Machine Learning

Davide Lanza

2019-2020

Outline The main goals of the course are to **(1)** obtain an overview of the subject, **(2)** gain some practical grasp of the topic by means of lab assignments, **(3)** get practice in the tools used for that purpose and **(4)** get some exposure to tasks such as reading and discussing a research paper.

The course is an introduction to machine learning. Principles and techniques will be given especially, but not exclusively, in the area of classification. Lab+homework assignments will give students hands-on experience in building simple learning models.

Professor Stefano Rovetta (stefano.rovetta@unige.it)
DIBRIS Valle Puggia, Via Dodecaneso 35 room 205.
Phone: (010 353) 6605

References Suggested books that cover topics included in the course:

- [**DUDA2001**] R.O. Duda, P.E. Hart e D.G. Stork, *Pattern Classification*, 2nd ed., J. Wiley & Sons, 2001
- [**BISHOP2006**] C.M. Bishop, *Pattern recognition and machine learning*, New York, Springer, 2006
- [**HASTIE2009**] T. Hastie, R. Tibshirani, J.H. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed., New York, Springer, 2009
- [**FLETCHER2000**] R. Fletcher, *Practical Methods of Optimization*, 2nd ed., John Wiley & Sons, 2000
- [**GOODFELLOW2016**] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press

Book that covers a popular Python framework, not included in the course but useful:

- [**CHOLLET2017**] F. Chollet, *Deep Learning with Python*, Manning, 2017

Topic	Reference
Introductory concepts	[DUDA2001] Ch.1
Probability concepts	[BISHOP2006] Ch.1 sec. 2
Bayes decision theory and classification	[DUDA2001] Ch.2
Naive Bayes classifier	[HASTIE2009] Ch.6 sec. 6.3
Linear regression	[BISHOP2006] Ch.3 sec. 1, [HASTIE2009] Ch.3 (more details than needed)
Parameter estimation and maximum likelihood	[DUDA2006] Ch.3 Sec. 1 and 2
Bias-variance decomposition	[BISHOP2006] Ch.3 Sec. 2, [HASTIE2009] Ch.7 Sec. 3
Nearest neighbour classifiers	[DUDA2001] Ch.4 Sec. 5
Decision trees	[DUDA2001] Ch.8 Sec. 1 to 4
Random forests	[HASTIE2009] Ch.15
Neural networks	[BISHOP2006] Ch.5 Sec. 5.1-5.5
Optimization concepts	[FLETCHER2000] Ch.1, Ch.2 Sec. 2.1-2.6
Deep learning	[GOODFELLOW2016] Several parts

Contents

1 Introduction **1**
1.1 Operators . . . 1
1.2 Machine learning problems . . . 2
1.2.1 Supervised learning . . . 3
1.2.2 Unsupervised learning . . . 4
1.2.3 ML techniques . . . 5
1.3 Perceptual problems . . . 5
1.3.1 Perceptual problems examples . . . 6
1.3.2 Data cleaning . . . 8
1.4 Learning problems . . . 8
1.4.1 Learning scenarios . . . 8
1.5 Linear threshold classifier . . . 9
1.5.1 Separable data requirement . . . 9
1.5.2 Learner model . . . 10
1.5.3 Activation functions . . . 11
Lab "zero" . . . 13

2 Bayesian classification **15**
2.1 Probability theory . . . 15
2.1.1 Distribution and density . . . 16
2.1.2 Conditional probability . . . 16
2.1.3 Total probability theorem . . . 17
2.1.4 Bayes theorem . . . 17
2.2 Bayesian Decision Theory . . . 18
2.2.1 Decision and loss . . . 18
2.2.2 Quality evaluation of discrete decision set . . . 18
2.2.3 Quality evaluation of continuous decision set . . . 19
2.2.4 Bayes decision criterion . . . 20
2.3 Bayesian classifiers . . . 20
2.3.1 Minimum-error-rate classification . . . 20
2.3.2 Classifier model . . . 20
2.3.3 Naive Bayes classifier . . . 21
Lab Report 1 – Naive Bayes . . . 23

3 Linear regression **29**
3.1 One-dimensional LR . . . 29
3.1.1 LR as an optimization problem . . . 30
3.1.2 Solution of the 1D-LR problem . . . 32
3.2 1D-LR with offset . . . 33
3.3 The multi-dimensional linear regression problem . . . 34
3.4 Numerical issues . . . 36
3.5 Summary . . . 37
Lab Report 2 – Linear Regression . . . 38

4 Optimization **45**
4.1 Minimization problem . . . 45
4.1.1 Convex sets and functions . . . 46
4.1.2 Gradient and Hessian . . . 47
4.1.3 Minimum and convexity conditions . . . 48

4.1.4 Taylor polynomials . . . 49
4.2 Optimization algorithms . . . 49
4.2.1 Case 1 - Descent techniques . . . 49
4.2.2 Case 2 - Newton-Raphson method . . . 50
4.2.3 Case “1.5” - Hybrid method . . . 51
4.2.4 Case 0 - Direct search . . . 51

5 Statistical learning 53
5.1 Statistics & parameter estimation . . . 53
5.1.1 Models . . . 53
5.1.2 Model parameter estimation . . . 55
5.1.3 Learning process . . . 55
5.2 Parametric methods . . . 57
5.2.1 Maximum Likelihood parameter estimation . . . 57
5.2.2 ML and MAP . . . 58
5.3 Non-parametric methods . . . 58
5.3.1 Nearest-neighbour classifiers . . . 59
5.3.2 k-Nearest-Neighbour classifier . . . 60
5.3.3 Decision trees . . . 63
5.3.4 Random forests . . . 64
Lab Report 3 – kNN Classifier . . . 66

6 Evaluation of classifiers 71
6.1 Estimation of the generalization ability . . . 71
6.1.1 Statistical learning . . . 71
6.1.2 Sampling effects . . . 71
6.1.3 Bias/variance decomposition . . . 71
6.1.4 How to control generalization? . . . 72
6.2 Computational (empirical) estimates of generalization . . . 72
6.2.1 Resampling methods . . . 72
6.2.2 Cross-validation . . . 72
6.2.3 Leave-one-out cross-validation . . . 73
6.2.4 Bootstrap . . . 74
6.3 Empirical evaluation of classifiers . . . 74
6.3.1 Contingency tables . . . 74
6.3.2 Confusion matrix, accuracy and error rate . . . 74
6.3.3 The dichotomic case . . . 75
6.3.4 Dichotomic case: more indexes . . . 76

7 Neural Networks 79
7.1 Brain and Neural Networks . . . 79
7.1.1 Biological inspiration . . . 79
7.1.2 Neural Network model . . . 80
7.2 Single layer neural networks . . . 83
7.2.1 Rosenblatt’s perceptron (1950s) . . . 83
7.2.2 Perceptron learning algorithm . . . 83
7.2.3 Widrow and Hoff’s Adaline (1960) . . . 85
7.2.4 Adaline learning algorithm (LMS algorithm) . . . 85
7.2.5 LMS with online learning . . . 86
7.2.6 LMS algorithm and MSE minimization . . . 86
7.2.7 Two further steps . . . 87
7.2.8 The linear separability problem . . . 87
7.3 Multilayer network . . . 89
7.3.1 Topologies, UAP and learning . . . 89
7.3.2 Sigmoid activation function . . . 90
7.3.3 Differentiability of activation functions . . . 92
7.3.4 Error back-propagation algorithm . . . 92
7.3.5 Output activation (softmax) layer for classification . . . 97
7.3.6 Information entropy and cross-entropy loss . . . 98
7.3.7 Cross-entropy objective with sigmoid activation . . . 100
7.3.8 Cross-entropy objective with softmax activation . . . 101

Lab Report 4 – Neural Networks . . . 102

8 Deep learning **105**
8.1 Depth and internal representation . . . 105
8.2 Convolutional neural networks . . . 106
8.2.1 CNN: Convolutional layer . . . 107
8.2.2 CNN: Pooling layer . . . 107
8.2.3 CNN: Output layer . . . 108
8.2.4 Training a CNN . . . 109
8.2.5 Regularization methods . . . 109
8.3 Information bottleneck & unsupervised learning . . . 109
8.3.1 Autoencoders . . . 110
8.3.2 Denoising autoencoders (DAEs) . . . 112
8.3.3 Restricted Boltzmann Machines . . . 112
8.4 Deep Neural Networks . . . 113
8.4.1 Examples of successful deep networks . . . 113
8.4.2 Frameworks, languages, libraries . . . 116
8.4.3 Tain deep networks without supercomputers . . . 116

David Stuz – Notes on Goodfellow's "Deep Learning" Textbook . . . 118

Chapter 1

Introduction

How computation meets the brain? Simulating the **final result** of brain processing ($\rightarrow$ Artificial intelligence). In order to simulate the **inner mechanics** of brain processing we use Artificial neural networks (**NN**). Modern machine learning (**ML**) was mostly developed for neural networks.

For some known task we already have an **algorithm**: sorting (insert sort, bubble sort, Shell sort, radix sort, heapsort, bogosort...), spectral analysis of periodic signals (FFT), database filtering (SQL SELECT) etc... For others we still don't have any known algorithm (recognizing faces, distinguishing between genuine works by a given author and fakes, recognizing emotions from gestures or facial expression, understanding speech, controlling a soft robotic [→Sant'Anna's robot]). So, many interesting problems are too complex to admit an algorithmic solution, or even a complete description. For these problems, **only data** are available (nowadays we have lots of data from lots of sources). ML is about using data to solve problems.

1.1 Operators

We will use vector and matrices in order to represent data (e.g. a pattern can be coded into a vector and a training set into a matrix). In order to use vector and matrices, we have to define some operators that state the available operations applicable to them.

Two main operation are defined:

- Vector sum: $\boldsymbol{u} \in \mathcal{V}, \boldsymbol{v} \in \mathcal{V} \Rightarrow \boldsymbol{u} + \boldsymbol{v} \in \mathcal{V}$
- Multiplication by a scalar: $\boldsymbol{u} \in \mathcal{V}, \boldsymbol{a} \in \mathcal{F} \Rightarrow a\boldsymbol{u} \in \mathcal{V}$

and, on real vectors ($\boldsymbol{u} \in \mathbb{R}^d$): $\quad \boldsymbol{u} = [u_1, u_2, ..., u_d] \quad \boldsymbol{v} = [v_1, v_2, ..., v_d]$

- Vector sum: $\boldsymbol{u} + \boldsymbol{v} = [u_1 + v_1, u_2 + v_2, ..., u_d + v_d]$
- Multiplication by a scalar: $a\boldsymbol{u} = [au_1, au_2, ..., au_d]$

Other operations are possible on real vectors:

- Scalar (inner, dot-) product between two vectors: outputs a scalar and is defined as:

$$\boldsymbol{u} \cdot \boldsymbol{v} = \sum_i u_i v_i$$

- (Euclidean) norm of a vector:

$$||\boldsymbol{u}|| = \sqrt{\sum_i u_i^2}$$

- (Euclidean) distance between two vectors:

$$d_E(\boldsymbol{u}, \boldsymbol{v}) = ||\boldsymbol{u} - \boldsymbol{v}|| = \sqrt{\sum_i (u_i - v_i)^2}$$

It is important to note the following 4 properties:

1. $||\boldsymbol{u}|| = \sqrt{\sum_i (u_i u_i)} = \sqrt{\boldsymbol{u} \cdot \boldsymbol{u}}$
2. $\boldsymbol{u} \cdot v = ||\boldsymbol{u}||||\boldsymbol{v}|| \cos\theta$ where θ is the angle between $\boldsymbol{u}$ and $\boldsymbol{v}$.
3. Therefore $\boldsymbol{u} \cdot v = 0$for orthogonal vectors ($\cos\theta = 0$).
4. If $||\boldsymbol{u}|| = 1$ and $||\boldsymbol{v}|| = 1$, then $||\boldsymbol{u} - \boldsymbol{v}|| = 2 - 2v \cdot u$

Moreover, we say that $\boldsymbol{u}$ is a **linear combination** of vectors $\boldsymbol{v}_i$ when

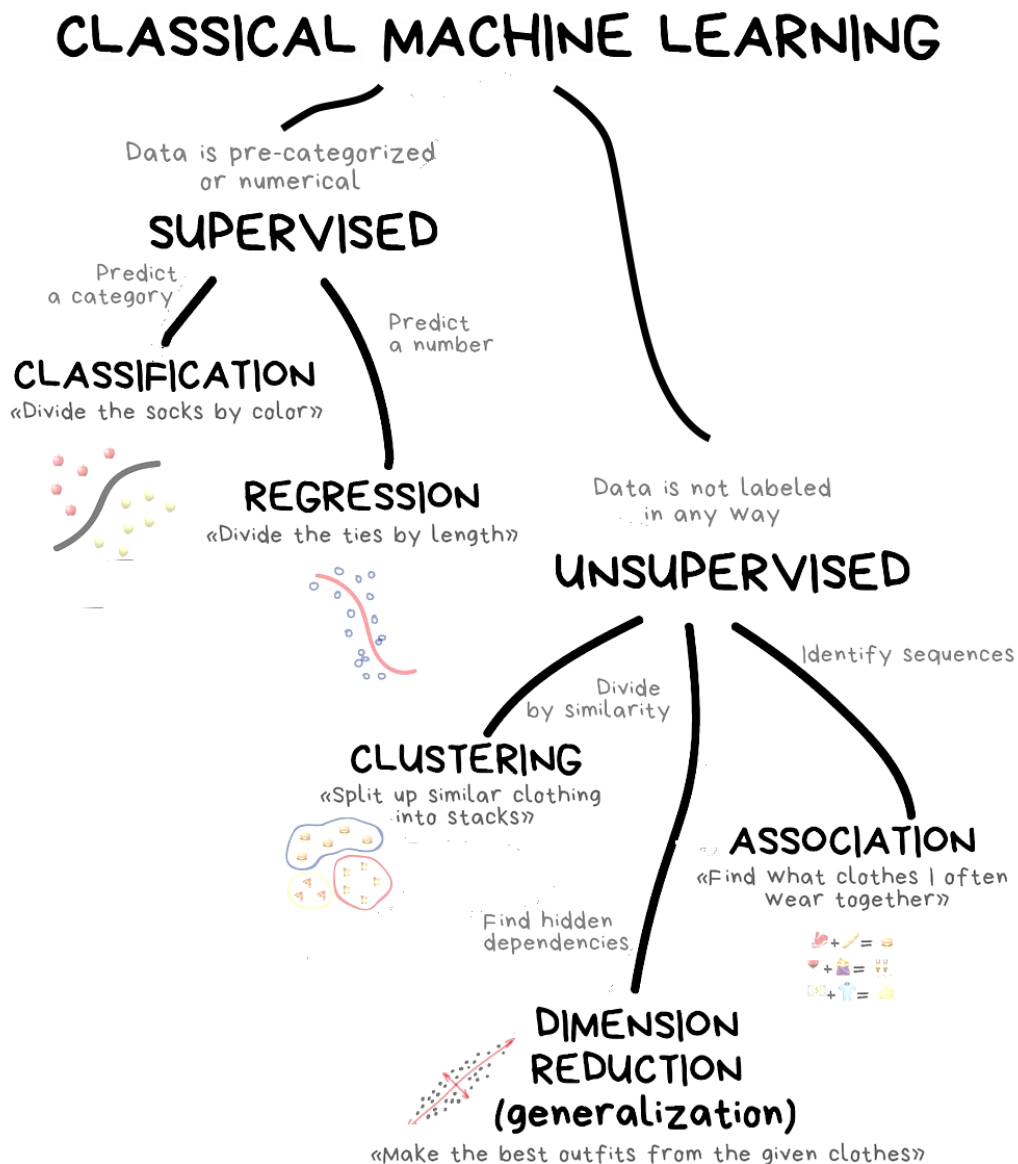
$$\boldsymbol{u} = \sum_i a_i \boldsymbol{v}_i$$

We also say that $\boldsymbol{u}$ is a **convex combination** of vectors $\boldsymbol{v}_i$ when

1. $\boldsymbol{u} = \sum_i a_i \boldsymbol{v}_i$ (a linear combination)
2. $\sum_i a_i = 1$ and $a_i > 0, \ \ \forall i$

1.2 Machine learning problems

The following scheme includes the main categories of classical machine learning problems:



1.2.1 Supervised learning

In supervised learning, an algorithm is employed to learn the mapping function from the input variable ($\boldsymbol{x}$) to the output variable (y); that is $y = f(\boldsymbol{x})$. The objective of such a problem is to **approximate the mapping function** (f) as accurately as possible such that whenever there is a new input data ($\boldsymbol{x}$), the output variable (y) for the dataset can be predicted.

The main difference between classification and linear regression is that the output variable in **regression** is numerical (or continuous) while that for **classification** is categorical (or discrete).

○ Regression

In machine learning, regression algorithms attempt to estimate the mapping function (f) from the input variables ($\boldsymbol{x}$) to numerical or **continuous** output variables (y). In this case, y is a **real value**, which can be an integer or a floating point value. Therefore, regression prediction problems are usually quantities or sizes.[1]

Examples of the common regression algorithms include **linear regression**, Support Vector Regression (**SVR**), and **regression trees**.

Some algorithms, such as logistic regression, have the name “regression” in their names but they are not regression algorithms.

Here is an example of a linear regression problem in Python:

```
import numpy as np
import pandas as pd

# importing the model
from sklearn.linear_model import LinearRegression
from sklearn.cross_validation import train_test_split

# importing the module for calculating the performance metrics of the model
from sklearn import metrics
data_path = ''http://www-bcf.usc.edu/~gareth/ISL/Advertising.csv''
# loading the advertising dataset
data = pd.read_csv(data_path, index_col=0)
array_items = ['TV', 'radio', 'newspaper']
#creating an array list of the items
X = data[array_items]
#choosing a subset of the dataset
y = data.sales
#sales

# dividing X and y into training and testing units
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
linearreg = LinearRegression()
#applying the linear regression model
linearreg.fit(X_train, y_train)
#fitting the model to the training data
y_predict = linearreg.predict(X_test)
#making predictions based on the testing unit
print(np.sqrt(metrics.mean_squared_error(y_test, y_predict)))
#calculating the RMSE number

#output gives the RMSE number as 1.4046514230328955
```

○ Classification

On the other hand, classification algorithms attempt to estimate the mapping function (f) from the input variables ($\boldsymbol{x}$) to **discrete or categorical** output variables ($\boldsymbol{y}$). In this case, $\boldsymbol{y}$ is a **category** that the mapping function predicts. If provided with a single or several input variables, a classification model will attempt to predict the value of a single or several conclusions.[2]

Examples of the common classification algorithms include **logistic regression**, **Naive Bayes**, **decision trees**, and **K Nearest Neighbors**.

[1] For example, when provided with a dataset about houses, and you are asked to predict their prices, that is a regression task because price will be a continuous output.

[2] For example, when provided with a dataset about houses, a classification algorithm can try to predict whether the prices for the houses “sell more or less than the recommended retail price.”

Here, the houses will be classified whether their prices fall into two discrete categories: above or below the said price.

Here is an example of a classification problem that differentiates between an orange and an apple:

```
from sklearn import tree

# Gathering training data

# features = [
#[155, ''rough''],
#[180, ''rough''],
#[135, ''smooth''],
#[110, ''smooth'']]
# (input to classifier)

features = [[155, 0], [180, 0], [135, 1], [110, 1]]
# scikit-learn requires real-valued features

# labels = [#''orange'',#''orange'',#''apple'',#''apple'']
# (output values)
labels = [1, 1, 0, 0]

# Training classifier
classifier = tree.DecisionTreeClassifier()
# using decision tree classifier
classifier = classifier.fit(features, labels)
# Find patterns in data

# Making predictions
print (classifier.predict([[120, 1]]))

# Output is 0 for apple
```

1.2.2 Unsupervised learning

How do you find the underlying structure of a dataset? How do you summarize it and group it most usefully? How do you effectively represent data in a compressed format? These are the goals of unsupervised learning, which is called “unsupervised” because you start with **unlabeled data** (there?s no Yes/No).

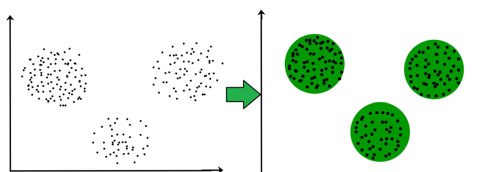
The two unsupervised learning tasks we will explore are **clustering** the data into groups by similarity and **reducing dimensionality** to compress the data while maintaining its structure and usefulness.

In contrast to supervised learning, it?s **not easy** to come up with **metrics** for how well an unsupervised learning algorithm is doing. “Performance” is often subjective and domain-specific.

◦ Clustering

Clustering is the task of dividing the population or data points into a number of groups such that data points in the same groups are more similar to other data points in the same group and dissimilar to the data points in other groups. It is basically a collection of objects on the basis of similarity and dissimilarity between them.

For example, the data points in the graph below clustered together can be classified into one single group. We can distinguish the clusters, and we can identify that there are 3 clusters in the below picture:



Of course, it is not necessary for clusters to be a spherical.

Algorithms developed to implement this technique are, for example, **K-mean Clustering** and **Hierarchical Clustering**.

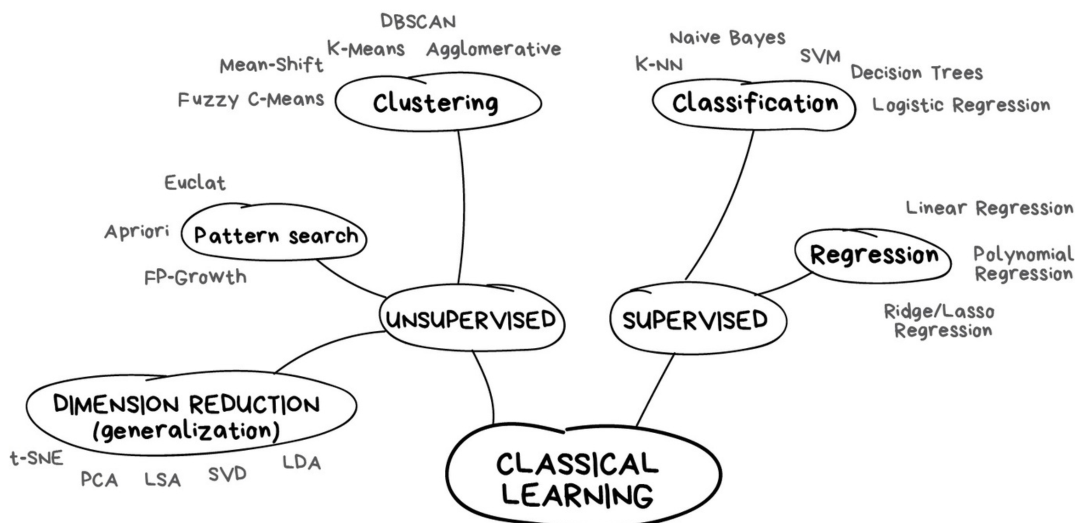
◦ Dimensionality reduction

Dimensionality reduction looks a lot like compression. This is about trying to reduce the complexity of the data while keeping as much of the relevant structure as possible.

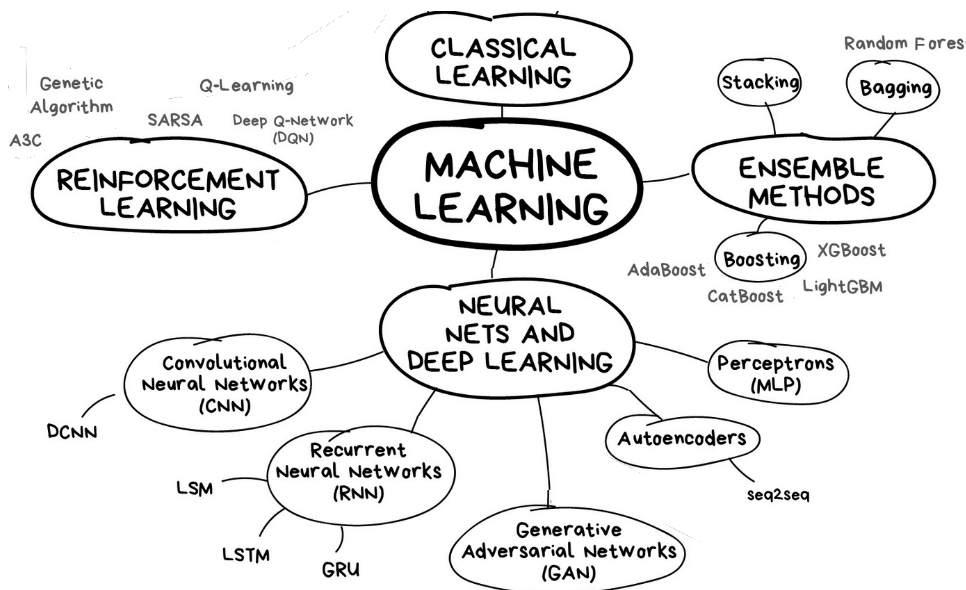
For example, if you take a simple 128 x 128 x 3 pixels image (length x width x RGB value), that?s 49,152 dimensions of data. If you?re able to reduce the dimensionality of the space in which these images live without destroying too much of the meaningful content in the images, then you?ve done a good job at dimensionality reduction.

Algorithms developed to implement this technique are, for example, Principal Component Analysis (**PCA**) and Singolar Value Decomposition (**SVD**).

1.2.3 ML techniques



Be careful though, classical learning is **just a subset** of the overall machine learning domain:



1.3 Perceptual problems

Perceptual tasks are problems related to perception. They have a typical structure, based on sets of individual measurements. It is generally difficult to write a program (an algorithm) to solve a perceptual task, so ML is commonly used in order to **learn from data**.

Which types of quantities we want to learn? Real values (one or more) or categorical values. Examples of **categorical information** are colours (red, green, blue, cyan, magenta, yellow, black ...), names (Socrates, Plato ...), truth values (True, False), in which there is no natural ordering, only quantitative information.

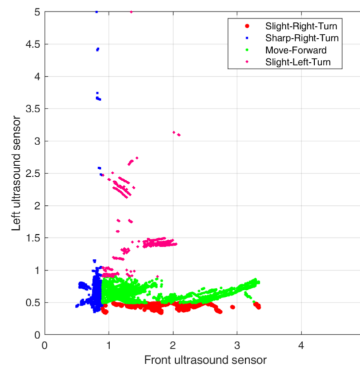
		Type of output	
		Quantitative	Nominal
Super-vised	Yes	Regression	Classification
	No	Low-dimensional mapping	Clustering

1.3.1 Perceptual problems examples

Autonomous robot Consider a wall-following robot that has to make decisions as to the direction to take, depending on a circular array of ultrasound sensors. The robot has 24 such sensors evenly spread over 360 degrees. The possible directions are:

Sharp-Left-Turn	Slight-Left-Turn	Move-forward	Slight-Right-Turn	Sharp-Right-Turn

The scitos G5 robot (metralabs.com/en/mobile-robot-scitos-g5/) is a multipurpose, modular platform for robotic research and development. he ultrasonic sensor?s output is available as a voltage in the range 0→5V. In the following graph we show a the minimum readings from two groups of sensors, on the forward and on the left. The colors are the solution of a perception problem, and correspond to the directions to take:



Iris recognition

- The Iris dataset has been in use since 1936
- Collected by botanist Edgar Anderson in 1935
- Used by statistician Sir Ronald A. Fisher in 1936

Sources:

- Edgar Anderson (1935).*The irises of the Gaspe Peninsula.* Bulletin of the American Iris Society 59: 25.
- Fisher, R.A. (1936). *The Use of Multiple Measurements in Taxonomic Problems.* Annals of Eugenics 7: 179-188.

It is possible to download it from the University of California - Irvine repository at:
http://archive.ics.uci.edu/ml/datasets/Iris

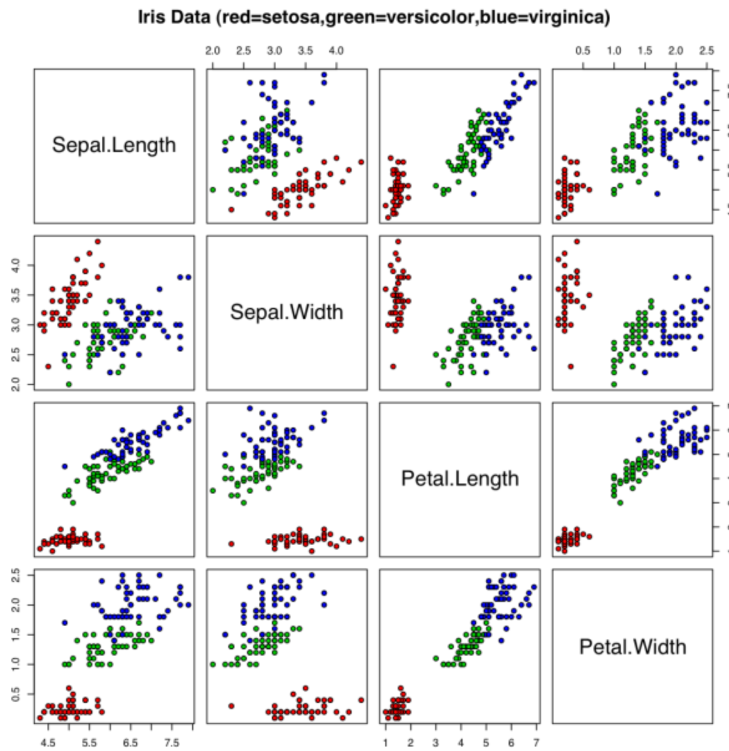
In this problem, the classes to identify are three:

Iris Setosa Iris Virginica Iris Versicolor

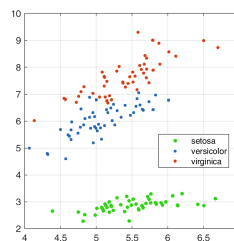
In the following Table it is possible to see the dataset:

Iris setosa				*Iris versicolor*				*Iris virginica*			
Sepal length	Sepal width	Petal length	Petal width	Sepal length	Sepal width	Petal length	Petal width	Sepal length	Sepal width	Petal length	Petal width
5·1	3·5	1·4	0·2	7·0	3·2	4·7	1·4	6·3	3·3	6·0	2·5
4·9	3·0	1·4	0·2	6·4	3·2	4·5	1·5	5·8	2·7	5·1	1·9
4·7	3·2	1·3	0·2	6·9	3·1	4·9	1·5	7·1	3·0	5·9	2·1
4·6	3·1	1·5	0·2	5·5	2·3	4·0	1·3	6·3	2·9	5·6	1·8
5·0	3·6	1·4	0·2	6·5	2·8	4·6	1·5	6·5	3·0	5·8	2·2
5·4	3·9	1·7	0·4	5·7	2·8	4·5	1·3	7·6	3·0	6·6	2·1
4·6	3·4	1·4	0·3	6·3	3·3	4·7	1·6	4·9	2·5	4·5	1·7

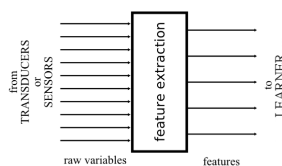
In the following Figure are shown the graphical representations of the dataset with the respective classification w.r.t. two of the 4 data types:



But if we multiply the length and height of petals and sepals in order to compute their area, we obtain a single graph instead:



Now we got a better graph, where classes are clearly separated: this is a **feature extraction**, a.k.a. apply a computation to raw data in order to enhance their division in classes.



ML does not need feature extraction, it works with raw data, but **data preparation could be very useful** anyway (e.g. data cleaning).

1.3.2 Data cleaning

- Change data types to make them suitable for your software (es. change strings into numerical codes)
- Remove data with out-of-range values, deal with **missing** data. For this we have several possible strategies, like:
  Removing observations (rows),
  Removing input variables (columns)
  Imputation of missing values
- Align timestamped data

1.4 Learning problems

Learning problems are divided in **representation** (learn to reproduce what is in your data) and **generalization** (learn to understand what your data represent). Solving the representation problem finds the best solution **for the training set**, while solving the generalization problem finds the best solution **for any data** from the same source that generated the training set.

- More clearly, **generalization** in this context is the ability of a learning machine to perform accurately on new, unseen examples/tasks: while classical optimization algorithms can minimize the loss on a training set, machine learning is concerned with minimizing the loss on unseen samples.
- On the other hand, **representation** more likely it's related feature detection, often attempting to preserve the information in their input but also transforming it in a way that makes it useful (for example as a pre-processing step before performing classification or predictions).

We define as "learning machine" (or "**learner**") not necessarily a real machine, but something that solves our problem, maybe a software program. We define as "task" the problem to be solved. We don't have a description of the problem, but data, so learning will be something like "**adjusting quantities inside the machine**" (e.g. algorithm parameters) in order to do a certain task.

We will then define the **hypothesis** as a specific learning machine that implements a certain task, e.g. a neural network that reads images and recognizes whether there is a known person (biometric recognition). The **hypothesis space** H is the set of all tasks that can be learned by a specific learning machine, e.g. the set of all classifiers that can be implemented by a specific neural network by setting its internal parameters. To make it more intuitive, we can say that the hypothesis space is a learning machine *before* learning and the hypothesis is a learning machine *after* learning a task.

1.4.1 Learning scenarios

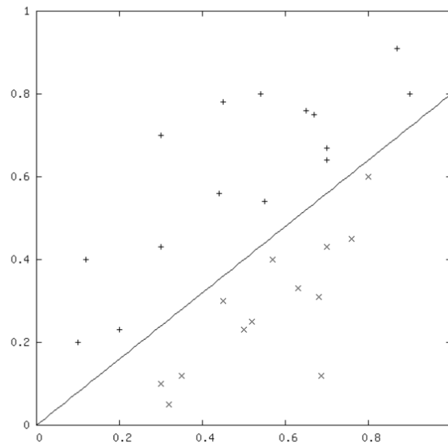
We have then to define three possible **scenarios**:

Sc.1 (useful mostly for reasoning):
  ◦ Learner = hypothesis space = H is fixed
  ◦ Data are fixed (population)
  ! Find correct learners in H (a representation task)
Sc.2 (not realizable):
  ◦ No data necessary: Probabilities are assumed to be known!
  ! Find best hypotesis space H and optimal learner
Sc.3 (your usual situation):
  ◦ Data will be stochastic but probabilities are not known
  ◦ Hypothesis space H fixed, chosen in advance
  ! Find learners in H which are correct for any possible realization of the data ($\rightarrow$ a generalization task)

Let's see now a basic example of classifier in **Scenario 1** (we know the whole population).

1.5 Linear threshold classifier

In a classification task, there exists a line (or a plane if $d = 3$, or a hyperplane if $d > 3$) such that object of a given class are all on the same side of the line/plane/hyperplane.



How we can write a linear equation like $y = wx$ in a general way? If we rewrite it as $-wx + y = 0$, we can define two vectors $\boldsymbol{x} = [x \quad y]$ and $\boldsymbol{w} = [-w \quad 1]$ and the general form will be:

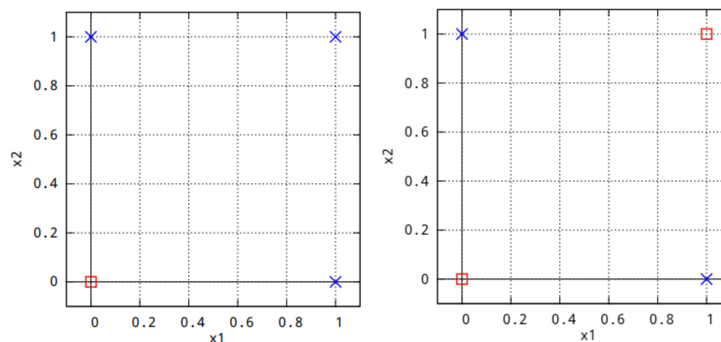
$$\boldsymbol{x} \cdot \boldsymbol{w} = 0$$

That, more generally, is a d-dimensional hyperplane (for $d = 1$ is a straight line). In linear classification we will use this **hyperplane as the learner**. This hyperplane will have the following properties:

- Homogeneous $\equiv$ passes through the origin
- The normal vector is $\boldsymbol{w}$
- Unit-length normal vector is $\hat{\boldsymbol{w}} = \dfrac{\boldsymbol{w}}{||\boldsymbol{w}||}$
- The positive side is represented by $\boldsymbol{x} \cdot \boldsymbol{w} > 0$
- A non-homogeneous hyperplane will have instead an equation like $\boldsymbol{x} \cdot w = \theta$

1.5.1 Separable data requirement

Let's consider the Logical NAND function (left) and the Logical XOR one (right). Let's encode it using two classes (0 and 1):



Consider the **Logical NAND** (left). We decide to solve using a separating hyperplane (a line in the 2-dim plane): on the positive side there will be one class (the 1, the red square) and on the negative side the other class (the 0, the blue crosses). From the (left) figure we can easily understand that infinite lines would solve the problem, and infinite parameter sets represent each equation (only looking at the sign!). Moreover, even the sign is arbitrarily assigned to the classes.

Consider this time the **Logical XOR** (right). In this case we don't have a solution, because it is not a linearly separable problem $\rightarrow$ **Linear separability assumption**

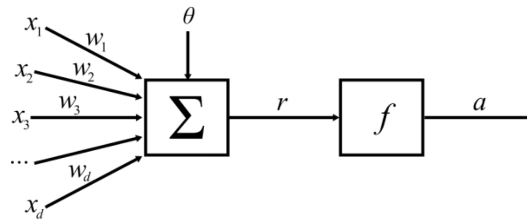
1.5.2 Learner model

Let's analyze a learner suited for linearly separable data:

$$r = \boldsymbol{x} \cdot \boldsymbol{w} - \theta \qquad a = f(r)$$

where:

- x is a d-dimensional vector of input values
- w is the corresponding (d-dimensional) vector of parameters
- $\cdot$ indicates scalar product
- r indicates the net, "integrated" input
- $f()$ is a nonlinear, monotonic activation function
- θ is a threshold
- a indicates the output.



We can then get rid of the threshold:

$$r' = r - \theta = \boldsymbol{x} \cdot \boldsymbol{w} \qquad a = f(r)$$

$$r - \theta = \sum_{i=1}^{d} w_i x_i - \theta = w_1 x_1 + w_2 x_2 + ... + w_d x_d - \theta$$

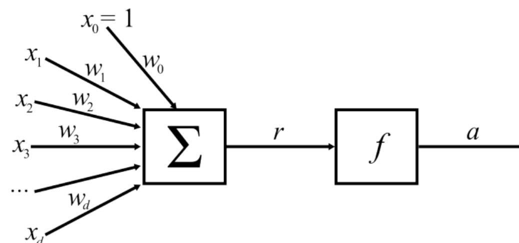
So, now let's call θ with another name:

$$\theta = -w_0 x_0 \ , \quad x_0 \equiv 1$$

This will lead to:

$$r - \theta = \sum_{i=1}^{d} w_i x_i + w_0 = \sum_{i=0}^{d} w_i x_i$$

So, if before θ was a threshold (subtracted), now the term w_0 is a bias, and it is summed as an offset value (r' is called r anyway):



Why we did this change? Because we like all parameters to be in one place!

Note: now the indexes for the components of x and w start at 0, not 1:

$$\boldsymbol{x} = [x_0, x_1, x_2, ..., x_d] \quad \boldsymbol{w} = [w_0, w_1, w_2, ..., w_d]$$

with $w_0 = -\theta =$ **bias** and $x_0 \equiv 1$.

1.5.3 Activation functions

For the presented classifier we can have different activation functions $f(r)$:

- **Heaviside step** (defined on $[-\infty, +\infty] \rightarrow \{0, +1\}$):

$$f(r) = \mathbf{1}(r) = \text{step}(r) = \Theta(r) = \begin{cases} +1 & r \geq 0 \\ 0 & r < 0 \end{cases}$$

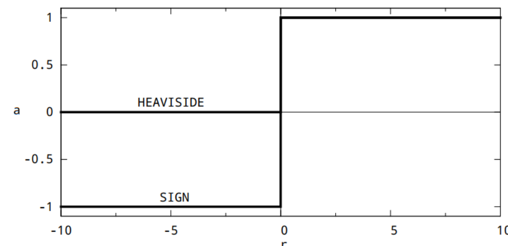
  (ties broken arbitrarily)

- **Signum function** (defined on $[-\infty, +\infty] \rightarrow \{-1, +1\}$):

$$f(r) = \text{sign}(r) = \begin{cases} +1 & r \geq 0 \\ -1 & r < 0 \end{cases}$$

  The signum function is a symmetrization in the interval $[-1, +1]$ of the Heaviside step function:

$$\text{sign}(r) = 2 \ \text{step}(r) - 1$$



- **Sigmoid** or logistic **function** (defined on $[-\infty, +\infty] \rightarrow \{0, +1\}$):

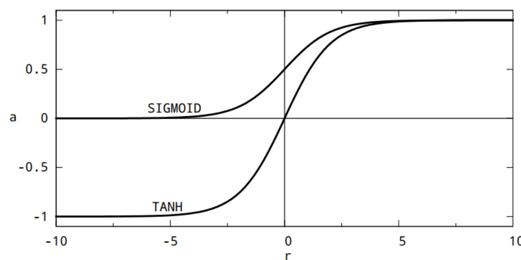
$$f(r) = \sigma(r) = \frac{1}{1 + e^{-r}}$$

- **Hyperbolic tangent** (defined on $[-\infty, +\infty] \rightarrow \{-1, +1\}$):

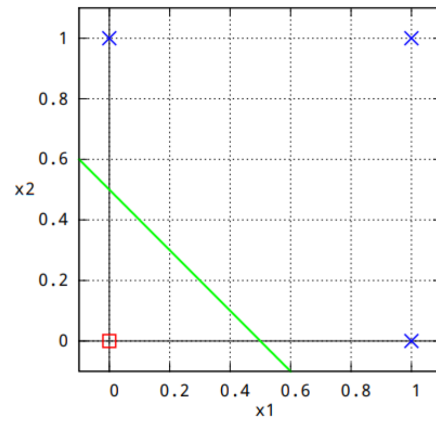
$$f(r) = \tanh(r) = \frac{1 - e^{-r}}{1 + e^{-r}}$$

  The hyperbolic tangent function function is a symmetrization in the interval $[-1, +1]$ of the sigmoid function:

$$\tanh(r) = 2 \ \sigma(2r) - 1$$



Let's make an example with the already seen Logical NAND function. In this case, as we can see from the following Figure, a good classifier is the one with $\boldsymbol{w} = [0.5, \ 1, \ 1]$ (with red square as positive value):


x2
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1
x1

Linear classifier demo

Lab “zero” for the Machine Learning course, EMARO

***Abstract*—Demo Matlab program with extensive comments**

I. MATLAB Implementation

Implementation of a simple, hand-designed linear classifier *Hand-designed* means that we don’t learn anything from the data, we just "invent" a reasonable classifier and see if it works.
First off, clear workspace (the memory of all variables):

```
clear
```

Create a "toy" training set with 10 random points within the square `[1,2]x[1,2]` (as class 1) and 10 points in `[0,1]x[0,1]` (as class 0)

```
x = 1 + rand(10,2);
x = [ x ; rand(10,2)]
```

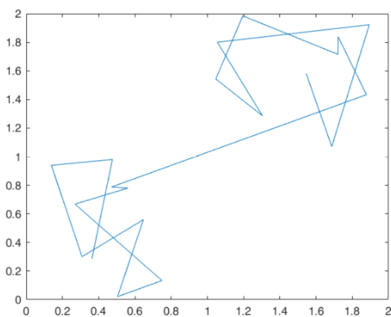
```
x = 20x2
1.54491.5824
1.68621.0707
1.89361.9227
1.05481.8004
1.30371.2859
1.04621.5437
1.19551.9848
1.72021.7157
1.72181.8390
1.87781.4333
```

Create corresponding vector of targets

```
t = [ones(10,1);zeros(10,1)];
```

Plot data

```
plot(x(:,1),x(:,2))
```



It is not very clear; plot again with suitable style (point markers rather than connecting lines) and different colors for different classes...
Plot class 1 in blue crosses:

```
plot(x(t==1,1),x(t==1,2),'xb');
```

(note indexing with logical expression)
Don’t replace figure, add next plots over old one:

```
hold on
```

Plot class 0 in red circles:

```
plot(x(t==0,1),x(t==0,2),'or');
```

Make sure that axes cover the full possible range;

```
axis([0 2 0 2])
```

Add grid:

```
grid on
```

Add figure title;

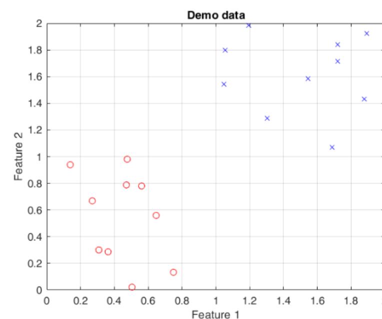
```
title('Demo data')
```

Add axis labels:

```
xlabel('Feature 1')
ylabel('Feature 2')
```

Adding plots to the figure is not necessary anymore, so:

```
hold off
```



Let’s try classifying with this hyperplane (actually, a line):

```
w = [1 1];
y = x*w' > 0;   % classification
```

Display the number of correctly classified points:

```
disp(sum(y==t))
```

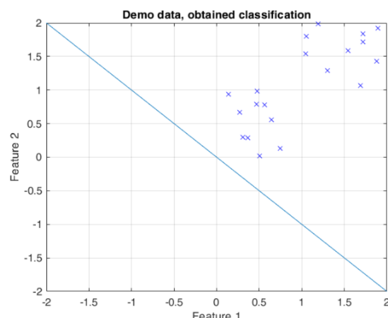
```
10
```

...not satisfactory. Let’s plot the data with the classification obtained rather than with the correct labels:

```
plot(x(y==1,1),x(y==1,2),'xb');
hold on
plot(x(y==0,1),x(y==0,2),'or');
axis([-2 2 -2 2])
grid on
title('Demo data, obtained classification...
    ')
xlabel('Feature 1')
ylabel('Feature 2')
```

Let's also plot the hyperplane:

```
xraster = -2:.1:2;
plot(xraster, -xraster*w(1)/w(2))
hold off
```



A bit of geometrical reasoning tells us that we need one more parameter in the hyperplane, otherwise it will always cross the origin and the figure shows that this is not ok. Let's add a column of all ones to the training set

```
xx = [ones(size(x,1),1) x];
```

The coefficient corresponding to the constant column will be the constant offset in the equation of the hyperplane

```
ww = [-2 1 1];
```

Let's classify with this hyperplane:

```
yy = linclass(xx,ww);
```

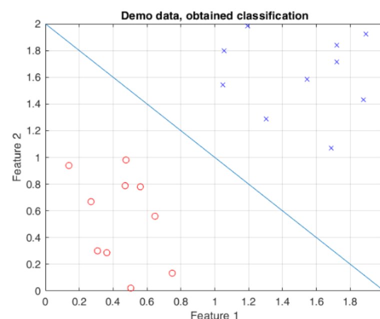
This time we use an external function to encapsulate the classifier, however simple it may be. See file `linclass.m` Display the result using a more sophisticated printing function:

```
fprintf('number of correct classifications...
    : %i out of %i total observations\n...
    ', ...
sum(yy==t), size(yy,1))
```

```
number of correct classifications: 20 out of
    20 total observations
```

Much better now. Let's see the new hyperplane in context:

```
plot(xx(yy==1,2),xx(yy==1,3),'xb');
hold on
plot(xx(yy==0,2),xx(yy==0,3),'or');
axis([0 2 0 2])
grid on
title('Demo data, obtained classification...
    ')
xlabel('Feature 1')ylabel('Feature 2')
xraster = 0:.1:2;
plot(xraster, -xraster.*ww(2)/ww(3)-ww(1))
hold off
```



Appendix

```
function y = linclass(x,w)
% linear threshold classifier
%   x: n x m data set to classify
%   w: 1 x m coefficient vector of ...
    hyperplane
y = x*w' > 0;
end

% this source code MUST be saved in a ...
    separate file named linclass.m

% typing 'help linclass' at the prompt ...
    will print the three lines of comments
% that follow the function header
```

Chapter 2

Bayesian classification

If with the example of the linear classifier seen in the previous Chapter we where in the Learning Scenario 1, we are now in **Scenario 2** (complete (probabilistic) knowledge). So, we will need a brief recall on probability theory.

2.1 Probability theory

Probability is an expression of uncertainty:

- **Frequentist probability**: Probability of an event as a generalization of the frequency of occurrence of that event in infinite repetition of an experiment (trial).
- **Subjective probability**: Probability of an event as a confidence in the fact that the event itself will occur, even in a single experiment.

Calling E an event from the set $\mathcal{E} = \{E_1, E_2, ..., E_N\}$ of all possible events (e.g. the outcomes of an experiment) we will indicate with $P(E)$ the probability of the event E. Then we will have the 3 **axioms of probability**:

1. $P(E) \geq 0$
2. $\sum_{i=1}^{N} P(E_i) = 1 \quad$ or $P(\mathcal{E}) = 1$
3. If E_i are mutually exclusive, then $P(E_i \cup E_j) = P(E_i) + P(E_j)$

Then other notable properties can be derived from the axioms:

1. $P(E) \leq 1$
2. If $E : i$ are not mutually exclusive, then $P(E_i \cup E_j) = P(E_i) + P(E_j) - P(E_i \cap E_j)$
3. $P(E_i \cap E_j) = P(E_i)P(E_j)$ for independent events.

Two events are **independent** if the outcome of one event does not influence the outcome of the second event. It is the **opposite of mutually exclusive**.

Example Let's consider a rolling die:

$$\mathcal{E} = \{1, 2, 3, 4, 5, 6\}$$

$$P(E_i) = P(E_j) \quad \forall i, j$$

$$\sum_i P(E_i) = 1 \Rightarrow P(E_i) = 1/6 \;\; \forall i$$

$$P(E < 3) = P(1 \cup 2) = P(1) + P(2) = 1/3$$

Example Let's consider two rolling dice:

$$\mathcal{E} = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

$$P(10) = P((4 \cup 6) \cup (5 \cup 5) \cup (6 \cup 4))$$

$$P(5 \cap 5) = P(5)P(5) = \frac{1}{6}\frac{1}{6} = \frac{1}{36}$$

Then the probability of the other two pairs (and of every other pair) is the same (if they are fair dice). So:

$$P(10) = P(4)P(6) + P(5)P(5) + P(6)P(4) = 3\,\frac{1}{36} = \frac{1}{12}$$

We can deduce this general law for dice:

$$P(\text{comb\&dice}) = \text{num.combinations} \cdot \left(\frac{1}{6}\right)^{\text{num.dice}}$$

2.1.1 Distribution and density

Let's now characterize better the probability:

- Regarding *continuous and discrete events*, let e indicate any event from a set $\mathcal{E}$. Given a specific event $\hat{e} \in \mathcal{E}$ (a specific "value" of e), we will call

  $$P_{\mathcal{E}}(\hat{e}) = P(e < \hat{e})$$

  the (cumulative) **distribution function** of events in $\mathcal{E}$.
- Regarding only *discrete events*, let e indicate any event from a numerable set $\mathcal{E}$ (e.g. an integer number in $\mathcal{E} = \{0, 1\}$. Given a specific event $\hat{e} \in \mathcal{E}$ (a specific "value" of e), we will call

  $$F_g(\hat{e}) = P(e < \hat{e})$$

  is the **probability mass function** of events in $\mathcal{E}$
- Regarding only *continuous events*, let e indicate any event from a numerable set $\mathcal{E}$ (e.g. an *real* number in $\mathcal{E} = \{0, 1\}$. A function f_g such that

  $$P_g(\hat{e}) = \int_{-\infty}^{\hat{e}} f_g(e)de$$

  is the **probability density function** of events in $\mathcal{E}$

The function $f_g(e)$ gives the **infinitesimal** ($\rightarrow$ "null") probability that a random event e has value $\hat{e}$. Its definite integral on a random interval $[\hat{e}_1, \hat{e}_2] \in \mathcal{E}$ is the **finite** probability that $\hat{e}_1 < e < \hat{e}_2$.

Intuitively, we can understand that discrete probabilities can be found by "counting", while continuous probabilities can be found by "measuring areas".

2.1.2 Conditional probability

We define the conditional probability as the probability of an event E **given** the knowledge that another event F has occurred:

$$P(E|F)$$

Let's take again the dice example. We had $P(10) = 1/12$. W.r.t. it, if we know that the first dice is 2, then

$$P(10|\text{first dice is 2}) = 0$$

while if we know that the first dice is 5, then

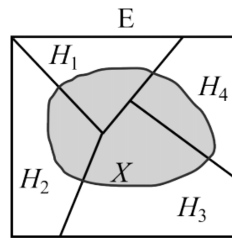
$$P(10|\text{first dice is 5}) = 1/6$$

2.1.3 Total probability theorem

Let's start with some jargon:

- H is a hypothesis
- $\{H_i\}$ is a set of alternative hypotheses ($H \in \{H_i\}$)
- X is an experimental observation
- $P(H)$ is the **a priori probability** of hypothesis H, so, the probability that H is true before seeing any experimental observation
- $P(H|X)$ is the **a posteriori probability** of hypothesis H after observing X
- $P(X|H)$ is the **likelihood** of observing X when H holds (H is verified, it's true, it's certain)
- $P(X)$ is the **marginal probability** of X, the probability of observing X in any case

This theorem states how to compute a marginal probability in a situation like the one illustrated here:



$$P(X) = \sum_i P(X|H_i)P(H_i)$$

2.1.4 Bayes theorem

This theorem gives the probability of a hypothesis after seeing an experimental observation:

$$P(H|X) = \frac{P(X|H)P(H)}{P(X)}$$

There is an alternative form of this theorem, that uses the total probability theorem and does not require $P(X)$:

$$P(H|X) = \frac{P(X|H)P(H)}{\sum_i P(X|H_i)P(H_i)}$$

$\sum_i P(X|H_i)P(H_i)$ is also known as a **partition function**.

An important use of probability functions is to compute the "most likely" or **expected value** of some random X. In the case of discrete events, it is a weighted sum (where ξ_i are the possible values of X):

$$\mathbb{E}\{X\} = \sum_i \xi_i F_X(\xi_i)$$

The symbol $\mathbb{E}\{\}$ is called **expectation operator**. For real-valued X we have instead:

$$\mathbb{E}\{X\} = \int_E \xi f_x(\xi) d\xi$$

2.2 Bayesian Decision Theory

2.2.1 Decision and loss

In a **decision problem** we will have

- c possible, mutually exclusive events, or "states of nature"

$$\{\omega_1, \omega_2, ..., \omega_c\}$$

- s possible actions or "decisions" that we may make

$$\{y_1, y_2, ..., y_s\}$$

- Input: One observation $\boldsymbol{x}$ (feature vector)
- Output: A decision y whose value is one of $\{y_1, y_2, ..., y_s\}$
- Reference (target): A true state of nature t whose value is one of $\{\omega_1, \omega_2, ..., \omega_c\}''$

There will be errors in the decision process. To measure the errors we have to define a **loss function**:

$$\lambda(y_j|t)$$

For problems with continuous output, λ it is an actual function, for example:

$$\lambda(y|t) = ||y - t||^2 \quad \leftarrow \text{Squared error loss function}$$

For problems with discrete and finite outputs (classification), it may be a loss matrix:

$$\Lambda = \left.\overbrace{\begin{bmatrix} \lambda_{11} & \lambda_{12} & ... & \lambda_{1c} \\ \lambda_{21} & \lambda_{22} & ... & \lambda_{2c} \\ & ... & & \\ \lambda_{s1} & \lambda_{s2} & ... & \lambda_{sc} \end{bmatrix}}^{\text{true state of nature } t}\right\} \text{decision } y$$

Bayes decision theory (or, 'classification according to the Reverend Bayes) deals with **prediction based on observed data**, for example, a doctor that diagnose a disease after visiting a patient: he records all observation and measurements into a patient record $\boldsymbol{x}$. Usually a doctor has some information available from his medicine textbooks and from his own experience, thanks to which he can determine:

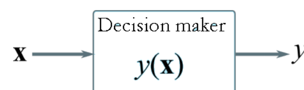
- The incidence of diseases $P(\omega_j)$
- The typical and not-so-typical signs and symptoms of diseases $P(\boldsymbol{x}|\omega_j)$

Then, from Bayes theorem we know that:

$$P(H|X) = \frac{P(X|H)P(H)}{\sum_i P(X|H_i)P(H_i)}$$

2.2.2 Quality evaluation of discrete decision set

So, how to evaluate the quality of a single decision? Suppose we have a decision maker, a rule that receives an observation $\boldsymbol{x}$ and emits a decision $y(\boldsymbol{x}) \in \{y_1, ..., y_c\}$. Given $\boldsymbol{x}$, the decision is fixed by the **decision rule** $y(x)$ (a mapping, function ...)



The cost of for the decision $y(x) = y_i$ will be defined considering a **conditional risk** (expected cost of a single decision, given $\boldsymbol{x}$):

$$R(y_i|\boldsymbol{x}) = \sum_{j=1}^{c} \lambda_{ij} P(\omega_j|\boldsymbol{x})$$

This is the expectation of the loss incurred in by deciding (taking action) y_i computed over all possible states of nature $\omega : j, \quad j = 1...c$.

Example Consider a Mars rover. The possible states of nature detected by his sensors will be:

$$\omega \in \{\omega_1, \omega_2, \omega_3\} = \{water, solidground, sand\}$$

We will then have their a priori probabilities:

$$P(\omega_1) = 0.2, \quad P(\omega_2) = 0.4, \quad P(\omega_3) = 0.4$$

The possible decisions that the rover can make are:

$$y \in \{y_1, y_2\} = \{rover_retract, rover_advance\}$$

We receive an input observation $\boldsymbol{x}$ from the sensors:

$$\boldsymbol{x} = \{groundTouchSens1, groundTouchSens2, groundOptSens, groundWhisker\}$$

We receive $\boldsymbol{x}$ with the correspondent likelihood values (note that **likelihoods may not sum up to 1** and that they are not mutually exclusive, so they are not "in direct competition with each other"):

$$P(\boldsymbol{x}|\omega_1) = 0.5, \quad P(\boldsymbol{x}|\omega_2) = 0.9, \quad P(\boldsymbol{x}|\omega_3) = 0.1$$

We will have then the following loss matrix:

$$\Lambda = \left.\overbrace{\begin{bmatrix} (\omega_1) & (\omega_2) & (\omega_2) & \\ \lambda_{11} & \lambda_{12} & \lambda_{13} & (y_1) \\ \lambda_{21} & \lambda_{22} & \lambda_{23} & (y_2) \end{bmatrix}}^{\text{(true state of nature } t)}\right\}(\text{decision } y) \quad = \begin{bmatrix} 0.1 & 1.0 & 4.0 \\ 2.0 & 0.1 & 0.1 \end{bmatrix}$$

So, the conditional risk of decision y_1 given the previous observation $\boldsymbol{x}$ is:

$$R(y_1|\boldsymbol{x}) = \sum_{j=1}^{3} \lambda : 1jP(\omega : j|\boldsymbol{x}) = 0.1 \cdot 0.5 + 1 \cdot 0.9 \cdot 4 \cdot 0.5 = 2.95$$

while the conditional risk of decision y_2 is:

$$R(y_2|\boldsymbol{x}) = \sum_{j=1}^{3} \lambda : 2jP(\omega : j|\boldsymbol{x}) = 2 \cdot 0.5 + 0.1 \cdot 0.9 + 0.1 \cdot 0.5 = 1.14$$

So, when we receive input $\boldsymbol{x}$, the decision that minimizes the conditional risk is y_2.

2.2.3 Quality evaluation of continuous decision set

When the decision is obtained from a deterministic, fixed function $y(\boldsymbol{x})$, when we have $\boldsymbol{x}$ we also know the decision $y(\boldsymbol{x})$. So we can compute the expected cost, where the expectation is taken over all possible inputs weighted by their probability density:

$$R = \int_{\mathcal{X}} R\left(y(\boldsymbol{x})|\boldsymbol{x}\right) p(\boldsymbol{x}) d\boldsymbol{x} \qquad \text{(Expected risk)}$$

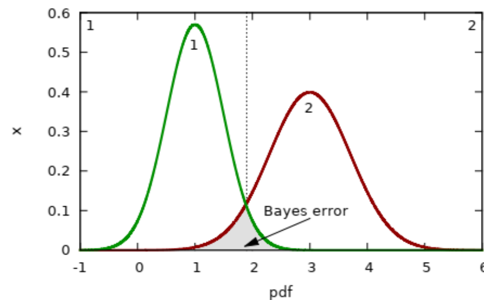
2.2.4 Bayes decision criterion

We can now state the so-called Bayes decision criterion (it's a theoretically **optimal** criterion, you can't do better than this!):

> To minimize R, given an input $\boldsymbol{x}$ the decision rule $y(\boldsymbol{x})$ should output the decision y that minimizes the conditional risk $R(y(\boldsymbol{x})|\boldsymbol{x})$ as often as possible.

A more precise (operational) version is the following one:

> To minimize R, given an input $\boldsymbol{x}$ the decision rule $y(\boldsymbol{x})$ should output the decision y that minimizes the expected risk R.



The Bayes error, the best possible error probability.

2.3 Bayesian classifiers

2.3.1 Minimum-error-rate classification

Classification is a decision problem with:

$$s \equiv c \qquad \{y_1, ..., y_s\} \equiv \{\omega_1, ..., \omega_c\}$$

I.e., there is no actual decision to take, we are only recognizing the state of nature (the **class**).

A common loss function for classification is the **zero-one loss**:

$$\lambda(y|t) = \begin{cases} 0 & \text{if } y = t \\ 1 & \text{if } y \neq t \end{cases}$$

For example, a zero-one loss matrix for a three class problem ($c = 3$) is:

$$\Lambda = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

What does the 0-1 loss mean? It means that all types of errors have the same cost ($= 1$) and that the correct classifications don?t have a cost. So, R is equal to the expected probability of error (proof: plug zeroes and ones in the definition of conditional cost). In facto, with the zero-one loss, we have the so-called **minimum-error-rate classification**.

2.3.2 Classifier model

But how to design a **classifier**? A classifier is a rule $y()$ that receives an observation $\boldsymbol{x}$ and outputs a class $\omega = y(\boldsymbol{x})$. The Bayes decision criterion states that y should minimize $R(y(x)|\boldsymbol{x})$. A natural idea would be:

- Build c blocks or "matched filters" g_i, $i : 1...c$ (the **discriminant functions**) that compute

$$g_1(\boldsymbol{x}) = -R(\omega_1|\boldsymbol{x}), ..., g_c(\boldsymbol{x}) = -R(\omega_c|\boldsymbol{x})$$

- Select ω_i that has maximum $g_i(\boldsymbol{x})$

The **decision region** is a subset of the data space with a given minimum-conditional-risk decision (i.e., the decision y is the same for all data in the region). Decision regions are separated by **decision boundaries** (or decision surfaces): the decision boundary between two regions (say $y = \omega_i$ and $y = \omega_j$) are defined by:

$$g_i(\boldsymbol{x}) = g_j(\boldsymbol{x})$$

In the case of the minimum-error-rate classifier (= using zero-one loss):

$$g_i(\boldsymbol{x}) = -\sum_{j=1, j\neq i}^{c} P(\omega_j|\boldsymbol{x}) = -\left(1 - P(\omega_i|\boldsymbol{x})\right) \qquad P(\omega_i|\boldsymbol{x}) \text{ from Total } P \text{ theorem}$$

Since a **classifier is defined by the decision boundaries**, we can note that the actual functions being compared need not be actually $g_i(\boldsymbol{x}) = R(\omega_i|\boldsymbol{x})$, they can be **any monotonic transformation** $g_i(\boldsymbol{x}) = f(R(\omega_i|\boldsymbol{x}))$ that **preserves decision boundaries**, for example:

$$g_i = \log R(\omega_i|\boldsymbol{x}) \qquad \text{or} \qquad g_i = \frac{1}{1 + e^{-R(\omega_i|\boldsymbol{x})}}$$

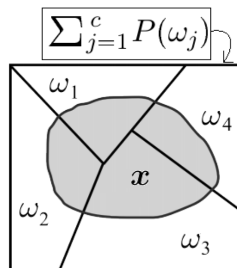
This gives us more freedom in building a classifier! We can use more general scores $f(R(\omega_i|\boldsymbol{x}))$ instead of actual conditional risks $R(\omega_i|\boldsymbol{x})$. So, which reasonable discriminant functions we can choose for the zero-one loss classifier? The following transformation

$$f(x) = x + 1$$

is monotonic and preserves decision boundaries, so we can avoid the useless "−1" term in the previously seen decision region formulation and have:

$$g_i(\boldsymbol{x}) = 1 - \sum_{j=1, j\neq i}^{c} P(\omega_j|\boldsymbol{x}) \rightarrow \boxed{g_i(\boldsymbol{x}) = P(\omega_i|\boldsymbol{x}) \qquad (P \text{ of } \omega_i \text{ given } \boldsymbol{x})}$$

Remember that, for $c = 4$, we have:



Here we see that if we give the same weight to all errors (0/1 loss) the discriminant functions are simply the probability of each class given the input – so **we take the one with maximum probability**!

2.3.3 Naive Bayes classifier

The classifier presented above is a quite sensible classifier due to its sensible criterion (taking the one with the maximum probability). A popular classifier is the so called **Naive Bayes classifier**, that is instead built using "wrong" discriminant functions based on "naive" (or even "idiot"):

- Recall that: $\boldsymbol{x} = [x_1, x_2, x_3, ..., x_d]$

- So (from Bayes' theorem):

$$P(\omega_i|\boldsymbol{x}) = P(\boldsymbol{x}|\omega_i)P(\omega_i) = P(x_1, x_2, x_3, ..., x_d|\omega_i) \cdot P(\omega_i)$$

- So, in general we have:

$$P(a, b|c) \neq P(a|c)\ P(b|c)$$

  but the **naive assumption** (wrong) is to consider them independent:

$$P(x_1, x_2, ..., x_d|\omega_i) \cdot P(\omega_i) = P(x_1|\omega_1)P(x_2|\omega_2) \dots P(x_d|\omega_d) \cdot P(\omega_i)$$

So, the result is that we have the following boundaries:

$$g_i(\boldsymbol{x}) = P(\omega_i)\left[P(x_1|\omega_i)P(x_2|\omega_i) \dots P(x_d|\omega_i)\right] = P(\omega_i) \prod_{j=1}^{d} P(x_j|\omega_i)$$

But if it's an "idiotic" classifier, why we use it? Because of the way how it "learns", that makes it particularly **handy when features are binary** (true/false):

> To "learn" $P(x_k|\omega i)$ it counts how often each value of x_k occurs in class !i in the training set.

So we will have:

$$g_i(\boldsymbol{x}) = P(\omega_i) \prod_{j=1}^{d} P(x_j|\omega_i)$$

$$P(x_k = \text{true}|\omega_i) = \frac{\text{number of times } x_k = \text{true in class } \omega_i}{\text{number of observations of class } \omega_i} = \frac{N_{\text{true},\omega_i}}{N_{\omega_i}}$$

$$P(x_k = \text{false}|\omega_i) = \frac{\text{number of times } x_k = \text{false in class } \omega_i}{\text{number of observations of class } \omega_i} = \frac{N_{\text{false},\omega_i}}{N_{\omega_i}}$$

Prior probability of classes:

$$P(\omega_i) = \frac{\text{number of observations in class } \omega_i}{\text{number of observations in the training set}} = \frac{N_{\omega_i}}{N}$$

Naive Bayes Algorithm Implementation and Performance Evaluation

Report for the Machine Learning course, EMARO

Davide Lanza
EMARO+ M2
Genoa, Italy
davidel96@hotmail.it

***Abstract*—In this report we will analyze a MATLAB implementation of the Naive Bayes Classification technique. After a short introduction, we will define the theoretical framework in which we will do our analysis. Considering these assumptions, we will introduce the Naive Bayes Classifier NBC and we will discuss our implementation, focusing on the Laplace smoothing variant. We will estimate then the accuracy of such classifiers w.r.t. a very small use-case dataset, and a classic, well-known dataset. Analyzing ad comparing the different results obtained we will show that Laplace smoothing does not enhance the accuracy – but it lowers it – when the amount of data is big and uniformly distributed, while enhance a lot the generalization power of the learner – reducing the overfitting relative to the train set – for small size datasets.**

***Index Terms*—Machine Learning, Naive Bayes Classification, Linear Classification, Laplace Smoothing, MATLAB**

I. Introduction

Thanks to the exponential increase of data available on the Internet, the volume of information available for perform machine learning task grows incredibly. Classification tasks – as the assignment of some input to one or more predefined categories based on the features it bears – are an important component in information management tasks.

In this paper, we focus on a specific classification algorithm suitable for probabilistic environments. First, we will discuss the principles of Bayes Theorem and the Naive Bayes algorithms. Then we will introduce its implementation on MATLAB. Finally, the testing results are analyzed, in order to evaluate the influence of the training-test dataset size ratio and the Laplacian smoothing factor.

The datasets used for the testing here are "The golf database" [1, p.521 Fig.1a] and the Jeff Schlimmer's Mushroom dataset[1] that includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the *Agaricus* and *Lepiota* families [2, pp. 500-525].

II. State of the Art

In this section, we will introduce the theoretical framework from which we will implement our Naive Bayes Classifier (NBC). Starting from the specific classification that we will perform (minimum-error-rate classification), we will introduce the Bayesian classifiers and the so-called "naive assumption" for the NBC.

A. Minimum-Error-Rate Classification

In a classification problem we will have c classes ω_i that will represent possible, mutually exclusive events or "state of nature", an observation that is a vector of f features $\boldsymbol{x} = [x_1, ..., x_f]$, and a decision y in output whose value is in the $\{\omega_i | 1 \leq i \leq c\}$ domain.[2]

To estimate the errors in the process we need to define a loss function $\lambda(y|t)$ that will evaluate the "distance" between the prediction $y \in \{\omega_i\}$ and the actual true value $t \in \{\omega_i\}$. Since in classification we will deal with discrete and finite outputs, we will use a loss matrix:

$$\begin{array}{c} [y/t] \\ (\omega_1) \\ (\omega_2) \\ \\ (\omega_c) \end{array} \begin{array}{c} \begin{array}{cccc} (\omega_1) & (\omega_2) & ... & (\omega_c) \end{array} \\ \begin{pmatrix} \lambda_{11} & \lambda_{12} & ... & \lambda_{1c} \\ \lambda_{21} & \lambda_{22} & ... & \lambda_{2c} \\ \vdots & \vdots & \ddots & \vdots \\ \lambda_{c1} & \lambda_{c2} & ... & \lambda_{cc} \end{pmatrix} \end{array} = \Lambda \quad (1)$$

In this report we will adopt a zero-one loss function $\lambda(y = \omega_i | t = \omega_j) = \delta_{ij}^{-1}$, [3] a common choice for classification purposes. Such a function assign to all kinds of errors the same cost, with null cost for exact prediction. Then, the zero-one loss matrix is:

$$\Lambda = \begin{pmatrix} 0 & 1 & ... & 1 \\ 1 & 0 & ... & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & ... & 0 \end{pmatrix} \quad (2)$$

To evaluate the quality of the classification y given an observation $\boldsymbol{x}$ we have to define a risk function $R(y = \omega_i | \boldsymbol{x})$ that is the expectation of the loss incurred in by classifying $y = \omega_i$ computed over all possible states of nature $\{\omega_j | j \leq i \leq c\}$:

$$R(y = \omega_i | \boldsymbol{x}) = \sum_{j=1}^{c} \lambda_{ij} P(\omega_j | \boldsymbol{x}) \quad (3)$$

[1] Available at https://archive.ics.uci.edu/ml/datasets/mushroom.

[2] In fact, a classification problem is a decision problem for which the decisions domain $\{y_i\} = \{\omega_i\}$.

[3] The δ_{ij}^{-1} is the inverse of the Kroneker delta δ_{ij}.

Since we use a zero-one loss function we are in the so-called minimum-error-rate classification, that means, the risk $R(y = \omega_i|\boldsymbol{x})$ is equal to the expected probability of error. Since we deal with expected probabilities, we need probability theory in order to design our classifier.

B. Bayesian Classifier

Using the Total Probability theorem it is possible to rewrite the risk function as follows:

$$R(y = \omega_i|\boldsymbol{x}) = \sum_{j=1, j\neq i}^{c} \lambda_{ij} P(\omega_j|\boldsymbol{x}) = P(\omega_i|\boldsymbol{x}) - 1 \quad (4)$$

The Bayes decision criterion [3] states that the prediction y should minimize the risk function. We can then implement our classifier with c discriminant functions g_i chosen as follows[4]:

$$g_i(\boldsymbol{x}) = R(y = \omega_i|\boldsymbol{x}) + 1 = P(\omega_i|\boldsymbol{x}) \quad (5)$$

The classifier will then select ω_i that has maximum $g_i(\boldsymbol{x})$, so, thanks t the zero-one loss,it will take the one with maximum probability.

C. The Naive Assumption

From Bayes' theorem we that

$$\begin{aligned} g_i(\boldsymbol{x}) &= P(\omega_i|\boldsymbol{x}) = P(\boldsymbol{x}|\omega_i)P(\omega_i) = \\ &= P(\omega_i)P(x_1, x_2, x_3, ..., x_d|\omega_i) \end{aligned} \quad (6)$$

In general, we have $P(a, b|c) \neq P(a|c)P(b|c)$, but the so-called "naive assumption" consist in consider them independent:

$$g_i(\boldsymbol{x}) = P(\omega_i) \prod_{j=1}^{d} P(x_j|\omega_i) \quad (7)$$

Given a dataset in the form $[\boldsymbol{x} \Rightarrow t]$, it is possible to train such a classifier according to the following rules:

$$P(\omega_i) = \frac{\text{count}_{\text{dataset}}(t = \omega_i)}{\text{size(dataset)}} \quad (8)$$

$$P(x_j = \alpha_k|\omega_i) = \frac{\text{count}_{\text{dataset}}(x_j = \alpha_k \wedge t = \omega_i)}{\text{count}_{\text{dataset}}(t = \omega_i)} \quad (9)$$

where $\{\alpha_k\}$ are the values that the observation feature x_j can assume.

III. Implementation

A. Canonical NBC

We have a dataset composed by a set of o observations $X = \{\boldsymbol{x}_i | 1 \leq i \leq o\}$ and by the respective truth values $\boldsymbol{t} = \{t_i | 1 \leq i \leq o\}$ such that $\Delta = [X, \boldsymbol{t}]$:

$$\Delta = \boxed{\begin{matrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_f^{(1)} & t_1 \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_f^{(2)} & t_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_1^{(o)} & x_2^{(o)} & \cdots & x_f^{(o)} & t_o \end{matrix}} \quad (10)$$

[4] Since a classifier is defined by the decision boundaries $g_i(\boldsymbol{x}) = g_j(\boldsymbol{x})$, the actual functions can be any monotonic transformation that preserves decision boundaries.

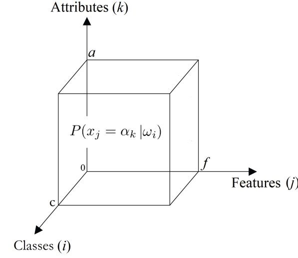


Figure 1: Graphical representation of P_A tensor.

Since every feature will assume a limited number of values, we defined an alphabet set $A = \{\alpha_i | 0 \leq i \leq a\}$ of a attributes that will code each feature value.[5] Then, defining $\Omega = \{\omega_i | 1 \leq i \leq c\}$ we will have:

$$\boxed{x_i \in A\,, \quad t_i \in \Omega} \quad (11)$$

Using a part test-train dataset ratio of $1 : 4$,[6] the classifier is trained with $\Delta_{train} = [X_{train}, t_{train}] \subset \Delta$. The NBC knowledge is represented by a vector P_Ω and by a three-dimensional tensor P_A:

$$P_\Omega = \begin{bmatrix} P(\omega_1) & P(\omega_2) & \cdots & P(\omega_c) \end{bmatrix}^T \quad (12)$$

$$P_{A_{ij}} = \begin{bmatrix} P(x_j = \alpha_1|\omega_i) & \cdots & P(x_j = \alpha_a|\omega_i) \end{bmatrix}^T \quad (13)$$

$$P_A = \begin{bmatrix} P_{A_{11}} & \dots & P_{A_{1f}} \\ P_{A_{21}} & \dots & P_{A_{2f}} \\ \vdots & \ddots & \vdots \\ P_{A_{c1}} & \dots & P_{A_{cf}} \end{bmatrix} \quad (14)$$

The training process that computes the values of the two matrices is described in Algorithm 1. After the training, using the knowledge NBC= $[P_\Omega, P_A]$ we are able to perform predictive classification on the dataset. Receiving in input an observation $\boldsymbol{x}$, the NBC will decode it by means of the alphabet A (from attributes α_k to the indexes k) and it will compute the discriminant functions in order to extract a prediction y (Algorithm 3).

B. NBC with Laplace Smoothing

The training Algorithm 1 is not perfect, and it has to be smoothed, because it initialize to 0 the probabilities in P_A for features unseen in Δ_{train}[7]. Not only, also the prediction Algorithm 3 presents this problem related to unseen features. For sake of simplicity, we will analyze a simple case. Let's say that after the training, the NBC would be able to predict two classes ω_1, ω_2 with $50 : 50$ probabilities. Now, consider

[5] This leads to have $a = \max_{\text{features}}(\text{different feature values})$ and then to have some null values in the probability tensors, but it simplifies a lot the implementation.

[6] The single instances for each fold are randomly selected by means of MATLAB function `cvpartition`.

[7] That is a consequence of choosing a single coding alphabet A for all the features with size a.

an observation $\boldsymbol{x} = [x_1, ..., x_f]$ which NBC rates very highly as ω_1, let's say $P(\boldsymbol{x}|\omega_1) = 0.99$ and $P(\boldsymbol{x}|\omega_2) = 0.01$. If we consider another observation $\bar{\boldsymbol{x}} = [x_1, ..., \bar{x}, ..., x_f]$ that is exactly the same as $\boldsymbol{x}$ except for the feature $\bar{x}$ expressing an unseen attribute, we will have $N_{\bar{x},\omega_i} = 0$ since there were no examples of $\bar{x}$ value in Δ_{train}. Then, suddenly we would have:

$$P(\bar{x}|\omega_i) = N_{\bar{x},\omega_i}/N_{\omega_i} = 0 \tag{15}$$

$$P(\bar{\boldsymbol{x}}|\omega_1) = P(\boldsymbol{x}|\omega_1)P(\bar{x}|\omega_1) = 0.99 \cdot 0 = 0 \tag{16}$$

$$P(\bar{\boldsymbol{x}}|\omega_2) = P(\boldsymbol{x}|\omega_2)P(\bar{x}|\omega_2) = 0.01 \cdot 0 = 0 \tag{17}$$

Despite $\boldsymbol{x}$ being strongly classified as ω_1, $\bar{\boldsymbol{x}}$ may be classified differently because $\bar{x}$ feature value provides a corresponding zero probability. A first solution to this problem would be to exclude the zero-valued probabilities from the prediction process, that is, basically, to assign a 1 to their probability. This will result in enhancing the weight of the *a priori* values P_Ω in the product seen in Equation 7 (see Algorithm 4).

This *a priori* solution is not the most general one, and could overweight the *a priori* probabilities ignoring the data (if a certain feature never assumed a specific attribute value along the entire dataset probably its probability will be almost 0, not 1). For this reason, implement Laplace smoothing allows us to give to unseen features a small non-zero probability for both classes, so that the posterior probabilities don't suddenly drop to zero, without over-weighting the *a priori* probabilities. To implement it in Algorithm 1 we should modified it inserting a smoothing factor ℓ (see Algorithm 2). During the training then has to be computed also a default probability vector that will be used when the feature values will be some attributes not included in the alphabet A:

$$P_d = \begin{bmatrix} \ell/(N_{\omega_1} + c\,\ell) \\ \vdots \\ \ell/(N_{\omega_c} + c\,\ell) \end{bmatrix} \tag{18}$$

This default value will be useful during the prediction phase as shown in Algorithm 5. If we want to interpret qualitatively the action of the Laplace smoothing, we can notice that, on one hand, a value $\ell > 1$ means lower probabilities P_d and in the P_A tensor, so this will result in trusting more the *a priori* belief rather than the training data. On the other hand, a value $\ell < 1$ means higher probabilities P_d and in the P_A tensor, so this will result in trusting more the training data rather than the *a priori* belief.

IV. Performance Evaluation

The datasets used for the testing here are "The golf database" [1, p.521 Fig.1a] and the Jeff Schlimmer's Mushroom dataset that includes descriptions of hypothetical samples corresponding to 23 species of gilled mushrooms in the *Agaricus* and *Lepiota* families [2, pp. 500-525]. The test were k-fold cross validation, with a $k = 3$ value (due to the small amount of samples of the first dataset). The dataset tested was $\Delta_{test} = \Delta - \Delta_{train}$, but every time MATLAB's `cvpartiton` function composed the test and training sets randomly.

A. The Golf Dataset

Outlook	Temperature	Humidity	Windy	Play
overcast	hot	high	FALSE	yes
overcast	cool	normal	TRUE	yes
overcast	mild	high	TRUE	yes
overcast	hot	normal	FALSE	yes
rainy	mild	high	FALSE	yes
rainy	cool	normal	FALSE	yes
rainy	cool	normal	TRUE	no
rainy	mild	normal	FALSE	yes
rainy	mild	high	TRUE	no
sunny	hot	high	FALSE	no
sunny	hot	high	TRUE	no
sunny	mild	high	FALSE	no
sunny	cool	normal	FALSE	yes
sunny	mild	normal	TRUE	yes

Table 1: The golf dataset [1]

Since this dataset contains only 14 samples, in order to achieve generalized results it has been trained and tested for 10000 iterations (each time the Δ sets were randomly assembled). This high number of samples allowed to have an average accuracy with 10^{-2} precision. A confusion matrix has been plotted in order to summarize and visualize the result. The one in Figure 2 is the one corresponding to the NBC without Laplace smoothing, while the one in Figure 3 is the one corresponding to the NBC with a $l = 1$ Laplace smoothing. The full results are available in Table 2, where the accuracy values w.r.t. Δ_{train} and Δ_{test} as test sets (of a NBC trained on Δ_{train}) is the ratio between the positive results and the total number of experiments (iterations·size(dataset)).

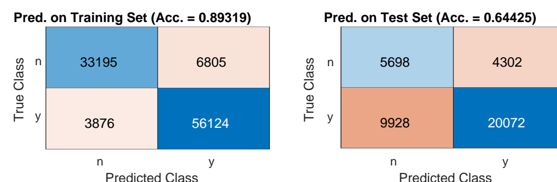


Figure 2: Golf dataset: Confusion matrix for the canonical NBC.

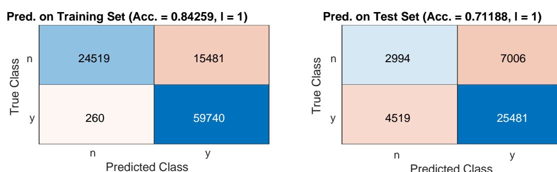


Figure 3: Golf dataset: Confusion matrix for the NBC with Laplace smoothing ($l = 1$).

B. The Mushroom Dataset

Since this dataset contains 8124 samples, it is possible to achieve generalized results with a single iteration. A confusion matrix has been plotted in order to summarize and visualize

l	Train accuracy	Test accuracy
♯	0.8940	0.6458
0	0.8922	0.6461
0.2	0.8903	0.6832
0.4	0.8745	0.6899
0.6	0.8593	0.7064
0.8	0.8466	0.7089
1	0.8438	0.7189
1.2	0.8320	0.7150
1.4	0.8243	0.7266
1.6	0.8111	0.7275
2	0.7959	0.7389
3	0.7747	0.7680
5	0.7268	0.7684
7	0.6845	0.7578
10	0.6362	0.7507
100	0.6000	0.7500
1000	0.6000	0.7500

Table 2: Golf dataset: Accuracy for testing Δ_{train} and Δ_{test} w.r.t. Laplace smoothing (the first line refers to the canonical NBC)

the result. The one in Figure 4 is the one corresponding to the NBC without Laplace smoothing, while the one in Figure 5 is the one corresponding to the NBC with a $l = 1$ Laplace smoothing. The full results has been computed this time with 100 iterations, and are available in Table 3.

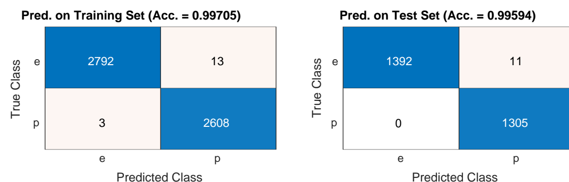


Figure 4: Mushroom dataset: Confusion matrix for the canonical NBC.

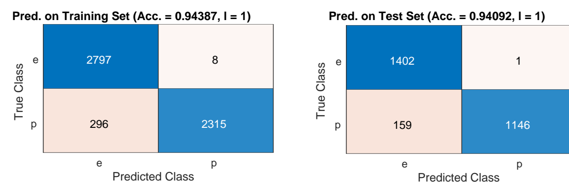


Figure 5: Mushroom dataset: Confusion matrix for the NBC with Laplace smoothing ($l = 1$).

V. Conclusions

From the analysis performed on the designed system, we confirm that NBC is a sound learner for a discrete, consistent and big datasets as the Mushroom one. For these kinds of datasets, Laplace smoothing does not enhance the accuracy, but it lowers it (Table 3). It has to be noted though that in the Mushroom dataset used, all the attributes of the received observations were probably already in the alphabet A. For a small dataset, instead, Laplace smoothing enhanced a lot the generalization power of the learner, reducing the overfitting relative to the train set. From Table 2 we can notice how around $l = 5$ the accuracies w.r.t. the test and the training sets are almost equal. That is a good achievement regarding these kind of problems, where huge amounts of data are unavailable.

l	Train accuracy	Test accuracy
♯	0.9971	0.9968
0	0.9971	0.9969
0.2	0.9658	0.9650
0.5	0.9526	0.9517
1	0.9447	0.9443
2	0.9392	0.9386

Table 3: Mushroom dataset: Accuracy for testing Δ_{train} and Δ_{test} w.r.t. Laplace smoothing (the first line refers to the canonical NBC)

VI. Future Work

Regarding the Mushroom case-study analysis, a better way to estimate the performances of Laplace smoothing would be using a Δ_{test} with a significant number of attributes not included in A, because here the data in Δ were quite uniformly distributed. Regarding the Golf case-study analysis, $\text{size}(\Delta) = 14$ is definitely too little in order to draw a conclusion w.r.t. small size datasets. Further analysis should test bigger datasets, comparing different $\text{size}(\Delta) = 10^i, 0 \leq i \leq 4$. Regarding the general architecture implemented here and the algorithms provided, loop unrolling would decrease significantly estimation time. Using dynamic-size matrix instead of a tensor whose third dimension size is fixed to a is not the best saving-space solution.

References

[1] L. C. Rivero, J. H. Doorn, and V. E. Ferraggine, *Handbook of Research on Innovations in Database Technologies and Applications: Current and Future Trends*. Hershey, PA, USA: IGI Global, 2009, ISBN: 978-1-605-66243-5.

[2] G. Lincoff, *National Audubon Society Field Guide to North American Mushroom*. NewYork, NY, USA: Alfred A. Knopf, 1981, ISBN: 978-0-394-51992-0.

[3] R. O. Duda, E. H. Hart, and D. G. Stork, *Pattern Classification*, 2nd. New York, NY, USA: John Wiley & Sons, 2001, ISBN: 978-0-471-05669-0.

Appendix

Algorithm 1 BNC Training (Canonical)

Require: $\Delta_{train}, A, \Omega$
  P_Ω, P_A initialize to 0
  $N = \text{count_rows}(\Delta_{train})$
  for $i = 1 \to c$ **do**
    $\text{mask}_i = \{[\boldsymbol{x}^{(m)}\ t_m] \in \Delta_{train} \mid t_m = \omega_i\}$
    $N_{\omega_i} = \text{count_rows}(\text{mask}_i)$
    $P_\Omega(i) = N_{\omega_i}/N$
    for $j = 1 \to f$ **do**
      for $k = 1 \to a$ **do**
        $\text{mask}_k = \{[\boldsymbol{x}^{(m)}\ t_m] \in \text{mask}_i \mid x_j^{(m)} = \alpha_k\}$

N_{x_k,ω_i} = count_rows(mask$_k$)
$P_A(i,j,k) = N_{x_k,\omega_i}/N_{\omega_i}$
end for
end for
end for
return P_Ω, P_A

Algorithm 2 BNC Training (Laplace smoothing)

Require: $\Delta_{train}, A, \Omega, \ell$
P_Ω, P_A initialize to 0
N = count_rows(Δ_{train})
for $i = 1 \to c$ **do**
mask$_i$ = $\{[\boldsymbol{x}^{(m)}\ t_m] \in \Delta_{train} \mid t_m = \omega_i\}$
N_{ω_i} = count_rows(mask$_i$)
$P_\Omega(i) = N_{\omega_i}/N$
$P_d(i) = \ell/(N_{\omega_i} + c\,\ell)$
for $j = 1 \to f$ **do**
for $k = 1 \to a$ **do**
mask$_k$ = $\{[\boldsymbol{x}^{(m)}\ t_m] \in \text{mask}_i \mid x_j^{(m)} = \alpha_k\}$
N_{x_k,ω_i} = count_rows(mask$_k$)
$P_A(i,j,k) = (N_{x_k,\omega_i} + \ell)/(N_{\omega_i} + c\,\ell)$
end for
end for
end for
return P_Ω, P_A, P_d

Algorithm 3 NBC Prediction (Canonical)

Require: $\boldsymbol{x}, P_A, P_\Omega, A, \Omega$
for $i = 1 \to c$ **do**
$g_i(\boldsymbol{x}) = P_\Omega(i)$
for $j = 1 \to f$ **do**
if $x_j \in A$ **then**
k = Decode(x_j, A)
$g_i(\boldsymbol{x}) = g_i(\boldsymbol{x})\ P_A(i,j,k)$
else
$g_i(\boldsymbol{x}) = 0$
end if
end for
end for
$y = \omega_i$ s.t. $\max_i(g_i(\boldsymbol{x}))$
return y

Algorithm 4 NBC Prediction (*a priori* version)

Require: $\boldsymbol{x}, P_A, P_\Omega, A, \Omega$
for $i = 1 \to c$ **do**
$g_i(\boldsymbol{x}) = P_\Omega(i)$
for $j = 1 \to f$ **do**
if $x_j \in A$ **then**
k = Decode(x_j, A)
if $P_A(i,j,k) \neq 0$ **then**
$g_i(\boldsymbol{x}) = g_i(\boldsymbol{x})\ P_A(i,j,k)$
end if
end if
end for
end for
$y = \omega_i$ s.t. $\max_i(g_i(\boldsymbol{x}))$
return y

Algorithm 5 NBC Prediction (Laplace smoothing)

Require: $\boldsymbol{x}, P_A, P_\Omega, P_d, A, \Omega$
for $i = 1 \to c$ **do**
$g_i(\boldsymbol{x}) = P_\Omega(i)$
for $j = 1 \to f$ **do**
if $x_j \in A$ **then**
k = Decode(x_j, A)
$g_i(\boldsymbol{x}) = g_i(\boldsymbol{x})\ P_A(i,j,k)$
else
$g_i(\boldsymbol{x}) = P_d(i)$
end if
end for
end for
$y = \omega_i$ s.t. $\max_i(g_i(\boldsymbol{x}))$
return y

Chapter 3

Linear regression

If in the previous cases we where in Scenario 1 and 2, we are now in **Scenario 3** (only data are available).

The situation is like an election prediction: in that case we don't have probabilities because of the high level of complexity of human behavior interpretation. In this situation, since the **data** will be **stochastic** and **probabilities** are **not known** the hypothesis space H will be chosen in advance. We will have to find then the best hypothesis for **any possible realization of the data** ($\rightarrow$ generalization task).

This is our **usual situation**, since now on we will work in this scenario.

In this Chapter[1] we will solve a **regression** problem, so we will approximate a functional dependency based on measured data (a typical **supervised** problem).

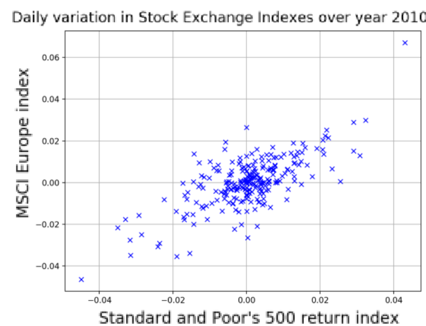
The data will be represented as:

$$\text{Observations} = \begin{pmatrix} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \dots \\ \boldsymbol{x}_N \end{pmatrix} \qquad \text{Target} = \begin{pmatrix} \boldsymbol{t}_1 \\ \boldsymbol{t}_2 \\ \dots \\ \boldsymbol{t}_N \end{pmatrix}$$

We will start from the one dimensional linear regression (**1D-LR**) in order to present then the variant with offset and the multidimensional variant (***n*D-LR**)

3.1 One-dimensional LR

Let's say that we want to predict the variation of the MSCI European index by observing Standard and Poor's 500 return index. So we have these 249 observations made in year 2010 (Source: UCI):



- Observation: x is the value of the variation of Standard and Poor's (SP) 500 return index on a given day.
- Target: t is the value of the variation of the MSCI European index (MSCI) on the same day.

$\rightarrow$ There is clearly some relationship between the two values, but not one-to-one.

[1]See either (or both) of [HASTIE2009] and [BISHOP2006] as reference.

A linear model for approximating the data $y(x)$ has to predict t given x:

$$t \simeq y \text{ where } y = wx$$

For instance :

$$t_1 \simeq y_1 \text{ where } y_1 = wx_1 \qquad t_2 \simeq y_2 \text{ where } y_2 = wx_2$$

So we want $y(x)$ to be similar to $t(x)$ for any x. Let's compute the parameter w then:

$$x_1 = 0.0159 \Rightarrow y_1 = 0.0159 \cdot w \text{ must approximate } t_1 = 0.0167$$

So:

$$w = \frac{0.0167}{0.0159} = 1.0503$$

However,

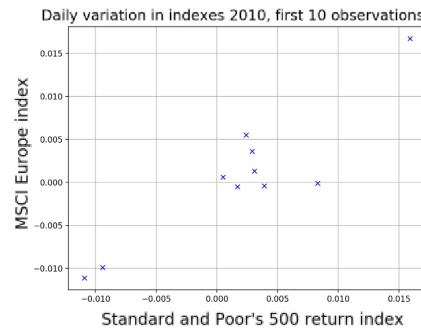
$$x_2 = 0.031 \Rightarrow y_2 = 0.0159 \cdot w \text{ must approximate } t_2 = 0.0013$$

So:

$$w = \frac{0.0013}{0.0031} = 0.4194$$

So, clearly this is not the best way to do it $\rightarrow$ we have to solve an **optimization problem**.

In fact, just consider the first 10 days of 2011:



We will have:

The first 10 observations:

No.	Date	SP	MSCI
1	2010-01-04	0.0159	0.0167
2	2010-01-05	0.0031	0.0013
3	2010-01-06	0.0005	0.0006
4	2010-01-07	0.0034	−0.0004
5	2010-01-08	0.0029	0.0036
6	2010-01-11	0.0017	−0.0005
7	2010-01-12	−0.0094	−0.0099
8	2010-01-13	0.0083	−0.0001
9	2010-01-14	0.0024	0.0055
10	2010-01-15	−0.0109	−0.0111

The first 10 equations:

$$\begin{cases} 0,0159\ w &= 0,0167 \\ 0,0031\ w &= 0,0013 \\ 0,0005\ w &= 0,0006 \\ 0,0039\ w &= -0,0004 \\ 0,0029\ w &= 0,0036 \\ 0,0017\ w &= -0,0005 \\ -0,0094\ w &= -0,0099 \\ 0,0083\ w &= -0,0001 \\ 0,0024\ w &= 0,0055 \\ -0,0109\ w &= -0,0111 \end{cases}$$

Here is clear the need for an optimization problem solution.

3.1.1 LR as an optimization problem

As we saw, we cannot hope to find a value for w that is good for all points. We should be satisfied with a value which is generally not so bad for most of the points. More rigorously:

> The expected (average) error should be low.

The idea is to quantify **how wrong** is each estimate using some **measure**, and make this measure as small as possible **on average**.

The so-called "measure" will be the **loss function**. which are the properties of a good loss?

- All errors give a positive contribution:

$$\lambda(t, y) = \lambda(-t, -y).$$

- λ has to be differentiable

Let's consider some options:

- Error $\lambda_E(t, y) = t - y \rightarrow$ is not an even function!
- Absolute error $\lambda_{AE}(t, y) = |t - y| \rightarrow$ is not differentiable!
- Square error $\lambda_{SE}(t, y) = (t - y) \rightarrow$ that is the simplest option that fits, in fact:
  - is even: $(t - y)^2 = (y - t)^2$
  - grows more than linearly, giving heavier weight to larger errors
  - is differentiable with respect to the model output:

$$\frac{d}{dy}\lambda_{SE}(t, y) = 2(y - t)$$

The **objective function** (or cost function) J will reflect our generic goal of minimize the mean value of the loss over the whole data set:

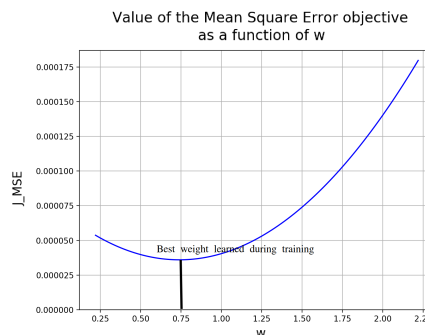
$$J = \frac{1}{N}\sum_{i=1}^{N}\lambda(t_l, y_l)$$

In the specific case of the square error loss we have J_{MSE}, that is the mean square error objective function:

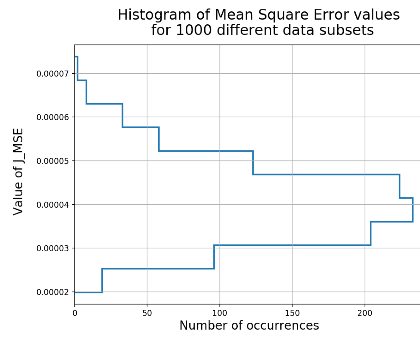
$$J = \frac{1}{N}\sum_{i=1}^{N}(t_l, y_l)^2$$

Since loss function $\lambda(t, y)$ is a function of the two arguments t and y, we can look at the objective in two complementary situations:

- Build the model: given the data set, the targets $t_1, ..., t_N$ are fixed $\rightarrow$ the objective **depends** only on the **model parameter**(s): given the values of y you change the value of w, it's the **learning part**: the objective is a function of the parameters in the model and the data are fixed



- Use the model: given a model, the parameters are fixed. However we can apply this model to various data set $\rightarrow$ the objective **depends** only on the **data**: once you fixed one of these hypothesis you "freeze" w and you have your model, then you'll have this **inference part**: the objective is a function of the data, which uses the (now fixed) model parameters



3.1.2 Solution of the 1D-LR problem

Let's see now some methods in order to obtain the solution to the 1D-LR problem.

The least squares method The problem for this method is to minimize J_{MSE} with respect to the parameters with fixed data. We know that J_{MSE} is a **parabola**, hence we have a **unique solution**:

$$w \quad \text{s.t.} \frac{d}{dw} J_{MSE} = 0$$

This is a necessary but **generally not sufficient** condition, **but** in this case it is a parabola, so, it is also necessary.

How to compute $\frac{d}{dw} J_{MSE}$? Since $y = wx$

$$\frac{d}{dw}\lambda_{SE}(t,y) = \frac{d}{dy}\lambda_{SE}(t,y)\frac{d}{dw}y$$

Here we have used the chain rule of differentiation to write the derivative:

$$\frac{df(g(x))}{dx} = \frac{df(y)}{dy}\bigg|_{y=g(x)} \frac{dg(x)}{dx} = \frac{d}{dy}(t-y)^2\bigg|_{y=wx} \frac{d}{dw}(-wx) = 2(t-xw) \quad -x$$

The only variable is w, the rest is fixed:

$$\frac{d}{dw}\lambda_{\text{SE}}(t,y) = 2(t-xw)(-x) = -2xt + 2x^2 w$$

$$\frac{d}{dw}J_{\text{MSE}} = \frac{d}{dw}\frac{1}{N}\sum_{l=1}^{N}\lambda_{\text{SE}}(t_l,y_l) = \frac{1}{N}\sum_{l=1}^{N}\frac{d}{dw}\lambda_{\text{SE}}(t_l,y_l) = \frac{1}{N}\sum_{l=1}^{N}2\left(x_l^2 w - x_l t_l\right),$$

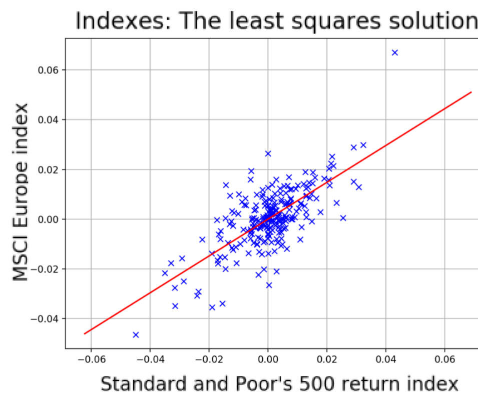
Exchange sum and derivative — where the constant coefficient $(2/N)$ is irrelevant and can be disregarded

This equation is solved by bringing w outside the sum, since it does not depend on l:

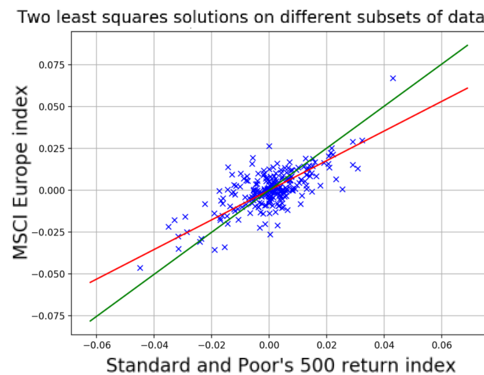
$$w\sum_{l=1}^{N}x_l^2 - \sum_{l=1}^{N}x_l t_l = 0 \qquad w = \frac{\sum_{l=1}^{N}x_l t_l}{\sum_{l=1}^{N}x_l^2}$$

Why it's a a linear equation even if there is a square? Because it is a square of a **constant value**, because it's data, not a variable.

This is the least square solution to the linear regression problem. You should not aim to solve it for perfect data, you'll never have it. For our example:



For two different subsets of data:

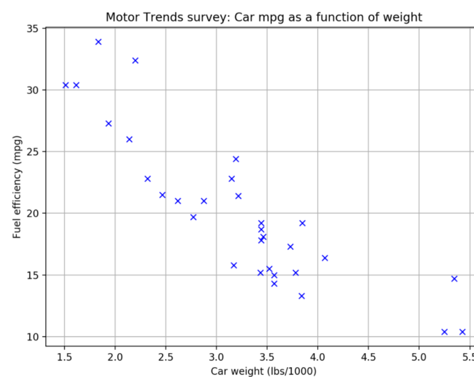


3.2 1D-LR with offset

Considering the "1974 'Motor Trends' car data (four columns)", that contains the result of a 1974 survey on some car models. *mpg* is miles-per-gallon, *disp* is displacement (in cu.in), *hp* is horse-power and *weight* is the total weight given in lbs/1000 (note that these are USA units):

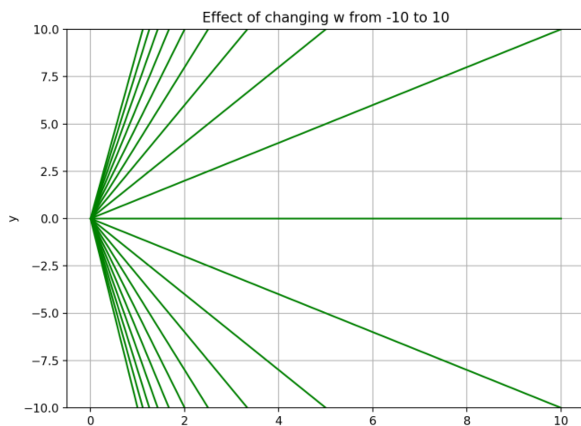
Model	mpg	disp	hp	weight
Mazda RX4	21	160	110	2.62
Mazda RX4 Wag	21	160	110	2.875
Datsun 710	22.8	108	93	2.32
Hornet 4 Drive	21.4	258	110	3.215
Hornet Sportabout	18.7	360	175	3.44
Valiant	18.1	225	105	3.46
Duster 360	14.3	360	245	3.57
Merc 240D	24.4	146.7	62	3.19

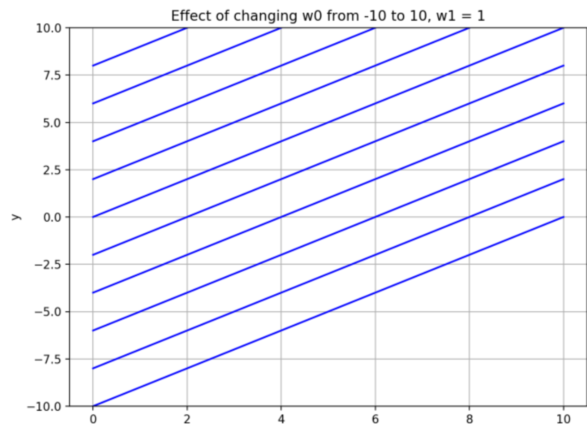
If we forecast *mpg* with *weight* we obtain this set on which use linear regression:



But now let's introduce a more flexible model adding an **offset**:

$y = wx$ $\qquad\qquad$ $y = w_1 x + w_0$



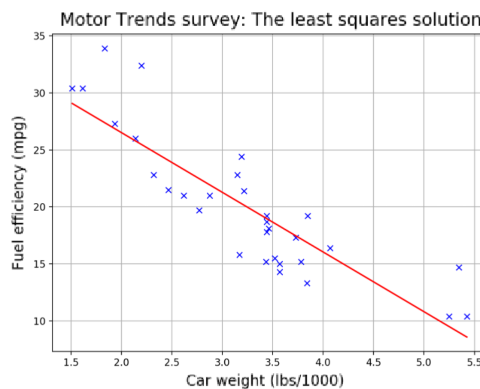


The **solution** in this case can be found by **centering** around the mean $\bar{x}$ of x and $\bar{t}$ of t:

$$\overline{x} = \frac{1}{N}\sum_{l=1}^{N} x_l \qquad \overline{t} = \frac{1}{N}\sum_{l=1}^{N} t_l \qquad w_1 = \frac{\sum_{l=1}^{N}(x_l - \overline{x})(t_l - \overline{t})}{\sum_{l=1}^{N}(x_l - \overline{x})^2} \qquad w_0 = \overline{t} - w_1 \overline{x}$$

Here, w_1 is the slope (**gain**) and w_0 the intercept, the offset (**bias**).

With this flexible model we're allowed to have solutions like:



We have switched from a linear to an **affine model** $y = w_0 + x w_1$.

3.3 The multi-dimensional linear regression problem

The data is now composed of d-dimensional vectors:

$$\begin{aligned}
\mathbf{x}_1 &= [\, x_{1,1}, x_{1,2}, \ldots, x_{1,d}\,] \\
\mathbf{x}_2 &= [\, x_{2,1}, x_{2,2}, \ldots, x_{2,d}\,] \\
&\ldots \\
\mathbf{x}_N &= [\, x_{N,1}, x_{N,2}, \ldots, x_{N,d}\,]
\end{aligned}$$

so we can organize them into a $N \times d$ matrix:

$$X = \begin{pmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \\ \vdots \\ \mathbf{x}_N \end{pmatrix} = \begin{pmatrix} x_{1,1} & x_{1,2} & \ldots & x_{1,d} \\ x_{2,1} & x_{2,2} & \ldots & x_{2,d} \\ x_{3,1} & x_{3,2} & \ldots & x_{3,d} \\ & & \vdots & \\ x_{N,1} & x_{N,2} & \ldots & x_{N,d} \end{pmatrix}$$

Since the data are now d-dimensional, we have d parameters in a d-dimensional vector:

$$\mathbf{w} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_d \end{pmatrix}$$

The linear model takes the d inputs of each observation and combines them by using the d parameters to produce one output:

$$\begin{aligned} y_1 &= x_{1,1}w_1 + x_{1,2}w_2 + \ \dots \ + x_{1,d}w_d \\ y_1 &= x_{2,1}w_1 + x_{2,2}w_2 + \ \dots \ + x_{2,d}w_d \\ &\dots \\ y_N &= x_{N,1}w_1 + x_{N,2}w_2 + \ \dots \ + x_{N,d}w_d \end{aligned}$$

This can be expressed as a matrix-vector multiplication between the data matrix X and the parameter vector $\boldsymbol{w}$:

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_N \end{pmatrix} = \begin{pmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,d} \\ x_{2,1} & x_{2,2} & \dots & x_{2,d} \\ x_{3,1} & x_{3,2} & \dots & x_{3,d} \\ & & \vdots & \\ x_{N,1} & x_{N,2} & \dots & x_{N,d} \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_d \end{pmatrix} = \begin{pmatrix} x_{1,1}w_1 & x_{1,2}w_2 & \dots & x_{1,d}w_d \\ x_{2,1}w_1 & x_{2,2}w_2 & \dots & x_{2,d}w_d \\ x_{3,1}w_1 & x_{3,2}w_2 & \dots & x_{3,d}w_d \\ & & \vdots & \\ x_{N,1}w_1 & x_{N,2}w_2 & \dots & x_{N,d}w_d \end{pmatrix} = X\mathbf{w}$$

Finally, our goal is to make this model $\boldsymbol{y}$ as similar as possible to the measured outputs for each observation, which again can be organized as a vector, this time N-dimensional:

$$\mathbf{t} = \begin{pmatrix} t_1 \\ t_2 \\ t_3 \\ \vdots \\ t_N \end{pmatrix}$$

We will have then the square error objective in matrix-vector form:

$$\begin{aligned} J_{\text{MSE}} &= \frac{1}{2}||\mathbf{t} - \mathbf{y}||^2 \\ &= \frac{1}{2}||\mathbf{t} - X\mathbf{w}||^2 \\ &= \frac{1}{2}(\mathbf{t} - X\mathbf{w})^T(\mathbf{t} - X\mathbf{w}) \\ &= \frac{1}{2}(\mathbf{t}^T - \mathbf{w}^T X^T)(\mathbf{t} - X\mathbf{w}) \\ &= \frac{1}{2}\left(\mathbf{t}^T\mathbf{t} - \mathbf{t}^T X\mathbf{w} - \mathbf{w}^T X^T\mathbf{t} + \mathbf{w}^T X^T X\mathbf{w}\right) \\ &= \frac{1}{2}||\mathbf{t}||^2 - \mathbf{w}^T X^T\mathbf{t} + \frac{1}{2}||X\mathbf{w}||^2 \end{aligned}$$

We can simplify the objective by disregarding the constant term $\frac{1}{2}||\boldsymbol{t}||^2$:

$$J_{\text{MSE}} = \cancel{\frac{1}{2}||\mathbf{t}||^2} - \mathbf{w}^T X^T\mathbf{t} + \frac{1}{2}||X\mathbf{w}||^2 = \frac{1}{2}||X\mathbf{w}||^2 - \mathbf{w}^T X^T\mathbf{t}$$

This is a **paraboloid**, a d-dimensional parabola, in the variables $\boldsymbol{w}$. It has a minimum when

$$\begin{cases} \partial J_{\text{MSE}}/\partial w_1 &= 0 \\ \partial J_{\text{MSE}}/\partial w_2 &= 0 \\ \partial J_{\text{MSE}}/\partial w_3 &= 0 \\ \vdots \\ \partial J_{\text{MSE}}/\partial w_d &= 0 \end{cases} \quad \text{or} \quad \nabla J_{\text{MSE}} = \mathbf{0}$$

where $\mathbf{0} = (0, 0, \ldots 0)^T$ and we can still solve this equation in closed form

But how we reach the **closed-form solution**? It can be proven that we can write:

$$\nabla J_{\text{MSE}} = \frac{\partial}{\partial \mathbf{w}} J_{\text{MSE}} = X^T X \mathbf{w} - X^T \mathbf{t} \longrightarrow \text{Setting } \nabla J_{\text{MSE}} = \mathbf{0} \text{ we get} \longrightarrow X^T X \mathbf{w} = X^T \mathbf{t}$$

By premultiplying both sides by $(X^T X)^{-1}$, we obtain the closed-form solution:

$$\boxed{\boldsymbol{w} = (X^T X)^{-1} X^T \boldsymbol{t}}$$

that is the matrix form of the **normal equations** for the **least squares problem**. This one matrix equation is **equivalent to a system** of $d+1$ simultaneous equations in $d+1$ unknowns. Checking the dimensions, everything works:

$$X \text{ is } (N \times d) \text{ and } X^T X \text{ is square } (d+1) \times (d+1).$$

$$(X^T X)^{-1} X^T = \textbf{Moore-Penrose pseudoinverse} \text{ of } X = X^\dagger$$

$$\rightarrow \boxed{\boldsymbol{w} = X^\dagger \boldsymbol{t}}$$

So, once we have solved a linear regression problem, we have received a new point $\boldsymbol{x}^*$ and we want to obtain the value $\boldsymbol{y}^*$ that **estimates the most probable value of the output**, how do we proceed? This is inference and works simply like this:

$$\boldsymbol{y}^* = \boldsymbol{w}^* \cdot \boldsymbol{x}^*$$

3.4 Numerical issues

There are some numerical issues related to this method:

- $X^T X$ **might not be invertible**. This is when the data have exactly linearly dependent components
- What if the **variables are correlated**?
- Even if $X^T X$ has full rank, in the case of correlated variables it will have a **high condition number** $\frac{\lambda_{first}}{\lambda_{last}}$
- It can be difficult to invert (numerical precision must be too high)
- This is a problem of **numeric instability**: even very small numeric errors are amplified by the condition number and cause large errors on the result

⇒ **SOLUTION**: Iterative computation by successive approximations (e.g., by gradient descent)

3.5 Summary

- Linear regression – an example of simple learning problem: forecast one continuous variable using observations (= one or more other variables) related to it

- Univariate linear regression: forecasting the most likely value of the target variable t as a linear function $y = w_1 x + w_0$ of the observed variable x
- The problem is a system of N linear equations in two unknowns w_1, w_0 – in general no solution
- Solved by minimizing an **objective function** that is the expectation of a chosen **loss function**
- We used the **square error loss** obtaining the **mean square error** objective
- The problem admits a closed-form solution

- Multivariate linear regression: Forecasting the most likely value of the target variable t as a linear function $y = \boldsymbol{w} \cdot \boldsymbol{x}$ of the observed vector variable $\boldsymbol{x}$
- The problem is a $N \times (d + 1)$ system of linear equations (N equations in $d + 1$ unknowns $\boldsymbol{w}$ – in general no solution if $d + 1 < N$)
- Again solved by minimizing the **mean square error** objective function
- The problem admits a closed-form solution using the **Moore-Penrose pseudo-inverse**
- However this solution may not exist or be low-quality due to numeric instability
- Iterative approximation methods exist (e.g., by gradient descent)

Linear Regression Implementation and Performance Evaluation

Report for the Machine Learning course, EMARO

Davide Lanza
EMARO+ M2
Genoa, Italy
davidel96@hotmail.it

***Abstract*—In this report will be analyzed a MATLAB implementation of the Linear Regression technique. After a short introduction, we will define the theoretical framework from which we derived our implementation, focusing on the variant with offset. We will estimate then the accuracy of such regressors w.r.t. a big and small datasets. Analyzing ad comparing the different results obtained for the mono- and multi-dimensional case, we will show that the offset variance lowers the error, but significantly only for small or unbalanced datasets.**

***Index Terms*—Machine Learning, Linear Regression, MATLAB**

I. Introduction

Thanks to the exponential increase of data available on the Internet, the volume of information available for perform machine learning task grows incredibly. Regression tasks – as the approximation of a functional dependency based on measured data – are an important component in information extraction and for predictive tasks.

In this paper, we focus on a specific regression method: the linear one. First, we will discuss the theoretical framework from which we derived our MATLAB implementation, then we will test it w.r.t. a high-dimension dataset and a low-dimension dataset regarding mono-dimensional linear regression problem, and w.r.t. a low-dimension dataset regarding multi-dimensional linear regression problem. Finally, the testing results are analyzed, in order to evaluate the influence of the training-test dataset size ratio and the incidence of the presence of an offset in the regressor.

The datasets used in this study are two. The first one is a high dimensional dataset and it is the Istanbul Stock Exchange dataset[1], which includes the returns of Istanbul Stock Exchange (BIST) with seven other international index (SP, DAX, FTSE, NIKKEI, BOVESPA, MSCE_EU, MSCI_EM) from Jun 5, 2009 to Feb 22, 2011 **Balaban2013**. The second one is a low dimensional dataset extracted from the 1974 Motor Trend US magazine, comprising fuel consumption and 10 aspects of automobile design and performance for 32 automobiles.[2].

[1] Available at https://archive.ics.uci.edu/ml/datasets/ISTANBUL+STOCK+EXCHANGE.

[2] It is the `mtcars` dataset of R.

II. Implementation

In this section, we will present the theoretical framework from which we have implemented our Linear Regressor (LR). Starting from the Multidimensional LR problem statement we will the introduce the modified solution with offset (MLR-O).

A. Problem statement

In a regression problem we want to approximate a functional dependency $t = f(\boldsymbol{x})$ with a LR that provides $y = LR(\boldsymbol{x})$ with the output y as much close as possible to the target t. In order to train the LR, we will have a dataset of o observations, each one is a vector of f features $\boldsymbol{x} = [x_1, ..., x_f]^T$ (if $f = 1$ we will have a one-dimensional linear regression problem $t = f(x)$). The LR is built in the following way

$$LR(\boldsymbol{x}) = [w_1, ..., w_f] \cdot \boldsymbol{x} = \boldsymbol{w}^T \cdot \boldsymbol{x} \tag{1}$$

where $\boldsymbol{w}$ is a column vector of parameters that divides each i-th dimension's plane $\{x_i, y\}$ with a line $y = w_i x_i$ passing through the origin (in fact, for one-dimensional cases $f = 1$ the LR is an actual straight line following as much as possible the distribution of observations in the function plane $\{x, y\}$).

B. Loss function and training equation

Since we cannot hope to find a value for $\boldsymbol{w}$ that is good for all points, ee have to define a loss function $\lambda(t, y)$ in order to minimize the expected (average) error. Since a good λ should be even, should grow more than linearly (in order to weight more bigger errors) and should be differentiable, we chase the Square Error one:

$$\lambda_{SE}(t, y) = \frac{1}{2}(t - y)^2$$

Given a dataset composed by a set of o observations $X = \{\boldsymbol{x}^{(i)T} | 1 \leq i \leq o\}$ and respective truth values $\boldsymbol{t} = \{t_i | 1 \leq i \leq o\}$ such that $\Delta = [X, \boldsymbol{t}]$

$$\Delta = \begin{array}{|ccccc|} x_1^{(1)} & x_2^{(1)} & \cdots & x_f^{(1)} & t_1 \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_f^{(2)} & t_2 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_1^{(o)} & x_2^{(o)} & \cdots & x_f^{(o)} & t_o \end{array} \tag{2}$$

the goal is to minimize the mean value of $\lambda_{SE}(t, y)$ over all the losses of each sample $\{\boldsymbol{x}^{(i)T}, t_i\}$ of the dataset, given the outputs $y_i = LR(\boldsymbol{x}_i)$:

$$J_{MSE}(\Delta) = \frac{2}{o}\sum_{i=1}^{o}(t_i - y_i)^2 =$$

$$= \frac{2}{o}\sum_{i=1}^{o}(t_i - \boldsymbol{w}^T\boldsymbol{x}^{(i)})^2 = \frac{2}{o}(\boldsymbol{t} - X\boldsymbol{w})^2 \qquad (3)$$

Due to the quadratic nature of the minimization problem, the following necessary condition for the existence of a minimum is also sufficient:

$$\frac{\partial}{\partial y}J_{MSE}(\Delta) = \frac{1}{o}X^T(\boldsymbol{t} - X\boldsymbol{w}) = 0 \qquad (4)$$

We can then obtain the following training equation for the MLR:

$$\boldsymbol{w} = \frac{X^T}{X^TX}\boldsymbol{t} = X^\dagger\boldsymbol{t} \qquad (5)$$

where $X^\dagger$ is the Moore-Penrose pseudoinverse **Moore1920Penrose1955**.

C. MLR with offset

To obtain a more sophisticated MLR, for which the lines each the dimension of the hyperspace does not necessarily pass through the origin. In this case, we have an additional term

$$LR\text{-}O(\boldsymbol{x}) = [w_0, w_1, ..., w_f] \cdot \begin{bmatrix} 0 \\ \boldsymbol{x} \end{bmatrix} = \boldsymbol{w}^T \cdot \boldsymbol{x} + w_0 \qquad (6)$$

For the MLR-O the training equation have to be centered around the mean $\langle\boldsymbol{x}\rangle$ of $\boldsymbol{x}$ and $\langle t\rangle$ of t

$$\langle\boldsymbol{x}\rangle = \frac{1}{o}\sum_{i=1}^{o}\boldsymbol{x}^{(i)} \qquad \langle t\rangle = \frac{1}{o}\sum_{i=1}^{o}\boldsymbol{t}_i \qquad (7)$$

$$X_c = X - \begin{bmatrix} \langle\boldsymbol{x}\rangle^T \\ \vdots \\ \langle\boldsymbol{x}\rangle^T \end{bmatrix} \qquad \boldsymbol{t}_c = \boldsymbol{t} - \begin{bmatrix} \langle t\rangle \\ \vdots \\ \langle t\rangle \end{bmatrix} \qquad (8)$$

so the equations are:

$$\boldsymbol{w} = X_c^\dagger\boldsymbol{t} \qquad (9)$$

$$w_0 = \langle t\rangle - \boldsymbol{w}^T \cdot \langle\boldsymbol{x}\rangle \qquad (10)$$

III. Performance Evaluation

The MLR-O has been tested for a one-dimensional problem on the Istanbul Stock Exchange data using the SP index as observation and the MSCE_EU index as target, for a one-dimensional problem on the Motor Trends car data using the car's weight as observation and the mpg (miles/gallon) as target, and for a multi-dimensional problem on the complete Motor Treend cars data, using three columns (displacement, horsepower and weight) in order to predict the mpg.

The solution have been obtained on different random subsets of the whole data set. The training-test ratio has been tried for $1:9$, $7:1$ and $9:1$ values, preparing the training and test set dataset as shown in the Algorithm 1, in order to preserve the temporal order of the subsequent samples (significant choice for the stock index case).

Algorithm 1: Dataset preparation

```
Require: Δ, n_folds, n_train
  Δ_train, Δ_test = ∅
  folds = Split(Δ, n_folds)
  indexes = Randperm(n_folds)
  for i = 1 → n_train do
    Append(folds(indexes(i))) to Δ_train
  end for
  for i = n_train + 1 → n_folds do
    Append(folds(indexes(i))) to Δ_test
  end for
  return Δ_train, Δ_test
```

The performance has been evaluated computing the mean square error $J_{MSE}(\Delta_{train})$ and $J_{MSE}(\Delta_{test})$ between prediction $\boldsymbol{y}$ and the target $\boldsymbol{t}$ after the training on Δ_{train}. In Appendix can be found the results for each case, in which the the LR has been tested for 10000 iterations, and the error in the graph has been successively averaged with the previous values in order to show the convergence of the average for each case.

A. LR on SP-MSCE_EU

Considering the case without offset for which the training-test ratio is 1:9 (LR trained on 10% of Δ), the error is low w.r.t. the training set, because the samples are less w.r.t. Δ, then find a line that will fit them will be easier than for the complete dataset. This leads to very different LR depending on the Δ_{train} used, as it is possible to see from the two cases in Figure 1 and in Figure 2, where two different Δ_{train} have been chosen (the red samples) along Δ (the blue dots).

Of course, a small training ratio as the 10% will lead to serious problems of overfitting and poor generalization, that is, a low error while testing Δ_{train} and a great error in testing Δ_{test}. In this case though, the dataset is big enough to avoid it, as it is possible to see from the results reported in Table 1, where J_{MSE} has been averaged on the 10000 iterations performed. In the Table are reported also the results for the LR-O version of the classifier, that seems to lowly enhance overfitting for low train ratios and lowly enhance generalization for high ratios.

Testing on	$\Delta_{train} : \Delta_{test}$	$\langle J\rangle$ for LR	$\langle J\rangle$ for LR-O
Δ_{train} Δ_{test}	$1:9$	$8.6319 \cdot 10^{-5}$ $9.397 \cdot 10^{-5}$	$8.4453 \cdot 10^{-5}$ $9.4914 \cdot 10^{-5}$
Δ_{train} Δ_{test}	$7:3$	$8.9178 \cdot 10^{-5}$ $9.1509 \cdot 10^{-5}$	$8.926 \cdot 10^{-5}$ $9.0654 \cdot 10^{-5}$
Δ_{train} Δ_{test}	$9:1$	$8.9608 \cdot 10^{-5}$ $8.9716 \cdot 10^{-5}$	$8.9703 \cdot 10^{-5}$ $8.829 \cdot 10^{-5}$

Table 1: Average MSEs for 1-D problem (SP-MSCE_EU)

Figure 1: LR plot for 1D problem (SP-MSCE_EU)

Figure 2: LR plot for 1D problem (SP-MSCE_EU)

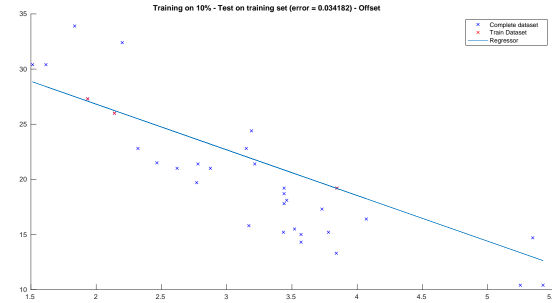


Figure 3: LR-O plot for 1D problem (MT Trends)

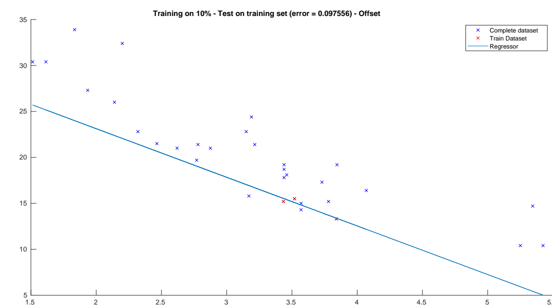


Figure 4: LR-O plot for 1D problem (MT Trends)

B. LR on MT Trends

For this low-dimension dataset, considering the case with offset for which the training-test ratio is 1:9 (training on 10% of Δ), the effect hypothesized for the previous case is strong and clearly visible in Figure 3 and in Figure 4, where the two LR-O are strongly different.

This has consequences that are clearly notable on the 10000 iterations results reported in Table 2, where a strong overfitting is notable for the $1:9$ case (training on only 3 samples along 32). The two error averages are almost balanced only for the $9:1$ case, for which, in the LR case, the final value is high. The offset here drastically enhance the results for each case, as reported in the Table.

Testing on	$\Delta_{train} : \Delta_{test}$	$\langle J \rangle$ for LR	$\langle J \rangle$ for LR-O
Δ_{train}	$1:9$	67.4494	1.1656
Δ_{test}		240.9808	63.7078
Δ_{train}	$7:3$	126.2138	8.4395
Δ_{test}		139.1455	10.5554
Δ_{train}	$9:1$	128.6478	8.7834
Δ_{test}		130.2314	8.8516.

Table 2: Average MSEs for 1-D problem (MT Trends)

C. MLR on MT Trends

For this low-dimensional dataset, we considered then the multidimensional case ($f = 3$). As it is possible to see from the 10000 iterations results reported in Table 3, both the MLR and the MLR-O performs better for high train rations (70% and 90%), but the overfitting is worse for the 10% case. Also here, the version with offset performs significantly better for all the cases.

Testing on	$\Delta_{train} : \Delta_{test}$	$\langle J \rangle$ for LR	$\langle J \rangle$ for LR-O
Δ_{train}	$1:9$	$3.2503 \cdot 10^{-2}$	$3.3581 \cdot 10^{-2}$
Δ_{test}		3677.0591	309.7325
Δ_{train}	$7:3$	70.0117	5.6216
Δ_{test}		91.1231	8.1223
Δ_{train}	$9:1$	73.108	6.001
Δ_{test}		73.3576	6.1095

Table 3: Average MSEs for 3-D problem (MT Trends)

IV. Conclusions

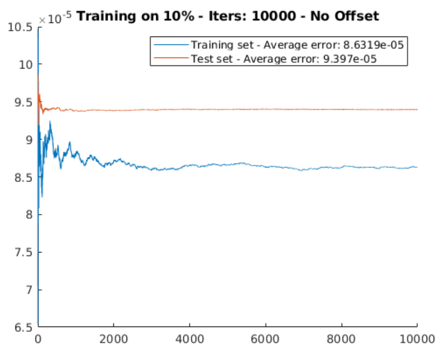
From the analysis performed, we confirm that LR is a sound learner for consistent, statistically centered, and big datasets as the Istanbul one, while the LR-O it does not increase significantly the already good performances. For small, statistically unbalanced datasets as the MT Trends one, if the LR does not performs well, the LR-O increases significantly the performances both in mono-dimensional (LR-O) and multi-dimensional (MLR-O) cases.

V. Future Work

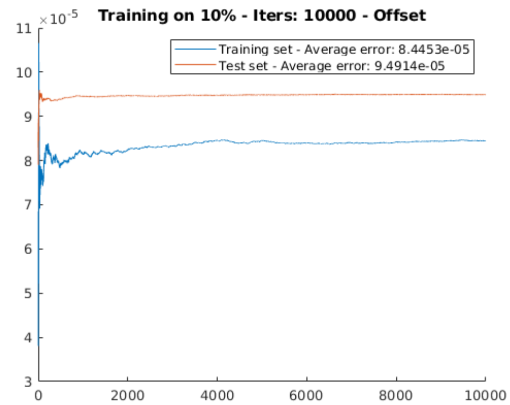
Regarding the SP-MSCE_EU case-study analysis, a better way to estimate the performances of linear regression should be to compare these results with a multidimensional problem $f > 1$. It should be interesting also to perform test for a big, statistically un-centered dataset. Regarding the MT Trends case-study analysis, $size(\Delta) = 32$ is definitely too little in order to draw a conclusion w.r.t. small size datasets. Further analysis should test bigger datasets, comparing different $size(\Delta) = 10^i$ $(0 \leq i \leq 4)$. Another variation should be testing it with a small, statistically centered dataset.

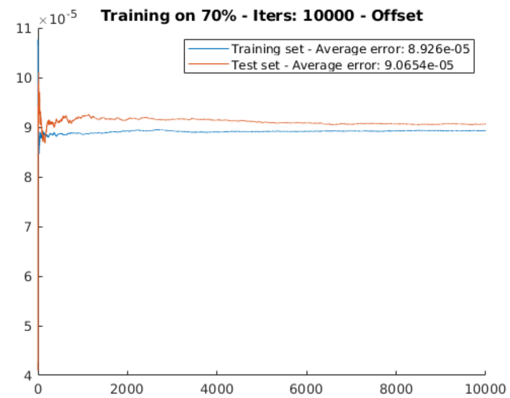
Appendix

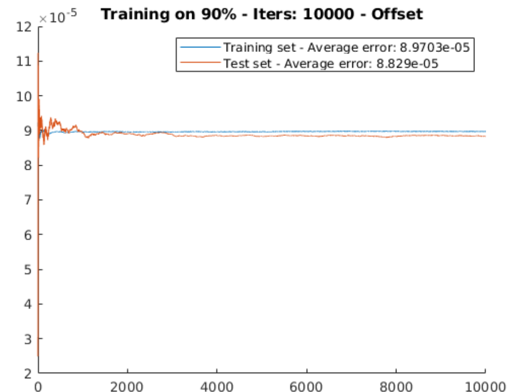
A. LR on SP-MSCE_EU



B. LR-O on SP-MSCE_EU

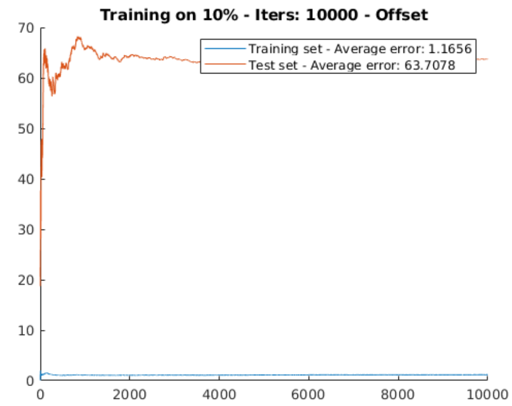


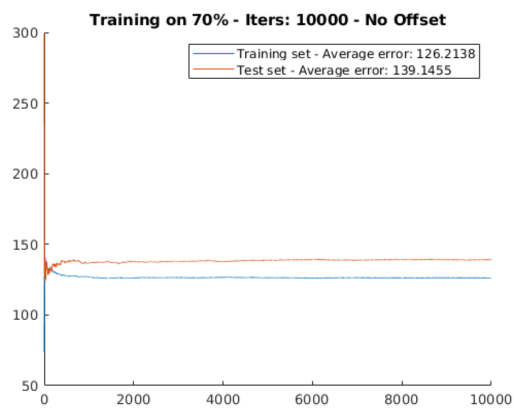


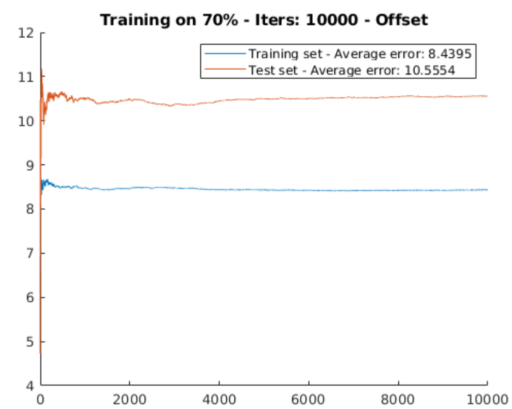


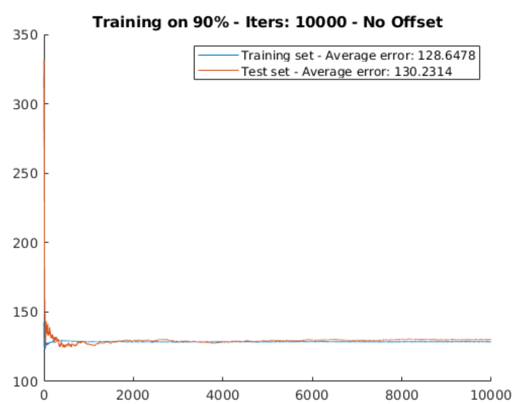
C. LR on MT Trends

D. LR-O on MT Trends



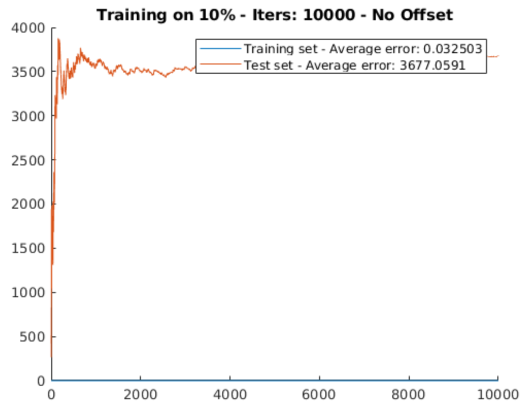


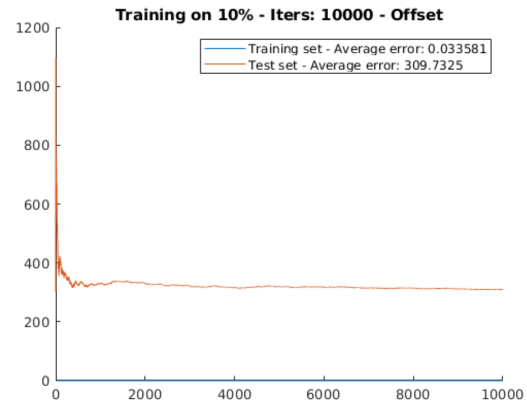


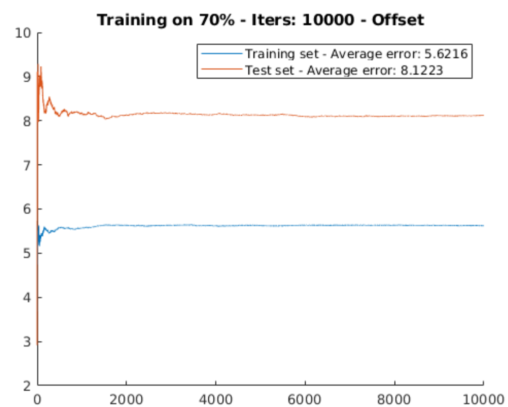


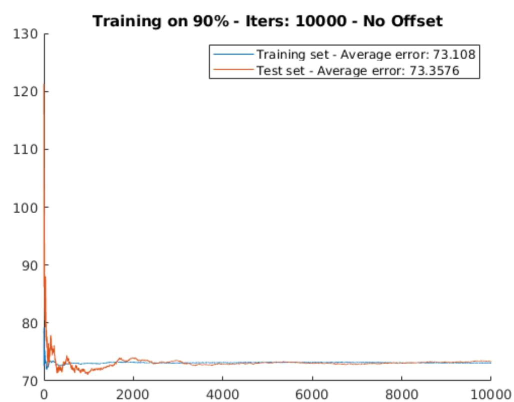
E. MLR on MT Trends

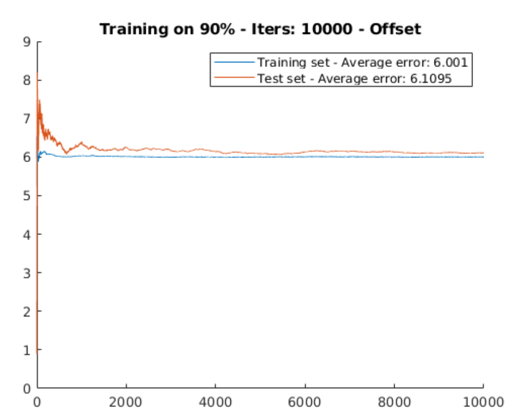
F. MLR-O on MT Trends











Chapter 4

Optimization

Why optimization is important? For example, it allows to plan in the most efficient way the maintenance interventions on a machine operating in a production line, scheduling them in order to reduce both the probability of failure and the cost of interventions.

The basic task of optimization is finding extrema of an **objective function**

$$E = f(\boldsymbol{w}) \qquad f : S \subset \mathbb{R}^m \to \mathbb{R}$$

An extreme is a point $\boldsymbol{w}^* \in S$ that may be a maximum or minimum.

A point w^* is a **minimum** if there is a neighborhood $\mathbb{R} \subseteq S$, where the following holds:

$$f(\boldsymbol{w}) \geq f(\boldsymbol{w}^*) \forall \boldsymbol{w} \in \mathbb{R}$$

So, a minimum is a point where f has a value smaller than in any other point in a given neighborhood.

A point w^* instead is a **maximum** if is a minimum of $-f$.

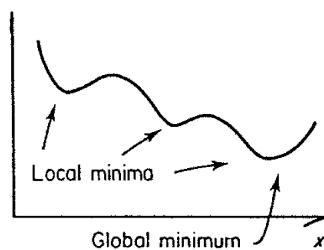
Because of the interchangeability of minima e maxima (from f to $-f$), we will talk only about **minimization**

4.1 Minimization problem

A minimum is **relative** if $\mathbb{R} \subseteq S$ strictly i.e., there is some other point in S (outside $\mathbb{R}$) where f has a smaller value than $f(\boldsymbol{w}^*)$.

A minimum is **absolute** if $\mathbb{R} = S$.

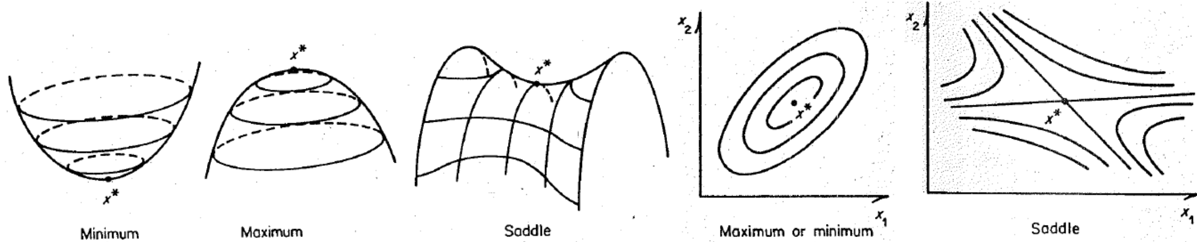
A relative minimum is a minimum only in a neighborhood (locally), so we also say “local” minimum for relative and “global” minimum for absolute.



What objective functions are we interested in? An objective function can also be termed **cost function** when we are minimizing it. A common type of cost function is an **error function** measuring the lack of quality of a learner (as we saw in the previous Chapter). A common type of objective function to maximize is a **performance function** measuring the quality of a learner.

Giving some other definitions:

- An **optimal solution** is an extremum of f, while an **optimal value** is $f(\boldsymbol{w}^*)$
- (non-standard, but useful) A **suboptimal solution** is an approximation to an optimal solution → value $\simeq f(\boldsymbol{w}^*)$
- A **feasible solution** is any point $\boldsymbol{w}$ which satisfies all hypotheses of the optimization problem (it might be an optimal solution).



The optimization problem is often made difficult by the presence of many local minima.

4.1.1 Convex sets and functions

A set $S \subset \mathbb{R}^m$ is convex if and only if, for any $\theta \in [0, 1]$

$$\forall \boldsymbol{v}, \boldsymbol{w} \in S \Rightarrow \theta \boldsymbol{v} + (1 - \theta)\boldsymbol{w} \in S$$

More generally, if for any $\theta_1 > 0, ..., \theta_n > 0$ such that

$$\forall \boldsymbol{v}_1, ..., \boldsymbol{v}_n \in S \Rightarrow \sum_k \theta_k \boldsymbol{v}_k \in S \qquad \textbf{Convex combination}$$

Properties:

- $\boldsymbol{v} \in \mathbb{R}^m$ (a single point) is convex
- $\emptyset = \{\}$ (the empty set) is convex
- $\mathbb{R}^m$ is convex

And if S_1 and S_2 are convex, then

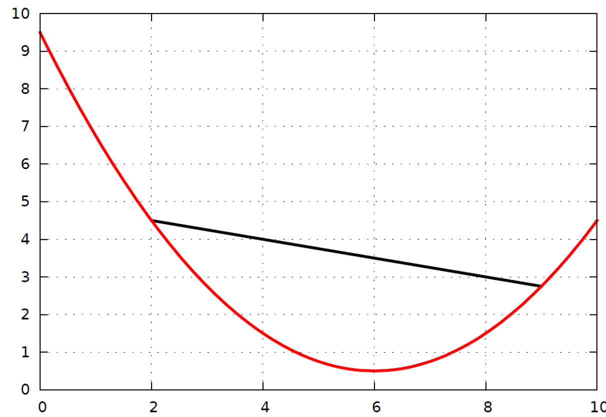
- $S_1 \cap S_2$ is convex
- $S_1 \cup S_2$ is **not necessarily convex** $(o + o = \infty)$

A function $f : S \in \mathbb{R}^m \to \mathbb{R}$ is convex if S is a convex set and if $\forall \boldsymbol{v}, \boldsymbol{w} \in S$, and with $0 \leq \theta \leq 1$:

$$f(\theta \mathbf{v} + (1 - \theta)\mathbf{w}) \leq \theta f(\mathbf{v}) + (1 - \theta) f(\mathbf{w})$$

i.e., if its epigraph is a convex set more generally for any $\theta_1 > 0, ..., \theta_n > 0$ such that $\sum_k \theta_k = 1$

$$\forall \mathbf{v}_1, \ldots, \mathbf{v}_n \in S \Rightarrow f\left(\sum_k \theta_k \mathbf{v}_k\right) \leq \sum_k \theta_k f(\mathbf{v}_k)$$



$$f(\boldsymbol{w}) \text{ concave } \Rightarrow -f(\boldsymbol{w}) \text{ convex}$$

But why should we be interested in convexity?

- Convexity is a property of a problem, not just of a loss function:
  - Objective
  - Learning machine on which the objective is computed (because we want to take derivatives)
  - **The parameter space** - remember that a function is not convex if its domain is not convex!
- Convexity is a good thing:
  - Uniqueness of extrema
  - Convergence of iterative algorithms
- Not all problems are convex:
  - Some learners guarantee a convex problem (e.g., SVM)
  - For non-convex problems we usually **cannot be sure whether a minimum is absolute (global) or relative (local)**
  - In non-convex optimization, often convex approximations (2nd order Taylor) are used iteratively

4.1.2 Gradient and Hessian

The **gradient** is a vector field:

$$\nabla f = \left[\frac{\partial f(\boldsymbol{w})}{\partial w_1}, \frac{\partial f(\boldsymbol{w})}{\partial w_2}, \dots, \frac{\partial f(\boldsymbol{w})}{\partial w_m} \right]^T$$

The derivative gives the rate of growth of a function of a scalar variable (negative sign $\rightarrow$ decreasing). The gradient scalar value (norm) gives the rate of maximum growth, while the direction gives the direction of maximum growth.

The gradient indicates the direction of maximum increase, and moving in the opposite direction $-\nabla f(\boldsymbol{w})$ we achieve the **maximum rate of decrease**. This observation is very useful in optimization techniques.

The **Hessian matrix** (or simply Hessian) is instead defined as:

$$H_f(\mathbf{w}) : \mathbb{R}^m \rightarrow \mathbb{R}^m \times \mathbb{R}^m \qquad \text{s.t.} \qquad h_{ij}(\mathbf{w}) = \frac{\partial^2 f(\mathbf{w})}{\partial w_i \partial w_j}$$

The Hessian matrix can be thought of as a list of m vectors

$$\boldsymbol{h}_i = \nabla(\nabla f(\boldsymbol{w}))_i$$

Derivative is a linear operator and the order of differentiation does not matter:

$$\frac{\partial^2 f(\mathbf{w})}{\partial w_i \partial w_j} = \frac{\partial}{\partial w_i}\left(\frac{\partial f(\mathbf{w})}{\partial w_j}\right) = \frac{\partial}{\partial w_j}\left(\frac{\partial f(\mathbf{w})}{\partial w_i}\right)$$

$\Rightarrow H$ is a symmetric matrix:

$$H_f(\mathbf{w}) = \begin{pmatrix} \frac{\partial^2 f}{\partial w_1^2} & \frac{\partial^2 f}{\partial w_1 \partial w_2} & \frac{\partial^2 f}{\partial w_1 \partial w_3} & \cdots & \frac{\partial^2 f}{\partial w_1 \partial w_m} \\ \frac{\partial^2 f}{\partial w_2 \partial w_1} & \frac{\partial^2 f}{\partial w_2^2} & \frac{\partial^2 f}{\partial w_2 \partial w_3} & \cdots & \frac{\partial^2 f}{\partial w_2 \partial w_m} \\ \frac{\partial^2 f}{\partial w_3 \partial w_1} & \frac{\partial^2 f}{\partial w_3 \partial w_2} & \frac{\partial^2 f}{\partial w_3^2} & \cdots & \frac{\partial^2 f}{\partial w_2 \partial w_m} \\ & & \vdots & & \\ \frac{\partial^2 f}{\partial w_m \partial w_1} & \frac{\partial^2 f}{\partial w_m \partial w_2} & \frac{\partial^2 f}{\partial w_m \partial w_3} & \cdots & \frac{\partial^2 f}{\partial w_m^2} \end{pmatrix} = \begin{pmatrix} \frac{\partial^2 f}{\partial w_1^2} & \frac{\partial^2 f}{\partial w_1 \partial w_2} & \frac{\partial^2 f}{\partial w_1 \partial w_3} & \cdots & \frac{\partial^2 f}{\partial w_1 \partial w_m} \\ & \frac{\partial^2 f}{\partial w_2^2} & \frac{\partial^2 f}{\partial w_2 \partial w_3} & \cdots & \frac{\partial^2 f}{\partial w_2 \partial w_m} \\ & & \frac{\partial^2 f}{\partial w_3^2} & \cdots & \frac{\partial^2 f}{\partial w_3 \partial w_m} \\ & & & & \vdots \\ & & & & \frac{\partial^2 f}{\partial w_m^2} \end{pmatrix}$$

4.1.3 Minimum and convexity conditions

Let's try now to characterize the minima:

> **Necessary first-order minimum condition**:
>
> $$\nabla E(\boldsymbol{w}^*) = 0$$
>
> This condition characterizes all points which are local minima, but also local maxima or saddle points (points which are minima along one direction and maxima along another direction).

If the function is a **convex function** this condition is also **sufficient**.

We can have though a cost function that is **locally convex**, that is, we have convexity only in a neighbourhood of $\boldsymbol{w}^*$ (intermediate situation):

> $\Rightarrow$ the first-order condition is then a **necessary and sufficient condition of local minimum**. Other local minima belong to different neighborhoods ("basins").

As for minima, we have different conditions of convexity:

- $f(\boldsymbol{w})$ is convex if $H_f(\boldsymbol{w})$ is **positive semidefinite** for all $\boldsymbol{w}$
- $f(\boldsymbol{w})$ is locally convex around a point $\boldsymbol{w}_0$ if $H_f(\boldsymbol{w})$ is positive semidefinite in a neighbourhood of $\boldsymbol{w}_0$

→ A matrix A is positive semidefinite (sometimes written $A \geq 0$) if

$$\forall \boldsymbol{v} \in \mathbb{R}^m \qquad \boldsymbol{v}' A \boldsymbol{v} \geq 0$$

Hessian generalizes the second derivative and "positive semidefinite" generalizes "non-negative"

Hence, to **check** whether a given point $\boldsymbol{w}$ is a **minimum**, we have:

- Necessary conditions of extremum: $\nabla f(\boldsymbol{w}) = 0$
- Necessary and sufficient condition of convexity: $H_f(\boldsymbol{w}) \geq 0$
- Sufficient conditions of minimum: $\nabla f(\boldsymbol{w}) = 0 \wedge H_f(\boldsymbol{w}) \geq 0$

4.1.4 Taylor polynomials

Before we start to see the various optimization algorithms, we need to recall a last concept: the Taylor polynomial. The Taylor polynomial of degree 2 for a scalar function $f(w)$ centered around w_0:

$$f(w) \approx f(w_0) + f'(w)\mid_{w=w_0}(w - w_0) + \frac{1}{2}f''(w)\mid_{w=w_0}(w - w_0)^2$$

The equivalent formula for a scalar field ($\boldsymbol{w} \in \mathbb{R}^m$):

$$f(\mathbf{w}) \approx f(\mathbf{w}_0) + \nabla f(\mathbf{w})\mid_{\mathbf{w}=\mathbf{w}_0}(\mathbf{w} - \mathbf{w}_0) + \frac{1}{2}(\mathbf{w} - \mathbf{w}_0)^T H\mid_{\mathbf{w}=\mathbf{w}_0}(\mathbf{w} - \mathbf{w}_0)$$

4.2 Optimization algorithms

We have different types of optimization algorithms regarding the case[1]:

- CASE 0: we only know $f(\boldsymbol{w}) \rightarrow$ computed values of the objective
- CASE 1: we know $f(\boldsymbol{w})$ and $\nabla_{\boldsymbol{w}} f \rightarrow$ computed values of the gradient
- CASE 2: we know $f(\boldsymbol{w}), \nabla_{\boldsymbol{w}} f$ and $H_f(\boldsymbol{w}) \rightarrow$ computed values of the Hessian

4.2.1 Case 1 - Descent techniques

In Case 1 we know $f(\boldsymbol{w})$ and $\nabla_{\boldsymbol{w}} f$ and **at each search point we approximate $f(\boldsymbol{w})$ with its <u>first-order</u> Taylor expansion**. We can use two techniques:

- finding zeros of the gradient by solving equations in closed form (usually hard)
- gradient descent

Regarding descent techniques, we iteratively descend toward the minimum

$$\mathbf{w}(\tau + 1) = \mathbf{w}(\tau) + \Delta\mathbf{w}(\tau)$$

by taking steps in the direction of the reverse gradient, $-\nabla f(\boldsymbol{w})$:

$$\Delta\mathbf{w}(\tau) = -\eta f \nabla(\mathbf{w}(\tau))$$

The **algorithm** is the following one:

1. Initialize: set $l = 0$ and select $\boldsymbol{w}(l = 0) = \boldsymbol{w}_0$
2. Compute the direction:
$$\nabla\boldsymbol{w}(l) \leftarrow \frac{-\nabla f(\boldsymbol{w}(l))}{||\nabla f(\boldsymbol{w}(l))||}$$
3. Compute (by **line search**) the appropriate **step size** η
4. Scale $\nabla\boldsymbol{w}(l) \leftarrow \eta\nabla\boldsymbol{w}(l)$
5. Perform step $\boldsymbol{w}(l+1) \leftarrow \boldsymbol{w}(l) + \nabla\boldsymbol{w}(l)$
6. Compute **convergence test**. If necessary, iterate from step 2.

How do we perform **line search**?

[1] cfr. Yurii Nesterov's *Lectures on Convex Optimization* regarding first and second order oracles.

- We identify the direction of decrease by the versor (vector with unit length)

$$\boldsymbol{z} = \frac{-\nabla E(\boldsymbol{w}_\tau)}{||\nabla r E(\boldsymbol{w}_\tau)||}$$

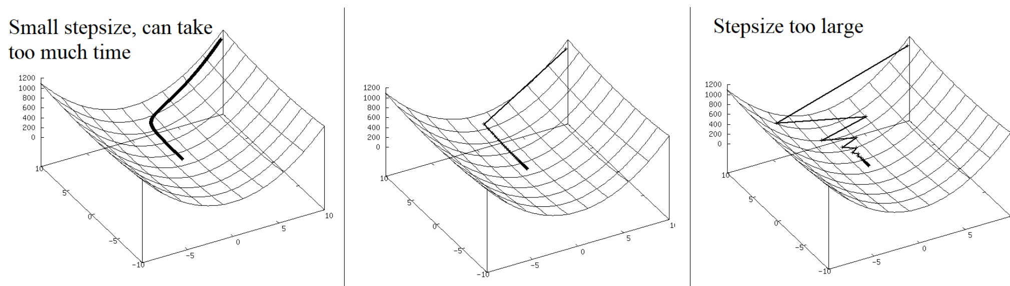
- We perform a minimization of the function $y(t) = f(\boldsymbol{z}t)$, which is a function of one variable

$$\eta = \min_t y(t)$$

⇒ This may not be practical or possible, for instance, in distributed implementations. So in these cases the value of η must be **guessed**.

⇒ There are also indirect methods which modulate η adaptively, changing it according to the variations of the cost function.

The step size η changes the behavior of the descent



The main **pro** of gradient descent is that is a simple techniques.

The **cons** are its unnecessarily slow convergence (always directed exactly as the negative gradient) and the fact that sometimes "is not going the good direction" (→ methods such as conjugate gradient correct this aspect)

4.2.2 Case 2 - Newton-Raphson method

In Case 2 we know $f(\boldsymbol{w}), \nabla_{\boldsymbol{w}} f$ and $H_f(\boldsymbol{w})$ and **at each search point we approximate $f(\boldsymbol{w})$ with its second-order Taylor expansion**. More sophisticated techniques are possible:

- (approximate) line search to find the optimal step size
- checking the necessary and sufficient condition of minimum
- Newton-type algorithms (faster steps)

  (...)

Regarding Newton-type algorithms, to find a zero of a generic function $g(x)$ they iterate as follows:

1. Start at x_0
2. At each step τ compute the update as

$$x_{\tau+1} = x_\tau - \frac{g(x_\tau)}{g'(x_\tau)}$$

   The update finds exactly the zero of the **1st order Taylor approximation of g in x_τ**... but the Taylor approximation is not g, so we repeat!

The necessary 1st order condition of minimum is that that minimum of f has to be a zero for its derivative f'. Hence, **to find a minimum** of a function $f(x)$, set

$$g(x) = f'(x) = \frac{\mathrm{d}f(x)}{\mathrm{d}x}$$

and then apply the Newton-Raphson method:

1. Start at x_0
2. At each step τ compute the update as

$$x_{\tau+1} = x_\tau - \frac{f'(x_\tau)}{f''(x_\tau)}$$

For **multidimensional** functions $f(\boldsymbol{w}) : \mathbb{R}^m \to \mathbb{R}$ we have:

$$\mathbf{w}_{\tau+1} = \mathbf{w}_\tau - \left[H_f(\mathbf{w}_\tau)\right]^{-1} \nabla f(\mathbf{w}_\tau)$$

The **pros** of such methods is that is simple but much faster than gradient descent.

The **cons** are that we need to compute the Hessian ($O(m^2)$ space complexity) and to invert it ($O(n^q)$ time complexity, with $2 < q \leq 3$ depending on the algorithm).

4.2.3 Case "1.5" - Hybrid method

We know $f(\mathbf{w})$ $\nabla_\mathbf{w} f$ and also some approximation to, or information about, $H_\mathbf{w} f$

Typical approach:

At each search point we approximate $f(\mathbf{w})$ with its first-order (not second-order) **Taylor expansion**

but we can **choose good directions and compute quicker step sizes** without having to store a full Hessian matrix at each step

- "Momentum"-based methods
- Adaptive step size methods (many)
- Conjugate gradient methods
- ADAM - "Adaptive momentum" algorithm

NOTE: How to spot a "Case 1.5" method?

In general, if you are using information from the current AND the previous iterations to compute the next step, then you are using second-order information.

On the other hand, if you are not storing a full Hessian matrix, then you are not in case 2.

You are in case 1.5 if you use second-order information but don't use *quadratic-complexity storage* (a matrix). I.e., if you just store some scalars or vectors in addition to $\nabla_\mathbf{w} f$.

4.2.4 Case 0 - Direct search

We only know $f(\mathbf{w}) \longrightarrow$ **NO TAYLOR EXPANSION AVAILABLE!**

Only **direct search** techniques are possible:
find promising points by using (meta)heuristics

- genetic algorithms
- particle swarm optimization
- ant colony optimization
- ...

or by random search

PRO: simple, and guaranteed to find the GLOBAL extrema
CON: ...only if infinite time is available!

Chapter 5

Statistical learning

We saw in Chapter 2 that, in order to evaluate a decision-maker, we can use the following formula for quality evaluation of continuous decision set:

$$R = \int_{\mathcal{X}} R\left(y(\boldsymbol{x})|\boldsymbol{x}\right) p(\boldsymbol{x}) d\boldsymbol{x} \qquad \text{(Expected risk)}$$

In this case though we are in Learning **Scenario 3** (only data available) – as for linear regression (Chapter 3) – so we are **evaluating** directly **from data**, because in real life we have **no access to probabilities**:

$$\hat{R} = \frac{1}{n}\sum_{i=1}^{n} R\left(y(\boldsymbol{x}_i)|\boldsymbol{x}_i\right) \qquad \boldsymbol{x}_i = \text{a data set or } \textbf{sample} \qquad \text{(Empirical risk)}$$

This empirical risk gives us an average over the available data, a Monte Carlo approximation of the expected risk R:

$$\begin{array}{ccc} \text{Probability} & \to & \text{Statistics} \\ R & \to & \hat{R} \end{array}$$

What is the problem with $\hat{R}$? The sample is randomly sampled, then every sample will be different:

$$\forall \text{ sample } \exists \text{ different } \hat{R} \implies \hat{R} \text{ is a random quantity itself}$$

Let's recall the difference between a random variable and its realization:

- A random (or stochastic) variable is a variable whose possible values are numerical outcomes of a random phenomenon. It does not have a value, but is described by a probability distribution function.

→ Probability represents the **true source** of the data

- A realization of a random variable is a numeric value that represents the outcome of one specific experiment or observation.

→ Realizations represent the finite data that **we observe** experimentally.

In order to deal with realizations (Scenario 3) and not directly with probabilities (Scenario 2) we will need empirical estimations → we will need statistics.

5.1 Statistics & parameter estimation

5.1.1 Models

Originally, "statistics" was the techniques for collecting data with the aim of governing a state. Now statistics is much more:

- it's the science of using empirical data to create models based on probability

- it's the science of induction of concepts from experiments.
- it's the science of discovery of the Laws of Nature / the science of modeling Nature.
- it's the science of scientific inquiry.

A **model** is an approximation to reality that retains only those characteristics that are useful to a certain purpose (a mathematical model, a physical model, a geometrical model, a probabilistic model, a "Rules of nature", a rendering, an identikit ...)

About models:

- Plato: Myth of the cave. We perceive real models only through partial experience.
- Aristotle, St. Thomas of Aquino: "*Nihil est in intellectu quod non prius in sensu*". Different perspective: no real models, only what we perceive contributes to our view of reality.
- Empiricists up to Popper: Experience is finite, therefore error=0 is impossible. Everything that is scientific must also be falsifiable, i.e., it must be accompanied by an evaluation of its limitations (for instance, a probability of error).

Statistics is about collecting experimental observations, then using them for estimating a model for the observed phenomenon. This model is a **probabilistic model**, and is then evaluated w.r.t. its **probability of error**.

Machine learning is about collecting experiences (→ a training set), then using them to learn some task (e.g., a classification rule) → a large part of **ML** is the same as **statistic inference** (though the terms are sometimes different). There are some assumptions related to the training set:

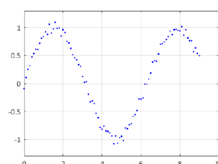
- it was obtained by random sampling.
- the observed phenomenon has a fixed statistical behaviour,
- each experiment is unrelated to any other.

We have now to introduce a fundamental property of the samples of the training set:

- If the individual patterns are **independent**:
  - The probability of any pattern does not depend on the probability of any other patterns
  - In particular, even if we sample sequentially, what we get at time t does not depend on what we've got at times $1, ..., t-1$
- and the individual patterns are also **identically distributed**:
  - Patterns come from a single data distribution: not more than one, not one that changes in time

⇒ we speak of independent, identically distributed sample (**i.i.d. sample**)

Not all problems produce i.i.d. data. For instance, sequential data are not independent (i.i.d in fact means that ordering does not matter and expectations and means make sense):

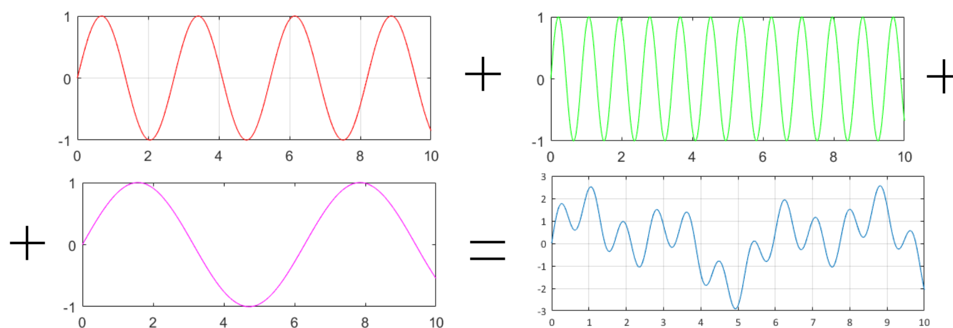


Time series

`Machine learn...`

Symbolic sequence

Why using probabilities? The source of my data is usually impredictable, due to random behaviour (**noise**) **or unknown behaviour**. Moreover, most of the times there are **unobservable effects** and variables → we model them as "randomness", and it is impossible to distinguish between randomness and unobservability. Consider this example:



5.1.2 Model parameter estimation

The two goals of statistics are to (1) **compute** something (a "statistic") based on the data and (2) **evaluate** the probability that the above computation is wrong (confidence)
$\rightarrow$ high confidence = good generalization.

What is parameter estimation? In models there are one (or more) quantities that we want to measure but **not directly accessible**. This could happen for two possible reasons:

- It is not a physically measurable quantity
- The measurements are affected by random variations or noise

Parameter estimation requires the following **"ingredients"**:

- The *data*: a training set
- The *learner*: a model depending on a set of parameters, to adapt (fit) it to the actual data
- The *loss*: a model for the uncertainty of the measurements (a noise model)

The **estimation process** takes two phases:

- Select an estimator suited to the problem
- Select the value of parameters that optimizes the estimator

Remember: The **estimator** is a statistic = a **function of the training set**.

After the estimation process, ha to be performed an **error evaluation**, that is, estimating the probability that an estimate is wrong. In the general case, a parameter w is estimated by a statistic $\hat{w}$. There is a random error in this estimate:

$$|w - \hat{w}|$$

In Machine Learning, **estimating generalization** means evaluating this error.

Then, the complete process is the following one:

1. Learning is statistical estimation
2. The training set is the sample
3. The result of learning is a classifier with some measured performance
4. Performance of learning depends on the sample: it is a statistic, a random variable itself
5. It is necessary to evaluate the probability of correctness of this statistic
6. Good generalization = correct statistic

5.1.3 Learning process

The learning process use the data to choose the learner that minimizes the loss with maximum confidence:

		Type of output	
		Quantitative	Nominal
Super-vised	Yes	Regression	Classification
	No	Low-dimensional mapping	Clustering

In **supervised** problems we can write **proper loss** functions to evaluate the discrepancy between our output and the target. In **unsupervised** problems we use **other quality indicators**:

- Examples of **loss functions for regression** (output and target are quantitative, i.e., real numbers):
  - Squared loss:
    $$\lambda(y,t) = (t-y)^2$$
    Its expectation is the mean squared error
  - Absolute loss:
    $$\lambda(y,t) = |t-y|$$
    Used in "median regression"
  - Huber loss:
    $$\lambda(y,t) = \begin{cases} (t-y)^2 & |t-y| < \epsilon \\ (2\cdot|t-y|-\epsilon)\epsilon & else \end{cases}$$
- Examples of **loss functions for classification** (output and target are nominal, but we may use scores in $[-1,+1]$) $\to$ losses may depend on $\mathbf{1}(y \neq t)$ (nominal) or on yt (real-valued scores) which is positive when correct
  - Zero-one loss:
    $$\lambda(y,t) = \mathbf{1}(\neq) = \begin{cases} 0 & t = y \\ 1 & else \end{cases}$$
    Its expectation is the mean error rate
  - Hinge loss:
    $$\lambda(y,t) = \max(0, 1-yt) = \begin{cases} 0 & yt > 1y \\ yt & else \end{cases}$$
    Sensitive to the magnitude of errors. Used in support vector machines.

There is a fundamental problem when dealing with learning. There are in fact two kinds of learning processes:

- **Inductive learning**: Learning from examples or data

  is the task of learning a function for every possible input, given only a finite set of example inputs. This is called **generalization**.
- **Deductive learning**: Learning from hypotheses and rules

While **deduction extracts**, but does not create, new knowledge. **Induction creates** new knowledge, and therefore is **not possible**. But then how does learning work, if it is not possible? Induction (generalization) only works if we have an **inductive bias** = an *a-priori* knowledge or assumption. Examples of inductive bias are the following ones:

Problem	Bias
Classification	Points in the same class are close to each other
Regression	Nearby points don't have very different outputs
Clustering	There are *k* compact clusters in the data
Mapping	The data are organized in a low-dimensional shape within the *d*-dimensional data space

Bias is not a bad thing, in fact it has been accumulated through successful evolution, and it is the basis of learning and even simple perception in living beings, including humans. A universal bias is Ockham's razor: "*Frustra fit per plura quod potest fieri per pauciora*"

But then, how to measure complexity?

5.2 Parametric methods

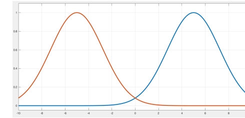
For these cases we have one type of bias, that is, **the parametric hypothesis**:

> The data are generated by a source whose probability density is *known* up to a *finite number* of parameters, and only these parameters (e.g., class center, class variance) are unknown.

For example, a binary classification problem belongs to these methods:

$$P(\omega_1) = P(\omega_2) = 0.5$$
$$P(\mathbf{x}|\omega_1) = \frac{1}{\sqrt{(2\pi)^d|\Sigma|}} \exp\left(-\tfrac{1}{2}(\mathbf{x}-\mathbf{m}_1)^T\Sigma^{-1}(\mathbf{x}-\mathbf{m}_1)\right)$$
$$P(\mathbf{x}|\omega_2) = \frac{1}{\sqrt{(2\pi)^d|\Sigma|}} \exp\left(-\tfrac{1}{2}(\mathbf{x}-\mathbf{m}_2)^T\Sigma^{-1}(\mathbf{x}-\mathbf{m}_2)\right)$$

$\mathbf{m}_1$, $\mathbf{m}_2$, Σ unknown



5.2.1 Maximum Likelihood parameter estimation

Consider a sample $X = \{x_1, ..., x_n\}$

The parametric assumption states $\underline{x}_l \sim p(\boldsymbol{x}|\theta)$ (read symbol "$\sim$" as: "distributed according to")

$p(\boldsymbol{x}|\theta)$ is a probability density function (pdf) that depends on a set of parameters θ

Suppose the data are fixed: what is the probability (density) of observing a given θ? We define the **Likelihood** of a given set of parameters:

$$\mathcal{L}(\theta, X) = p(X|\theta) \overset{\text{if i.i.d. hypothesis holds}}{=\!=\!\longrightarrow\!=} \prod_{l=1}^{n} p(\boldsymbol{x}_l|\theta)$$

The **Maximum Likelihood criterion** states that we have to select the θ that maximizes $\mathcal{L}(\theta, X)$. But what *really* is the likelihood?

$$\mathcal{L}(\theta, X) \overset{\text{should be}}{=} p(\theta|X) \overset{\text{Bayes}}{=} \frac{p(X|\theta)\cancel{p(\theta)}}{\cancel{p(X)}} = p(X|\theta) = \mathcal{L}(\theta, X) = \prod_{l=1}^{n} p(\boldsymbol{x}_l|\theta)$$

We performed that "simplification" because if all parameter values are uniformly probable, $p(\theta)$ is constant (does not depend on θ), and the denominator does not depend on θ at all $\rightarrow$ this is maximized when $\mathcal{L}(\theta, X) = p(X|\theta)$ is maximum.

Since for maximization, monotonically increasing transformations don't change the solution, we can define the **log-likelihood**:

$$L(\theta, X) = \ln \mathcal{L}(\theta, X) = \ln \prod_{l=1}^{n} p(\boldsymbol{x}_l|\theta) = \sum_{l=1}^{n} \ln p(\boldsymbol{x}_l|\theta)$$

Since for maximization, neither a constant change the solution, we can define the **average log-likelihood**:

$$\hat{L}(\theta, X) = \frac{1}{n} L(\theta, X) = \frac{1}{n} \sum_{l=1}^{n} \ln p(\boldsymbol{x}_l|\theta)$$

Remark: If the $p(\boldsymbol{x}_l|\theta)$ are of type $a\ e^b$, the logarithm will cancel out the exponential and only leave $b + \ln(a)$.

Estimation of univariate Gaussian See the following example:

$$p(x|\mu,\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right) \quad \text{one-dimensional (= univariate) Gaussian density}$$

Parameters are: $\theta = [\mu,\sigma]$

Suppose we have a training set $X = \{x_1,\dots,x_n\}$. Then:

$$L(\theta,X) = \ln\prod_{l=1}^{n}\frac{1}{\sqrt{2\pi}\sigma}\exp\left(-\frac{(x_l-\mu)^2}{2\sigma^2}\right) = -\frac{n}{2}\ln\sqrt{2\pi} - n\ln\sigma - \sum_{l=1}^{n}\frac{(x_l-\mu)^2}{2\sigma^2}$$

Maximization $\Rightarrow$ derivatives must vanish.

Let's find the values of μ and σ for which the derivatives of the log-likelihood are null.

Differentiate $L(\theta,X)$ w.r.t. μ:

$$\frac{\partial}{\partial\mu}\left(-\frac{n}{2}\ln\sqrt{2\pi} - n\ln\sigma - \sum_{l=1}^{n}\frac{(x_l-\mu)^2}{2\sigma^2}\right) = -2\sum_{l=1}^{n}\frac{x_l-\mu}{2\sigma^2} = 0 \iff \sum_{l=1}^{n}x_l - \sum_{l=1}^{n}\mu = \sum_{l=1}^{n}x_l - n\mu = 0 \iff \mu = \frac{1}{n}\sum_{l=1}^{n}x_l$$

Differentiate $L(\theta,X)$ w.r.t. σ:

$$\frac{\partial}{\partial\sigma}\left(-\frac{n}{2}\ln\sqrt{2\pi} - n\ln\sigma - \frac{1}{\sigma^2}\sum_{l=1}^{n}\frac{(x_l-\mu)^2}{2}\right) = -\frac{n}{\sigma} + \frac{1}{\sigma^3}\sum_{l=1}^{n}(x_l-\mu)^2 = 0 \iff n\sigma^2 - \sum_{l=1}^{n}(x_l-\mu)^2 = 0 \iff \sigma^2 = \frac{1}{n}\sum_{l=1}^{n}(x_l-\mu)^2$$

(since $\frac{1}{\sigma} > 0$)

So, the maximum likelihood estimate of the parameters of a univariate Gaussian density is:

$$\hat{\mu} = \frac{1}{n}\sum_{l=1}^{n}x_l \qquad \hat{\sigma} = \frac{1}{n}\sum_{l=1}^{n}(x_l-\hat{\mu})^2$$

5.2.2 ML and MAP

By Bayes' theorem, $p(\theta|X) = \frac{p(X|\theta)p(\theta)}{p(X)}$ So we can say that $\arg\max_\theta p(\theta|X) = \arg\max_\theta p(X|\theta)$

only when $p(\theta)$ is (assumed to be) a constant.

If $p(\theta)$ is constant: **Maximum likelihood (ML) criterion**

$$\hat{\theta} = \arg\max_\theta p(X|\theta) = \arg\max_\theta p(\theta|X) = \arg\max_\theta \ln p(\theta|X) = \dots$$

(any monotonically increasing transformation)

If $p(\theta)$ is not constant but known: **Maximum a-posteriori probability (MAP) criterion**

$$\hat{\theta} = \arg\max_\theta p(\theta|X) = \arg\max_\theta \frac{p(X|\theta)p(\theta)}{p(X)} = \arg\max_\theta p(X|\theta)p(\theta)$$

(Bayes' formula)

5.3 Non-parametric methods

Non-parametric statistics makes **no assumptions about probability distributions**. In fact, w.r.t the same problem:

- parametric statistics: assuming that two random variables have a Gaussian probability density, decide whether they have the same mean and variance
- non-parametric statistics: *decide whether two random variables have the same probability distribution*

Non-parametric models have a **complexity** (model size, number of parameters) that is **not pre-defined**, but depends on the data. In fact, w.r.t the same problem:

- parametric model: assuming that data from two classes have a Gaussian probability density with the same variance, find the separating hyperplane that ensures the best expected risk

- non-parametric model: *find a classification rule that, given a training set from two classes, ensures the best expected risk*

Two **basic non-parametric classifiers** that we will see in this Chapter are the nearest-neighbour classifiers and the decision trees. They don't explicitly implement a model with parameters, but directly **build a discrimination rule from data**.

5.3.1 Nearest-neighbour classifiers

Cover TM, **Hart** PE (1967). "*Nearest neighbor pattern classification*".
IEEE Transactions on Information Theory. 13 (1):21-27.

Consider a given dataset

$$\Delta = \boxed{\boldsymbol{x}_1 \quad \cdots \quad \boldsymbol{x}_l \quad \cdots \quad \boldsymbol{x}_f \quad \boldsymbol{t}} = \boxed{\begin{matrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_l^{(1)} & \cdots & x_f^{(1)} & t_1 \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_l^{(2)} & \cdots & x_f^{(2)} & t_2 \\ \vdots & & \ddots & \cdots & \cdots & & \vdots \\ x_1^{(o)} & x_2^{(o)} & \cdots & x_l^{(0)} & \cdots & x_f^{(o)} & t_o \end{matrix}}$$

where f is the number of features for each observation and o is the number of observations. For example, with $f = 2$ and $o = 10$:

Age and sex ($\boldsymbol{x}$)	Likes ice cream $t = f(\boldsymbol{x})$
(22,M)	yes
(23,M)	yes
(21,F)	yes
(18,M)	yes
(19,F)	yes
(25,F)	no
(27,M)	no
(29,F)	no
(31,F)	no
(45,M)	no

In our example, y can assume only two values, "yes" and "no". If we describe the alphabet of the possible attributes of the output as $A = \{''no'', ``yes''\}$, we'll have $A[1] = ``yes''$ and $A[0] = ``no''$.

Consider also and a **"query" point** $\bar{\boldsymbol{x}}$ that is given (e.g. (22,F)). For that query point, the classifier will find the nearest neighbour minimizing the distance from each sample of the dataset $\boldsymbol{x}_l$ (e.g. we want to understand if a 22-old female likes ice cream):

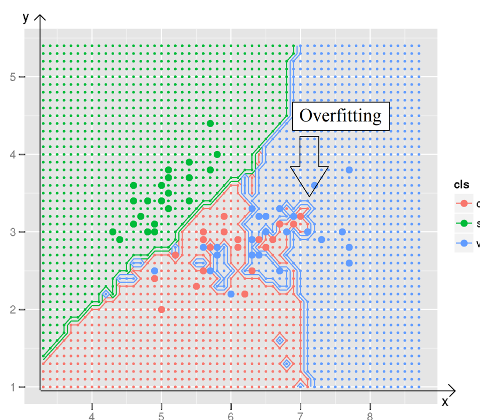
$$n = \arg\min ||\boldsymbol{x}_l - \bar{\boldsymbol{x}}||$$

This n is the index w.r.t. the possible values of the output (A). Then, the **output of our classifier** will be

$$y = A[n]$$

Properties of NN:

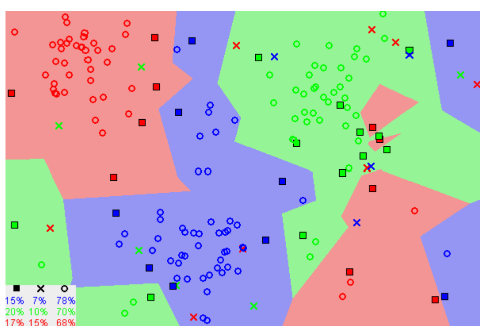
- '$O(1)$ "learning" complexity (just store the training set!)
- **but** $O(nd)$ classification complexity
- (sort of) works even for regression, with $t \in \mathbb{R}$
- Theoretical guarantee: For $n \to \infty$, error rate $\leq 2 \cdot$ Bayes error
- **but** may "overfit", i.e., decision regions may have jagged borders



5.3.2 k-Nearest-Neighbour classifier

The k-nearest neighbor (kNN) algorithm is a nearest neighbor classifier, that can though (as already seen for all these classifiers) implement supervised machine learning algorithm and solve **both classification and regression problems**.

The kNN algorithm assumes that similar things exist in close proximity. In other words, similar things are near to each other (e.g. "Birds of a feather flock together"):



Notice in the image above that most of the time, similar data points are close to each other. The kNN algorithm hinges on this assumption being true enough for the algorithm to be useful. kNN captures the idea of similarity (sometimes called distance, proximity, or closeness) calculating the distance between points on a graph.

There are other ways of calculating distance, and one way might be preferable depending on the problem we are solving. However, the straight-line distance (also called the Euclidean distance) is a popular and familiar choice.

kNN Algorithm:

1. Load the data (a training set $X = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_l, ..., \boldsymbol{x}_m\}$ and a "query" point $\bar{\boldsymbol{x}}$ that are given)
2. Initialize k to your chosen number of neighbors
3. For each example $\boldsymbol{x}_l$ in the data
   (a) Calculate the distance between the query point and the current $||\boldsymbol{x}_l - \bar{\boldsymbol{x}}||$
   (b) Add the distance and the index of the example to an ordered collection
4. Sort the ordered collection of distances and indices from smallest to largest (in ascending order) by the distances
5. Pick the first k entries from the sorted collection
$$\{n_1, ..., n_k\} = \text{top-}k||\boldsymbol{x}_l - \bar{\boldsymbol{x}}||$$
6. Get the labels of the selected k entries

7. If regression, return the mean of the k labels

$$y = \text{mean}\{t_{n_1}, ..., t_{n_k}\},$$

8. If classification, return the mode of the k labels (like a majority voting)

$$y = \text{mode}\{t_{n_1}, ..., t_{n_k}\},$$

Here we have a Pyhton implementation of it:

```
from collections import Counter
import math

def knn(data, query, k, distance_fn, choice_fn):
  neighbor_distances_and_indices = []

  # 3. For each example in the data
  for index, example in enumerate(data):
  # 3.1 Calculate the distance between the query example and the current
  # example from the data.
  distance = distance_fn(example[:-1], query)

  # 3.2 Add the distance and the index of the example to an ordered collection
  neighbor_distances_and_indices.append((distance, index))

  # 4. Sort the ordered collection of distances and indices from
  # smallest to largest (in ascending order) by the distances
  sorted_neighbor_distances_and_indices = sorted(neighbor_distances_and_indices)

  # 5. Pick the first K entries from the sorted collection
  k_nearest_distances_and_indices = sorted_neighbor_distances_and_indices[:k]

  # 6. Get the labels of the selected K entries
  k_nearest_labels = [data[i][1] for distance, i in k_nearest_distances_and_indices]

  # 7. If regression (choice_fn = mean), return the average of the K labels
  # 8. If classification (choice_fn = mode), return the mode of the K labels
  return k_nearest_distances_and_indices , choice_fn(k_nearest_labels)

def mean(labels):
  return sum(labels) / len(labels)

def mode(labels):
  return Counter(labels).most_common(1)[0][0]

def euclidean_distance(point1, point2):
  sum_squared_distance = 0
  for i in range(len(point1)):
  sum_squared_distance += math.pow(point1[i] - point2[i], 2)
  return math.sqrt(sum_squared_distance)

def main():
  # Regression Data
  # Column 0: height (inches) | Column 1: weight (pounds)
  reg_data = [[65.75, 112.99],[71.52, 136.49],[69.40, 153.03],[68.22, 142.34],
              [67.79, 144.30],[68.70, 123.30],[69.80, 141.49],[70.01, 136.46],
              [67.90, 112.37],[66.49, 127.45],]

  # Question:
  # Given the data we have, what's the best-guess at someone's weight if they are 60
      ↪ inches tall?
  reg_query = [60]
  reg_k_nearest_neighbors, reg_prediction = knn(reg_data, reg_query, k=3, distance_fn=
      ↪ euclidean_distance, choice_fn=mean)

  # Classification Data
  # Column 0: age | Column 1: likes pineapple
  clf_data = [ [22, 1], [23, 1], [21, 1], [18, 1],
               [19, 1], [25, 0], [27, 0], [29, 0],
               [31, 0], [45, 0],]
  # Question:
  # Given the data we have, does a 33 year old like pineapples on their pizza?
  clf_query = [33]
  clf_k_nearest_neighbors, clf_prediction = knn(clf_data, clf_query, k=3, distance_fn=
      ↪ euclidean_distance, choice_fn=mode)

if __name__ == '__main__':
  main()
```

Choosing the right value for k To select the k that's right for the data, we have to run the kNN algorithm several times with different values of k and choose the k that **reduces the number of errors** we encounter while maintaining the algorithm's ability to accurately make predictions when it's given data it hasn't seen before.

Here are some things to keep in mind:

- As we **decrease** the value of k to 1, our predictions become **less stable**. Just think for a minute, imagine $k = 1$ and we have a query point surrounded by several reds and one green (let's say the top left corner of the colored plot above), but the green is the single nearest neighbor. Reasonably, we would think the query point is most likely red, but because $k = 1$, kNN incorrectly predicts that the query point is green.
- Inversely, as we **increase** the value of k, our predictions become **more stable** due to **majority voting/averaging**, and thus, more likely to make more accurate predictions (up to a certain point).

  Eventually, we begin to witness an **increasing number of errors**. It is at this point we know we have pushed the value of K **too far**.
- In cases where we are taking a majority vote (e.g. picking the mode in a classification problem) among labels, we usually make k **an odd number** to have a tiebreaker.

Advantages:

- The algorithm is simple and easy to implement.
- There's no need to build a model, tune several parameters, or make additional assumptions.
- The algorithm is versatile. It can be used for classification, regression, and search.

Disadvantages:

- The algorithm gets significantly slower as the number of examples and/or predictors/independent variables increase.

kNN in practice kNN's main disadvantage of becoming significantly **slower as the volume of data increases** makes it an impractical choice in environments where predictions need to be made rapidly. Moreover, there are faster algorithms that can produce more accurate classification and regression results.

However, provided you have sufficient computing resources to speedily handle the data you are using to make predictions, kNN can still be useful in solving problems that have solutions that depend on identifying similar objects. An example of this is using the kNN algorithm in **recommender systems**, an application of kNN-search.

Recommender Systems At scale, this would look like recommending products on Amazon, articles on Medium, movies on Netflix, or videos on YouTube. Although, we can be certain they all use more efficient means of making recommendations due to the enormous volume of data they process. However, we could replicate one of these recommender systems on a smaller scale using what we have learned here in this article. Let us build the core of a movies recommender system.

Question we are trying to answer: Given our movies data set, what are the 5 most similar movies to a movie query?

Gather movies data: we could use some movies data from the UCI Machine Learning Repository, IMDb's data set, or painstakingly create our own:

Movie	ID	Movie Name	IMDB	Rating	Drama	Thriller	Comedy	History
				...				

When we run the algorithm, we see that, for a query movie, it recommends k films.

Variants

- Weighted nearest neighbours (weight proportional to distance)
- Condensed NN (choice of $c < n$ observations rather than all n data), $O(cd)$
- Approximated NN (distance is approximated with a computation time) $< O(d)$

5.3.3 Decision trees

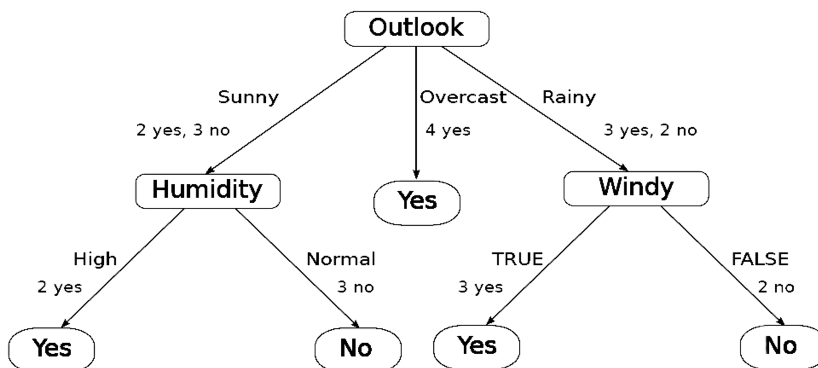
Decision trees are a non-parametric classifier for **categorical data**. There exist many variants (e.g. ID3, C4.5, CART). The main idea behind this classifier is is following one:

1. Select one feature **(attribute, variable)**
2. Partition the training set according to its possible values (**levels**)
3. For each of the parts of the training set so obtained (one per level)
   (a) if it contains data from one class only, then decide for that class
   (b) else if it contains data from more than one class and **all features have been used** then decide for the majority class
   (c) else apply again the same procedure using a different feature

Consider the following example starting from this dataset:

Outlook	Temperature	Humidity	Windy	Play
overcast	hot	high	FALSE	yes
overcast	cool	normal	TRUE	yes
overcast	mild	high	TRUE	yes
overcast	hot	normal	FALSE	yes
rainy	mild	high	FALSE	yes
rainy	cool	normal	FALSE	yes
rainy	cool	normal	TRUE	no
rainy	mild	normal	FALSE	yes
rainy	mild	high	TRUE	no
sunny	hot	high	FALSE	no
sunny	hot	high	TRUE	no
sunny	mild	high	FALSE	no
sunny	cool	normal	FALSE	yes
sunny	mild	normal	TRUE	yes

The tree we can obtain is the following:



In order to use **induction (learning) of decision trees** (→ the **ID3** decision trees) we use have to use a measure of "good class separation". Let's introduce the following definitions:

Information entropy $H(X) = \sum_{t \in \text{classes}} -p(t) \ln p(t)$

- X a data set
- t the targets in X
- $p(t)$ the proportion of target t in data set X

Information gain $IG(X, \{S1, \ldots, Sn\}) = H(X) - \sum_{Si} \frac{|Si|}{|S|} H(Si)$

- $\{S1, \ldots, Sn\}$ is the partition obtained by choosing a given feature
- Si is the subset of data that have the ith value (level) for that feature
- IG is the amount of information obtained by using that feature

Hence, in order to ensure **"good class separation"** we will have to select the feature that gives the **largest information gain**.

The **pros** of such a classifier is that is easily interpretable (only one feature at a time) and allows a very quick learning process.

The **cons** are that it is suboptimal (a greedy algorithm), it splits on features with many values are more likely, with few data splits may be on irrelevant features (noise) and it is not suitable for quantitative (e.g., real-valued) features.

Some **remedies** to the cons are some variations and extensions (like the C4.5, CART versions):

Bias toward many-valued features

- Normalize IG by dividing it by the **intrinsic information**:

$$-\sum_i \frac{|Si|}{|S|} \ln \frac{|Si|}{|S|}$$

Cardinality of the set Si → number of elements

which is higher for features with many levels

- Alternatives to IG as criteria for splitting (e.g., Gini index, minimum variance)

Irrelevant splits

- **Prune** the tree according to some criterion

Real-valued features

- Divide value ranges into sub-ranges (quantization)
- ...then use the sub-ranges as categorical features

5.3.4 Random forests

A random forest is a set of random trees, trained on randomly **resampled training sets**. It is a type of **ensemble classifier**. The method used is the following one:

1. Receive one training set X of cardinality n patterns
2. Create b new training sets of size n by **randomly sampling** from X with replacement
3. Train one **random tree** on each of the b new training sets
4. **Inference**: Given one query pattern:
   (a) Classify the pattern with each tree
   (b) Make the final decision by **majority voting** among the b individual decisions

How to perform random resampling? One way is random resampling with replacement (**bootstrap**):

> Given one training set X of cardinality n, a **bootstrap sample** is another traning set $X(i)$ obtained by randomly **sampling with replacement** n times from X:
>
> $X = [\, A\, B\, C\, D\, E\,]$
>
> $X^{(1)} = [A\, A\, C\, D\, E\,]$
>
> $X^{(2)} = [\, B\, C\, C\, D\, E\,]$
>
> $X^{(3)} = [\, A\, A\, B\, D\, D\,]$
>
> $X^{(4)} = [\, A\, B\, C\, E\, E\,]$
>
> $X^{(5)} = [\, A\, A\, A\, C\, C\,]$
>
> There are $\begin{bmatrix} 2n-1 \\ n-1 \end{bmatrix}$ possible bootstrap samples.
>
> (This can be approximated as $(n\pi)^{\frac{1}{2}} 2^{2n-1}$, so we know it is $\propto 2^{2n}$)

We can also proceed **aggregating bootstrap samples**. In decision forests in fact, we aggregate b trees trained on b different bootstrap samples. Aggregating learners that have been fitted on bootstrap samples is called **B**ootstrap **agg**regat**ing** (or **bagging**). Bagging reduces variance, the sensitivity to sampling (dependence on the particular training set).

We can as well use **Learning random trees**:

The decision tree induction rule will tend to select always the most discriminating features (it is their job after all!)

So trees will resemble each other and will not be very independent.

Bagged trees should use different features ⇒ **we use random sampling also for features!**

- **Select one feature from a subset of k randomly chosen features**
- Partition the training set according to its possible values (**levels**)
- For each of the parts of the training set so obtained (one per level),

    1. if it contains data from one class only,
    then decide for that class;
    2. else if it contains data from more than one class and all features have been used,
    then decide for the majority class;
    3. else apply again the same procedure using a different feature

Rules of thumb for k: $k \approx \lfloor \sqrt{d} \rfloor$ for classification, $k \approx \lfloor \sqrt{d/3} \rfloor$ for regression.
(but **never trust rules of thumb**...)

kNN Classifier Tuning

Report for the Machine Learning course, EMARO

Davide Lanza
EMARO+ M2
Genoa, Italy
davidel96@hotmail.it

***Abstract*—In this report we will analyze a MATLAB implementation of the k-Nearest Neighbors classifier. After a short introduction, we will define the theoretical framework from which we derived our implementation, focusing on this specific NN classifier. We will then test its accuracy w.r.t. the classic MNIST dataset, analyzing and comparing the different results obtained for different k values, in order tu tune the optimal value for this specific classification task.**

***Index Terms*—Machine Learning, Supervised Learning, Classification, Nearest Neighbors, kNN classifier, MATLAB, MNIST dataset**

I. Introduction

Thanks to the exponential increase of data available on the Internet, the volume of information available for perform machine learning task grows incredibly. Classification tasks are an important component in information extraction and for predictive tasks.

In this report, we focus on a specific technique, suitable for both classification and regression task, but designed mostly for the first case: the nearest-neighbor classifiers. First, we will discuss the theoretical framework from which we derived our MATLAB implementation, then we will test it w.r.t. the MNIST database of handwritten digits[1], which includes a training set of 60000 examples, and a test set of 10000 examples, that is a subset of a larger set available from the National Institute of Standards and Technology (NIST). The digits have been size-normalized and centered in 28x28 gray scale images, and are a standard benchmark for machine learning tasks.

II. Model

In this section, we will present the theoretical framework from which we have implemented our k-Nearest-neighbour classifier (kNN)[2], the common nearest neighbors classifier. The kNN can though (as a NN classifier) implement supervised machine learning algorithm and solve both classification and regression problems.[3]

Every NN classifier assumes that similar things exist in close proximity: that is the starting bias in order to allow learning from data, i.e. most of the time similar data points are close to each other. The kNN exploit this bias with a similarity check between the known elements of the dataset and a queried unknown element. Normally, this similarity is a distance, computed between points on a graph. In this regard, the Euclidean distance is a intuitive and common choice, and is the one we adopted here. More in detail, the kNN Algorithm works as follow:

1) Load the training dataset set

$$\Delta = \boxed{\boldsymbol{x}_1 \ \cdots \ \boldsymbol{x}_l \ \cdots \ \boldsymbol{x}_f \ \boldsymbol{t}} =$$

$$= \begin{array}{|cccccccc|} x_1^{(1)} & x_2^{(1)} & \cdots & x_l^{(1)} & \cdots & x_f^{(1)} & & t_1 \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_l^{(2)} & \cdots & x_f^{(2)} & & t_2 \\ \vdots & & \ddots & \cdots & \cdots & & & \vdots \\ x_1^{(o)} & x_2^{(o)} & \cdots & x_l^{(0)} & \cdots & x_f^{(o)} & & t_o \end{array}$$

composed by o observations $\boldsymbol{x}$, each one composed by f features, and by the corresponding o target values t
2) Load the input "query" point $\bar{\boldsymbol{x}}$
3) Initialize k to the chosen number of neighbors
4) For each example $\boldsymbol{x}_l$ in the data
   a) Compute the euclidean distance between the query point and the current

$$d(\boldsymbol{x}_l, \bar{\boldsymbol{x}}) = ||\boldsymbol{x}_l - \bar{\boldsymbol{x}}||$$

5) Sort the array of distances and indices from the smallest d to largest, in ascending order
6) Pick the first k entries from the sorted array, and from them obtain the respective t values

$$\{t_1, ..., t_k\} = \underset{\{t\}}{\text{top-}k}\,(||\boldsymbol{x}_l - \bar{\boldsymbol{x}}||)$$

7) If it is a classification task, return the mode of the k chosen targets:

$$y = \text{mode}\{t_1, ..., t_k\},$$

8) If it is a regression task, return the mean of the k chosen targets

$$y = \text{mean}\{t_1, ..., t_k\},$$

In our case, given the nature of the task related to the MNIST dataset, we will perform only a classification task, where the kNN learner will have to discriminate between 10 different digits.

	Cipher									
k	1	2	3	4	5	6	7	8	9	0
1	0.99471	0.96124	0.96039	0.96130	0.96412	0.98538	**0.96498**	**0.94455**	0.95837	0.99285
2	0.99559	0.95833	0.95643	0.95213	0.95067	0.98434	0.95914	0.92402	0.95143	0.99285
3	0.99735	**0.96414**	0.96633	**0.96741**	0.96412	0.98538	0.96400	0.94250	**0.96035**	**0.99387**
4	0.99735	0.96027	0.96435	0.96232	0.96860	**0.98643**	0.96303	0.93737	0.95936	0.99081
5	**0.99823**	0.95833	0.96534	0.95926	0.96300	**0.98643**	0.96303	0.94045	0.95540	0.99285
10	0.99647	0.95155	0.96534	0.95621	**0.97197**	0.98538	0.95719	0.94147	0.95341	0.99183
15	0.99647	0.94089	0.96435	0.95010	0.96300	0.98434	0.95330	0.93326	0.95639	0.98979
20	0.99559	0.93023	**0.96930**	0.95010	0.96636	0.98329	0.95233	0.93531	0.95143	0.98979
30	0.99559	0.92441	0.96732	0.94501	0.96188	0.98121	0.94649	0.92813	0.95242	0.98979
40	0.99559	0.91763	0.96237	0.93991	0.95627	0.98121	0.94163	0.92402	0.95044	0.98775
50	0.99559	0.91375	0.96138	0.93686	0.95291	0.98121	0.94066	0.91375	0.95044	0.98775

Table 1: Classification accuracy for single cipher

k	Accuracy
1	0.9691
2	0.9630
3	**0.9709**
4	0.9693
5	0.9686
10	0.9673
15	0.9635
20	0.9626
30	0.9595
40	0.9560
50	0.9538

Table 2: Classification total accuracy

III. Implementation

We have a dataset composed by 7000 observations, each one composed by $f = 28 \times 28 = 784$ features. In order to test it, we split the dataset with a $6 : 1$ train-test ratio, and we studied the behavior of the kNN while performing a classification tasks with different values of $K = [1, 2, 3, 4, 5, 10, 15, 20, 30, 40, 50]$. Every classification can be seen as composed by 10 sub-tasks: distinguish the digit w.r.t. the remaining 9. That is why the results have been reported in confusion matrices[4] (see Appendix) that allow to highlight the accuracy for every sub-task (see also Table 1), together with the general accuracy (see also Table 2).

Regarding the MATLAB implementation, the computation of the 10000 queries of the test dataset Δ_{test} is not computationally easy. With X $= \Delta_{train}$ and x_query$\in \Delta_{test}$, the whole training using the function

```
vecnorm(X - x_query,2,2)
```

in order to compute the euclidean distance took 1.30h $\sim$ to compute 10000 iterations, while with

```
pdist2(X,x_query,'euclidean')
```

it took 1h $\sim$ (these tests has been done using a HP 250 G1 with Inter Core i3 processor). In order to fasten the training process, it has been used the C-based MATLAB function

```
DNorm2(X-x_query,2) [5]
```

which reduced the computation time to almost half an hour for the entire training.

IV. Tuning

To find the best k suitable for the data, we had to run the kNN classifier several times with the different values of k shown. As seen from Table 2, the better overall accuracy has been obtained with $k = 3$ (the graph w.r.t. k is given in Appendix), while in Table 1 are available the relative maxima.

As we kept the value of k neat to 1, the predictions were less stable, then the accuracy was worse. In fact ,for the limit case $k = 1$, the single nearest neighbor is the only discriminant for classification, so, a queried "4" near to several other "4" but with a single "1" as nearest neighbor, will be misclassified as "1". Inversely, as te value of k increased, the predictions became more stable: we were in a situation where the outcome was reached due to a majority voting dynamic. Then, more likely we would have had more accurate predictions. The problem with this increase in value of k is that, at a certain point, we began to witness an increasing number of errors because of the excessive inclusion in the majority voting of samples not really "near" to our queried one. For this peculiar case, the MNIST dataset classification perform better with a low value of k ($k = 3$), but still we can see the worsening of performances for $k < 3$.

V. Conclusions

Tuning the kNN classifier is not a complex task theoretically, but it requires a lot of computational power for big datasets. As shown, the behavior can be easily grasped from the accuracy results, also w.r.t. the single classes. Regarding this last aspect, is interesting to notice how, in this case, for each digit there are different optimal values. This suggests that a further study will have to analyze how to preprocess the data in order to eliminate as much as possible these differences. Further work has to be done as well in preprocessing, in order to reduce the size of the input and ease even more the computational weight.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998. DOI: 10.1109/5.726791.

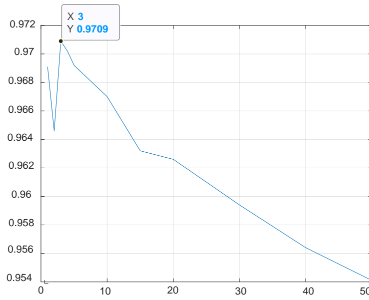
[2] T. M. Cover and P. E. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, Jan. 1967. DOI: 10.1109/TIT.1967.1053964.

[3] N. S. Altman, "An introduction to kernel and nearest-neighbor nonparametric regression," *The American Statistician*, vol. 46, no. 3, pp. 175–185, 1992. DOI: 10.1080/00031305.1992.10475879.

[4] K. M. Ting, *Encyclopedia of machine learning*, Springer, Ed. 2001, ISBN: 978-0-387-30164-8.

[5] M. FileExchange. (2010). Dnorm2 function, [Online]. Available: https://mathworks.com/matlabcentral/fileexchange/29035-dnorm2.

Appendix



Confusion matrix for K = 1 - Acc. = 0.9691

True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	0
1	1129	3		1	1	1				
2	6	992	5	1		2	16	3		7
3	1	2	970	1	19		7	7	3	
4	7			944		3	5	1	22	
5	1		12	2	860	5	1	6	4	1
6	2			3	5	944				4
7	14	6	2	4			992		10	
8	1	3	14	5	13	3	4	920	5	6
9	5	1	6	10	5	1	11	1	967	2
0	1	1			1	3	1			973

Confusion matrix for K = 2 - Acc. = 0.9646

True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	0
1	1129	3		1	1	1				
2	8	988	6	1		1	18	2		8
3	1	5	964	1	19	1	8	5	5	1
4	5			940		5	5	1	24	2
5	1		17	2	850	6	1	4	6	5
6	2			6	3	942		1		4
7	17	5	1	4			991		10	
8	2	5	17	7	20	2	3	906	6	6
9	5	2	6	8	4	1	12	2	963	6
0	1	1			2	2	1			973

Confusion matrix for K = 3 - Acc. = 0.9709

True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	0
1	1132	3								
2	7	995	2	1			15	2		10
3	1	4	975	1	12	1	7	3	6	
4	4			949		4	4		21	
5	1		10	2	861	5	1	4	4	4
6	3	1		3	3	944				4
7	20	4		1			992		11	
8	2	3	13	7	13	3	4	919	5	5
9	5	2	8	9	2	1	8	2	968	4
0	1	1			1	2	1			974

Confusion matrix for K = 4 - Acc. = 0.9702

True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	0
1	1133	2								
2	5	989	4	1			18	4		11
3	2	2	973	1	14	1	8	4	5	
4	8			945		4	2		22	1
5			10	2	864	7	1	1	3	4
6	3			4	3	943				5
7	20	3		4			992		9	
8	2	5	12	6	9	5	4	921	5	5
9	4	2	5	8	4	1	11	1	970	3
0	1	1			1	3	1	1		972

Confusion matrix for K = 5 - Acc. = 0.9692

True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	0
1	1133	2								
2	8	989	3			2	16	4		10
3	2	2	974	1	13	1	7	4	6	
4	6			945		4	2	1	22	2
5			9	2	860	7	1	4	4	5
6	3			3	1	946				5
7	21	4		3			989		11	
8	3	4	12	4	9	6	5	919	6	6
9	7	3	9	7	3	1	10	2	963	4
0	1	1			1	2	1			974

Confusion matrix for K = 10 - Acc. = 0.967

True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	0
1	1132	2				1				
2	11	981	2			2	18	5		13
3	3	2	975	1	11	1	7	7	3	
4	11			937		6	1	1	25	1
5			6	1	866	7	1	1	6	4
6	4			3	2	943				6
7	26	4		2			983		13	
8	4	4	11	6	7	4	6	918	8	6
9	6	3	6	9	3	1	10	2	963	6
0	1	1			2	3	1			972

Confusion matrix for K = 15 - Acc. = 0.9632

True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	0
1	1131	2	1			1				
2	15	966	4	1		3	20	8		15
3	3	2	974	1	14		7	5	4	
4	13			934		5	2	1	26	1
5	1		8	1	862	10	1		6	3
6	4			3	1	942		1		7
7	28	3		2			980		15	
8	4	5	12	5	11	5	9	909	7	7
9	6	2	8	9	2	1	10	1	964	6
0	1	1			2	5	1			970

Confusion matrix for K = 20 - Acc. = 0.9626

True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	0
1	1130	2	1			2				
2	18	961	7	2		3	19	6		16
3	3	2	976	1	12		7	5	4	
4	13			935		5	2	1	25	1
5	1		8	1	863	10	1		5	3
6	4			3	1	942		1		7
7	27	4		2			979		16	
8	4	4	13	5	10	3	8	911	9	7
9	6	2	10	10	2	1	11	1	959	7
0	1	1			2	5	1			970

Confusion matrix for K = 30 - Acc. = 0.9594

True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	0
1	1130	2	1			2				
2	19	955	8	2		4	21	7		16
3	3	2	977	1	11		6	5	5	
4	15			926		8	3	2	28	
5	3		9	1	857	11	2		5	4
6	5			3	2	940				8
7	32	3		2			973		18	
8	5	3	16	7	12	3	6	905	9	8
9	7	3	8	9	2	1	10		961	8
0	1	1			2	5	1			970

Confusion matrix for K = 40 - Acc. = 0.9564

True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	0
1	1130	2	1			2				
2	26	948	6	2		4	20	7		19
3	4	2	974	1	12		8	5	4	
4	16			924		9	1	1	31	
5	6		8	1	852	12	2	1	7	3
6	5			3	2	940				8
7	34	4		2			969		19	
8	5	3	18	8	13	5	7	899	8	8
9	7	3	7	7	5	1	11		960	8
0	1	1			2	7	1			968

Confusion matrix for K = 50 - Acc. = 0.9541

True Class \ Predicted Class	1	2	3	4	5	6	7	8	9	0
1	1130	2	1			2				
2	28	944	7	2		4	20	7		20
3	4	2	970	1	14	1	9	5	4	
4	17			921		9	2	1	32	
5	8		8	1	850	12	2	1	7	3
6	5			3	2	940				8
7	37	2		2			966		21	
8	6	3	17	10	14	6	7	893	9	9
9	7	3	9	7	2	1	13		959	8
0	1	1			2	7	1			968

Chapter 6

Evaluation of classifiers

6.1 Estimation of the generalization ability

6.1.1 Statistical learning

Learning a mapping $t = t(\boldsymbol{x})$ (e.g., classification) consists of selecting a function $y = y(\boldsymbol{x})$ from a hypothesis space H:

$$\text{Ideal case: } t(\boldsymbol{x}) \in H \text{ and } y \equiv t$$

Normally this is not assured and we can only find $(\boldsymbol{x}) \in H$ such that $y \simeq t \rightarrow$ we measure the quality by a **loss function**:

$$\lambda(y|t) \qquad \text{Expected risk: } R = \mathbb{E}\{\lambda\}$$

Learning is done by using a training set $X = \{\boldsymbol{x}_1, ..., \boldsymbol{x}_n\}$ with targets $T = \{t_1, ..., t_n\}$ and **minimizing the empirical risk**:

$$\text{Minimization: minimize } \hat{R} = \text{Empirical risk} = \text{Monte Carlo estimate of } R \text{ on } X$$

6.1.2 Sampling effects

A training set is a sample of the whole population. Since samples are realizations of a random variable, they will be different each time (we already know that learning machines "learn" different rules if we change the data).

6.1.3 Bias/variance decomposition

It is possible to decompose the expected square error objective J_{MSE} as the sum of three terms:

$$J_{MSE} = B + V + N$$

- B : **bias** – the expected difference between the error of the best model and the error of our model
- V : **variance** – the variance of our model?s error? i.e. the expected squared variation w.r.t. the bias (cfr Gaussian)
- N : **noise** – a residual, irreducible quantity, the amount of stochasticity in the data (e.g. training sets having different targets for very similar inputs)

For classification it is possible but more involved; underlying concepts are the same.

The bias/variance tradeoff is due to the fact that B can be lowered by using a more adaptable model (but then the learner will fit the given training set: **overfitting**) while the variance can be lowered by using a less adaptable model (but then the learner will have a low performance: **underfitting**). So:

- low bias = good performance (on the training set)
- low variance = reliable performance in inference: **generalization**

6.1.4 How to control generalization?

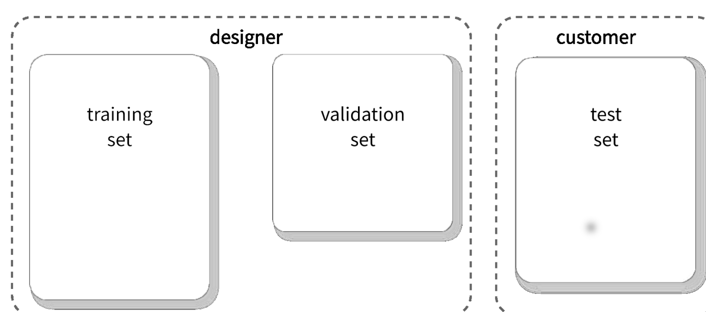
Several strategies are possible, divided in two main categories:

1. Evaluate quality empirically (ex post), and if necessary use this estimate for designing a new and better learner
2. Estimate quality theoretically (ex ante), and use this estimate for designing the learner once and for all

6.2 Computational (empirical) estimates of generalization

WARNING: We are MEASURING quality, NOT IMPROVING IT!

The standard procedure for empirical evaluation is to divide the dataset in:



The designer trains the machine using a **training set** and then evaluates the quality of the machine using an independent **validation set** (**independent** = the observations must be present either in the training set or in the test set, not both). The validation set is often called **hold-out set** when obtained by holding out a fraction of the observations of the training set. If the quality of the machine does not reach a prescribed quality level, its structure (hypothesis space) is changed and training is repeated until satisfactory.

Once the machine is finalized, its quality is evaluated on another, **independent test set**.

Which size does the data sets have to have? A small training set will lead to **overfitting** (bad generalization) while a small validation or test set will lead to an **unreliable quality evaluation**.

Reminder: We are MEASURING quality – NOT IMPROVING IT.
Learning does not depend on validation and test set size

6.2.1 Resampling methods

But what if the available **data are not enough**? "Modern" statistics takes advantage of computers to allow simulated experiments. **Resampling** is generating many new samples from a given available sample. The quantity R of interest is measured many times and its distribution is estimated. From the estimate we can compute confidence intervals and therefore we can estimate **generalization ability**.

6.2.2 Cross-validation

To perform a cross-validation:

1. Split your sample into a training set and a test set.
2. Evaluate R on the test set
3. This is called simply cross-validation

→ This can be done during iterative training.

When the test value of R starts to grow, we are overfitting **it's time to stop** even if the optimization has not converged yet!

6.2.3 Leave-one-out cross-validation

To perform the Leave-one-out cross-validation:

1. Split your sample into n training sets of size $n-1$ and corresponding n test sets of size 1.

$$X \to \begin{cases} X_1 = X - \{\mathbf{x}_1\} \\ X_2 = X - \{\mathbf{x}_2\} \\ X_3 = X - \{\mathbf{x}_3\} \\ \vdots \\ X_n = X - \{\mathbf{x}_n\} \end{cases}$$

2. Evaluate R on each test set

$$\{\, R(\mathbf{x}_1), R(\mathbf{x}_2), R(\mathbf{x}_3), \ldots, R(\mathbf{x}_n) \,\} = S$$

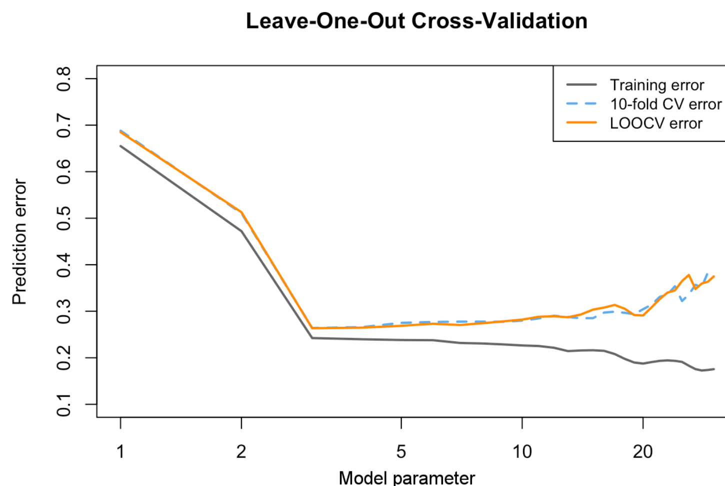
3. Your final estimate of R is the average $\mathbb{E}(S)$ of all these one-point estimates

→ You can also compute standard deviation, confidence intervals...

→ This is called leave-one-out cross validation and is heavily used.

A more general case is k training sets of size n/k and corresponding test sets of size $n = n - n/k$ → leave-k-out or **k-fold cross-validation**.

For example:



The leave-one-out algorithm is the following one:

Input: Data set X of n rows
1: **for** l: 1...n **do**
2: Remove $\mathbf{x}_l$ from X, obtaining training set X_l with $n-1$ rows
3: Train learning machine M_l
4: Compute validation cost $R(\mathbf{x}_l)$ using machine M_l
5: Add $R(\mathbf{x}_l)$ to set S
6: **end for**
7: Train final learning machine M on X
8: Compute statistics on S (average cost, standard deviation of cost, min/max/percentiles of cost, ...)
Output: Machine M characterized by statistics over S

6.2.4 Bootstrap

A **bootstrap sample** is made by sampling with replacement n times from the original training set X.

Take a lot of **bootstrap samples** (all of size n)

In each bootstrap sample, some patterns will be repeated and some will be absent.

The bootstrap estimate of R is the average of $\hat{R}$ on all the bootstrap samples.

6.3 Empirical evaluation of classifiers

There are specific **quality indexes** for classification. In this part we consider indexes that measure classification quality, and for that counting errors is usually not enough.

6.3.1 Contingency tables

In a contingency table, entries indicate number of co-occurrences of pairs of events from two event sets $E_1, E_2, ..., E_k$ and $F_1, F_2, ..., F_h$:

$$\begin{array}{c} \\ F_1 \\ F_2 \\ \vdots \\ F_h \end{array} \begin{array}{c} \begin{array}{cccc} E_1 & E_2 & \dots & E_k \end{array} \\ \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1k} \\ c_{21} & c_{22} & \dots & c_{2k} \\ & \vdots & & \\ c_{h1} & c_{h2} & \dots & c_{hk} \end{bmatrix} \end{array}$$

The entry ij represents number of experiments where events E_i and F_j have been observed together.

From a contingency table we can obtain the following types of information about co-occurrence of events:

- **counts**, its entries c_{ij}
- **frequencies**, if we normalize its entries: $f_{ij} = c_{ij}/N$
- **probabilities**, if we consider frequency as empirical estimates of probability: $p_{ij} \simeq f_{ij}$

6.3.2 Confusion matrix, accuracy and error rate

The confusion matrix C is a contingency table for classification outputs vs. actual classes. Hence, the entry c_{ij} represents number of times that the classifier decided ω_j when in fact the true class was ω_i. Usually C is a **square matrix**.

$$\text{sum of all entries } \sum_i \sum_i c_{ij} = \text{ total number of experimental observations } = \text{ sample size } n$$

Example: Classify web robots ("bots") vs human visitors in a web site:

Target class	Output class: Non-bot	Output class: Bot	
Non-bot	56.3%	3.7%	*93.8 %*
Bot	0.8%	39.1%	*98.0%*
	98.5%	*91.3%*	**95.5%**

Confusion matrix for multi-layer perceptron classification.

Target class	Output class: Non-bot	Output class: Bot	
Non-bot	58.5%	1.6%	*97.4 %*
Bot	0.3%	39.6%	*99.1%*
	99.4%	*96.2%*	**98.1%**

Confusion matrix for support vector machine classification.

Accuracy and error rate Given the confusion matrix C we can define:

$$n_{\text{correct}} = \sum_{i=1}^{k} c_{ii} = \text{tr}C \qquad n_{\text{err}} = n - n_{\text{correct}}$$

where trC = trace of C (sum of diagonal elements)

The normalized indexes are:

the frequency (rate) of correct classification, or **accuracy** $f_{\text{correct}} = \frac{n_{\text{correct}}}{n}$ the **error frequency** (**error rate**) $f_{\text{err}} = \frac{n_{\text{err}}}{n}$

6.3.3 The dichotomic case

When we have to classify whether an input is of a given class positive outcome or 1, vs. negative outcome or 0, the confusion matrix is:

		Output	
		0	1
Target	0	NR CORRECT NEG	NR FALSE POS
	1	NR FALSE NEG	NR CORRECT POS

- cases 00 and 11 : correct classifications
- case 01 : false positive
- case 10 : false negative

Proportions (or frequencies):

$$f_{01} = c_{01}/n \quad \text{and} \quad f_{10} = c_{10}/n$$

Probabilities:

$$P(01) \approx f_{01} \quad \text{and} \quad P(10) \approx f_{10}$$

For this case in statistics, special historical names are used:

- Event 01 or false positive = error of the first kind or **type I error**
- Event 10 or false negative = error of the second kind or **type II error**
- $P(01) = \alpha$
- $P(10) = \beta$

Example: Recognize publications written by T. Bayes in the whole Web. There is only one document by Bayes and (suppose) 10^{12} other documents on the Web:

Case 1: All correctly recognized.

$$\text{Accuracy} = \frac{10^{12}+1}{10^{12}+1} = 1$$

Case 2: We always classify as "not-by-Bayes".

$$\text{Accuracy} = \frac{10^{12}}{10^{12}+1} = \frac{1000000000000}{1000000000001} \approx 1$$

→ **We got one of the two classes completely wrong (100% errors), yet the accuracy was almost perfect!**

6.3.4 Dichotomic case: more indexes

Sensitivity and specificity As we saw, we need other indexes when accuracy is not enough:

$$\underset{\text{probability to correctly recognize class 1}}{\textbf{sensitivity} \text{ or } \textbf{true positive rate}} = \frac{\text{prob. true pos}}{\text{prob. true pos} + \text{prob. false neg}} = \frac{P(11)}{P(11)+P(10)} \approx \frac{c_{11}}{c_{11}+c_{10}}$$

$$\underset{\text{probability to correctly recognize class 0}}{\textbf{specificity} \text{ or } \textbf{true negative rate}} = \frac{\text{prob. true neg}}{\text{prob. true neg} + \text{prob. false pos}} = \frac{P(00)}{P(00)+P(01)} \approx \frac{c_{00}}{c_{00}+c_{01}}$$

A good classifier has both high sensitivity and high specificity.

Precision and recall Both express the fraction of true positives over two different sets (those judged as positive, and those actually positive):

precision
or **positive predictive value**
(The fraction of positive outputs that was actually positive)

$$= \frac{P(11)}{P(11)+P(01)} \approx \frac{c_{11}}{c_{11}+c_{01}}$$

recall (sensitivity)

$$= \frac{P(11)}{P(11)+P(10)} \approx \frac{c_{11}}{c_{11}+c_{10}}$$

***F* measure** A combined index called the F measure is also defined:

$$F = 2\frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}} = \frac{2P(11)^2}{2P(11)+P(10)+P(01)} \approx \frac{2c_{11}^2}{2c_{11}+c_{10}+c_{01}}$$

A single-number evaluation

A generalized version exists:

$$F_\beta = (1+\beta^2)\frac{\text{precision} \times \text{recall}}{\beta^2\text{precision} + \text{recall}}$$

the basic F-measure is when $\beta = 1$
so it can also be named **F1 measure**

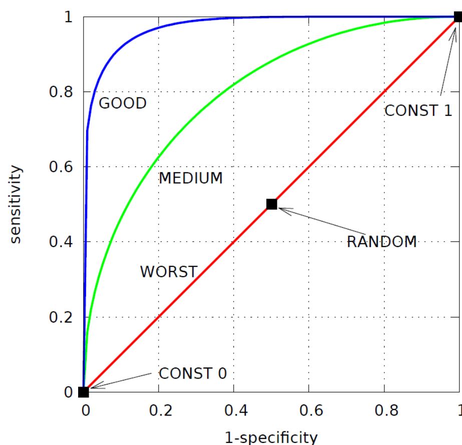
Remark! Neither precision/recall nor F take the probability of being correct on the negative class into account (i.e., P_{00} or c_{00} do not appear anywhere), and this is an advantage when it is too easy to be correct on the negative class!

Receiver operating characteristic curve (ROC) For a binary classifier whose operation **depends on some parameter**, for instance a threshold θ, changing this parameter changes the performance of the classifier itself - and changes C:

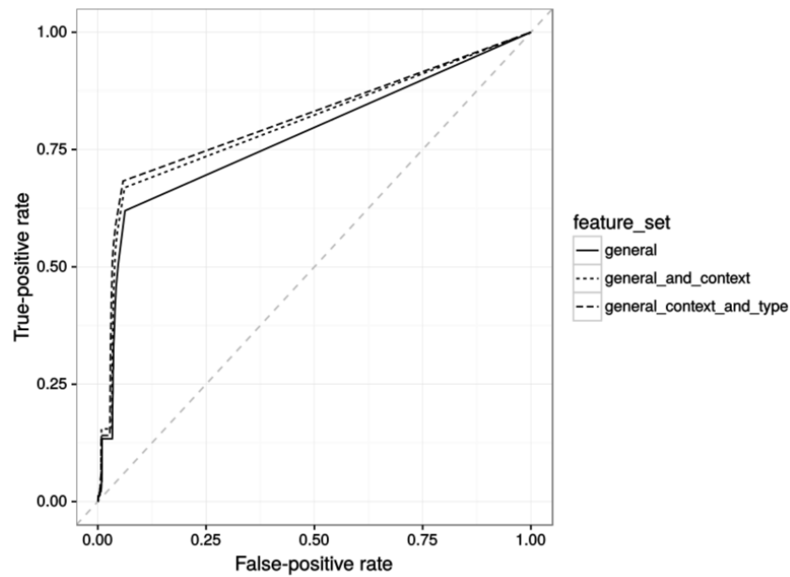
$$\text{Sensitivity } \frac{P(11)}{P(11)+P(10)} \text{ as the abscissa}$$

$$\text{False positive rate} = 1 - \text{specificity} = \frac{P(01)}{P(00)+P(01)} \text{ as the ordinate}$$

We plot for varying values of θ

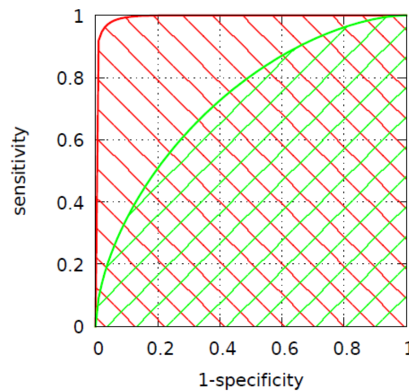


Example: consider the vandalism detectors on Wikidata (three different feature sets)[1]



← some vandalized pages are left
some legitimate pages are removed→

AUC analysis The optimal R.O.C. is the one that runs closers to the left and top boundaries. **This is also the one enclosing the largest area**:



The **Area Under the (r.o.c.) Curve** is another quality index for a classifier.

Remark! why not using these indexes directly as **objective functions** to be optimized for learning? Because They cannot be expressed as the expectation (mean) of a loss function. For instance $F1$:

$$F1 = \frac{2c_{11}^2}{2c_{11} + c_{10} + c_{01}} = \sum_{l=1}^{n} \lambda(y_l, t_l) \qquad \Rightarrow \qquad \lambda(y_l, t_l) = \quad ???$$

[1] from https://meta.wikimedia.org/wiki/Research:Building_automated_vandalism_detection_tool_for_Wikidata

Chapter 7

Neural Networks

Neural Networks are computational models of network of neurons that focus on **functionality** rather than plausibility. They are built of **several** layer, each composed by homogeneous units (performing a **simple, non-linear** computation). Layers are cascaded, and connected to each other in various patterns by **adaptable weights**.

Definition: Artificial Neural networks (ANNs) are networks of several interconnected units, each with a simple behavior, most often without memory and capable of a single input-to-output transformation. There are usually several connections, and their properties are summarized in parameters called weights.

A neural network is a "universal" model for learning mappings from inputs to outputs

	Computers	CNNs
Parallel	Weakly	Heavily
Processors	Complex	Simple units
Velocity	Very fast	Not very fast
Connection	Weakly connected	Heavily connected
Synchronicity	Synchronous	Asynchronous
Data	Digital	Analog

7.1 Brain and Neural Networks

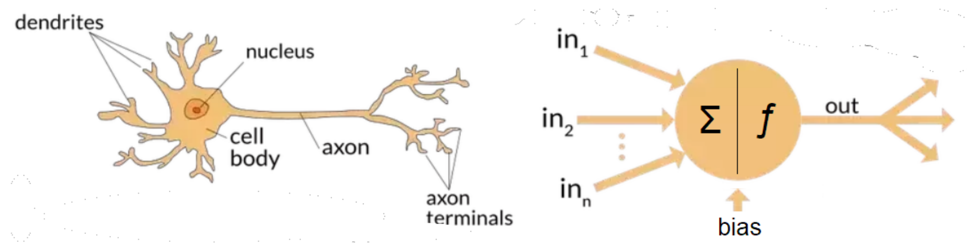
7.1.1 Biological inspiration

ANNs were inspired by the biological processes scientists were able to observe in the brain back in the 50s (although they do differ from their biological counterparts in several ways).

The idea behind **perceptrons** (the predecessors to artificial neurons) is that it is possible to **mimic certain parts of neurons**, such as dendrites, cell bodies and axons using simplified mathematical models of what limited knowledge we have on their inner workings: signals can be received from dendrites, and sent down the axon once enough signals were received. This outgoing signal can then be used as another input for other neurons, repeating the process.

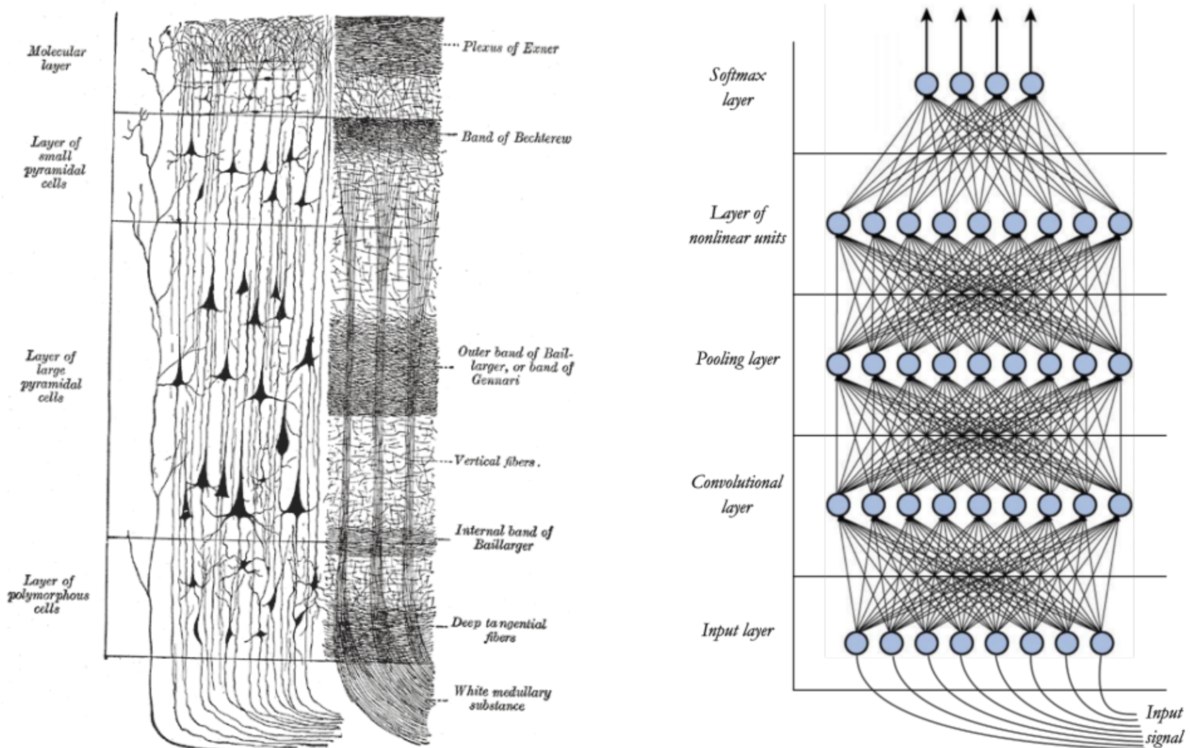
Some signals are more important than others and can trigger some neurons to fire easier. Connections can become stronger or weaker, new connections can appear while others can cease to exist. We can mimic most of this process by coming up with a function that receives a list of weighted input signals and outputs some kind of signal if the sum of these weighted inputs reach a certain bias.

Note that this simplified model does not mimic neither the creation nor the destruction of connections (dendrites or axons) between neurons, and ignores signal timing. However, this restricted model alone is powerful enough to work with simple classification tasks: invented by Frank Rosenblatt, the perceptron was originally intended to be a custom-built mechanical hardware instead of a software function:



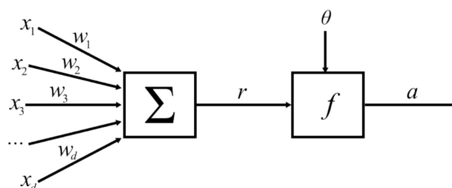
Regarding the signal generation and transmission in a neuron, we can identify an **input** (integration of membrane potential), a **decision** (excitation, firing) and an **output** (signal over axon). We will pretend that the information (however carried) is a single number.

Regarding the architecture of the brain, we can identify several types of neurons: **afferent neurons** (or sensory or receptor neurons), **efferent neurons** (or motor or effector neurons) and **interneurons** (or relay neurons, or association neurons, or bipolar neurons).



7.1.2 Neural Network model

In order to introduce the NN model, we can use the same representation of the first classifier seen (linear classifier):



that has:

- r is the net input
- f is the activation function
- a is the activation value

We have:

$$r = \boldsymbol{x} \cdot \boldsymbol{w} \qquad a = f(r - \theta)$$

- $\boldsymbol{x}$ is a d-dimensional vector of input values
- $\boldsymbol{w}$ is the corresponding (d -dimensional) vector of synaptic weights, that and are either positive (exciting) or negative (inhibiting)
- $\cdot$ indicates scalar product
- r indicates the net input on the neuron membrane
- $f()$ is a nonlinear function
- θ is a threshold
- a indicates the activation value of the membrane potential, or action potential; that is, the output of the neuron.

Examples of **activation functions** f are

- **Heaviside step** (defined on $[-\infty, +\infty] \to \{0, +1\}$):

$$f(r) = \mathbf{1}(r) = \text{step}(r) = \Theta(r) = \begin{cases} +1 & r \geq 0 \\ 0 & r < 0 \end{cases}$$

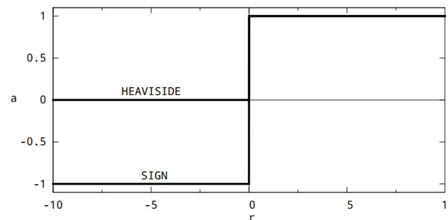
  (ties broken arbitrarily)

- **Signum function** (defined on $[-\infty, +\infty] \to \{-1, +1\}$):

$$f(r) = \text{sign}(r) = \begin{cases} +1 & r \geq 0 \\ -1 & r < 0 \end{cases}$$

  The signum function is a symmetrization in the interval $[-1, +1]$ of the Heaviside step function:

$$\text{sign}(r) = 2\ \text{step}(r) - 1$$



- **Sigmoid** or logistic **function** (defined on $[-\infty, +\infty] \to \{0, +1\}$):

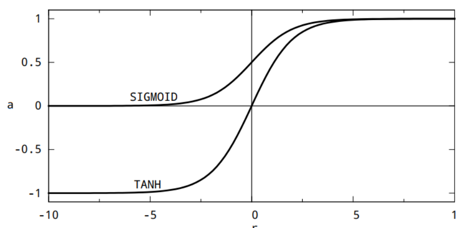
$$f(r) = \sigma(r) = \frac{1}{1 + e^{-r}}$$

- **Hyperbolic tangent** (defined on $[-\infty, +\infty] \to \{-1, +1\}$):

$$f(r) = \tanh(r) = \frac{1 - e^{-r}}{1 + e^{-r}}$$

  The hyperbolic tangent function function is a symmetrization in the interval $[-1, +1]$ of the sigmoid function:

$$\tanh(r) = 2\ \sigma(2r) - 1$$

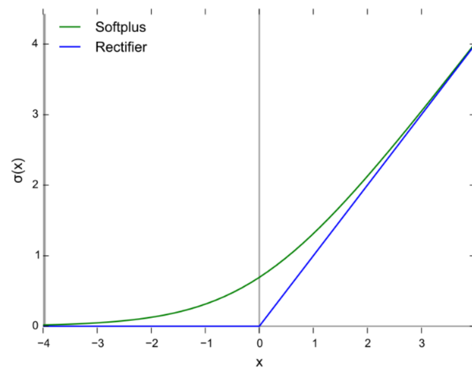


- **Ramp** or Rectified-linear (defined on $[-\infty, +\infty] \rightarrow \{0, +1\}$)[1]:

$$f(r) = max(0, r)$$

- **Softplus** (defined on $[-\infty, +\infty] \rightarrow \{-1, +1\}$):

$$f(r) = \ln(1 + e^r)$$



There are also **multi-neuron activation functions**, that work on k neurons and produce k interdependent values. So, given the stimuli $r_1, ..., r_k$, compute the output of neuron i:

- **Max**:

$$f(r_i) = \begin{cases} +1 & r_i = \max_i(r_1, ..., r_k) \\ 0 & \text{else} \end{cases}$$

- **Softmax**:

$$f(r_i) = \frac{e^{r_i}}{\sum_{j=1}^{k} e^{r_j}}$$

Neuron assemblies Many units combine to form one collective output. Here k neurons produce 1 scalar value.

- ReLU and Softplus can be considered as **approximations to the output sigmoids of many neurons, summed** together
- Max pooling: A small set of units (e.g., $k = 3$) gives one output which is the **maximum of the k individual outputs**.

  Other functions can be used in place of max (e.g., average or median) but in many applications max gives the best results

Learning in neural networks Learning in (biological) neurons occurs by slow modification of synaptic strength → similarly, learning in (formal) neurons occurs by slow modification of weights. Usually there is an iterative procedure that gradually changes weights according to the experience at steps $t = 0, 1, ...$

General formula: $w_{\tau+1} = w_{\tau+1} + \Delta w_{\tau+1} \rightarrow$ whole set of weights $\rightarrow \boldsymbol{w}_{\tau+1} = \boldsymbol{w}_{\tau+1} + \Delta \boldsymbol{w}_{\tau+1}$

Hebb's hypothesis on learning as synaptic modification:

> "When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes part in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased."
> – Donald O. Hebb, *The Organization of Behavior* (1949)
>
> (or, as a proverb, 'Cells that fire together, wire together')

[1] A unit with the rectifier-linear activation is nicknamed a **rectified-linear unit** or ReLU

7.2 Single layer neural networks

High-level (= abstract) models of the neuron function were proposed by Warren Mc Culloch and Walter Pitts in 1943:

$$y = f(w\Delta x - \theta)$$

where the activation function $f()$ could be:

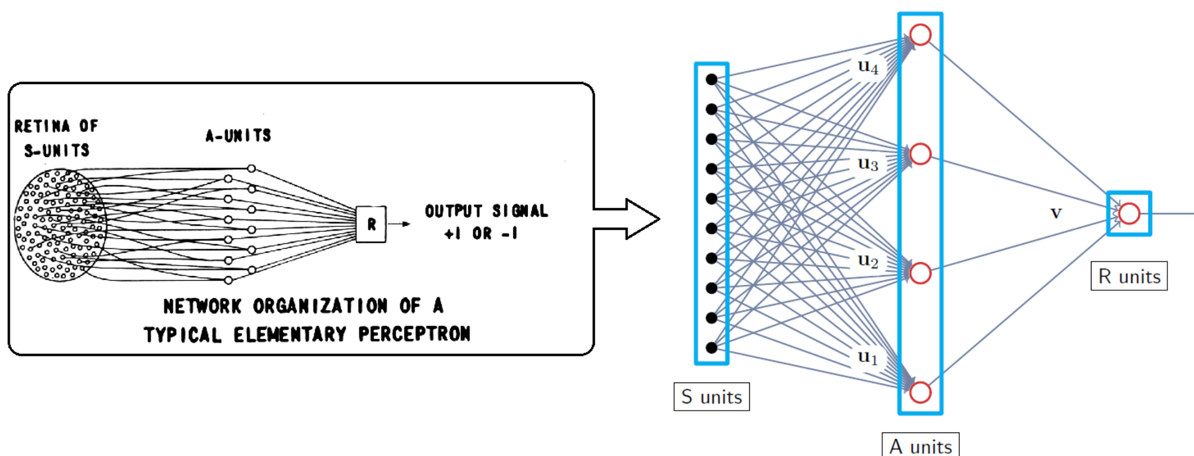
- the sign $(-1/+1)$ function
- or the step $(0/+1)$ function

(Remember that we can get rid of θ)

We will cover two ancient, simple models – Perceptrons and Adaline – in order to introduce several concepts as simple iterative learning rule, learning without memory (by-pattern algorithms), learning as optimization and in the end we will show why we need for more complex models in order to solve complex problems.

7.2.1 Rosenblatt's perceptron (1950s)

In late 1950s, Frank Rosenblatt introduced a network composed of the units that were enhanced version of McCulloch-Pitts Threshold Logic Unit (TLU) model. Rosenblatt's model of neuron, a perceptron, was the result of merger between two concepts from the 1940s, McCulloch-Pitts model of an artificial neuron and Hebbian learning rule of adjusting weights[2]. In addition to the variable weight values, the perceptron model added an extra input that represents **bias**.



In the architecture schematic, S-units are the **Sensory** units, A-units the **Associative** (linear $(a = r)$, randomly connected, random *weights*) and R-units the **Response** units (*linear threshold* units)

The perceptron is effectively a single-unit network:

$$\begin{cases} a_1 = \mathbf{u}_1 \cdot \mathbf{x} \\ a_2 = \mathbf{u}_2 \cdot \mathbf{x} \\ a_3 = \mathbf{u}_3 \cdot \mathbf{x} \\ a_4 = \mathbf{u}_4 \cdot \mathbf{x} \end{cases} \quad \text{or} \quad \mathbf{a} = \mathbf{U}^T\mathbf{x}, \quad \mathbf{U} = \begin{pmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \mathbf{u}_3 & \mathbf{u}_4 \end{pmatrix}$$

$$y = \mathbf{v} \cdot \mathbf{a} = \mathbf{v}^T\mathbf{U}^T\mathbf{x} = (\mathbf{v}^T\mathbf{U}^T)\mathbf{x}$$

$(\mathbf{v}^T\mathbf{U}^T)$ is a weight vector $\mathbf{w} = \mathbf{v}^T\mathbf{U}^T$

$$y = \operatorname{sgn}(\mathbf{w} \cdot \mathbf{x})$$

7.2.2 Perceptron learning algorithm

The perceptron learning algorithm is a **by-pattern algorithm**: the weight modifications are computed upon receiving each individual pattern.

[2] Bose, N. K. and Liang, P. "*Neural Network Fundamentals with Graphs, Algorithms, and Applications*". McGraw-Hill, New York, NY, 1996.

We assume a linear threshold unit (perceptron) with

$$\text{Activation: } f(r) = \text{sign}(f : \mathbb{R}^{d+1} \rightarrow \{-1, 1\}) \qquad \text{Bias: } w_0$$

For the **initialization**, we take a training set of N patterns $\boldsymbol{x}_l$ with targets $\boldsymbol{t}_l$ and then we assign some initial values to weights $\boldsymbol{w}$ (e.g. random in $[-1, 1]$) and finally we start with iteration step $l = 1$.

> An error-corrective reinforcement system (error correction system) is a training procedure in which the magnitude of η is 0 unless the current response of the perceptron is wrong, in which case, the sign of η is determined by the sign of the error. In this system, reinforcement is 0 for a correct response, and negative for an incorrect response, or, more generally, $\eta = f(R^* - r^*)$ where R^* is the required response, r^* is the obtained response, and f is a sign-preserving monotonic function, such that $f(0) = 0$.

The perceptron **learning algorithm** is the following one:

(init) Perform initializations

1. INPUT: Read input pattern number l, $\mathbf{x}_l$
2. Compute output: $r_l = \mathbf{w} \cdot \mathbf{x}_l$, $a_l = f(r_l)$
3. Compute output error: $\delta_l = t_l - a_l$
4. Compute correction: $\Delta\mathbf{w}_l = \delta_l \mathbf{x}_l$
5. Apply correction (weight update): $\mathbf{w}_{l+1} = \mathbf{w}_l + \Delta\mathbf{w}_l$
6. Increase l, go back to 1 when $l > n$
7. Go to INPUT

Regarding the **convergence** of perceptron learning:

> *"If the training set is linearly separable, the perceptron learning procedure will find separating hyperplane for it in a finite number of steps."*

If the training set is not linearly separable, the procedure does not converge.

> Given an elementary α-perceptron, a stimulus world W, and any classification $C(W)$ for which a solution exists; let all stimuli in W occur in any sequence, provided that each stimulus must reoccur in finite time; then beginning from an arbitrary initial state, an error correction procedure (quantized or non-quantized) will always yield a solution to $C(W)$ in finite time, with all signals to the R-unit having magnitudes at least equal to an arbitrary quantity $\delta \geq 0$.

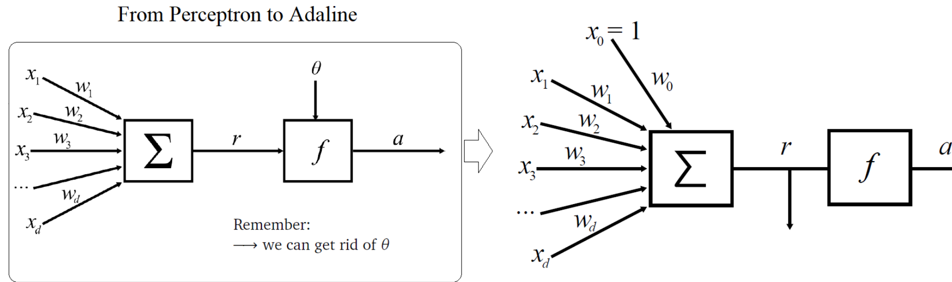
The perceptron learning in Matlab can be implemented in the following way:

We assume that targets are in $\{-1, +1\}$ and are stored in vector `t` of length `n = size(x,1)` and that x has been pre-processed as `x = [ x | ones(n,1) ]`:

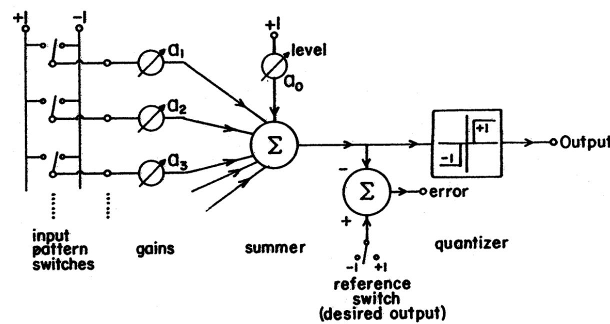
```
n = size(x,1);
d = size(x,2);
w = rand(1,d+1);
l = 1;
while (there_are_errors)
        r = w*x(l,:)' ;
        a = sgn(r) ;
        delta = t(l)-a ;
        dw = delta*x(l,:) ;
        w = w + dw ;
        l = l + 1 ;
        if (l>n)
                l=1 ;
        end
end
```

7.2.3 Widrow and Hoff's Adaline (1960)

The Adaline is a perceptron, but in addition to a, **also r is available** at the outside $\rightarrow$ **differentiable cost function**. According to our previous convention we have:



In the **original schematic** inputs were $\{-1;+1\}$ (binary, not required), output was binary AND analog and weights where potentiometers or memistors:



7.2.4 Adaline learning algorithm (LMS algorithm)

In Adaline's learning algorithm:

- r is used in the learning process rather than a
- r is a continuous function of the weights

$\rightarrow$ We have a quantitative evaluation of the error

The **Least Means Squares (LMS) algorithm** converges to a solution even for non-linearly separable problems, and achieves more robust solutions than the Perceptron algorithm!

Least Means Square rule In the following is described the update step rule:

At time step l the adaline with weights $\mathbf{w}_l$ receives pattern $\mathbf{x}$

The weight vector is changed by summing an updating step $\Delta\mathbf{w}_l$:

$$\Delta\mathbf{w}_{l+1} = \mathbf{w}_l + \Delta\mathbf{w}_l$$

The updating step is proportional to the product

$$\underset{(t-r)}{\text{(error on the linear output)}} \quad \times \quad \underset{\mathbf{x}}{\text{(input vector)}}$$

A kind of "correlation" (cfr. signal processing) between error and input

Proportionality is through a predefined coefficient η, so the rule for a generic time step is:

$$\Delta\mathbf{w} = \eta(t-r)\mathbf{x} \qquad \text{(subscript omitted for clarity)}$$

If we shorten using $\delta = (t-r)$ the rule is written as

$$\Delta\mathbf{w} = \eta\delta\mathbf{x} \qquad \text{hence the nickname "delta rule"}$$

Having to compare continuous quantities, we use the **squared error loss**, so the objective is:

$$J = J_{\text{mse}} = \frac{1}{n}\sum_{l=1}^{n}\frac{1}{2}(t-r)^2 = \frac{1}{2n}\sum_{l=1}^{n}(t-\mathbf{w}\cdot\mathbf{x})^2$$

The **necessary** minimum condition for this cost function is $\nabla J = 0$, and it is also **sufficient** because J_{MSE} is convex on $\mathbb{R}^{d+1}$:

$$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial w_1} \\ \frac{\partial J}{\partial w_2} \\ \vdots \\ \frac{\partial J}{\partial w_d} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{0}$$

$$\frac{\partial J}{\partial w_i} = \frac{1}{2n}\frac{\partial}{\partial w_i}\sum_{l=1}^{n}(t_l-\mathbf{x}_l\cdot\mathbf{w})^2 = \frac{1}{2n}\sum_{l=1}^{n}-2(t_l-\mathbf{x}\cdot\mathbf{w})\frac{\partial \mathbf{x}_l\cdot\mathbf{w}}{\partial w_i} = -\frac{1}{n}\sum_{l=1}^{n}(t_l-\mathbf{x}_l\cdot\mathbf{w})\frac{\partial x_{li}w_i}{\partial w_i}$$

$$= -\frac{1}{n}\sum_{l=1}^{n}(t_l-\mathbf{x}_l\cdot\mathbf{w})x_{li} = -\frac{1}{n}\sum_{l=1}^{n}\delta_l x_{li} \qquad \text{for } \delta_l = (t_l-\mathbf{x}_l\cdot\mathbf{w})$$

The **learning rule** we just obtained is the following one, and it's a rule for **batch training**:

$$\Delta w_i = \frac{1}{n}\sum_{l=1}^{n}\delta_l x_{li} \quad \text{or} \quad \Delta\mathbf{w} = \frac{1}{n}\sum_{l=1}^{n}\delta_l\mathbf{x}_l$$

In this case, it converges in one step to the global minimum of $J = J_{MSE}$, but this is possible **only when we have all the training set at once**! If we don't have it, we have to use online learning.

7.2.5 LMS with online learning

Delta rule for the l-th pattern $\boldsymbol{x}_l$ At each pattern $\boldsymbol{x}_l$, a small updating step $\Delta\boldsymbol{w}_l$ is applied (**online learning** or **learning by pattern**):

$$\mathbf{w}_{l+1} = \mathbf{w}_l + \Delta\mathbf{w}_l \qquad \Delta\mathbf{w}_l = \eta\delta_l\mathbf{x}_l$$

where

- $\delta_l = t_l - r_l$
- η "small"
- $\mathbf{w}_l$ weights at step l
- $\mathbf{w}_{l+1}$ weights that will be used at the next step $l+1$

7.2.6 LMS algorithm and MSE minimization

Does the LMS algorithm really correspond to Mean Square Error (MSE) minimization?
YES, but **convergence is on average**, not deterministic.

The iterative minimization of the squared error pass through these steps:

- Initialization: $\mathbf{w}_{1i} = 0 \quad \forall i : 1\ldots d \quad$ or $\quad \mathbf{w}_1 = \mathbf{0}$
- Gradient descent step: $\mathbf{w}_{l+1} = \mathbf{w}_l + \eta(-\nabla J(\mathbf{w}_l))$
- Estimate of the gradient: $-\frac{\partial J}{\partial w_{li}} = (t_l - \mathbf{w}_l\cdot\mathbf{x}_l)x_{li} \quad \forall i : 1\ldots d \quad$ or $\quad -\hat{\nabla} J = (t_l - \mathbf{w}_l\cdot\mathbf{x}_l)\mathbf{x}_l$

The term

$$\delta_l \boldsymbol{x}_l = (t_l - \boldsymbol{x}_l \cdot \boldsymbol{w})\boldsymbol{x}_l$$

is an **estimate of the (negative) gradient based on just one sample**. If the learning step η is small enough, this converges **on average** to the unique minimum:

> For $l \to \infty$ it will keep on oscillating, but always around the MSE solution.

If a probabilistic model of input is known (**autocorrelation function**), bounds for η can be obtained.[3]

7.2.7 Two further steps

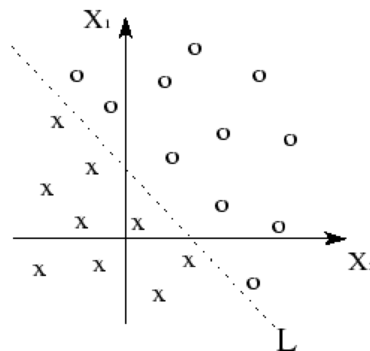
1. The single-neuron "networks" seen so far only implement linear threshold classifiers (cfr. *Perceptrons: An Introduction to Computational Geometry* by Marvin Minsky, Seymour A. Papert, The MIT Press, 1987 expanded edition)

   How is it possible to do better?
   (cfr. *Parallel Distributed Processing: Explorations in the Microstructure of Cognition* by David E. Rumelhart, James L. McClelland and the PDP research Group, The MIT Press, 1987)

2. What is the general relationship between optimization (seen in previous lectures) and learning-by-pattern algorithms (seen here)?

7.2.8 The linear separability problem

Consider two-input patterns (X_1, X_2) being classified into two classes as shown:



Each point with either symbol of x or o represents a pattern with a set of values (X_1, X_2). Each pattern is classified into one of two classes. Notice that these classes can be separated with a single line L. They are known as linearly separable patterns. Linear separability refers to the fact that classes of patterns with n-dimensional vector $\mathbf{x} = (x_1, x_2, ..., x_n)$ can be separated with a single decision surface. In the case above, the line L represents the decision surface.

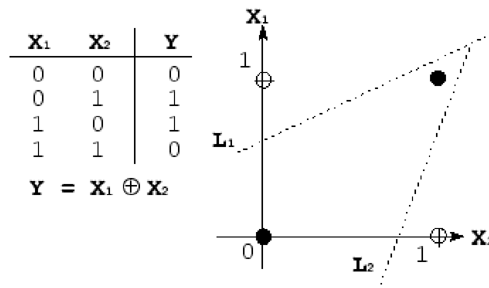
The processing unit of a single-layer perceptron network is able to categorize a set of patterns into two classes as the linear threshold function defines their linear separability. Conversely, the two classes must be linearly separable in order for the perceptron network to function correctly[4]. Indeed, this is the main limitation of a single-layer perceptron network.

The most classic example of linearly inseparable pattern is a logical exclusive-OR (XOR) function:

[3] A 1-hour lecture by Widrow who explains the LMS algorithm and the Adaline can be found at:

- Part 1 – The LMS algorithm: https://www.youtube.com/watch?v=hc2Zj55j1zU
- Part 2 – ADALINE: https://www.youtube.com/watch?v=skfNlwEbqck

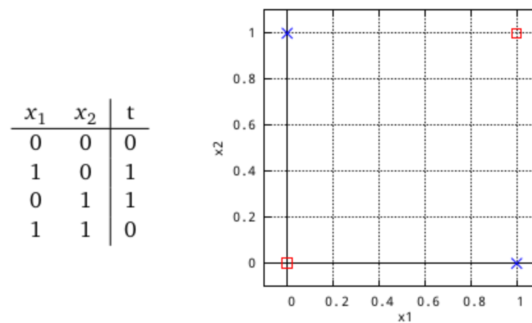
[4] see Haykin, Simon."*Neural Networks: A Comprehensive Foundation, second edition*". Prentice-Hall, Upper Saddle River, NJ, 1999.



X1	X2	Y
0	0	0
0	1	1
1	0	1
1	1	0

The one in figure is the illustration of XOR function that two classes, 0 for black dot and 1 for white dot, cannot be separated with a single line. The solution seems that patterns of (X_1, X_2) can be logically classified with two lines L_1 and L_2[5]

Then, how to overcome the linear separability problem? With feature engineering, using more complex neurons and designing multiple active layers → all are methods to change or **enlarge the hypothesis space**. Then, starting from this situation:



x_1	x_2	t
0	0	0
1	0	1
0	1	1
1	1	0

1) Using **feature engineering:**

We employ our knowledge of the problem to build new features

x_1	x_2	$x_3 = x_1 x_2$	t
0	0	0	0
1	0	0	1
0	1	0	1
1	1	1	0

$$w_0 + x_1 w_1 + x_2 w_2 + \boxed{x_3 w_3} = t$$

$$\begin{cases} w_0 + 0 \cdot w_1 + 0 \cdot w_2 + \boxed{0 \cdot w_3} = 0 \\ w_0 + 1 \cdot w_1 + 0 \cdot w_2 + \boxed{0 \cdot w_3} = 1 \\ w_0 + 0 \cdot w_1 + 1 \cdot w_2 + \boxed{0 \cdot w_3} = 1 \\ w_0 + 1 \cdot w_1 + 1 \cdot w_2 + \boxed{1 \cdot w_3} = 0 \end{cases} \qquad \mathbf{w} = [\,0\;1\;1\;-2\,]$$

2) Using a **nonlinear network**:

We employ our knowledge of the problem to suitably transform the discriminant function

$$V = \begin{bmatrix} 0 & v \\ v & 0 \end{bmatrix}$$

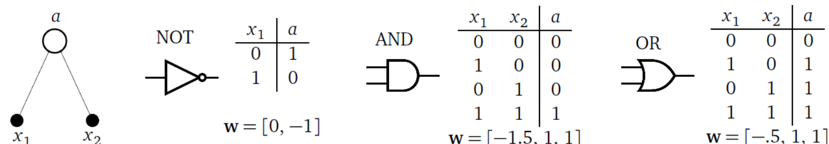
$$\Downarrow$$

$$r = \mathbf{w} \cdot \mathbf{x} + \mathbf{x}^T V \mathbf{x} \quad \Rightarrow \quad r = x_0 w_0 + x_1 w_1 + x_2 w_2 + x_1 x_2 (2v)$$

$$r = x_0 w_0 + x_1 w_1 + x_2 w_2 + x_1 x_2 w_3 \qquad \text{or, if } w_3 = 2v$$

3) Using a **multi-layer network**:

With a single layer network we can model some logic ports (correspondent to the linear separable ones):



NOT:

x_1	a
0	1
1	0

AND:

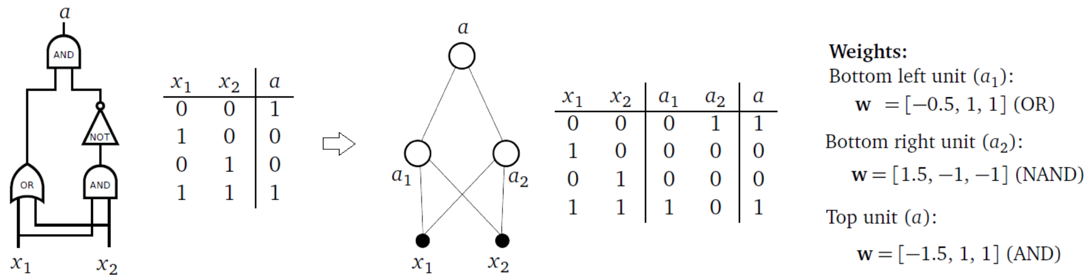
x_1	x_2	a
0	0	0
1	0	0
0	1	0
1	1	1

OR:

x_1	x_2	a
0	0	0
1	0	1
0	1	1
1	1	1

[5]See Beale, R. and Jackson, T."*Neural Computing: An Introduction*". Hilger, Philadelphia, PA, 1991.

But to build a XOR we can use a multilayer network with different ports, and then we can express this as a multilayer network:

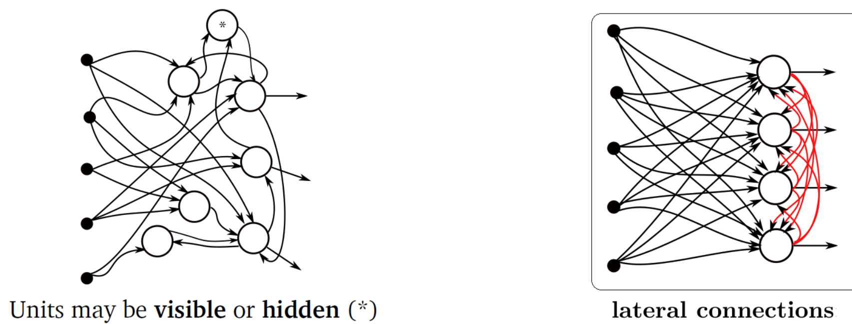


This inspired the design of multilayer neural networks (cfr. David Rumelhart, James McClelland and Geoffrey Hinton).

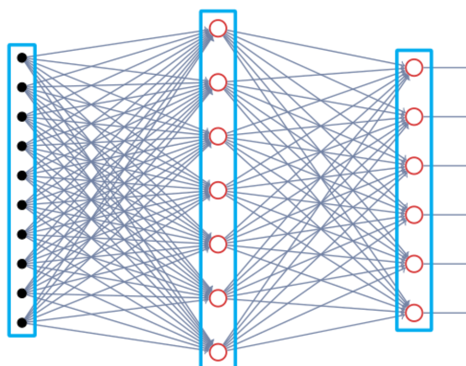
7.3 Multilayer network

7.3.1 Topologies, UAP and learning

Network topologies The most general case of topology includes feedback hidden units, that does not allow to rewrite the network in a layered way (a special type of feedback are **lateral connections**):



The least general topology type is the **feed-forward, multi-layer** one: any general feed-forward topology can be rewritten as a multi-layer topology and any feed-forward topology can be rewritten as a fully connected topology, possibly with some connections having weight $\equiv 0$:



The **universal approximation property** (UAP) states that "*a feed-forward network with a single hidden layer containing a finite number of neurons can approximate continuous functions on compact subsets of* $\mathbb{R}^d$."

→ Cybenko, G. (1989) *Approximations by superpositions of sigmoidal functions*, in "Mathematics of Control, Signals, and Systems", 2(4), 303–314

Activation functions must not be polynomial (including linear), but it **is the topology that has the UAP** regardless of activation function.

→ Hornik, K. (1991) *Approximation Capabilities of Multilayer Feedforward Networks*", in "Neural Networks", 4(2), 251–257

This is an **existence theorem**, not a constructive one: it does not tell us anything about

- whether a particular function can be learned from data
- or how many units are necessary for representing the function within a given error

When applying a learning algorithm, a multi-layer networks will adapt its weights to approximate a given input-output mapping.

The UAP guarantees that any mapping can be learned, even non-linearly-separable ones!

→ this works because each layer can be seen as **learning features = learning an internal representation of the data** for the following layer.

Learning Learning can be cast as **optimization** of a suitable **objective or cost function** (e.g., classification accuracy), but most optimization methods rely on

- either the necessary minimum condition $\nabla f = 0$
- or on the direction of the gradient ∇f

→ requirement: f must be **at least differentiable**:
(even better if also convex, but that's not possible with neural networks)

- λ differentiable w.r.t. the output
- output differentiable w.r.t. the weights

Even if f is differentiable, **for hidden units we cannot compute the loss**
(i.e. an error term δ like the MSE $(t-a)^2$)
→ requirement: we need a way to do this (we will need the *error back-propagation algorithm*, but before let's analyze some activation function to see if they are at least differentiable)

7.3.2 Sigmoid activation function

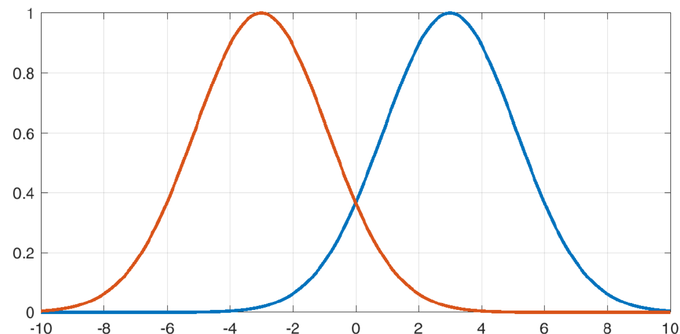
The *Sigmoid* function is a popular differentiable activation function. In order to obtain it, we have to start by writing the discriminant function for a problem with two Gaussian, spherical, equal-variance classes:

Translation of the origin, rotation of axes.

1-dimensional symmetrical problem in x with only two parameters:

$$p(x|\omega_1) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[\frac{(x-\mu)^2}{2\sigma^2}\right]$$

$$p(x|\omega_2) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[\frac{(x+\mu)^2}{2\sigma^2}\right]$$



For the Bayes theorem we have:

$$P(\omega_1) = \frac{p(x|\omega_1)P(\omega_1)}{p(x|\omega_1)P(\omega_1) + p(x|\omega_2)P(\omega_2)} \qquad P(\omega_2) = \frac{p(x|\omega_2)P(\omega_2)}{p(x|\omega_1)P(\omega_1) + p(x|\omega_2)P(\omega_2)}$$

We want a 2-class discriminant function $g(x)$, hence we design it as follows (removing the factors $1/\sqrt{2\pi}\sigma$):

$$g(x) \quad = \quad P(\omega_1) - P(\omega_2) \quad = \quad \frac{\exp\left[\frac{(x-\mu)^2}{2\sigma^2}\right]}{\exp\left[\frac{(x-\mu)^2}{2\sigma^2}\right] + \exp\left[\frac{(x+\mu)^2}{2\sigma^2}\right]} - \frac{\exp\left[\frac{(x+\mu)^2}{2\sigma^2}\right]}{\exp\left[\frac{(x-\mu)^2}{2\sigma^2}\right] + \exp\left[\frac{(x+\mu)^2}{2\sigma^2}\right]}$$

$$g(x) = \frac{\exp\left[-\frac{x^2+\mu^2}{2\sigma^2}\right]\exp\left[\frac{x\mu}{\sigma^2}\right] - \exp\left[-\frac{x^2+\mu^2}{2\sigma^2}\right]\exp\left[-\frac{x\mu}{\sigma^2}\right]}{\exp\left[-\frac{x^2+\mu^2}{2\sigma^2}\right]\exp\left[\frac{x\mu}{\sigma^2}\right] + \exp\left[-\frac{x^2+\mu^2}{2\sigma^2}\right]\exp\left[-\frac{x\mu}{\sigma^2}\right]}$$

The common positive factor $\exp\left[-\frac{x^2+\mu^2}{2\sigma^2}\right]$ cancels out:

$$g(x) \quad = \quad \frac{e^{\frac{x\mu}{\sigma^2}} - e^{-\frac{x\mu}{\sigma^2}}}{e^{\frac{x\mu}{\sigma^2}} + e^{-\frac{x\mu}{\sigma^2}}}$$

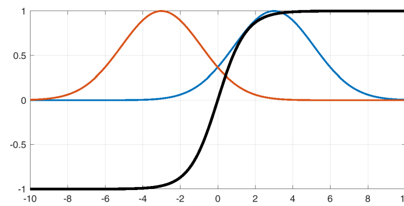
We replace x with the score $r = \mathbf{x} \cdot \mathbf{w}'$

We can absorb the factor μ/σ^2 into the norm of $\mathbf{w}'$: $\mathbf{w} = \frac{\mu}{\sigma^2}\mathbf{w}'$

We obtain

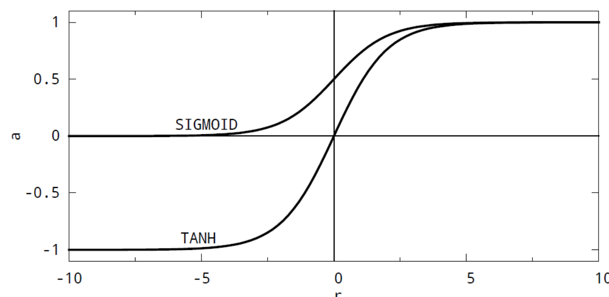
hyperbolic tangent activation, $\tanh(r)$

$$g(r) = \frac{e^r - e^{-r}}{e^r + e^{-r}} \qquad r = \mathbf{x} \cdot \mathbf{w}$$



Thanks to the hyperbolic function $tanh(r)$ we can define the **sigmoid function**:

$$\sigma(r) = \frac{1}{1+e^{-r}} = \frac{\tanh(r)+1}{2}$$



7.3.3 Differentiability of activation functions

The sigmoid is the solution of the **logistic equation**:

$$y' = y(1-y)$$

Therefore, by definition, for $\sigma(r)$ and $tanh(r)$ we have:

$$\frac{\partial \sigma(r)}{\partial r} = \sigma(r)(1-\sigma(r)) \qquad \frac{\partial \tanh(r)}{\partial r} = 1-\tanh^2(r)$$

But what about the other activation functions?

- For the **softplus**:

$$f(r) = \log(1+e^r) \qquad \frac{\partial a_j}{\partial r_j} = \frac{1}{1+e^{-r}} = \sigma(r)$$

- The "sharp" **ReLU** is not differentiable in 0, but we can use a **subderivative** to write:

$$f(r) = \max\{0, r\} \qquad \frac{\partial a_j}{\partial r_j} = \begin{cases} 0, & r \leq 0 \\ 1, & r > 0 \end{cases} = \theta(r)$$

- For the **softmax**, a_i depends on $r_1, ..., r_c$, not just r_j . So we have to write two cases:

$$(1)\ \frac{\partial a_j}{\partial r_j} = a_j(1-a_j) \qquad (2)\ \frac{\partial a_j}{\partial r_i} = -a_j a_i, \quad i \neq j$$

7.3.4 Error back-propagation algorithm

The error back-propagation algorithm it is a learning algorithm for multi-layer networks. It was discovered by Amari/Werbos/Parker/Rumelhart/Hinton/Williams from 1974 to 1986 and the name appears in Rosenblatt's "*Principles of Neurodynamics*" in 1962. Basically, it's a clever **application of the chain rule** of differential calculus:

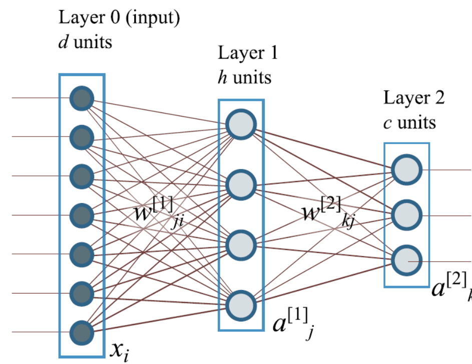
> We can perform **gradient descent** in a **distributed way** and **without actually computing** derivatives for some activation functions $(sigmoid, tanh, softmax)$
>
> → The responsibility for errors is back-propagated from the outputs back inside the network, and distributed among the hidden layers.

How it works? It starts from the chain rule:

$$\frac{\mathrm{d}f(g(x))}{\mathrm{d}x} = \left.\frac{\mathrm{d}f(y)}{\mathrm{d}y}\right|_{y=g(x)} \frac{\mathrm{d}g(x)}{\mathrm{d}x} \quad \text{Where is the "chain"?} \quad \frac{\mathrm{d}f(g(h(x)))}{\mathrm{d}x} = \frac{\mathrm{d}f(g)}{\mathrm{d}g} \nearrow \frac{\mathrm{d}g(h)}{\mathrm{d}h} \nearrow \frac{\mathrm{d}h(x)}{\mathrm{d}x}$$

which, for instance, can be used to prove that

$$\frac{\partial \sigma(r)}{\partial w_i} = \frac{\mathrm{d}\sigma(r)}{\mathrm{d}r}\frac{\partial r}{\partial w_i} = \sigma'(r)x_i = \sigma(r)(1-\sigma(r))x_i$$



The symbols:

Constants that describe the network

d	number of input units
h	number of hidden units
c	number of output units
n_w	total number of weights, $n_w = (d+1)h + (h+1)c$

Indexes used for addressing individual units

i	index for input components
j	index for hidden units
k	index for output units

Quantities coming from the training set

x_i	i-th component of input pattern $\mathbf{x}$
t_k	k-th component of target $\mathbf{t}$(can be a vector!!)

Quantities computed during the forward pass

$r^{[1]}{}_j$	net stimulus of the j-th hidden unit
$r^{[2]}{}_k$	net stimulus of the k-th output unit
$a^{[1]}{}_j$	j-th hidden unit activation value
$a^{[2]}{}_k$	k-th component of output

Weights to be optimized in learning

$w^{[1]}{}_{ji}$	weight to j-th hidden unit from i-th input unit $[(d+1)\times h]$
$w^{[2]}{}_{kj}$	weight to k-th output unit from j-th hidden unit $[(h+1)\times c]$

The loss function We can use any differentiable loss function. The default choices are:

- We use the **square** loss function $\lambda(t_k, a_k^{[2]}) = (a_k^{[2]} - t_k)^2$ for **regression**
- We use the **cross-entropy** loss (more later) for **classification**
- We use other losses for special requirements (e.g. insensitivity to small errors-noise or to large errors-outliers)

(Non convex cost function)

The objective function In theory it should be the **expected loss** (plus penalties or regularisation terms, as needed):[6]

$$J(W) = \int_{\mathscr{X}} \frac{1}{2}\frac{1}{c}\sum_{k=1}^{c} \lambda\left(t_k(\mathbf{x}), a^{[2]}{}_k(\mathbf{x})\right) p(\mathbf{x}) d\mathbf{x} \qquad \mathscr{X} \text{ (the input space) indicates all possible inputs } \mathbf{x}$$

$J(W)$ is known only through its estimate on the training set:

[6]For the next few passages we will write the objective as a function J of the set of parameters (connection weights), that is expressed as follow:

$$\text{For brevity } W = \left[\left\{w_{ji}^{[1]}\right\} \mid \left\{w_{kj}^{[2]}\right\}\right] \text{ and objective } J(W)$$

$$J(W) \approx \hat{J}(W) = \frac{1}{n}\sum_{l=1}^{n}\frac{1}{2}\frac{1}{c}\sum_{k=1}^{c}\lambda\left(t_k(\mathbf{x}_l), a^{[2]}{}_k(\mathbf{x}_l)\right)$$

X indicates a training set of n observations

This is a summation of n terms similar to the following:

$$\hat{J}_l(W) = \frac{1}{2}\frac{1}{c}\sum_{k=1}^{c}\lambda\left(t_k(\mathbf{x}_l), a^{[2]}{}_k(\mathbf{x}_l)\right)$$

To study the algorithm, we only need to consider **one pattern**

- For training **online (= by pattern)**, we will apply immediately the Δw
- For training **by epoch**, we will sum several Δw and apply them only at the end of each pass (**a training epoch**).
- For training **by batch**, we will sum several Δw and apply them only after some % of a complete pass.
- The term **minibatch** refers to batch training with small batches so that it is more similar to online.

From now on we will write just $\hat{J} = \hat{J}_l(W)$ for brevity.

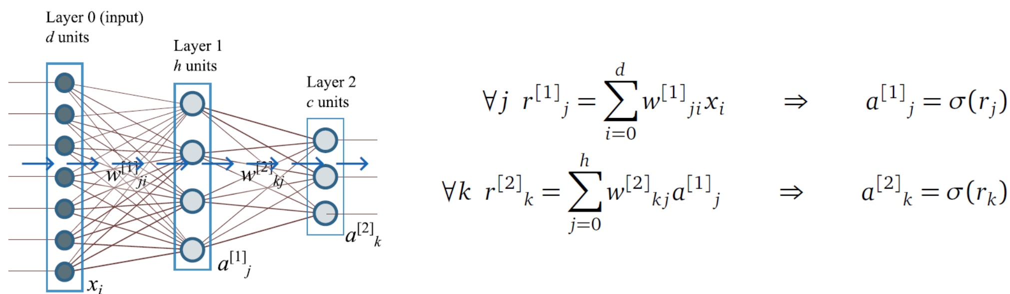
As a first application of the chain rule, we can write

$$\frac{\partial \hat{J}}{\partial w} = \frac{1}{c}\sum_{k=1}^{c}\left[\frac{\partial \hat{J}}{\partial y}\lambda(t_k, y)\right]\left[\frac{\partial}{\partial w}a^{[2]}{}_k\right] = \frac{1}{c}\sum_{k=1}^{c}\lambda'\left(t_k, a^{[2]}{}_k\right)\frac{\partial}{\partial w}a^{[2]}{}_k$$

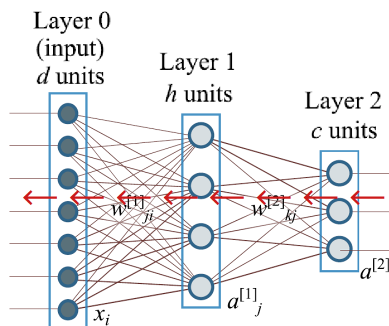
$\Rightarrow$ once we know the derivative of the loss, the rest does not depend on it – it is the same for any loss.

The operation of the multilayer perceptron is divided in **two steps**:

1. Activation **forward-propagation**



2. Error **back-propagation**



We recall here the **gradient descent**:

For all weights: $\mathbf{w}(\tau+1) = \mathbf{w}(\tau) - \eta\nabla_{\mathbf{w}}\hat{J}$

For one specific weight w: $w(\tau+1) = w(\tau) - \eta\frac{\partial \hat{J}}{\partial w}\hat{J}$

Hence to back-propagate the error and update the weights, we start from computation of partial derivatives, i.e., the gradient of the error:

- w is generically any of the weights of the network, i.e., it can stand for $w_{ji}^{[1]}$ or $w_{kj}^{[2]}$ (we will distinguish the two cases).
- We need all the components of the gradient $\nabla \hat{J}$. These are $\dfrac{\partial \hat{J}}{\partial w}$ for all possible w
- We have to compute

$$\frac{\partial a_k^{[2]}}{\partial w}$$

(depending on whether w is a $w^{[2]}$ or a $w^{[1]}$ we will have different expansions of the above expression)

- Then we can compute the **hidden-to-output weights** $w_{kj}^{[2]}$:

$$\frac{\partial \hat{J}}{\partial w^{[2]}{}_{kj}} = \frac{1}{c}\sum_{q=1}^{c} \lambda'\left(t_k, a^{[2]}{}_k\right) \frac{\partial a^{[2]}{}_q}{\partial r^{[2]}{}_q} \frac{\partial r^{[2]}{}_q}{\partial a^{[1]}{}_j}$$

We can drop all terms not depending on k, those with $q \neq k$, and consider only the one with $q = k$:

$$\frac{\partial \hat{J}}{\partial w^{[2]}{}_{kj}} = \frac{1}{c}\lambda'\left(t_k, a^{[2]}{}_k\right) \frac{\partial a^{[2]}{}_k}{\partial r^{[2]}{}_k} \frac{\partial r^{[2]}{}_k}{\partial a^{[1]}{}_j}$$

We plug in quantities known from the forward pass:

$$\frac{\partial \hat{J}}{\partial w^{[2]}{}_{kj}} = \frac{1}{c}\lambda'\left(t_k, a^{[2]}{}_k\right) \sigma'(r^{[2]}{}_k) a^{[1]}{}_j$$

If we define $\delta_k = \lambda'\left(t_k, a^{[2]}{}_k\right)\sigma'(r^{[2]}{}_k)$

i.e., $\dfrac{\partial \hat{J}}{\partial w^{[2]}{}_{kj}} = \dfrac{1}{c}\left[\ \lambda'\left(t_k, a^{[2]}{}_k\right)\sigma'(r^{[2]}{}_k)\ \right] a^{[1]}{}_j = \delta_k a^{[1]}{}_j$

What we obtain is a generalization of the "delta" term which we have seen in the delta rule by Widrow and Hoff. **Generalized delta rule** for the **hidden-to-output weights**:

$$\boxed{\Delta w^{[2]}{}_{kj} = -\eta \delta_k a^{[1]}{}_j}$$

- But there is a **problem** with the input-to-hidden weights $w_{kj}^{[2]}$: **not all terms are readily available**. Hence, we use again the chain rule to find another formulation for $\dfrac{\hat{J}}{w_{ji}^{[1]}}$

$$\frac{\partial \hat{J}}{\partial w^{[1]}{}_{ji}} = \frac{1}{2}\frac{1}{c}\sum_{k=1}^{c} \frac{\partial \lambda\left(t_k, a^{[2]}{}_k\right)}{\partial w^{[1]}{}_{ji}} = \frac{1}{c}\sum_{k=1}^{c} \lambda'\left(t_k, a^{[2]}{}_k\right) \frac{\partial a^{[2]}{}_k}{\partial r^{[2]}{}_k} \frac{\partial r^{[2]}{}_k}{\partial a^{[1]}{}_j} \frac{\partial a^{[1]}{}_j}{\partial w^{[1]}{}_{ji}}$$

Now the quantities appearing in the last equation are available, again from either the forward pass or theory:

$$\lambda'\left(t_k, a^{[2]}{}_k\right)\frac{\partial a^{[2]}{}_k}{\partial r^{[2]}{}_k} = \delta_k \qquad \frac{\partial r^{[2]}{}_k}{\partial a^{[1]}{}_j} = w^{[2]}{}_{kj} \qquad \frac{\partial a^{[1]}{}_j}{\partial w^{[1]}{}_{ji}} = \frac{\partial a^{[1]}{}_j}{\partial r^{[1]}{}_j}\frac{\partial r^{[1]}{}_j}{\partial w^{[1]}{}_{ji}} = \sigma'(r^{[1]}{}_j) x_i$$

Substituting it we have:

$$\frac{\partial \hat{J}}{\partial w^{[1]}{}_{ji}} = \frac{1}{c}\sum_{k=1}^{c}\left[\lambda'\left(t_k, a^{[2]}{}_k\right) \quad \frac{\partial a^{[2]}{}_k}{\partial r^{[2]}{}_k}\right]\left[\frac{\partial r^{[2]}{}_k}{\partial a^{[1]}{}_j}\right]\left[\frac{\partial a^{[1]}{}_j}{\partial w^{[1]}{}_{ji}}\right] =$$

$$= \frac{1}{c}\sum_{k=1}^{c}\left[\delta_k \; \sigma'(r^{[2]}{}_k)\right]\left[w^{[2]}{}_{kj}\right]\left[\sigma'(r^{[1]}{}_j) \; x_i\right]$$

We can further manipulate the expression, by first isolating the terms which do not contribute to the summation:

$$= \left[\frac{1}{c}\sum_{k=1}^{c}\delta_k\sigma'(r^{[2]}{}_k)w^{[2]}{}_{kj}\right]\left[\sigma'(r^{[1]}{}_j)x_i\right]$$

and then identifying the generalized delta for the input-to-hidden weights:

$$\delta_j = \left[\frac{1}{c}\sum_{k=1}^{c}\delta_k\sigma'(r^{[2]}{}_k)w^{[2]}{}_{kj}\right]$$

We obtain in the end the **generalized delta rule** for the **input-to-hidden weights**:

$$\boxed{\Delta w^{[1]}{}_{ji} = -\eta\delta_j x_i}$$

amazingly similar in form to that for the hidden-to-output weights

Hece, the **general algorithm** is the following one:

For each input pattern $\mathbf{x}$:

1. **Forward propagation (activation):**
   - $r_j = \sum_{i=0}^{d} w^{[1]}{}_{ji}x_i$
   - $a^{[1]}{}_j = \sigma(r_j)$
   - $r_k = \sum_{j=0}^{h} w^{[2]}{}_{kj}a^{[1]}{}_j$
   - $a^{[2]}{}_k = \sigma(r_k)$
2. **Back propagation (error and correction):**
   - $\delta_k = \lambda'\left(t_k, a^{[2]}{}_k\right)\sigma'(r_k)$
   - $\Delta w^{[2]}{}_{kj} = -\eta\delta_k a^{[1]}{}_j$
   - $\delta_j = \left[\frac{1}{c}\sum_{k=1}^{c}\delta_k\sigma'(r_k)w^{[2]}{}_{kj}\right]$
   - $\Delta w^{[1]}{}_{ji} = -\eta\delta_j x_i$
3.
   - BY PATTERN (online): immediately update $\mathbf{w} = \mathbf{w} + \Delta\mathbf{w}$
   - BY EPOCH (batch): accumulate $\Delta\mathbf{w}_{\text{tot}} = \Delta\mathbf{w}_{\text{tot}} + \Delta\mathbf{w}$ and wait until end of the epoch, then apply $\Delta\mathbf{w}_{\text{tot}}$
   - BY EPOCH with minibatches: accumulate $\Delta\mathbf{w}_{\text{tot}} = \Delta\mathbf{w}_{\text{tot}} + \Delta\mathbf{w}$ and wait until end of the minibatch, then apply $\Delta\mathbf{w}_{\text{tot}}$

Batch	Online	Minibatch

totDW = 0
for l = 1:n
  compute forward pass
  compute DW
  totDW = totDW + DW
end
W = W − eta*totDW
	for l = 1:n
	  compute forward pass
	  compute DW
	  W = W − eta*DW
	end
		fix BS = minibatch size
		totDW = 0
		for l = 1:n
		  compute forward pass
		  compute DW
		  totDW = totDW + DW
		  if l>1 **and** mod(l, BS-1)==0
		    W = W − eta*totDW
		    totDW = 0
		  end
		end

Recall:

- For the logistic sigmoid:
$$\sigma'(r) = r(1-r)$$
- For the hyperbolic tangent:
$$\tanh'(r) = 1 - r^2$$
- For the "softplus" $f(r) = \log(1+e^r)$:
$$f' = \frac{1}{1+e^{-r}} = \sigma(r)$$
- For the ReLU $F(r) = \max\{0, r\}$
$$F' \approx \begin{cases} 0, & r \leq 0 \\ 1, & r > 0 \end{cases} = \theta(r) \quad \text{(Heaviside step)}$$
- For the softmax:
$$\frac{\partial a_j}{\partial r_j} = a_j(1-a_j) \qquad \frac{\partial a_j}{\partial r_i} = -a_j a_i$$

7.3.5 Output activation (softmax) layer for classification

For multi-class classification we have to solve a c-class problem with $c > 2$, and one output unit is not sufficient. We have then to use c units, designed with **one-hot encoding**:

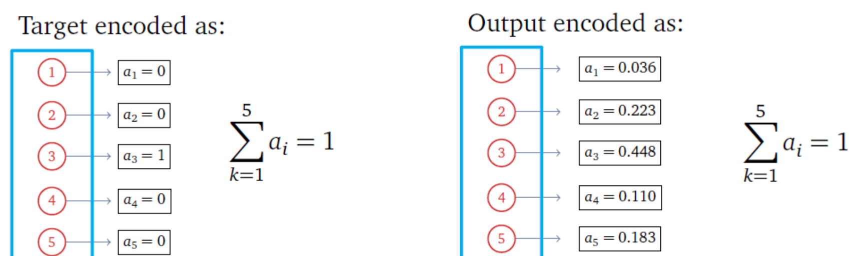
$$\begin{array}{cccccccccc} & [& 0 & 0 & \dots & 0 & 1 & 0 & \dots & 0 &] \\ \text{Unit} \rightarrow & & 1 & 2 & \dots & j-1 & j & j+1 & \dots & c & \end{array}$$

This layer of c units obeys the **probabilistic constraint**

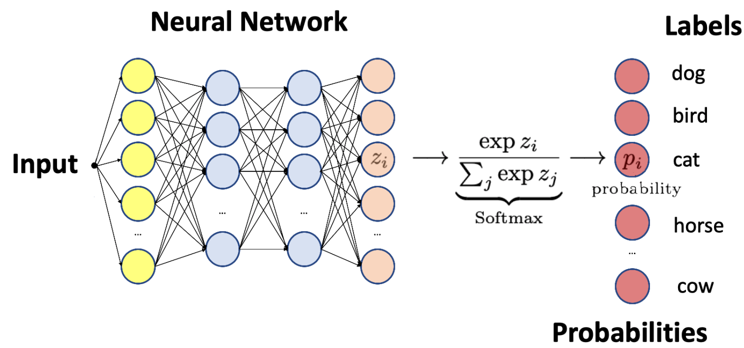
$$\sum_{j=1}^{c} a_j = 1$$

$\rightarrow$ the target $\boldsymbol{t}$ and the output $\boldsymbol{a}$ now are **vectors** of length c.

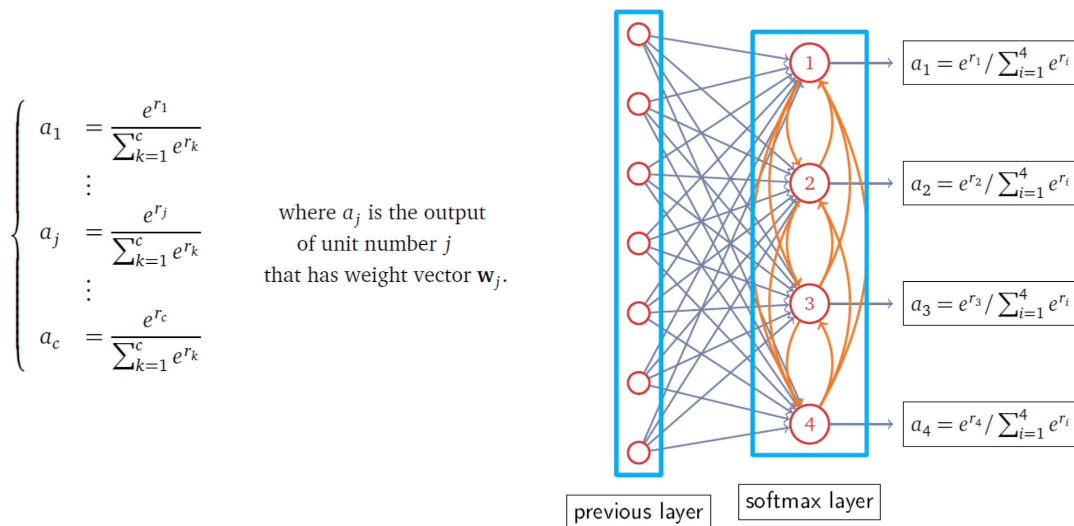
Example: Suppose to have a number of classes $c = 5$ and suppose that the output prediction is class = 3. We can have the following situation:



How we can implement the one-hot encoding? For classification, we can use **as activation a softmax layer** (different notation in the following figure):



But how does the softmax **activation layer** works? It obeys the probabilistic constraint by definition:



7.3.6 Information entropy and cross-entropy loss

Given ω a **categorical variable** with levels $\omega_1, ..., \omega_c$, Let's call ω_j a **symbol**. Symbols are generated i.i.d. with a probability mass function:

$$P = \{P_1, ..., P_c\} \qquad \text{i.e. } Pr(\omega = \omega_j) = P_j$$

Frequentist probability ("counting symbols")[7]:

$$P_j = \lim_{N \to \infty} \frac{n_j}{N}$$

- N sample size
- $n_j \leq N$ is number of times ω_j appears in the sample

Suppose we draw infinite samples of finite size N from the source (N realizations of the process ω). What is the value of N that, in the long run, ensures an average of 1 appearance of symbol ω_j? In machine learning, the answer to this question is the required number of samples, which is called **sample complexity**.

[7]Frequentist probability or frequentism is an interpretation of probability. It defines an event's probability as the limit of its relative frequency in many trials. Probabilities can be found (in principle) by a repeatable objective process (and are thus ideally devoid of opinion). This interpretation supports the statistical needs of many experimental scientists and pollsters.

Consider this:

$P_j = \frac{n_j}{N}$ e.g. 1 .125

What should be the sample size if we want to see the symbol exactly once (on average?)

$P_j = \frac{1}{N/n_j}$ e.g. 8 1

The sample size that (in the long run) guarantees 1 observation is the denominator:

$$\frac{N}{n_j} = \frac{1}{P_j}$$

How much information is carried by a symbol?

The **information carried by symbol** ω_j is the number of digits needed to write

$$\frac{1}{P_j} \quad \text{(minus one)}$$

(Information measures the **unexpectedness** of a symbol)

Hence the number of zeros required to represent a value is

$$I_j = \log_2 \frac{1}{P_j} = -\log_2 P_j$$

The number of digits is the **number of zeros + 1**. The unit is **bits** (or shannon, Sh).

For simplifying calculations we adopt radix e
so the actual unit is the nat (1 bit $=\log 2 \approx 0.693$ nat).
Recall that **all logarithm bases > 1 are equivalent up to a multiplicative constant**:
$\log_a X = \log_b X \times (1/\log_b a)$

So, the process ω generates a different symbol at each realization. These symbols occur with different probabilities → different quantities of information. Finally, the overall source can be characterized by the information of all the symbols that it produces. We now have to define the information entropy of the source:

The **information entropy** of a source of symbols is the **expected information** of the symbols:

$$H(P) = \mathbb{E}\{\omega\} = \sum_{j=1}^{c} P_j I_j = -\sum_{j=1}^{c} P_j \log_2 P_j$$

(Entropy measures the **unpredictability** of a source)

We can now define **cross-entropy**, i.e. a measure of the information from one source when the information of symbols is measured for a different source.

Two examples:

1. Digital communication system with $P = \{P_1, \ldots, P_n\}$ but are encoded according to an estimated probability $Q = \{Q_1, \ldots, Q_n\}$ different from P.
2. The symbols are the targets in a classification task, but they are estimated by a learner.
   (In this case probabilities are **conditional**, functions of the inputs x: $P = \{P_1(x), \ldots, P_c(x)\}$, $Q = \{Q_1(x), \ldots, Q_c(x)\}$, $P_j(x) = \Pr(\omega = \omega_j | x)$.)

Cross entropy:

$$H(P|Q) = -\sum_{j=1}^{c} P_j \log Q_j \ .$$

Cross-entropy, alternate form (by summing and subtracting $P_j \log P_j$ to each term in the summation):

$$H(P|Q) = -\sum_{j=1}^{c} P_j \log P_j - \sum_{j=1}^{c} P_j \log \frac{Q_j}{P_j} = H(P) + D(P|Q)$$

$D(P|Q)$ is the **Kullback-Leibler divergence** from P to Q, the surplus of information needed to describe the source P when using the source Q.

(Cross entropy measures the unpredictability of a source, when we can measure a different (e.g., approximate) source)

Now, how to define the required **cross-entropy loss function**? Consider the following situation:

- binary classification task, $c = 2$
- neural network with one sigmoidal unit at its output
- Targets $t \in \{0, 1\}$
- We can interpret these as the "probabilities" of being of positive class (actually "certainties")
- Given an input x
  - $P_1(x)$ = probability of x in class 1 = $t(x)$
  - $P_0(x)$ = probability of x in class 0 (or NOT in class 1) = $1 - t(x)$:

  $P(x) = \{P_0(x), P_1(x)\} = \{\ 1 - t(x)\ ,\ t(x)\ \}$
- Same (but real-valued) for network outputs a:
  $Q(x) = \{Q_0(x), Q_1(x)\} = \{\ 1 - a(x)\ ,\ a(x)\ \}$

Then, the loss function can be expressed in function of the cross-entropy between output and target:

$$\begin{aligned} \lambda(t, a) &= H(P, Q) \\ &= -\ P_0 \log Q_0\ -\ P_1 \log Q_1 \\ &= -(1 - t)\log(1 - a) - t \log a \end{aligned}$$

remarkably simple to compute:
one of the two terms always vanishes
(the first when $t = 1$, the second when $t = 0$)

7.3.7 Cross-entropy objective with sigmoid activation

As an objective function for classification task, we can derive from the ideal cross-entropy function the expected cross-entropy:

Ideal:

$$f = \int \lambda(t(x), a(x)) p(x) \, dx$$

Empirical:

$$\hat{f} = \frac{1}{n} \sum_{l=1}^{n} \lambda(t(x_l), a(x_l)) = \frac{1}{n} \sum_{l=1}^{n} (-(1 - t(x_l)) \log(1 - a(x_l)) - t(x_l) \log a(x_l))$$

Measures how different the two probability mass functions are, in expectation (on average)

The differentiation can be made as follows:

By the chain rule: $$\frac{\partial \lambda(t, a(w))}{\partial w} = \frac{\partial \lambda(t,a)}{\partial a}\frac{\partial a(w)}{\partial w}$$

Derivative of λ with respect to a

$$\frac{\partial \lambda(t,a)}{\partial a} = -(1-t)\frac{\partial}{\partial a}\log(1-a) - t\frac{\partial}{\partial a}\log a = \frac{1-t}{1-a} - \frac{t}{a} = (a-t)\boxed{\frac{1}{a(1-a)}}$$

We can observe that the second factor is $\frac{1}{\text{derivative of a sigmoid}}$

From the back propagation algorithm we recall that:

For each input pattern **x**:

① **Forward propagation (activation):**

- $r_j = \sum_{i=0}^{d} w^{[1]}{}_{ji} x_i$
- $a^{[1]}{}_j = \sigma(r_j)$
- $r_k = \sum_{j=0}^{h} w^{[2]}{}_{kj} a^{[1]}{}_j$
- $a^{[2]}{}_k = \sigma(r_k)$

② **Back propagation (error and correction):**

- $\delta_k = \boxed{\lambda'(t_k, a^{[2]}{}_k)}\ \sigma'(r_k)$
- $\Delta w^{[2]}{}_{kj} = -\eta \delta_k a^{[1]}{}_j$
- $\delta_j = [\frac{1}{c}\sum_{k=1}^{c} \delta_k \sigma'(r_k) w^{[2]}{}_{kj}]$
- $\Delta w^{[1]}{}_{ji} = -\eta \delta_j x_i$

And when the loss is cross-entropy we obtain:

$$\frac{\partial \lambda(t,a)}{\partial r} = \frac{\partial \lambda(t,a)}{\partial a}\frac{\partial a}{\partial r} = (a-t)\frac{1}{a(1-a)}\, a(1-a) = (a-t)$$

→ **the non-linearity (sigmoid)** in the output **do not contribute** to the gradient (*exp* and *log* cancel out each other)

7.3.8 Cross-entropy objective with softmax activation

We recall here the derivative of the **softmax**:

$$\frac{\partial a_j}{\partial r_k} = \begin{cases} a_j(1-a_j) & j=k \\ -a_j a_k & j \neq k \end{cases}$$

(remember that each activation a_i depends on all other activations a_k)

Then, the derivative of cross-entropy with softmax is:

$$\lambda(\mathbf{t},\mathbf{a}) = -\sum_{j=1}^{c} t_j \log a_j$$

$$\frac{\partial \lambda(\mathbf{t},\mathbf{a})}{\partial r_k} = -\sum_{j=1}^{c} t_j \frac{\partial \log a_j}{\partial r_k} = -\sum_{j=1}^{c}\frac{t_j}{a_j}\frac{\partial a_j}{\partial r_k} = -\frac{t_j}{a_k}\frac{\partial a_k}{\partial r_k} - \sum_{j\neq k}\frac{t_j}{a_j}\frac{\partial a_j}{\partial r_k} = -\frac{t_j}{a_k}a_k(1-a_k) + \sum_{j\neq k}\frac{t_j}{a_j}a_j a_k = \dots$$

$$\dots = -t_j(1-a_k) + \sum_{j\neq k} t_j a_k = -t_j + t_j a_k + \sum_{j\neq k} t_j a_k = -t_j + \left[t_j a_k + \sum_{j\neq k} t_j a_k \right] = \dots$$

$$\dots = -t_j + a_k \underbrace{\sum_j t_j}_{\to 1} = a_j - t_j$$

→ this is **extremely simple and the same as the sigmoid**

Neural Networks

Report for the Machine Learning course, EMARO

Davide Lanza
EMARO+ M2
Genoa, Italy
davidel96@hotmail.it

***Abstract*—In this report we will analyze MATLAB's own neural network tools, contained in a library called Neural Networks Toolbox.**

***Index Terms*—Machine Learning, Neural Networks, Multilayer Networks, Classification, Regression, MATLAB**

INTRODUCTION

Thanks to the exponential increase of data available on the Internet, the volume of information available for perform machine learning task grows incredibly. Learning-from-data tasks are an important component in information extraction and for predictive tasks. In this report, we focus on neural networks, a function fitting tool suitable for both classification and regression tasks. There is proof that a fairly simple neural network can fit any practical function, and this makes it extremely powerful. MATLAB's Deep Learning Toolbox provides a framework for designing and implementing deep neural networks with algorithms, pretrained models, and apps.

I. FEEDFORWARD MULTI-LAYER NETWORKS

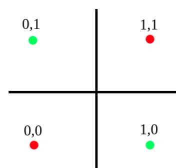
We defined a set of o input vectors ($\dim \boldsymbol{x}_i = f$) as columns in a matrix

$$\mathcal{I}nputs = [\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_o]$$

and another set of o target vectors ($\dim \boldsymbol{t}_i = c$) so that they indicate the classes to which the input vectors are assigned

$$\mathcal{T}argets = [\boldsymbol{t}_1, \boldsymbol{t}_2, ..., \boldsymbol{t}_o]$$

Notice that the number of rows c in $\mathcal{T}$ is the number of classes to be recognized in the classification task, the number of rows f in $\mathcal{I}$ is the number of features of every observation in the dataset and the number of columns o is the number of observations. For binary classification problems $c = 2$, there are only two classes (e.g. "True" or "False"), then we set each scalar target value to either 0 or 1, indicating which class the corresponding input belongs to. For example, you can define the two-class exclusive-or classification problem as follows:



$$\mathcal{I} = [(0,0)\ \ (0,1)\ \ (1,0)\ \ (1,1)] = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

$$\mathcal{T} = [False\ True\ True\ False] = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

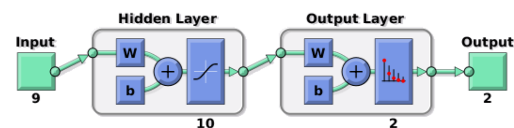


Figure 1: Neural network $n = 10$

Using the MATLAB neural network pattern recognition app `nprtool`, we trained a network made by n hidden neurons on the breast cancer dataset[1] provided with the toolbox ($o = 699, f = 9, c = 2 \rightarrow$ the two associated categories with each input vector are "benign" or "malignant"). The results for a single-hidden layer network with $n = 10$ (train on 70% of the dataset, validation on 15% and test on 15%) are shown in Figure 2

The accuracy results for different settings of the network w.r.t. the breast cancer dataset are shown in Table 1.

n	Train	Validation	Test	α_{train}	α_{val}	α_{test}
5	70%	15%	15%	98,16%	95,24%	95,24%
10	70%	15%	15%	96,52%	98,09%	98,09%
10	80%	10%	10%	97,32%	97,14%	92,86%
20	70%	15%	15%	97,75%	96,19%	95,23%
100	70%	15%	15%	98,77%	98,09%	93,33%

Table 1: Breast cancer overall accuracy α results

We tested the network as well on the thyroid dataset[2] ($o = 7200, f = 21, c = 3 \rightarrow$ the three associated categories with each input vector are "normal, not hyperthyroid", "hyperfunction" and "subnormal functioning"). The accuracy results for different settings of the network are shown in Table 2.

[1] Data donated by Olvi Mangasarian, available from the UCI Machine Learning Repository: Murphy,P.M., Aha, D.W. (1994). *UCI Repository of machine learning databases* http://www.ics.uci.edu/ mlearn/MLRepository.html. Irvine, CA: University of California, Department of Information and Computer Science.

[2] From Garavan Institute. T.his data is available from the UCI Machine Learning Repository.

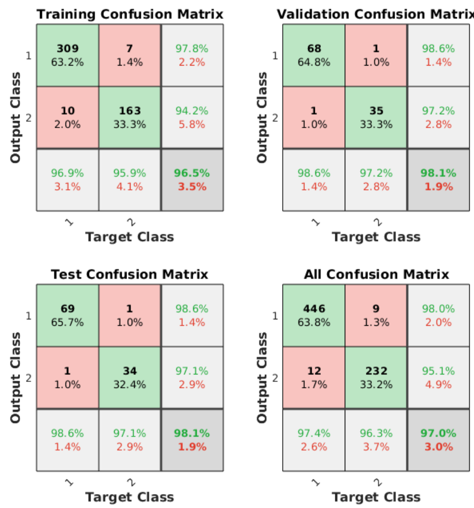


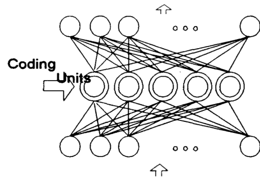
Figure 2: Breast cancer training results confusion matrices ($n = 10$, train 70%, validation 15%, test 15%)

n	Train	Validation	Test	α_{train}	α_{val}	α_{test}
5	70%	15%	15%	94,56%	92,40%	93,70%
10	70%	15%	15%	94,11%	94,26%	93,98%
10	80%	10%	10%	94,45%	94,31%	93,06%
20	70%	15%	15%	93,12%	95,00%	93,98%
100	70%	15%	15%	92,52%	92,96%	92,50%

Table 2: Thyroid overall accuracy α results

II. Autoencoder

The simplest autoencoder network is a multi-layer perceptron neural network which has one input layer, one hidden layer ($n < o$), and one output layer ($c = o$):



The autoencoder is trained using the same pattern as both the input and the target:

$$\mathcal{I} = \mathcal{T}$$

Note that in this case we don't have any classes or other mapping to learn. This is a special case of unsupervised training. In fact, it is sometimes called "self-supervised", since the target we use is the input pattern itself.
We trained the multilayer perceptron as an autoencoder for the MNIST data [1]. MATLAB provides a separate function that is used as follows:

```
myAutoencoder  =  trainAutoencoder(myData,n);
myEncodedData  =  encode(myAutoencoder,myData);
```

The autoencoder learns an internal, compressed representation for the data thanks to its hidden layer. Since it is unsupervised learning, the goal is to learn the different MNIST classes from unlabeled data (dataset extract in Figure 3).

Figure 3: Full dataset extract

The experiments has been performed with reduced dataset that were subsets of different classes $x_1, x_2, ..., x_{10}$. Each reduced training set was composed only by 2 classes (e.g., "1" and "8" digits) hence the number of hidden layers was $n = 2$, and each hidden layer had to "encode" one class (e.g., "1" and "8"). The results obtained from a dataset composed only by two "easy-to-distinguish" digits "1" and "8" (Figure 4) are shown in Figure 5.

Figure 4: "1" & "8" dataset extract

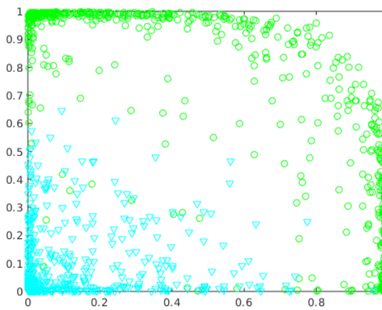


Figure 5: "1" & "8" dataset autoencoder results

Then, the $n = 2$ autoencoder has been tested on a dataset composed only by two "difficult-to-distinguish" digits "5" and "6" (Figure 6) are shown in Figure 7.

Figure 6: "5" & "6" dataset extract

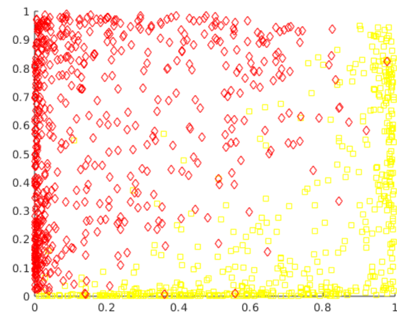


Figure 7: "5" & "6" dataset autoencoder results

The autoencoder can be trained in order to encode more classes. In order to still be able to plot the results, we implemented a 3 hidden units autoencoder and we test it for three digits encoding (see Figure 8).

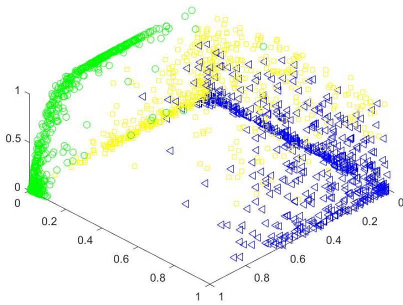


Figure 8: "0" & "1" & "5" dataset extract

III. Conclusions

Regarding the first task, the results match what we were expecting in the first place form the theory: an increasing number of neurons leads to worse performances on smaller dataset, but can bring some improvements in bigger ones, of course only until a certain threshold .

Regarding the second task, the results show linear separation for easy-to-distinguish classes (e.g. "1" and "8" digits in Figure 5). Instead, with difficult-to-distinguish pairs (e.g. "3" and "6" digits in Figure 7) the results are way worse.

Hence, train a neural network is a complex task, that does not have a unique, structured, technique. In fact, different situations, tasks, architectures and datasets lead to different solutions, due to the various differences underneath. The weight initialization, the value of the biases, the random dataset fold splitting that leads to randomly assembled training datasets result leads to a various range of results. To obtain accurate results from neural networks, re-train and parameter modification is mandatory, as shown by the results reported here.

References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998. DOI: 10.1109/5.726791.

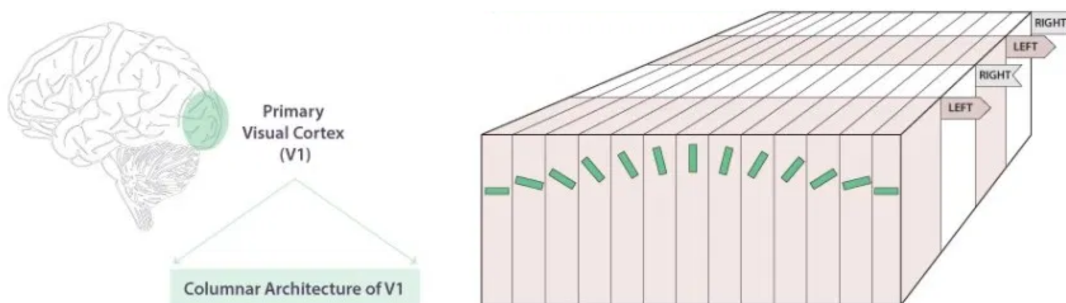
Chapter 8

Deep learning

Main reference:
Ian Goodfellow, Yoshua Bengio and Aaron Courville. *Deep Learning*. MIT Press (2016)
Freely available (but not for download) at http://www.deeplearningbook.org/

8.1 Depth and internal representation

Hubel and Wiesel placed electrodes in animals brains (visual cortex). They discovered the columnar organization of neurons:



Each layer in a cortical colum extracts **features** from the input it receives from the previous layer. These features are more and more abstract:

Edges → Simple shapes → Composite shapes → Eyes, mouths, noses → ... → Grandmother

(The Grandmother Cell hypothesis)

Neural networks stores **internal representations** in hidden layers, but to develop such a **hierarchy** may require many layers (**deep networks**).

But regarding learning, which are the **limits** of a multi-layer network?

- Error back-propagation does not work well with very deep structures

  Vanishing/exploding gradient phenomenon: at each layer, the backpropagated components of the gradient become exponentially smaller or larger (sums of sums of sums of sums of ... , concentration effects).

→ To avoid the problem: use shallow networks (theoretically sufficient)

Nevertheless, there is a **representational advantage in depth** and in the 80s and early 90s some works proved that:

> "*Some logical functions, that can be implemented with a depth of k layers, require exponentially more units if reduced to* $k-1$ *layers*"

In the 2010s:

> "*Dependent inputs (variables) need very deep networks*" (P. Baldi)

And more recently:

> *"Quantitative bounds for neural networks similar to those for logical functions. For a given accuracy $\leq \epsilon$ we have:"*
>
> $$M \text{ units in } k \text{ layers } \Rightarrow O(2^M) \text{ units in } k-1 \text{ layers}$$

But how to go deep? How to train a very large and deep network? There are several strategies:

- Reduce the number of parameters (weights) by forcing some special connectivity pattern
- Randomly remove some parameters during learning
- Pool values together, chose only a value from each pool (e.g. the maximum)
- Use unsupervised learning on most layers except the last ones

Inductive bias: some hypotheses in the hypothesis space are made less probable than others
Strong inductive bias: some hypotheses have **zero** probability

The **deep learning structures** which are commonly used are the following ones:

- Convolutional neural networks
- Restricted Boltzmann Machines
- Autoencoders
- Fully-connected multi-layer perceptrons (usually as the final stage)
  These basic networks are **stacked**: each acts as an individual layer in a deep hierarchical structure

8.2 Convolutional neural networks

$x(t)$ a signal, $h(t)$ another signal (e.g., filter response)

$$(x * h)(t) = \int_{-\infty}^{+\infty} x(\tau)h(t-\tau)d\tau$$

x_i a discrete signal, h_i another discrete signal

$$(x * h)_i = \sum_{j=-\infty}^{+\infty} x_j h_{i-j}$$

h usually has limited support (is "short" or local). If h is even, it does not matter whether it is reversed or not (if not, it is more properly a cross-correlation) ... but since we are learning it, it does not matter!

The convolution is a dynamic system's with response h to an input signal x, or it is the output of a filter applied to a signal. The signal h used to apply the transformation is often called a **kernel**.

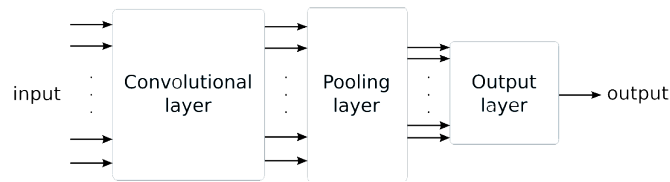
Convolutional neural networks (CNNs) are translation-independent neural networks. They are used especially when dealing with large-dimensionality, natural signals (full images, video, sound from the real world).

CNN = Multi-layer perceptron with two strong simplifications:

1. A fixed structure and a fixed, reduced connectivity pattern
2. Weight on the hidden layer are tied or shared, i.e., they are the same few values, but replicated across the layer

and some specializations:

1. Use of pooling
2. Output nonlinearity is usually ReLU



8.2.1 CNN: Convolutional layer

In a convolutional layer each unit is connected to a **limited-size receptive field** (e.g. for an image it can be a 3×3 pixel patch) and each unit has the same weights as the others (**shared weights**).

- Mathematically, the layer implements a **discrete convolution** (**followed by nonlinearity**) where the convolution **kernel is the weights, learned from the data**:

$$r_j = \sum_{i=1}^{d} x_{i+j} w_{3-i+j} \qquad a_j = f(r_j)$$

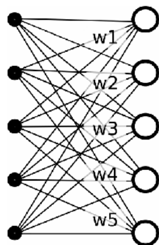
- As a signal processor, the layer applies a finite impulse response filter w to a sampled signal x

The **nonlinearity** that follows the convolution is **usually a ReLU**, and it works as a **"Detector"**.

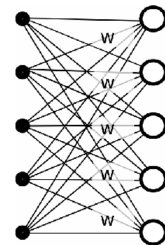
There is an **optimization advantage of sharing weights**: for 3×3 patches from a 100×100 image, we optimize only 9 weights instead of 100000!

There are different kinds of layer connections, in the follow images we compare 3 of them with the fully connected on:

Fully connected network (not a CNN)

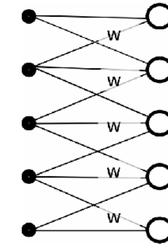


Weight sharing



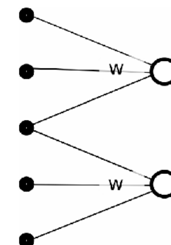
Here W is the kernel

Localized kernels



Connections don't span the whole signal

Downsampling (stride)



The kernel is applied intermittently, not on all inputs
Also possible to have non-contiguous elements (dilation)

More complex arrangements, but using the same concepts, are used for multidimensional signals (e.g., images: 2D; color images: 3D; video: 3D)

8.2.2 CNN: Pooling layer

A pooling layer is used after convolutional layer to reduce dimensionality. It works on a set of units (e.g., 4) and **replace the full information** (e.g., 4 output values) **with one scalar**:

- *Max pooling*: take the maximum
- Other types of pooling: *Average pooling* or others...
- Pooling works well and acts as a **subsampling or downsampling** possibly non-linear, e.g., with a *max* function for the max pooling or an *average* function for the average one.

- Pooling introduces **invariance to small changes**, that is one of the main features of CNNs:

8.2.3 CNN: Output layer

The output layer is a **fully-connected layer** [1] which takes as inputs all the outputs of the pooling layer. Its usual **neuron model** is:

$$r = \boldsymbol{x} \cdot \boldsymbol{w}$$
$$a = a(r)$$

The **activation nonlinearity** of the output layer can be *ReLU*, *sigmoid* or *tanh*. Usually for deep networks **ReLU** is used (so there is **no positive saturation**).

An example of full CNN architecture is the following one (**AlexNet**[2]):

[1]The output fully-connected layer is not part of the "previous CNN", in fact it can be substituted or retrained adapting the underlying trained CNN to different classification task

[2]AlexNet is the name of a convolutional neural network, designed by Alex Krizhevsky, and published with Ilya Sutskever and Krizhevsky's PhD advisor Geoffrey Hinton. AlexNet competed in the ImageNet Large Scale Visual Recognition Challenge on September 30, 2012. The network achieved a top-5 error of 15.3%, more than 10.8 percentage points lower than that of the runner up. The original paper's primary result was that the depth of the model was essential for its high performance, which was computationally expensive, but made feasible due to the utilization of graphics processing units (GPUs) during training.

8.2.4 Training a CNN

Training a CNN requires means training a **huge number of weights**, and it needs huge training sets. Training is operated normally by **stochastic gradient descent using minibatches**.

8.2.5 Regularization methods

Regularization is a way to **simplify the objective function** so as to make it smoother and with less local minima (easier to train). There are different methods:

- **Early stopping**: measure validation error during training; stop when it starts to increase
- **Reducing network capacity**: Smaller networks (number of layers, layer size), shared weights, ...
- **Weight decay**:

  $$objective + h||w||_2 \qquad objective + h||w||_1$$

  h is a coefficient to balance the penalty term with respect to the main objective function

A particular class of regularization methods are the **stochastic regularization methods**, which change at each training step and work on average:

- **Dropout**: at each training step, some units are "dropped out" (removed) with some probability p. Usually $p = 0.5$ (except for input units). In inference, all nodes are used, but outputs are scaled by $1 - p$
- **DropConnect**: individual connections are dropped out rather than whole units
- **Stochastic pooling**: in pooling, units in the pool are randomly chosen
- **Augmented data**: artificially created data are added to the training set. These are usually obtained by randomly deforming the original data or by adding noise to them.

A alternative approach is to use **unsupervised layers**. In the architecture then, we have to put:

Cascaded networks of unsupervised layers trained one after the other
+
Final classification layer

→ after unsupervised training layer-by-layer, the whole network is finally trained with a few iterations of **error back-propagation**.

8.3 Information bottleneck & unsupervised learning

Now, consider this example. We have to move in a new house, but in the new house the library is way smaller that in the previous house. Hence, we have to save only the **most relevant part** of the information (the most relevant book) and throw away the others:

This is basically the problem called **information bottleneck**.[3]

[3] *cfr.* DJ Strouse and David Schwab, *The deterministic information bottleneck*, 2016 http://arxiv.org/abs/1604.00268 and the related article https://www.inference.vc/representation-learning-and-compression-with-the-information-bottleneck/

In order to **decide which aspects of observed data are relevant information**, and what aspects can be thrown away, we can use two balancing criteria:

- **compactness of representation**, measured as the *compressibility*: number of bits needed to store the representation, and
- **information the representation retains** about some behaviorally relevant variables.

But this pretty much assumes a supervised learning setting, that is, it is assumes we know what the behaviorally relevant variables are and how they are related to observed data, or at least we have to have data to learn or approximate the joint distribution $p(x, y)$ between observed and relevant variables. The question remains how could one possibly do something like this in the unsupervised, or at least semi-supervised setting. How we can solve it:

- Using statistics, maximizing entropy
  – Coding theory
  – Stochastic complexity and minimum description length
- Minimizing errors (e.g., $||\boldsymbol{t} - \boldsymbol{a}||^2$):
  – Autoencoders
  – PCA
  – Rate-distortion theory

There two most popular **unsupervised learning techniques** that try to solve this problem are:

- **Maximum likelihood**

  Maximum (marginal) likelihood looks at the marginal distribution of observed data $p(x)$, and tries to build a model $q(x)$ that approximates this in the $KL[p|q]$ sense. The representations we then use are often extracted from the conditional distributions of some hidden variables y conditioned on observations $q(y|x)$.

  An important fact to notice is that the same marginal model $q(x)$ can be represented as the marginal of an infinite number of joint distributions $q(x, y) = q(y|x)q(x)$. You can represent a Gaussian $q(x)$ as just a Gaussian, without hidden variables, as the end-point of a Brownian motion, or even as the output of some weird nonlinear neural network with some noise fed in at the top. Therefore, unless we have some other assumptions, **the likelihood alone can't possibly tell apart these different representations**.

- **Autoencoders**

  The same criticism to maximum likelihood applies to other unsupervised criteria, for example, autoencoders or denoising autoencoders (DAE).

  DAE learns about the data distribution $p(x)$, and it will build representations that are useful to solve the denoising task. It is related to the information bottleneck in that it solves the same trade-off between compression and retaining information.

  However, instead of retaining information about behaviorally relevant variables, it tries to retain information about the data x itself. This is really the key limitation, as **it cannot (without further assumptions) tell which aspects of the data are behaviorally relevant** and which aren't. It has to assume that everything is equally relevant.

To summarise, a key limitation of using unsupervised criteria for representation learning is the following:

> *Maximum likelihood (or autoencoding), without strong priors, is the same as assuming that every bit, each pixel, of your raw observed data is equally relevant behaviourally.*

Hence, to learn behaviorally useful representations in a fully unsupervised way, **we need priors and assumptions** about the representation itself. **Sometimes** these priors are **encoded** in the architecture of neural networks, sometimes they can be **incorporated** more explicitly.

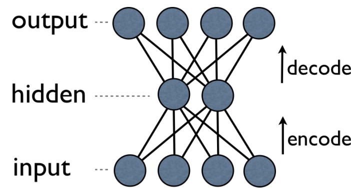
But let's analyze more in depth the second structure we introduced: the autoencoders.

8.3.1 Autoencoders

An autoencoder is a special case of a multi-layer perceptron charcterized by two aspects:

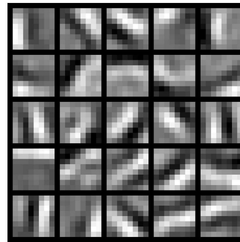
- **Structure**: number of units in the input layer = number of units in the output layer > number of hidden units
- **Learned task**: an autoencoder is trained to approximate the identity function (= replicate its input at the output)

An autoencoder is **not a classifier**!



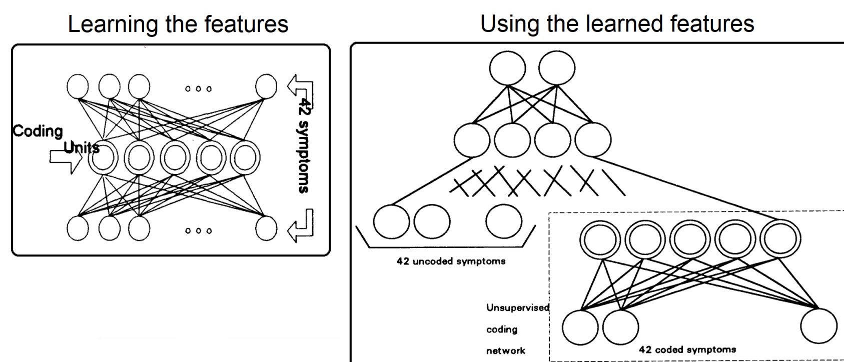
What is interesting in autoencoders is not the output value (is a lower-quality approximation to the input) but the **pattern present on the hidden layer**. Since we don?t use any target (the target coincides with the input), the autoencoder task is **unsupervised** (sometimes termed "self-supervised").

Example: learned features from a set of images:



Example: recognizing handwritten digits

Another example of an autoencoder for learning features from symbolic data is the one used to **diagnose Lyme disease** from patient records. The problem in this case is that many features (observed signs and symptoms) are binary and very sparse. How to learn them:



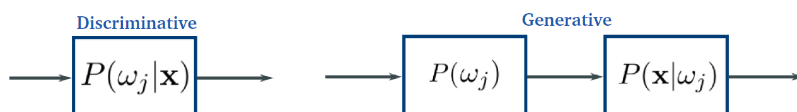
It has been proved that **an autoencoder with linear activations learns the principal components**. This is because the objective is the mean squared reconstruction error of a lower-rank representation, the same as PCA.

8.3.2 Denoising autoencoders (DAEs)

The aforementioned DAEs are trained with **random values (noise)** added to the training set and, in training, noise is averaged out. In operation, the resulting network is more stable ("robust") to noise.

8.3.3 Restricted Boltzmann Machines

A restricted Boltzmann machine (RBM) is a generative **stochastic artificial neural network that can learn a probability distribution** over its set of inputs. In figure we can see the difference between a discriminative and a generative (this case) structure:

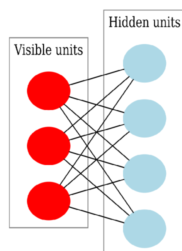


RBMs were initially invented under the name Harmonium by Paul Smolensky in 1986, and rose to prominence after Geoffrey Hinton and collaborators invented fast learning algorithms for them in the mid-2000. RBMs have found applications in dimensionality reduction, classification, collaborative filtering, feature learning, topic modelling and even many body quantum mechanics. They can be trained in either supervised or unsupervised ways, depending on the task.

As their name implies, RBMs are a **variant of Boltzmann Machines**. The "**unrestricted**" BMs have the following characteristics:

- binary-valued units
- bi-directional connections
- symmetric weight (equal in the two directions): a pair of nodes from each of the two groups of units (commonly referred to as the "visible" and "hidden" units respectively) may have a symmetric connection between them
- general topology (feedback possible)
- no connections between nodes within a group
- "**unrestricted**" Boltzmann machines **may have connections between hidden units**.

The **restricted** version has the limitation that its topology must be a **bipartite graph**, and this makes it more tractable[4]



Called $\boldsymbol{v} = [v_i]$ and $\boldsymbol{h} = [h_i]$ the visible and hidden unit activation values, $w_{i,j}$ the weight between v_i and h_j, a_i and b_i the biases of visible and hidden units, we can then we can define an "**energy**" that is:

$$E(\mathbf{v},\mathbf{h}) = -\sum_i a_i v_i - \sum_j b_j h_j - \sum_i \sum_j v_i w_{i,j} h_j$$

Then, the **probability of any possible network state** is

$$P(\mathbf{v},\mathbf{h}) = \frac{1}{Z} e^{-E(\mathbf{v},\mathbf{h})} \qquad \text{with } Z \text{ partition function (normalizer)}$$

[4]This restriction allows for more efficient training algorithms than are available for the general class of Boltzmann machines, in particular the gradient-based contrastive divergence algorithm.

Since intra-layer connections are not present, probability of activation of one unit does not depend on that of other units in the same layer – only in the other layer:

$$P(v_i = 1|\mathbf{h}) = \frac{1}{1 + e^{-(a_i + \sum_j w_{i,j} h_j)}} \qquad P(h_j = 1|\mathbf{v}) = \frac{1}{1 + e^{-(b_j + \sum_i w_{i,j} v_i)}}$$

To train a RBM we use an algorithm called **contrastive divergence** which uses random sampling from the probabilities (computed as above):

- Apply one input
- Compute probability $P(\mathbf{h}|\mathbf{v})$ - Sample from it to generate hidden configuration
- Compute a positive update step $\Delta\mathbf{w}^+ = \mathbf{v}\mathbf{h}^T$ (**outer product**)
- Generate one possible input $\mathbf{v}'$ from the hidden configuration
- Compute probability $P(\mathbf{h}'|\mathbf{v}')$
- Compute a negative update step $\Delta\mathbf{w}^- = \mathbf{v}'\mathbf{h}'^T$
- Apply update: $\mathbf{w} \leftarrow \mathbf{w} + \eta(\Delta\mathbf{w}^+ - \Delta\mathbf{w}^-)$

This does not optimize any explicit objective function!

8.4 Deep Neural Networks

A **Deep Belief Network** (DBN) is a sequence of **Restricted Boltzmann Machines** (RBMs), in which **each RBM can be trained independently** of the following ones (greedy strategy) and the **last layer can be a classifier**.

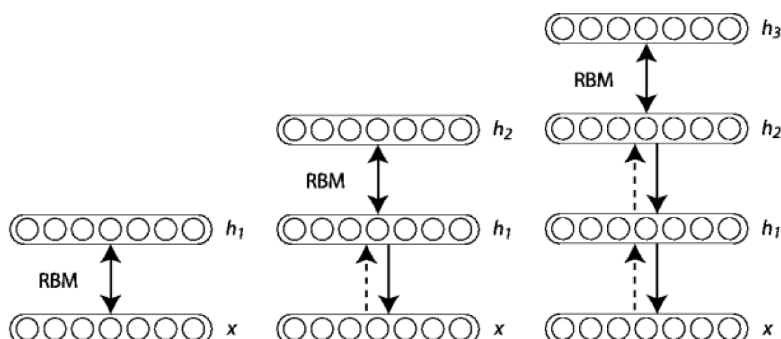
Deep networks can be built out of RBMs, but also out of **autoencoders**. Autoencoders have been found to be less insensitive to random noise.

Training RBMs of large size is not simple, but there are tricks to make the task easier.

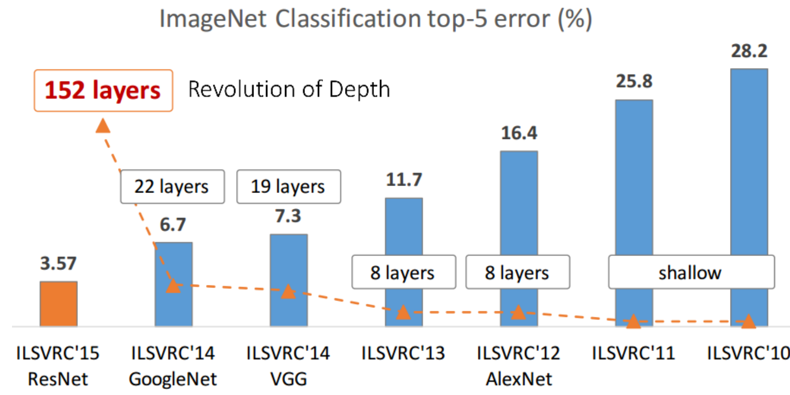
8.4.1 Examples of successful deep networks

Deep Belief Networks (2006) Fully connected multi-layer perceptrons whose layers are trained as "restricted Boltzmann machines" or as "autoencoders". It is the first model of the recent deep learning wave.

KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. 2012. p. 1097-1105.

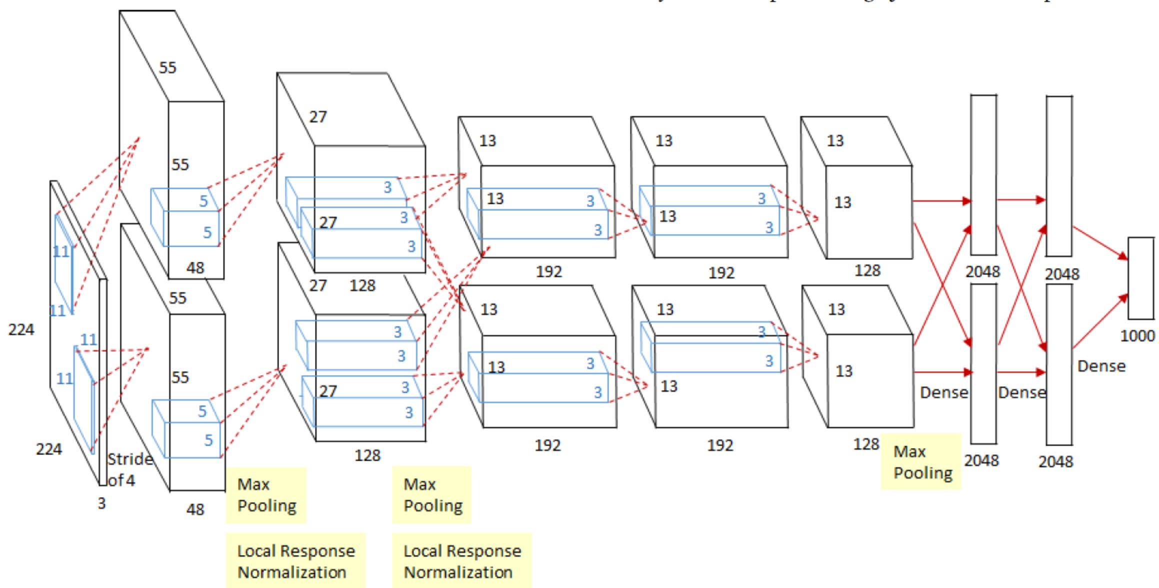


ImageNet competition ImageNet is an online resource with millions of images tagged with nouns from the WordNet ontology. It is also a competition with challenges that were consistently won by deep neural networks:



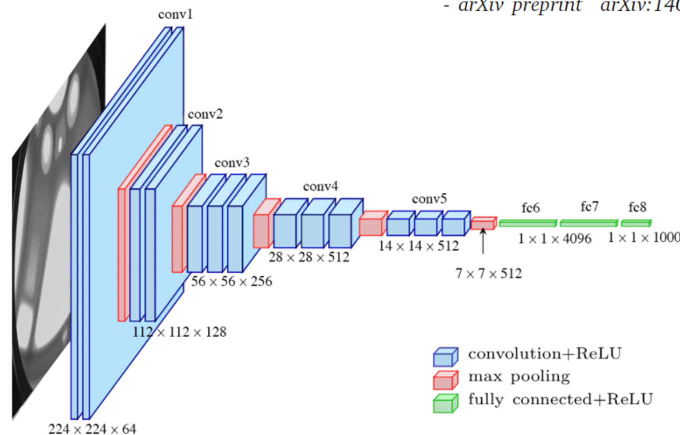
AlexNet (2012) This convolutional network considerably improved the current state-of-the-art performance in the ImageNet competition using 60 million parameters, 650 000 neurons, 5 convolutional layers and 3 fully connected layers with 1000-unit softmax as the output:

KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. Imagenet classification with deep convolutional neural networks. In: Advances in neural information processing systems. 2012. p. 1097-1105.



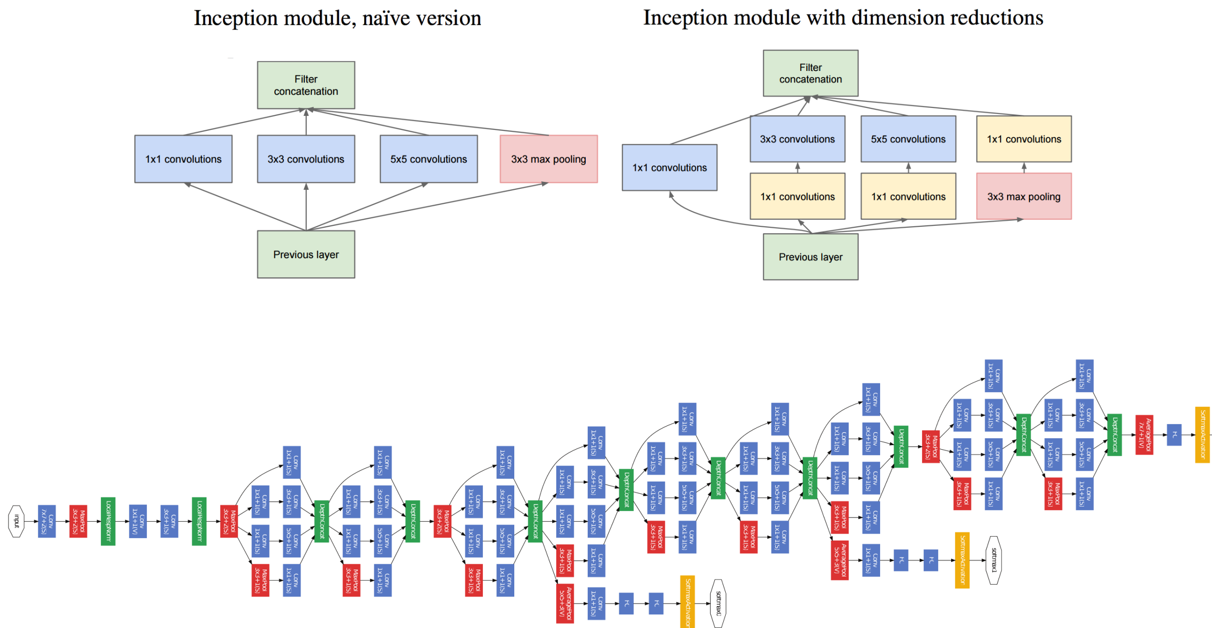
VGG networks (2014) This convolutional network by the Visual geometry Group (VGG) at Oxford was based on solving a localization and a classification task using depth (achieving 1st and 2nd places in the Imagenet challenge 2014, respectively). The depths of 16 and 19 layers proved to be the best:

K Simonyan, A Zisserman. Very deep convolutional networks for large-scale image recognition - arXiv preprint arXiv:1409.1556, 2014



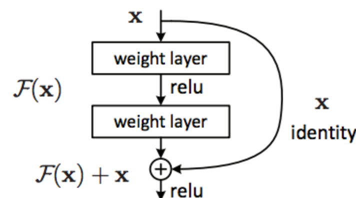
GoogLeNet (2015) Instead of simply going in depth, Google built a network by composing several replicas and variants of an "inception module" that included several alternative convolutional layers (with different topologies) and then max pooling. Around 100 layers and yet an order of magnitude less parameters than AlexNet:

SZEGEDY, Christian, et al. Going deeper with convolutions. In: Proceedings of the IEEE Conf. on Computer Vision and Pattern Recognition. 2015. p. 1-9.



Microsoft ResNet (2015) Based on "residual blocks" which learn the difference between input and its transformed version (less variability, less information, easier to learn):

HE, Kaiming, et al. Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition. 2016. p. 770-778.



Generative Adversarial Nets (2014) These nets are not designed for winning ResNet challenges

GOODFELLOW, Ian, et al. Generative adversarial nets. In: Advances in neural information processing systems. 2014. p. 2672-2680.

Twin model made of two multi-layer perceptrons:

- A generative network that produces examples from a learned probability distribution
- A discriminative model that tries to classify both the training examples and the artificial samples generated

→ Useful for approximating the data (e.g., generating art)

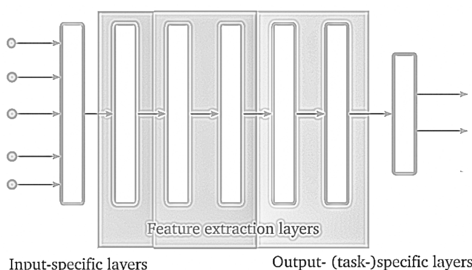
8.4.2 Frameworks, languages, libraries

- **Caffe** http://caffe.berkeleyvision.org/
  C++ Deep learning library with several front-ends (e.g., for Python)
- **Scikit-Learn** http://scikit-learn.org/stable/
  Machine learning in Python based on Matlab-like libraries (numpy, scipy, matplotlib)
- **TensorFlow** https://www.tensorflow.org/
  Dataflow programming using tensors as the basic data type
- **Theano** http://deeplearning.net/software/theano/
  Efficient Python library for computation and machine learning
- **Torch** http://torch.ch/
  GPU-oriented scientific computing with wide support for machine learning algorithms
- **Keras** http://keras.io/
  Keras is a high-level neural networks API, written in Python and using other libraries as the computation engine
- **Orange** https://orange.biolab.si/
  Visual programming for machine learning and data mining. Extensible using Python.

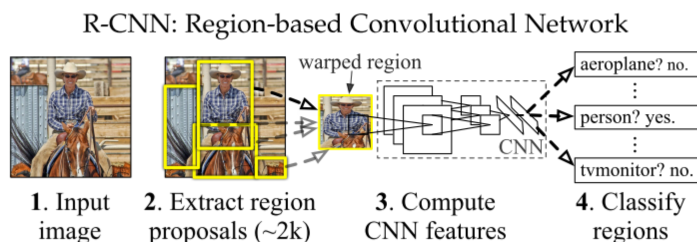
8.4.3 Tain deep networks without supercomputers

Training a deep network requires using training sets with more than millions of observations, optimizing millions of parameters and employing as many computing cores as possible (maybe GPU) for days or weeks.

Some popular **deep networks** are **available as off-the-shelf components** in software libraries. They are pre-trained for specific tasks so any user can employ them without having to do the optimization:



Directly using pre-trained networks works if your task is the same as the original task on which the network was trained. There are many networks available for tasks such as locating regions-of-interest for object tracking, for example the R-CNN:



To **fine-tune** a pre-trained network works if your task is similar as the original task on which the network was trained:

- It is necessary that in your task
  – the input and output data types are the same (e.g., input = images, output = classes)
  – the input and output sizes are the same
- as the original task on which the network was trained
  – Get a pre-trained network
  – Set a small learning rate η
  – Train the network on your data set

Transfer learning from a pre-trained network works if your task is similar as the original task on which the network was trained (as fine tuning), but with some differences:

- It is necessary that in your task
  – the input data type is the same (e.g., images)
  – the input size is the same
- as the original task on which the network was trained.
  – Get a pre-trained network
  – Remove the final layers
  – **Train a shallow network using the features provided by the remaining layers** using your data set

Matlab's deep learning toolbox contains a lot of pre trained models like:

- **Model for ResNet-50 Network**
  Pretrained Resnet-50 network model for image classification
- **Model for GoogLeNet Network**
  Pretrained GoogLeNet network model for image classification
- **Model for AlexNet Network**
  Pretrained AlexNet network model for image classification
- **Model for VGG-16 and VGG-19 Networks**
  Pretrained VGG-16 and VGG-19 network models for image classification
- **Importer for models created with other tools**
  Caffe, Keras

Also lots of models available from other sources

I AM DAVIDSTUZ

14th MAY 2017

Notes on Goodfellow's "Deep Learning" Textbook

In this article, I collect my notes on Ian Goodfellow's Deep Learning textbook. While the basic chapters are not included, the notes can be understood as short summaries of the corresponding chapters.

Ian Goodfellow's Deep Learning textbook quickly became a standard for generations to come. Although I was introduced to most of the cencepts a few years earlier — mainly through seminars at RWTH Aachen University —, I still took the chance and read most of the chapters. In this article, I want to present some of my notes I took while reading the textbook. Note that an online version of the book is available here.

As I was familiar with the basics of machine learning and feed-forward neural networks, I started with chapter 7. Additionally, I skipped chapters 12, 13 and 14 as they are of less interest for me.

Chapter Notes

Click on a chapter to open the corresponding notes.

I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. Chapter 7, MIT Press, 2016. (https://davidstutz.de/notes-on-goodfellows-deep-learning-textbook/#panel-1580681411-3615)

In chapter 7, Goodfellow et al. discuss several regularization techniques for deep learning in detail, and give valuable interpretations and relationships between them. They also revisit the goal of regularization. In particular, they characterize regularization as an approach to trade increased bias for reduced variance. The ultimate goal is to take a model where variance dominates the error (e.g. overfitted models) and reduce the variance in the hope to only slightly increase the bias. In the following, some regularization techniques are discussed, focussing on the practical insights provided by Goodfellow et al. instead of describing the techniques in detail.

Norm regularization (L_2 and L_1 regularization). Usually only the weights are regularized, not the biases. Therefore, it might also be interesting to regularize the weights in different layers differently strong. Two useful interpretations of L_2 regularization are the following:

- L_2 regularization shrinks the weights during training. However, Goodfellow et al. also show that components of the weights that correspond to directions that do not contribute to reducing the cost are decayed away faster than more useful directions. This is the result of the analysis of the eigenvectors and eigenvalues of the Hessian matrix (assuming a local, quadratic and convex approximation of the cost function).
- Using L_2 regularization, the training set is perceived to have higher variance. As a result, weight components corresponding to features that have low covariance with the target output compared to the perceived variance shrink faster.

Data augmentation. Goodfellow et al. briefly discuss the importance and influence of data augmentation to increase the size of the training set. Unfortunately, they give little concrete examples or references on this topic. They mostly focus on adding noise to either the input units or the hidden units. In a separate section on noise robustness, they also discuss the possibility to add noise to the weights in order to make the final model more robust to noise. Later this topic is also related to adversarial training, where training samples are constructed to "fool" the network while being similar to existing training samples. Here, as well, they do not give many concrete examples.

Early stopping. Beneath discussing the interpretation of early stopping as regularizer, Goodfellow et al. also focus on the problem utilizing early stopping while still being able to train on the full training set (as early stopping requires a part of the training set as validation set). The first approach discussed involves retraining the model on the full training set and training for approximately as many iterations as before when training with early stopping. The second approach fine-tunes the model on the full training set and stops when the training error reaches the training error when training was stopped using early stopping.

Dropout. Goodfellow et al. discuss the two important interpretations: dropout as bagging, and dropout as regularizer. In the first case, the most valuable insight provided is how to get the advantage of training with dropout at testing time, i.e. how to approximate the ensemble prediction.

I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. Chapter 8, MIT Press, 2016. (https://davidstutz.de/notes-on-goodfellows-deep-learning-textbook/#panel-1580681411-3618)

In chapter 8, Goodfellow et al. discuss optimization for deep neural networks. In particular, they focus on three different aspects:

1. How learning differs from classical optimization;
2. Why learning deep neural networks is difficult;
3. And concrete optimization techniques (both "fixed" learning rate and adaptive learning rate) as well as parameter initialization schemes.

How learning differs from classical optimization. One of the most important difference between learning and optimization is that learning tries to indirectly optimize a (usually) intractable measure while for optimization the goal is to directly optimize the objective at hand. In learning, the overall goal is to minimize the expected generalization error, i.e. the risk. However, as the generating distribution is usually unknown (and/or intractable), the empirical risk on a given training set is minimized instead. Furthermore, in learning, the empirical risk is usually not minimized directly, for example if the loss is not differentiable. Instead, a surrogate loss is minimized and optimization is usually stopped early to prevent overfitting and improve generalization. This is in contrast to classical optimization where the objective is usually not replaced by a surrogate objective. Lastly, in learning the optimization problem can usually be decomposed as a sum over the training samples.

Due to computational limits and considerations regarding generalization and implementation, researchers have early argued about so-called stochastic (mini-batch) optimization schemes. This discussion is usually specific to the task of learning and cannot be generalized to classical optimization. Some of the arguments made by Goodfellow et al. are summarized in the following points:

1. The standard error of a mean estimator is $\frac{\sigma}{\sqrt{n}}$ where σ is the true standard deviation and n the number of samples. Thus, there is less than linear returns in using more samples for estimating statistics (e.g. the gradient).
2. Small batches can offer regularization effects due to the introduced noise. However, this usually requires a smaller learning rate and induces slow learning.

Why learning deep neural networks is difficult. Goodfellow et al. discuss several well-known problems when training deep neural networks. However, they also give valuable insights of how these problems are related and approached in practice. The obvious argument is that used optimization techniques assume access to the true required statistics, such as the true gradient. However, in practice, the gradient is noisy and usually estimated based on stochastic mini-batches. Furthermore, the problem can be ill-conditioned, i.e. the corresponding Hessian matrix may be ill-conditioned. Goodfellow et al. provide an intuitive explanation based on a second-order Taylor expansion of the gradient descent update. Then, it can easily be shown that a gradient descent update of $-\epsilon g$, with g begin the gradient, adds $\frac{1}{2}\epsilon^2 g^T Hg - \epsilon g^T g$ to the cost. However, this may get positive. In particular, Goodfellow et al. describe the case that $g^T g$ stays mostly constant during training while $g^T Hg$ may increase by an order of magnitude. They also describe that monitoring both values during training might be beneficial (the former is also useful to detect whether learning has problems with local minima).

Another important discussion provided by Goodfellow et al. is concerned with the importance of local minima and saddle points. The main insight is that in high-dimensional spaces, local minima become rare and saddle points become much more frequent. This can be explained by the corresponding Hessian matrix. For local minima, all eigenvalues need to be positive, while for saddle points, both positive and eigenvalues are present. Obviously, the latter case becomes more frequent in high-dimensional spaces. Furthermore, local minima are much more likely to have low cost. This, too, can be explained by the Hessian being more likely to have only positive eigenvalues in low cost areas. Therefore, in high-dimensional spaces, saddle-points are the more serious problem in learning. Still, both cases induce a difficulty and require optimization methods to be tuned to the specific learning task.

Lastly, Goodfellow et al. discuss the case of cliffs in the energy landscape corresponding to the learning problem. In particular, cliffs refer to extremely steep regions (which may occur suddenly in more or less flat regions). These cliffs usually cause extremely high gradient and may result in large jumps made by the gradient descent update, potentially increasing the cost. However, they also provide a simple counter-measure (see Chapter 11): gradient clipping. Gradient clipping is usually done by either clipping the individual entries of the gradient vector at a maximum value, or clipping the gradient vector as whole at a specific norm. Both approaches seem to work well in practice.

Basic optimization techniques. As basis for the remaining chapters, Goodfellow et al. discuss stochastic gradient descent with momentum as well as Nesterov's accelerated gradient method. Details can be found in the chapter. Especially, the detailed explanation of the momentum term can be recommended.

Some interesting arguments made by Goodfellow et al. concern the convergence rate. It is well known that the convergence rate of stochastic gradient descent in the strictly convex case is $\mathcal{O}\left(\frac{1}{k}\right)$. However, the generalization error cannot be reduced faster than $\mathcal{O}\left(\frac{1}{k}\right)$ such that, from the machine learning perspective, it might not be beneficial to consider algorithms offering faster convergence (as this may correspond to overfitting).

Weight initialization schemes. Goodfellow et al. discuss several weight initialization schemes without going into too much details. Instead of looking at the individual schemes, value can be found in the discussed heuristics - especially as they explicitly state that initialization (and optimization) is not well understood yet. The following presents an unordered list detailing some of the discussed heuristics:

- An important aspect of weight initialization is to break symmetry, i.e. units with the same or similar input should have different initial weights as otherwise they would develop very similarly during training. This motivates random initialization using a high-entropy distribution - usually Gaussian or uniform.
- Biases are usually initialized to constant values. In many cases, 0 might be sufficient, however for saturating activation functions or output activation function non-zero initialization should be considered.
- The right balance between large initial weights and not-too-large initial weights is important. While too large weights may cause gradient explosion (if no clipping is used), large weights ensure that activations and errors (in forward and backward pass, respectively) are still numerically significant (i.e. distinct from 0).
- Independent of the initialization scheme used, it is recommended to monitor the activations and gradients of all layers on individual mini-batches. If activations or gradients in specific layers vanish, the scale or range of initialization may need to be adapted.

Adaptive learning rate techniques. Goodfellow et al. discuss several techniques including AdaGrad, Adam and RMSProp (with and without momentum). However, they are not able to answer the question which of the techniques is to be preferred.

Meta algorithms. Finally, Goodfellow et al. discuss a set of meta algorithms to aid optimization. The most interesting part is concerned with batch normalization. In particular, they are able to provide an extremely intuitive motivation. Using any gradient descent based optimization technique, the computed updates for a particular layer assume that preceeding layers do not change - which of course is wrong. They also discuss that batch normalization - through normalizing the first and second moments - implicitly reduces the expressive power of the network, especially when using linear activation functions. This results in the model being easier to train. To avoid the restrictions in expressive power, batch normalization applies a reparameterization to allow non-zero mean and non-unit variance.

I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. Chapter 9, MIT Press, 2016. (https://davidstutz.de/notes-on-goodfellows-deep-learning-textbook/#panel-1580681411-3632)

In chapter 9, Goodfellow et al. discuss convolutional networks in quite some detail. However, instead of focussing on the technical details, they discuss the high-level interpretations and ideas.

For example, they motivate the convolutional layer by sparse interactions, parameter sharing and equivariant representations. With sparse interactions, they refer to the local receptive field of individual units within convolutional networks (as the used kernels are usually small compared to the input size). Parameter sharing is achieved by using the same kernel at different spatial locations, such that neighboring units use the same weights. In this regard, some of the discussed alternative uses of convolution in neural networks are interesting. For example tiled convolution where different kernels are used for neighboring units by cycling through a fixed number of different kernels. Unshared convolution is also briefly discussed. Finally, equivariant representation refers to the translation invariance of the convolution operation.

Regarding pooling, they focus on the invariance introduced through pooling. However, they do not discuss the different pooling approaches used in practice. Unfortunately, They also don't give recommendations of when to use pooling and which pooling scheme to use. In contrast, they discuss the interpretation of pooling as infinitely strong prior. An interesting interpretation where pooling is assumed to place an infinitely strong prior on units invariant to local variations (like small translations or noise). In the same sense, convolutional layers place an infinitely strong prior on neighboring units having the same weights.

Finally, the importance of random and unsupervised features is briefly discusses. Here, an interesting reference is [1] where it is shown that random features work surprisingly well.

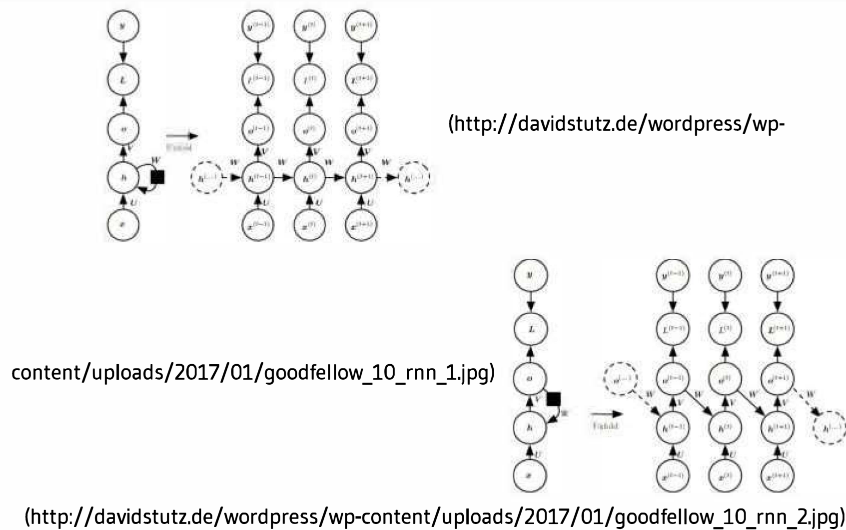
They conclude with a longer discussion of the biological motivation of convolutional networks given by neuroscience. While most of the discussed aspects are well-known, it is an interesting summary of different aspects motivating research in convolutional networks. Some of the main points is to distinguish simple and complex cells, and the simple cells in particular can often be modeled using Gabor filters. Another interesting insight is the low resolution used by the human eye. Only individual locations are available in higher resolution. Unfortunately, Goodfellow et al. do not provide many references how this model of attention can be implemented in modern convolutional networks.

[1] A. M. Saxe, P. W. Koh, Z. Chen, M. Bhand, B. Suresh, A. Ng. *On random weights and unsupervised feature learning*. ICML, 2011

I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. Chapter 10, MIT Press, 2016. (https://davidstutz.de/notes-on-goodfellows-deep-learning-textbook/#panel-1580681411-3625)

In chapter 11, Goodfellow et al. give an introduction to recurrent neural networks as well as corresponding further developments like long short-term memory networks (LSTM). During their discussion they focus on three different schemes (Goodfellow et al. call it "patterns") of recurrent neural networks (of which the first two schemes are illustrated in Figure 1):

- Recurrent neural networks producing an output at each iteration and the hidden units are connected through time.
- Recurrent neural networks producing an output at each iterations where the output is propagated to the hidden units in the next time step.
- Recurrent neural networks that read a complete sequence and then produce a single output.



(http://davidstutz.de/wordpress/wp-content/uploads/2017/01/goodfellow_10_rnn_1.jpg)

(http://davidstutz.de/wordpress/wp-content/uploads/2017/01/goodfellow_10_rnn_2.jpg)

Figure 1 (**click to enlarge**): Recurrent neural network where the hidden units are propagated through time (left) and where only the output is propagated through time (right). As detailed by Goodfellow et al., the second option represents strictly lower expressiveness in terms of which functions can be modeled.

The key idea that makes recurrent neural networks interesting, is that the parameters (i.e. W, U, V ...) are shared across time, allowing for variable length sequences to be processed. In addition, parameter sharing is important to generalize to unseen examples of different lengths. The general equations corresponding to the first scheme are as follows:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$

$$h^{(t)} = \tanh(a^{(t)})$$

$$o^{(t)} = c + Vh^{(t)}$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

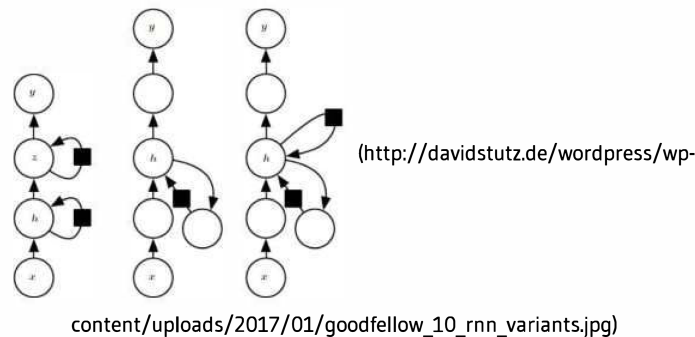
Relating to Figure 1, on top of the output $o^{(t)}$ a loss $L^{(t)}$ is applied which implicitly performs the softmax operation to compute $\hat{y}^{(t)}$ and computes the loss with regard to the true output $y^{(t)}$.

Recurrent neural networks are generally trained using error backpropagation through time, which describes error backpropagation applied to the individual networks starting from the last time step and going back to the first time step. As the parameters across time are shared, the gradients with respect to the involved parameters represent sums over time. For example, regarding W the gradient is easily derived (by recursively applying the chain rule) as

$$\nabla_W L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}}\right) \nabla_{W^{(t)}} h_i^{(t)}$$

when considering the first model of Figure 1.

While the presented recurrent network is shallow - having only one hidden layer - deep recurrent neural networks can add multiple additional layers. Interestingly, one has many options of how these additional layers are connected through time. Goodfellow et al. illustrate this freedom using Figure 2. Note that the black square indicates a time delay of one time step for unfolding (see chapter 10.1 for details) the model.



(http://davidstutz.de/wordpress/wp-content/uploads/2017/01/goodfellow_10_rnn_variants.jpg)

Figure 2 (**click to enlarge**): Examples of possible deep recurrent network architectures.

Towards the end of the chapter, Goodfellow et al. focus on learning long-term dependencies. The described problem corresponds to exploding or vanishing gradients (with respect to time) when training recurrent neural networks for long sequences. Beneath simple techniques such as gradient clipping (also see chapter 8), several model modifications are discussed that simplify learning long-term dependencies. Among these models, Goodfellow et al. also discuss long short-term memory (LSTM) models. Other approaches include skip connections and leaky units.

I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. Chapter 11, MIT Press, 2016. (https://davidstutz.de/notes-on-goodfellows-deep-learning-textbook/#panel-1580681411-3634)

Chapter 11 is among the most interesting chapters for deep learning practitioners that already have some background on the involved theory and algorithms. What Goodfellow et al. call "Practical Methodology" can best be described as a loose set of tips and tricks for approaching deep learning problems. They give the following, general process that should be followed:

1. Define the problem to be solved including metrics used to access whether the problem was solved; it is also beneficial to define expected results in terms of the chosen metrics.
2. Get a end-to-end prototype running that includes the selected metric.
3. Incrementally do the following: diagnose a component (or aspect) that causes the system to under perform (e.g. hyperparameters, bugs, low-quality data, not enough data, model complexity etc.) and fix it.

Surprisingly, this approach has many parallels with modern, agile software engineering principles (e.g. prototyping, iterative development, risk focus).

Goodfellow et al. then discuss some of these aspects in detail. The most interesting points are made on diagnosing a running end-to-end system:

- Visualize the results: do not focus on the quantitative results in terms of the selected metrics, also visualize the results to asses them qualitatively. This also includes looking at examples that are considered very difficult or very easy.
- Always monitor training and test performance: also discussed in chapter 7, training and test performance may give important clues regarding hyperparameters or regularization such as early stopping. However, it might also help to decide whether bugs cause problems or underfitting/overfitting is a problem.
- Try a tiny dataset: try a smaller or easier training set; this might be helpful to see whether bugs exist.
- Monitor activations and gradients: monitoring the activations may provide clues about the model complexity and activation functions. Together with monitoring the gradients, e.g. the magnitude, might be helpful to asses optimization performance, problems with the hyperparameters etc.

I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. Chapter 15, MIT Press, 2016. (https://davidstutz.de/notes-on-goodfellows-deep-learning-textbook/#panel-1580681411-3639)

In Chapter 15, Goodfellow et al. consider representation learning, focussing on unsupervised pre-training. Specifically, they discuss when and why unsupervised pre-training may help the subsequent supervised task. The two discussed interpretations are:

- Unsupervised pre-training acts as regularizer by limiting the initial parameters to a specific region in parameter space.
- Unsupervised pre-training helps to learn representation characterizing the input distribution; this may help learning mappings from input to output.

Regarding the first interpretation, it was originally assumed to help optimization by avoiding poor local minima. However, Goodfellow et al. emphasize that it is known by now that local minima aren't a significant problem in deep learning. That may also be one of the reasons why unsupervised pre-training isn't as popular anymore (especially compared to supervised pre-training or various forms of transfer learning). However, unsupervised pre-training may make optimization more deterministic. Goodfellow et al. specifically argue that unsupervised pre-training causes deep learning to consistently reach the same "solution". This suggests that unsupervised pre-training reduces the variance of the learned estimator. It is hard to say when unsupervised pre-training is beneficial when using this interpretation.

The second interpretation gives more clues about when unsupervised pre-training may be beneficial. For example, if the initial representation is poor. Goodfellow et al. name the example of word representations and also argue that there is less benefit for vision tasks as discrete images already represent an appropriate representation of the data. When thinking of unsupervised pre-training (or semi-supervised training) as identifying the underlying causes of the data, success may depend on the causal factors involved and the data distribution. For example, assumptions such as sparsity or independence may or may not be present regarding the causes. Furthermore, from a uniform data distribution, no useful representation can be learned. From a highly multi-modal distribution, unsupervised pre-training may already identify the different modes without knowing the semantics.

Overall, the chapter gives a good, high-level discussion of unsupervised pre-training and representation learning in general without going into algorithmic details. Two important takeaways are the two presented interpretations that can be used to reason about unsupervised pre-training.

I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. Chapter 16, MIT Press, 2016. (https://davidstutz.de/notes-on-goodfellows-deep-learning-textbook/#panel-1580681411-3642)

In Chapter 16, Goodfellow et al. briefly recap directed and undirected graphical models including d-separation, factor graphs and ancestral sampling. However, I found that there are better textbooks or chapters on graphical models. A similarly brief introduction can be found in [1] and an extensive discussion is available in [2].

In the end, they relate graphical models to deep learning yielding some interesting insights. While traditional graphic models usually have fewer unobserved variables and tend to have sparse connections such that exact inference is possible, the deep learning approach usually focusses on having many hidden, latent variables with dense connections in order to learn distributed representation. Exact inference is usually not expected to be possible and even marginals are not tractable. It is usually sufficient to be able to draw approximate samples and efficiently compute the gradient of the underlying energy function (while the energy itself does not need to be tractable).

Finally, they briefly introduce restricted Boltzmann machines (RBMs) (note that there might be more detailed discussions available). An RBM is an energy-based model with binary hidden and visible variables, h and v, respectively:

$$E(v,h) = -b^Tv - c^Th - v^TWh$$

where b, c and W are real-valued parameters that are learned. Note that there is no interaction between any two hidden variables or any two visible variables (as illustrated by the $-b^Tv$ and $-c^Th$ terms). Instead, only parts of visible and hidden variables are, usually densely, connected through the weight matrix W. The individual conditional distributions are easily computed by:

$$P(h_i = 1|v) = \sigma(v^TW_{i,j} + b_i)$$

Overall, this allows for efficient Gibbs sampling. Furthermore, the energy is linear in all of its parameters such that the derivatives are easy to derive.

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.
[2] D. Koller, N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. Chapter 17, MIT Press, 2016. (https://davidstutz.de/notes-on-goodfellows-deep-learning-textbook/#panel-1580681411-3645)

In chapter 17, after discussing structured predictions using graphical models in Chapter 16, Goodfellow et al. briefly introduce basic Monte Carlo methods for sampling. While these methods might not be new to most students in computer vision and machine learning, I found a repetition of the concepts quite useful. Nevertheless, I want to emphasize that there are more

appropriate readings regarding the details of Monte Carlo methods.

The basic idea of Monte Carlo Sampling is NOT to sample from a distribution, but to approximate the expectation of a function $f(x)$ under a distribution $p(x)$. If it is possible to draw from $p(x)$, the basic approach is to use the estimator

$$\hat{s}_n = \frac{1}{n}\sum_{i=1}^n f(x^{(i)})$$

with samples $x^{(i)}$ drawn from $p(x)$. This estimator is not biased, and given finite variance, i.e. $Var[f(x^{(i)})] < \infty$, the estimator converges to the true expected value for an increasing number of samples. If it is not possible to sample from $p(x)$, an alternative distribution $q(x)$ can be introduced and the same estimator can be used:

$$\hat{s}_n = \frac{1}{n}\sum_{i=1}^n \frac{p(x^{(i)})f(x^{(i)})}{q(x^{(i)})} \text{ for } x^{(i)} \sim q$$

The estimator is still unbiased, and the minimum variance is obtained for

$$q^*(x) = \frac{p(x)|f(x)|}{Z}$$

where Z is the partition function. Generally, low variance is achieved for $q(x)$ being high whenever $p(x)|f(x)|$ is high.

Goodfellow et al. also briefly discuss Markov Chain Monte Carlo for sampling and Gibbs Sampling. However, I found that there are better resources to study these techniques.

I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. Chapter 18, MIT Press, 2016. (https://davidstutz.de/notes-on-goodfellows-deep-learning-textbook/#panel-1580681411-3647)

In Chapter 18, after briefly introducing Monte Carlo methods in chapter 17 and graphical models in Chapter 16, Goodfellow et al. discuss the problem of computing the partition function. Most (undirected) graphical models, especially deep probabilistic models, are defined by an unnormalized probability distribution $\tilde{p}(x)$, often based on an energy. Computing the normalization constant Z, i.e. the partition function, is often intractable:

$$\int \tilde{p}(x)dx$$

Another problem, is that the partition function usually depends on the model parameters, i.e. $Z = Z(\theta)$ for parametersθ. Thus, the gradient of the log-likelihood (in order to maximize the likelihood) decomposes into the so-called positive and negative phases:

$$\nabla_\theta \log p(x;\theta) = \nabla_\theta \log \tilde{p}(x;\theta) - \nabla_\theta \log Z(\theta)$$

Under certain regularity conditions (see the textbook for details), which can usually be assumed to hold for machine learning models, the gradient can be rewritten as

$$\nabla_\theta \log Z = E_{x\sim p(x)}[\nabla_\theta \log \tilde{p}(x)]$$

The derivation for the discrete case is rather straight-forward by computing the derivative of $\log(Z)$ and assuming that $p(x) > 0$ for all x. This identity is the basis for various Monte Carlo based methods for maximizing the likelihood. Also note that the two phases have intuitive interpretations. In the positive phase, $\log(\tilde{p}(x))$ is increased for x drawn from the data; in the negative phase, the partition function Z is decreased by decreasing $\log(\tilde{p}(x))$ for x drawn from the model distribution.

Following this brief introduction, Goodfellow et al. focus on the discussion of contrastive divergence. In general, contrastive divergence is based on a naive Markov Chain Monte Carlo algorithm for maximizing the likelihood: For the positive phase, samples from the data set are used to compute $\nabla_\theta \log(\tilde{p}(x;\theta))$; for the negative phase, a Markov Chain is burned in to provide samples $\{\tilde{x}_1,\ldots,\tilde{x}_M\}$ used to estimate $\nabla_\theta \log(Z)$ as

$$\frac{1}{M}\sum_{i=1}^M \nabla_\theta \log \tilde{p}(\tilde{x}^{(i)};\theta)$$

As burning in a Markov Chain, initialized at random, in each iteration is rather computational expensive, contrastive divergence initializes the Markov Chain using samples from the data set. This approach is summarized in Algorithm 1. Note that Gibbs Sampling is used (see gibbs_sampling in Algorithm 1, details can be found in the textbook, Chapter 17).

```
contrastive_divergence(
    ϵ // step size
    k // number of Gibbs steps
)
while not converged
    sample a minibatch {x^(1),...,x^(m)}
    g := 1/m ∑_{i=1}^m ∇_θ log p̃(x^(i), θ)
```

```
for i = 1,...,M
    \tilde{x}^{(i)} := x^{(i)}
for i = 1,...,k
    for j = 1,...,m
        x̃^(j) :=gibbs_update(x̃^(j))
g := g − (1/m) Σ_{i=1}^m ∇_θ log p̃(x̃^(i), θ)
θ := θ + εg
```

This results in a convenient approximation to the expensive negative phase. However, it also introduces the problem of spurious modes - regions with low probability under the data distribution but high probability under the model distribution. Unless a lot of iterations are performed, a Markov Chain initialized with samples from the data set will usually not reach these spurious models. Thus, the negative phase fails to suppress these regions. Overall, it might be likely to get samples that do not resemble the data.

While contrastive divergence implicitly approximates the partition function (without providing an explicit estimate of it), other methods try to avoid this estimation problem. Goodfellow et al. discuss approaches including pseudo-likelihood, score matching and noise-contrastive estimation. For example, the main idea behind pseudo-likelihoods is that the partition function is not needed when considering ratios of probabilities:

$$\frac{p(x)}{p(y)} = \frac{\frac{1}{Z}\tilde{p}(x)}{\frac{1}{Z}\tilde{p}(y)} = \frac{\tilde{p}(x)}{\tilde{p}(y)}$$

The same is applicable to conditional probabilities. These considerations result in the pseudo-likelihood objective:

$$\sum_{i=1}^{n} \log p(x_i|x_{-i})$$

where x_{-i} corresponds to all variables except for x_i. Goodfellow et al. note that maximizing the pseudo-likelihood is asymptotically consistent with maximizing the likelihood. The remaining discussed approaches often use similar approaches for avoiding the partition function. Details can be found in the chapter.

I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. Chapter 19, MIT Press, 2016. (https://davidstutz.de/notes-on-goodfellows-deep-learning-textbook/#panel-1580681411-3653)

In Chapter 19, Goodfellow et al. discuss approximate inference as optimization. While concrete examples, especially regarding the deep models discussed in Chapter 20, are missing, the main idea behind approximate inference is discussed in more detail. As motivation, they illustrate why the posterior distribution, i.e. $p(h|v)$ where v are visible and h are hidden variables, is usually intractable in layered models. Figure 1 shows the discussed examples, corresponding to a semi-restricted Boltzmann machine on the left, a restricted Boltzmann machine in the middle, and a directed model on the right. In all three cases the posterior is intractable due to interactions between the hidden variables - directly or indirectly.

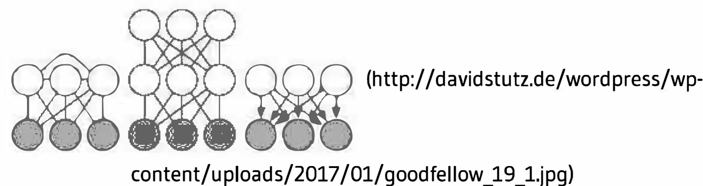

(http://davidstutz.de/wordpress/wp-content/uploads/2017/01/goodfellow_19_1.jpg)

Figure 1 (**click to enlarge**): Illustration of three graphical models as commonly used for deep learning. In all three cases, the direct or indirect interactions between hidden variables prevent the posterior from being tractable.

In order to approximate $p(h|v;\theta)$ with θ being parameters, the main idea behind approximate inference is based on the evidence lower bound $\mathcal{L}(v,\theta,q)$ on $\log p(h|v;\theta)$:

$$\mathcal{L}(c,\theta,q) = \log p(v;\theta) - D_{KL}\left(q(h|v)|p(h|v;\theta)\right)$$

Here, q is an arbitrary distribution defining the tightness of the lower bound. Specifically, if q and p are almost equal, the lower bound becomes exact. This is due to the definition of the Kullback-Leibler divergence:

$$D_{KL}\left(q(h|v)\,|\,p(h|v;\theta)\right) = E_(h \sim q)\left[\log\left(\frac{q(h|v)}{p(h|v;\theta)}\right)\right]$$

Rewriting the evidence lower bound using the definition of the Kullback-Leibler divergence and using basic logarithmic identities gives:

$$\mathcal{L}(v,\theta,q) = E_{h\sim q}[\log p(h,v)] + H(q)$$

with $H(q)$ being the entropy. Inference can, thus, be stated as optimizing for the ideal q. When restricting the family of distributions, $\mathcal{L}(v, \theta, q)$ may become tractable.

Variational inference means to choose q from a restricted set of families. The mean field approximation defines q to factor as follows:

$$q(h|v) = \prod_i q(h_i|v)$$

In the discrete case, the distribution q can be parameterized by vectors of probability, resulting in a rather straight-forward optimization problem. In the continuous case, calculus of variation is applicable. Researchers have early derived a general fix point equation to use. Specifically, fixing all $q(h_j|v)$ for $j \neq i$, the optimal $q(h_i|v)$ is given by the normalized distribution corresponding to:

$$\tilde{q}(h_i|v) = \exp\left(E_{h_{-i} \sim q(h_{-i}|v)}[\log \tilde{p}(v, h)]\right)$$

This equation is frequently referred to in practice wherever the mean field approximation is used.

Unfortunately, Goodfellow et al. discuss the discrete and continuous case of the mean field approximation in a rather technical way given two specific examples not related to the deep models described in Chapter 20 (at least personally, I see no benefit in having read the examples).

I. Goodfellow, Y. Bengio, A. Courville. *Deep Learning*. Chapter 20, MIT Press, 2016. (https://davidstutz.de/notes-on-goodfellows-deep-learning-textbook/#panel-1580681411-3660)

In Chapter 20, Goodfellow et al. discuss deep generative models. First, a short note on the reading order for Chapters 16 to 20. Chapter 16 can be skipped with basic knowledge on graphical models, or should be replaced by an introduction to graphical models such as [1]. For chapter 17, discussing Monte Carlo methods, I recommend falling back to lecture notes or other resources to get a more detailed introduction and a better understanding of these methods. In Chapter 18, the sections on the log-likelihood gradient as well as contrastive divergence are rather important and should be read before starting with Chapter 20. Similarly, Chapter 19 introduces several approximate inference mechanisms used (or built upon) in Chapter 20.

The discussion first covers (restricted) Boltzmann machines. As energy-based model, the joint probability distribution is described as

$$P(x) = \frac{\exp(-E(x))}{Z}$$

with

E(x) = -x^TUx - b^Tx

where U is a weight matrix and b a bias vector. The variables x are all observed in this simple model. Obviously, Boltzmann machines become more interesting when introducing latent variables. Thus, given observable variables v and latent variables h, the energy can be defined as

$$E(v, h) = -v^T R v - v^T W h - h^T S h - b^T v - c^T h$$

where R, W and S are weight matrices describing the interactions between the variables and b and c are bias vectors. As the partition function of Boltzmann machines is intractable, learning is generally based on approaches approximating the log-likelihood gradient (e.g. contrastive divergence as discussed in Chapter 18).

In practice, Boltzmann machines become relevant when restricting the interactions between variables. This leads to restricted Boltzmann machines where any two visible/hidden variables are restricted to not interact with each others. As consequence, the matrix R and S vanish:

$$E(v, h) = -b^T v - c^T h - v^T W H$$

An interesting observation, also essential for efficient learning of restricted Boltzmann machines, is that the conditional probabilities $P(h|v)$ and $P(v|h)$ are easy to compute. This follows from the following derivation:

$$P(h|v) = \frac{P(h,v)}{P(v)}$$

$$= \frac{1}{P(v)} \frac{1}{Z} \exp\{b^T v + c^T h + v^T W h\}$$

$$= \frac{1}{Z'} \exp\{c^T h + v^T W h\}$$

$$= \frac{1}{Z'} \exp\{\sum_j c_j h_j + \sum_j v^T W_{:j} h_j\}$$

$$= \frac{1}{Z'} \prod_j \exp\{\sum_j c_j h_j + v^T W_{:j} h_j\}$$

Where the definition of the conditional probability and the energy E(v, h where substituted. As the variables h are binary, we can take the unnormalized distribution and directly normalize it to obtain:

$$P(h_j = 1|v) = \frac{\tilde{P}(h_j=1|v)}{\tilde{P}(h_j=0|v)+\tilde{P}(h_j=1|v}$$

$$= \frac{\exp\{c_j+v^T W_{:j}\}}{\exp\{0\}+\exp\{c_j+v^T W_{:j}\}}$$

$$= \sigma\left(c_j + v^T W_{:j}\right)$$

Efficient evaluation and differentiation of the unnormalized distribution and efficient sampling make restricted Boltzmann machines trainable using algorithms such as contrastive divergence.

[1] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.