1 Structural Risk Minimization (SRM) & VC-dimension

The feasibility of supervised learning depends on the following key question:
Does a training sample consisting of N independent and identically distributed examples

$$(\mathbf{x}_1, d_1), (\mathbf{x}_2, d_2), ..., (\mathbf{x}_N, d_N)$$

contain sufficient information to construct a learning machine capable of good generalization performance? The answer to this fundamental question lies in the method of *structural risk minimization*, described by Vapnik (1982, 1998).

Here we consider a second measure of the complexity of H, called the Vapnik-Chervonenkis dimension of H (VC dimension, or $VC(H)$, for short). As we shall see, we can state bounds on sample complexity that use $VC(H)$ rather than $\mid H \mid$. These bounds allow us to characterize the sample complexity of many infinite hypothesis spaces, and can be shown to be fairly tight.

■ Shattering & VC-dim
The VC dimension measures the complexity of the hypothesis space H, not by the number of distinct hypotheses $\mid H \mid$, but instead by the number of distinct instances from X that can be completely discriminated using H. To make this notion more precise, we first define the notion of shattering a set of instances.

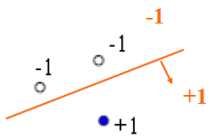
Given X (instance space), H (hypothesis space) and a binary classification, there are 2^N possible **dichotomies** (partitions or labeling of the N points by -1 or $+1$). A particular dichotomy is **represented** in H if there exists a hypothesis h in H that realizes the dichotomy.

Def: *H shatters X iff H can represent all the possible dichotomies on X (0 errors).*

- the points in X can be separated by an h in H in all the possible way
- for every possible dichotomy of X, there exists a consistent hypothesis h in H

Example:

- 3 points in IR^2

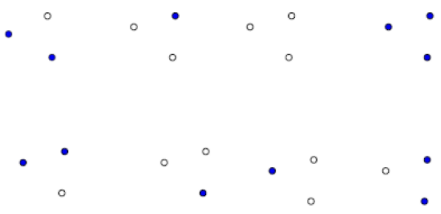


- H as set of lines $h(\boldsymbol{x}) = sign(\boldsymbol{w}\boldsymbol{x} + w_0), \boldsymbol{x} \in IR^2$

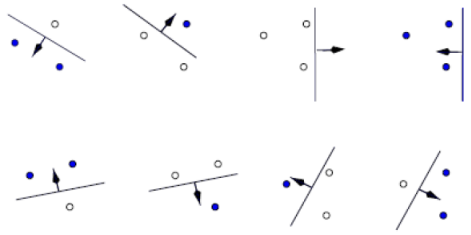
■ A dichotomy is a particular labeling (in {-1,+1}) of the points

- This specific dichotomy can be represented in H :
  - there exist a line that correctly separates the points

- All the possible 2^N dichotomies on 3 point in IR^2



- The class of linear functions H shatters X



The ability to shatter a set of instances is closely related to the inductive bias of a hypothesis space. Recall that an unbiased hypothesis space is one capable of representing every possible concept (dichotomy) definable over the instance space X. Put briefly, an unbiased hypothesis space H is one that shatters the instance space X. What if H cannot shatter X, but can shatter some large subset S of X? Intuitively, it seems reasonable to say that the larger the subset of X that can be shattered, the more expressive H. The VC dimension of H is precisely this measure.

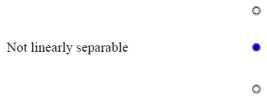
Def: *The **VC dimension** of a class of functions H is the maximum cardinality of a set (configuration) of points in X that can be shattered by H.*

$VC(H) = p$

- H *shatters* at least one set (configuration) of p points
- H *cannot shatter* any set (configuration) of $p + 1$ points

If arbitrarily large (but finite) sets of X can be shattered by H, then VC is infinite.

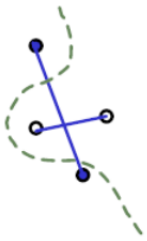
Note that however not all the possible configuration of N (in the example $N = 3$) points can be shattered. But it is sufficient to find one configuration of N points which is separable for every labeling.



In general the VC dimension of a class of linear (separator/decision) hyperplanes (LTU) in a **n**-dimensional space is **n+1**.

so the VC-dim of a linear function is:

- **VC(H)≥3** (shown on the 3 points)
- **VC(H)<4**



Can always draw six lines between pairs of four points.
Two of those lines will cross.
If we put points linked by the crossing lines in the same class they can't be linearly separated (do you remember the XOR?)

So a line can shatter 3 points but not 4

- Conclusion: **VC(H) = 3**, H hyperplanes in IR^2 (LTU)

■ **Number of parameters and VC-Dim** Is VC the number of parameters? It can be related but not really the same:

- Of course, we may add redundant free parameters
- there exist models with one parameter and infinite VC dim

Moreover, for nearest neighbor VC-dim is infinite: The VC dimension of a 1-nearest-neighbor classifier is infinite. Given any set of points, it will correctly classify 100% of those points since each point is the closest point to itself. In this case the number of parameters in the k-nn is infinite, because is infinite the number of patterns (N) and the number of parameters is given by N/k. Infact, with $K = 1$ we have overfitting with K-nn and the error is really high and the VC-dim is very high.

In neural networks the number of units (so more weights to the net) can give an high complexity and an high VC-dim.

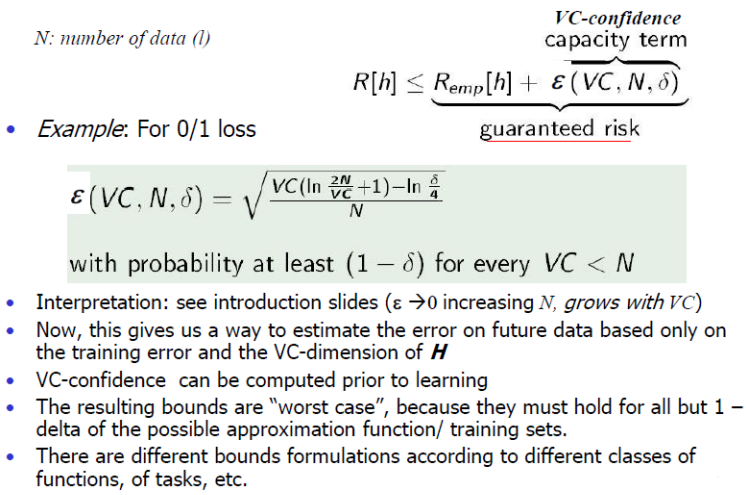
Other Note on K-NN and VC-Dim
At first glance, k-nearest neighbors has a single parameter, that is k, the number of neighbours to be included in deciding on the majority-vote predicted classification.

However, the effective number of parameters to be fit is not really k but more like n/k. This is because the approach effectively divides the region spanned by the training set into approximately n/k parts and where each part is governed by the majority vote classifier.

To form an analogy with a parameterised model (e.g. GLM) if you have a dataset of size n 50, a choice of k=25 for kNN results in an effective number of parameters of about 2 and is comparable in the extent of smoothing / degree of freedom to a linear regression fit with two coefficients.

P.S. For kNN, the effective number of parameters / the degree of complexity is known as Vapnik–Chervonenkis (VC) dimension.

■ **SLT: Analytical Bound on R**



Example of consequences:

- For many reasonable hypothesis classes (e.g., linear approximators) the VC dimension is linear in the number of "parameters" of the hypothesis.
- This shows that to learn "well", we need a number of examples that is linear in the VC dimension (so linear in the number of parameters, in this case).

■ **Structural Risk Minimization**
SRM uses VC dimension as a controlling parameter for minimizing the generalization bound on R. Assuming

finite VC we can define a nested structure of models- hypothesis spaces according to the VC-dim as in the following

$$H_1 \subseteq H_2 \subseteq ... \subseteq H_n$$

$$VC(H_1) \leq VC(H_2) \leq ... \leq VC(H_n)$$

In effect,the size of H_n is a measure of the machine capacity.

■ **SRM for model selection**

Growing VC-dim: empirical (TR) error decreases, VC-confidence increases

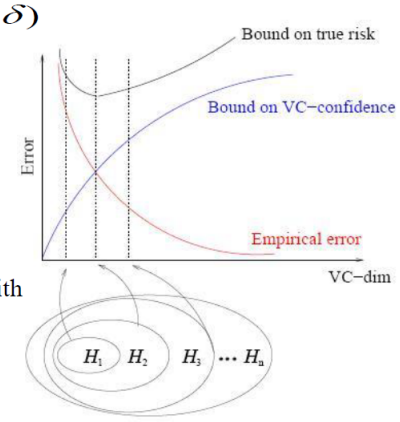
Structural Risk Minimization: find a trade-off

$$R \leq R_{emp} + \varepsilon(1/l, VC, 1/\delta)$$

- $\mathcal{H}_1 \subseteq \mathcal{H}_2 \subseteq \cdots \subseteq \mathcal{H}_n$
- $VC(\mathcal{H}_1) \leq \cdots \leq VC(\mathcal{H}_n)$

e.g. NN with increasing number of Hidden units

Model selection: choose the model (h) with the better bound on the true risk



This doesn't work in practice, infact the VC-Conf can result very conservative: hundreds of times larger than the empirical overfitting effect.

■ **SRM Approaches**

$$R \leq R_{emp} + \epsilon(1/l,\, VC,\, 1/\delta)_{VC-confidence}$$

For instance two practical approaches

- Choose appropriate structure/complexity, fix the model (and hence the VC-confidence), minimize TR error (e.g. can be used in NN)
  - However regularization due to training heuristic can further introduce a implicit SRM implementation (early stopping, ...)
- Fix the TR error, automatically minimize the VC-confidence: how ? We will see.