ABSTRACT

The force/torque sensor is an important tool that gives to the robots the ability to interact with the environments. Calibration is essential for these sensors to convert the raw sensor values to accurate forces and torques measurements. However, in practice, the calibration of multi-axis force/torque sensor requires complex multi-step data processing, due to the coupling effects and nonlinearity of the sensors. Moreover, accuracy cannot be guaranteed. To solve this problem, various Machine Learning and Deep Learning approaches have been used and compared in terms of accuracy, rate and number of parameters.

ABSTRACT

Un sensore di forza/coppia è un importante strumento che fornisce al robot l'abilità di interagire con l'ambiente in cui sono immersi. La calibrazione è essenziale per questi sensori per convertire i valori grezzi in misure accurate di forze e coppie. Tuttavia, in pratica, il sensore di forza/coppia multi-asse necessita di un complesso processamento multi-step, a causa degli effetti accoppiamento e non linearità dei sensori. Inoltre, l'accuratezza non può essere garantita. Per risolvere questo problema, diversi algoritmi di Machine Learning e Deep Learning sono stati impiegati e confrontati in termini di accuratezza, frequenza, e numero di parametri.

Contents

Chapter 1 Introduction

1.1 Problem Description

Service robots are entering our daily and this will become more pervasive in the near future. Many roboticist emphasize the need of service robotics concerning the elderly population in Europe, Japan and USA. They think that machines equipped with physical and cognitive capabilities could take care of people physically and cognitively limited or disabled. Robots have shown to be capable of autonomous driving in environments of different complexities and constraints. Concerning household scenarios, robots already clean apartments. Other tasks as folding towels, preparing pancakes, or clear a table have been demonstrated on research platforms.

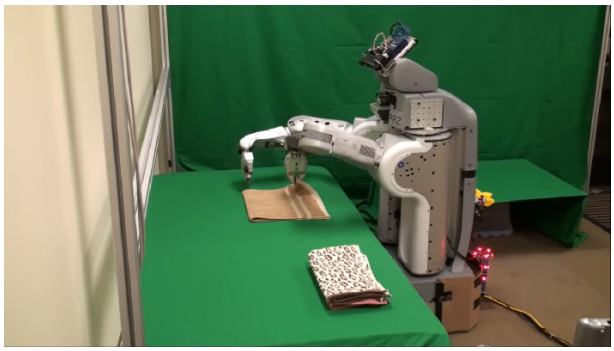


Figure 1(A) Robot folding towels. (B) Vacuum-cleaner

However, to let the robots perform these tasks in unstructured and open-ended environments, despite unexpected events and uncertainty in perception and execution, is matter of research effort and there are still many open problems that need to be solved. Service robots of new generation will interact with objects of different weights, dimension and shape and tasks, and they will need to be capable of adapting the grasp configuration to any kind of object, avoiding its slippage even in presence of external disturbances applied to the object. Such capability requires the ability to modulate the grasp force to allow the robot to manipulate both rigid objects and deformable or fragile ones that have to be grasped with the

minimum force required to hold them without causing excessive deformation or breakage. In order to regulate the grasp force, the measurements of contact forces and moments as well as contact locations at each finger so that external and internal forces can be estimated.

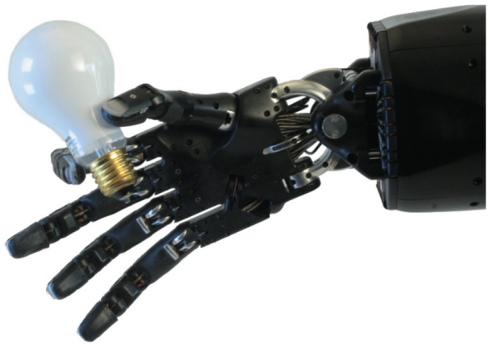


Figure 2 Robot grasping fragile object

Force and torque sensors are widely used in robotic manipulation. The sensor used in this thesis is a 6-axis force/torque sensor based on optoelectronic technology. Force/Torque sensor calibration is the process of mapping raw voltage data to force and torque wrench. In recent years, deep-learning has been actively studied in the field of computer science. The powerful characteristics of deep learning algorithms is that learning is possible by training single DNN model including a nonlinear activation function. The purpose of this thesis is to compare DNN (Deep Neural Network) with various Machine Learning algorithms in terms of prediction accuracy, prediction rate and number of parameters to tune.

1.2 Outline

The thesis is structured as follows:

- **Chapter 2 – Force Sensors**

  Chapter 2 introduces force sensors, their structure, their working principle. Then SunTouch tactile sensor is described.

- **Chapter 3 – Dataset for Calibration**

  Chapter 3 focuses on the construction of training set and its pre-processing using PCA and decimation.

- **Chapter 4 – Machine Learning Algorithms**

  Chapter 4 describes several Machine Learning algorithms like Regression Trees, KNN, AdaBoost, Random Forest, showing their results.

- **Chapter 5 – Gaussian Processes for Regression**

  Chapter 5 illustrates Gaussian Processes, how they works and how the prediction step works. Finally, simulation results are shown.

- **Chapter 6 – Deep Learning Algorithms**

  Chapter 6 introduces Neural Networks, its structure, training using backpropagation and regularization to obtain better testing error. In the end, results of different Neural Networks structure are shown.

- **Chapter 7 – Conclusion**

  Chapter 7 highlights the differences between different algorithms, their limitations and their strength. Therefore, some future development are introduced.

Chapter 2 Force Sensors

Measurement of a force or torque is usually reduced to measurement of the strain induced by the force (torque) applied to an **extensible element** of suitable features. Therefore, an indirect measurement of force is obtained by means of measurements of small displacements. The basic component of a force sensor is the **strain gauge** that uses the change of electric resistance of a wire under strain.

2.1 Strain Gauge

A strain gauge takes advantage of the physical property of electrical conductance and its dependence on the conductor's geometry. When an electrical conductor is stretched within the limits of its elasticity such that it does not break or permanently deform, it will become narrower and longer, which increases its electrical resistance end-to-end. Conversely, when a conductor is compressed such that it does not buckle, it will broaden and shorten, which decreases its electrical resistance end-to-end. From the measured electrical resistance of the strain gauge, the amount of induced stress may be inferred.

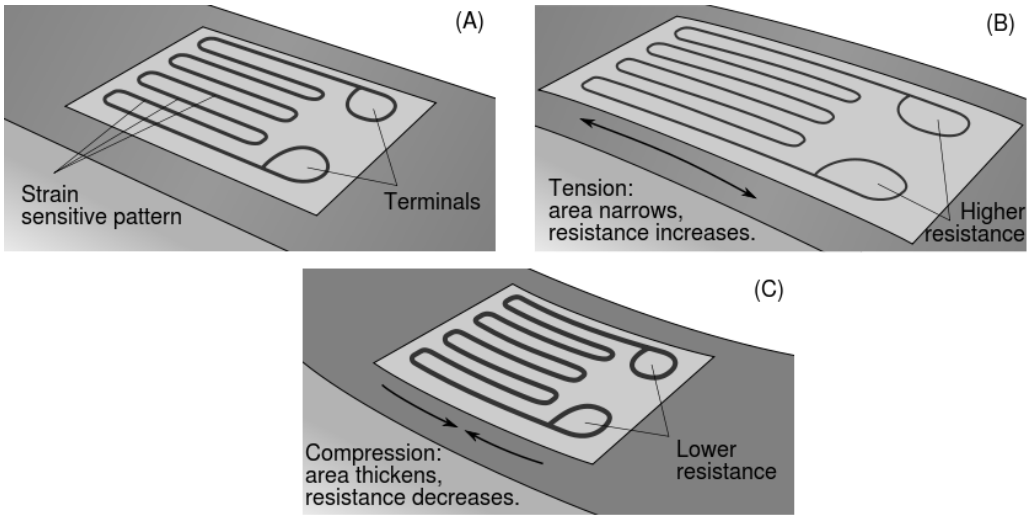
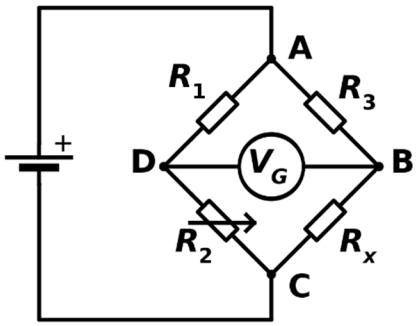


Figure 3 Visualization of the working concept behind the strain gauge on a beam under exaggerated bending.

The strain gauge is chosen in such a way that the resistance R_S changes linearly in the range of admissible strain for the extensible element. To transform changes of resistance into an electric signal, the strain gauge is inserted in one arm of a **Wheatstone bridge**, which is balanced in the absence of stress on the strain gauge itself. From Fig.4 it can be understood that the voltage balance in the bridge is described by:



$$V_g = \left(\frac{R_2}{R_2 + R_1} - \frac{R_x}{R_x + R_3}\right) V_s$$

Figure 4 Wheatstone bridge

If temperature variations occur, the wire changes its dimension without application of any external stress. To reduce the effect of temperature variations on the measurement output, it is worth inserting another strain gauge in an adjacent arm of the bridge, which is glued on a portion of the extensible element not subject to strain. Finally, to increase bridge sensitivity, two strain gauges may be used which have to be glued on the extensible element in such a way that one strain gauge is subject to traction and the other to compression; the two strain gauges then have to be inserted in two adjacent arms of the bridge.

2.2 Shaft torque sensor

In order to employ a servomotor as a torque-controlled generator, an indirect measurement of the driving torque is typically used, e.g., through the measurement of armature current in a permanent-magnet DC servomotor. If it is desired to guarantee insensitivity to change of parameters relating torque to the measured physical quantities, it is necessary to resort to a direct torque

measurement. The torque delivered by the servomotor to the joint can be measured by strain gauges mounted on an extensible apparatus interposed between the motor and the joint, e.g., a hollow shafting. Such apparatus must have low torsional stiffness and high bending stiffness, and it must ensure a proportional relationship between the applied torque and the induced strain. The measured torque is that delivered by the servomotor to the joint, and thus it does not coincide with the driving torque. This measurement does not account for the inertial and friction torque contributions as well as for the transmission located upstream of the measurement point.

2.3 Wrist force sensor

When the manipulator's end-effector is in contact with the working environment, the force sensor allows the measurement of the three components of a force and the three components of a moment with respect to a frame attached to it. As illustrated in Fig. 5, the sensor is employed as a connecting apparatus at the wrist between the outer link of the manipulator and the end-effector. The connection is made by means of a suitable number of extensible elements subject to strain under the action of a force and a moment. Strain gauges are glued on each element, which provide strain measurements.

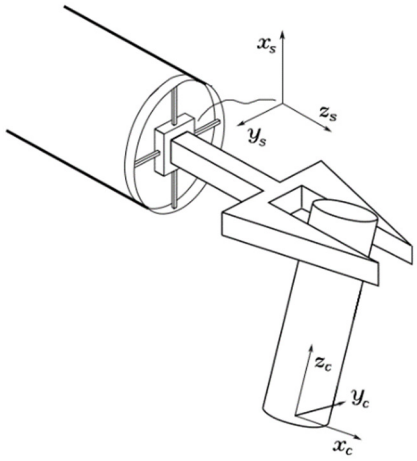


Figure 5 Use of a force sensor on the outer link of a manipulator

The elements have to be disposed in a keen way so that at least one element is appreciably deformed for any possible orientation of forces and moments. Furthermore, the single force component with respect to the frame attached to the sensor should induce the least possible number of deformations, so as to obtain good structural decoupling of force components. Since a complete decoupling cannot be achieved, the number of significant deformations to reconstruct the six components of the force and moment vector is greater than six. A typical force sensor is that where the extensible elements are disposed as in a **Maltese cross**; this is schematically indicated in Fig. 6. The elements connecting the outer link with the end-effector are four bars with a rectangular parallelepiped shape. On the opposite sides of each bar, a pair of strain gauges is glued that constitute two arms of a Wheatstone bridge; there is a total of eight bridges and thus the possibility of measuring eight strains.

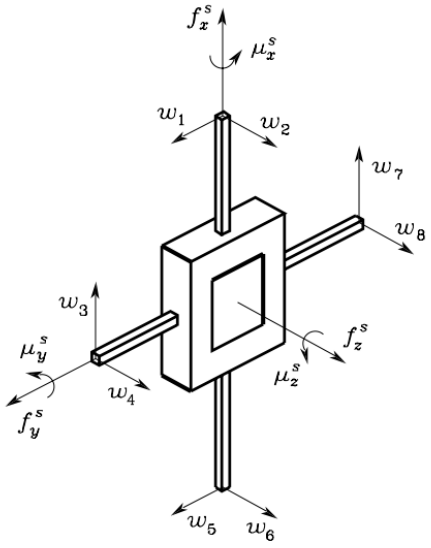


Figure 6 Schematic representation of a Maltese-cross force sensor

The matrix relating strain measurements to the force components expressed in a Frame s attached to the sensor is termed **sensor calibration matrix**. Let w_i, for $i = 1, \dots, 8$, denote the outputs of the eight bridges providing measurement of the strains induced by the applied forces on the bars according to the direction

specified in figure to the directions specified in Fig. 6. Then, the calibration matrix is given by the transformation:

$$\begin{bmatrix} f_x^s \\ f_y^s \\ f_z^s \\ \mu_x^s \\ \mu_y^s \\ \mu_z^s \end{bmatrix} = \begin{bmatrix} 0 & 0 & c_{13} & 0 & 0 & 0 & c_{17} & 0 \\ c_{21} & 0 & 0 & 0 & c_{25} & 0 & 0 & 0 \\ 0 & c_{32} & 0 & c_{34} & 0 & c_{36} & 0 & c_{38} \\ 0 & 0 & 0 & c_{44} & 0 & 0 & 0 & c_{48} \\ 0 & c_{52} & 0 & 0 & 0 & c_{56} & 0 & 0 \\ c_{61} & 0 & c_{63} & 0 & c_{65} & 0 & c_{67} & 0 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \\ w_6 \\ w_7 \\ w_8 \end{bmatrix}$$

Finally, it is worth noticing that force sensor measurements cannot be directly used by a force/motion control algorithm, since they describe the equivalent forces acting on the sensors which differ from the forces applied to the manipulator's end-effector (Fig. 6). It is therefore necessary to transform those forces from the sensor Frame s into the constraint Frame c

$$\begin{bmatrix} f_c^c \\ \mu_c^c \end{bmatrix} = \begin{bmatrix} R_s^c & 0 \\ S(r_{cs}^c)R_s^c & R_s^c \end{bmatrix} \begin{bmatrix} f_s^s \\ \mu_s^s \end{bmatrix}$$

which requires knowledge of the position r_{cs}^c of the origin of Frame s with respect to Frame c as well as of the orientation R_s^c of Frame s with respect to Frame c. Both such quantities are expressed in Frame c, and thus they are constant only if the end-effector is still, once contact has been achieved.

2.4 SunTouch

The sensor is based on the use of optoelectronic technologies and it aims to overcome difficulty of the integration into small spaces, high costs, repeatability and complex conditioning electronics. The sensor has different capabilities, i.e. it can measure the six components of the force and torque vectors applied to it, and it can be used as a tactile sensor providing a spatial and geometrical information about the contact with a stiff external object. A deformable elastic layer is

positioned above a matrix of sensible points (the **taxels**) to transduce the force and torque vectors into deformations, which are then measured, by the sensible points. Furthermore, the signals provided by the taxels, which are spatially distributed below the deformable layer, constitute a spatially distributed information that will also allow to estimate the size and orientation of the **contact surface** between the external surface of the sensor and the objects in contact with it. The contemporary knowledge of all this information is essential for a use of the sensor in robotic applications where objects of different size and dimension have to be manipulated by robotic hands.

2.4.1 Working principle

The tactile sensor basic idea is to use a deformable layer positioned above a discrete number of sensible points (called "taxels"), in order to transduce the external force and moment, applied to the sensor, into deformations, which are measured by the taxels. The taxels, spatially distributed below the deformable layer, provide a set of signals corresponding to a distributed information (called "tactile map") about the sensor deformations. Practically, the deformable layer transduces an external force and/or torque into a deformation of its bottom facet through its stiffness. An external force applied to the deformable layer produces local variations of the bottom surface of the elastic material and the couples of optical devices measure the deformations in a discrete number of points. In particular, these deformations produce a variation of the reflected light intensity and, accordingly, of the **photo-current** flowing into the photo-detector. The deformations can be positive or negative, i.e. the photo-current can locally increase or decrease depending on amplitudes of tangential and normal force components, as well as on torque components. The position of the $k-th$ taxel can be identified with the (x_k, y_k) coordinates of the centre position of the taxel. Denoting with v_k the voltage variation of the $k-th$ taxel, $v_k > 0$ denotes an

increasing distance (and then a decreasing photo-current), while $v_k < 0$ denotes a decreasing distance (and then an increasing photo-current) between the reflecting surface and the electronic layer (obviously $v_k = 0$ denotes no variation).

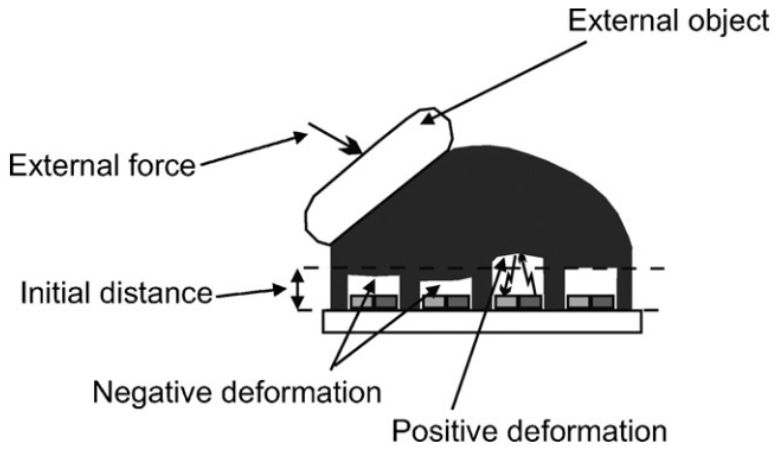


Figure 7 Sketch of the working principle.

The whole tactile map allows, after a calibration procedure, to estimate contact force and moment together with information about the orientation of the contact surface and object properties. The taxels have been developed by using optoelectronic technology, and in particular each sensing point is constituted by an emitter and a receiver, mounted side by side, working in reflection mode. The soft pad has been realized by using the silicone molding technology with the molds made with a high resolution 3D printing manufacturing process.

2.5 Technical Features

The sensor is mainly constituted by three components: a **Printed Circuit Board** (PCB), a rigid grid and a deformable cap. For each, the emitter/receiver couple is constituted by a unique optoelectronic component: a Surface Mount Technology (SMT) photo-reflector. In particular, the optoelectronic section of the PCB integrates 25 taxels, organized in a 5 × 5 matrix. For each taxel, the conditioning electronics is constituted by two resistors: one to drive the LED and a second to

transduce the photocurrent measured by the PT into a voltage directly compatible with an Analog-to-Digital (A/D) converter. The same 12-bit A/D converters (manufactured by Analog Devices) with 16 channels and a **Serial Peripheral Interface** (SPI), used in previous works, has been integrated in the PCB design. In particular, an interfacing section constituted by the microcontroller PIC16F1824, manufactured by Microchip Technology, has been added on a separate rigid board, connected to the previous described part via a flexible section. The integration of the microcontroller allows to obtain a fully integrated sensor with a programmable device used to interrogate the sensor via a standard serial interface already available in most commercial grippers. The board is completed by a standard low-noise voltage regulator with an input voltage range up to 12 V (typical range of supply voltage available on commercial grippers) and an output voltage equal to 3.3 V to supply the whole PCB.

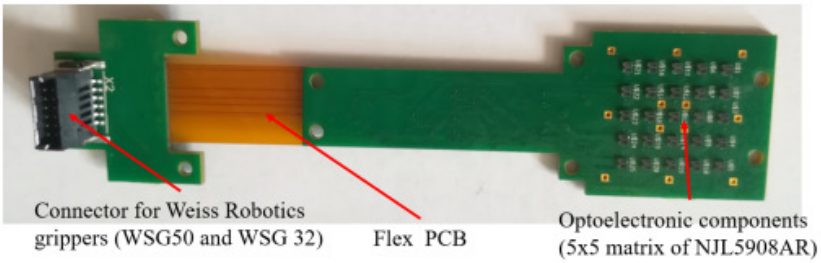


Figure 8 The tactile sensor PCB: layout with dimensions a top view

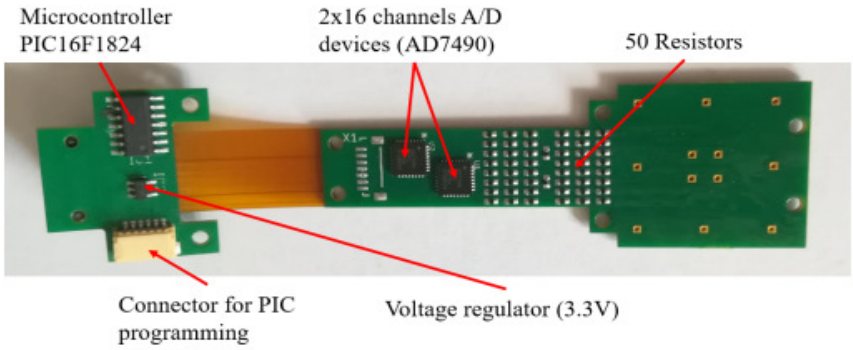
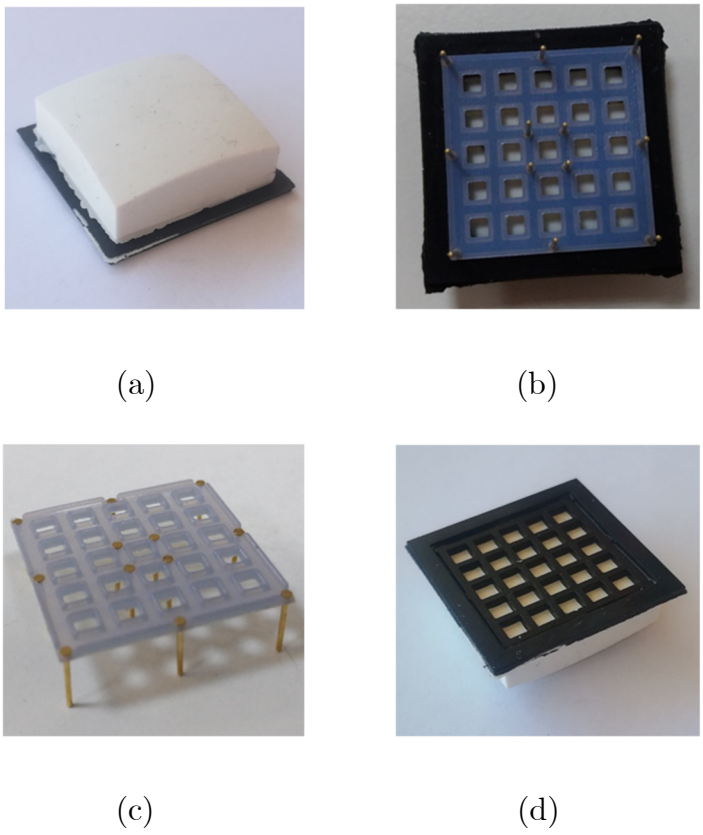


Figure 9 The tactile sensor PCB: layout with dimensions: a bottom view

2.6 Characteristics of the deformable pad

A mechanical structure constituted by the deformable layer and the rigid grid is connected above the PCB. The deformable layer is mainly made of white **silicone** with a domed top side and a square base, as shown by the picture in Fig.10(a).



(a) (b)

(c) (d)

Figure 10 Pictures of deformable layer and rigid grid

The mechanical properties of the silicone determine the full-scale and the sensitivity of the sensor. This prototype uses a shore hardness of 26 A. When external forces and/or moments are applied to the deformable layer, they produce vertical displacements of the white ceilings for all cells. The distances between the top of photo-reflectors and the white surfaces change, by producing variations of the **reflected light** and, accordingly, of the voltage signals measured by the PTs. The addition of the third component (i.e., the rigid grid) became necessary due to the electromechanical characteristic of the optical components. In particular, photo-reflector has a **non-monotonic characteristic** (see Fig. 11), which relates

the measured voltage to the distance of a reflecting surface positioned in front of the component.

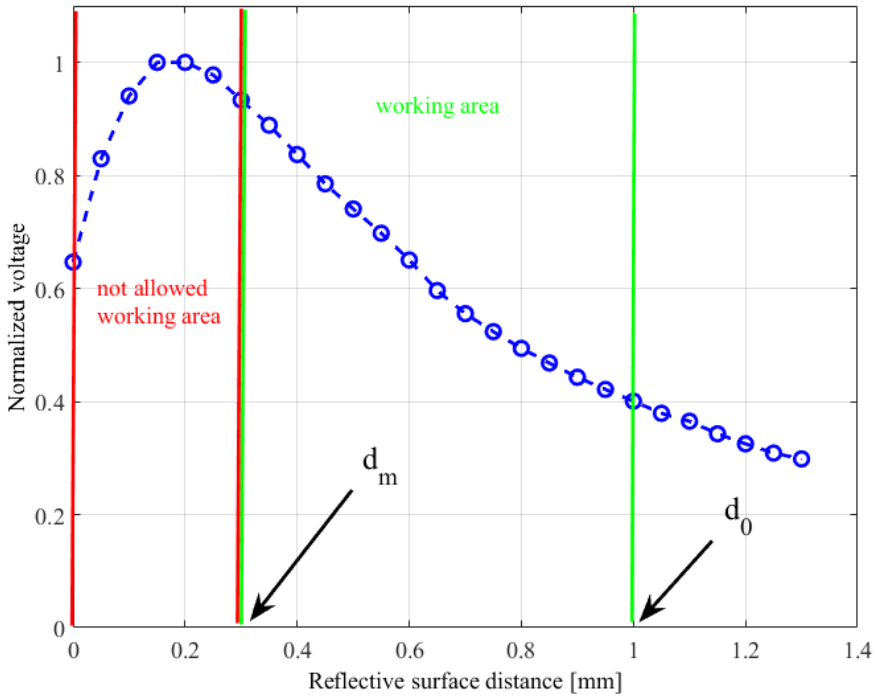


Figure 11 Characteristic for a single taxel: normalized voltage vs reflective surface distance

As a consequence, the rigid grid has to ensure that the reflecting surface never reaches distances from the component that fall into the non-monotonic area, highlighted by the red bars in Fig. 11. Taking into account that the height of a component is $0.5\ mm$, the rigid grid has been designed with a thickness of $0.8\ mm$. With this choice the minimum reachable distance between a reflecting surface and a photo-reflector is $d_m = 0.3\ mm$. On the other side, the silicone layer have been designed so that, in rest condition, the sum of the grid thickness and of the cell walls fixes the white ceilings at an initial distance $d_0 = 1\ mm$ from the emitting surface of the optical components. The integrated design of these two components allows to force the photo-reflectors to work in the monotonic working area, highlighted by the green bars in Fig. 11.

2.7 Integration of the sensor in a commercial gripper

The connector compatible with the sensor port available on the commercial grippers WSG-series, manufactured by Weiss Robotics, has been integrated, in order to provide the 5V voltage supply to the sensor and for the physical implementation of the serial interface. The assembled force/tactile sensor is finally fixed inside an aluminum case suitably designed to house the sensor and for the mechanical connection to the WSG-series flange. Fig. 12 reports a picture of the sensorized finger fully integrated with the gripper.

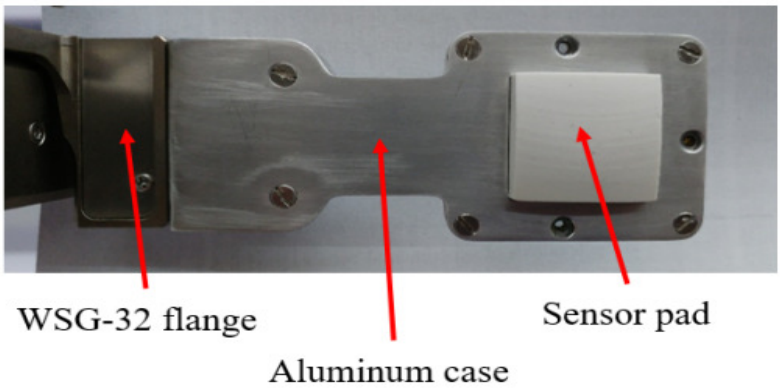


Figure 12 Picture of the sensorized finger fully integrated with the WSG-32 gripper

The microcontroller section available on the PCB allows two possible connections to exchange data with the main PC. In the fully integrated version, the PCB takes the voltage supply directly from the sensor port available on the WSG-series flange. The same port is used to implement a standard serial communication between the gripper and the sensor. The microcontroller interrogates the A/D converters via an SPI interface and transmits the raw via its serial port. The gripper is programmable by using the **LUA** programming language, that is an interpreted language suitably designed for embedded applications. The second possible connection from the microcontroller to the robot control PC foresees the use of a standard USB-to-serial converter with an external cable, that directly

connects the microcontroller to the main PC. In this case, the power supply and the serial transmission are implemented directly from the PC.

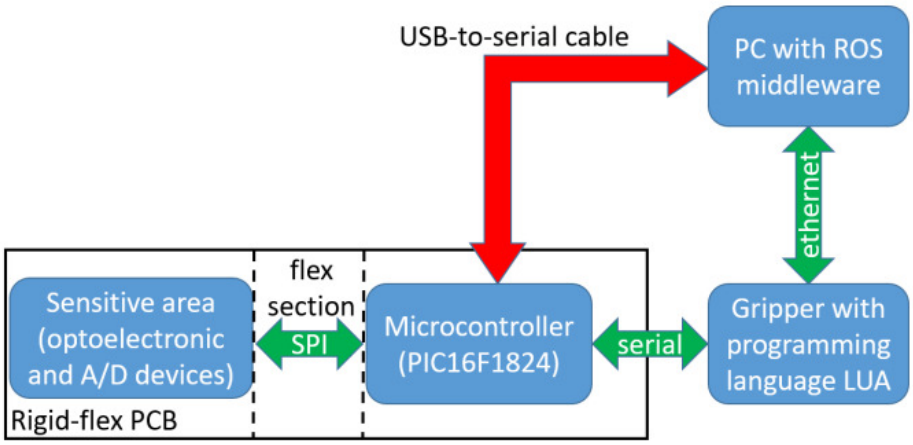


Figure 13 Data flow scheme of possible connections from the sensor to the control PC

The solution to limitations related to the serial port latency time is to interface the microcontroller with a **serial-to-WIFI adapter**, in order to use a wireless connection directly with the PC. On the control PC, two different ROS nodes have been developed: one to interact with the gripper, if the first solution is selected, and another one to directly interact with the microcontroller in the second case. In both cases the ROS nodes receive raw data (i.e., the 50 bytes acquired by the A/D converters) and the first elaboration consists in the reconstruction of actual voltage values, which are published to be available for the whole ROS network.

Chapter 3 Dataset for Calibration

In general, force/torque sensor calibration is the process of mapping raw sensor data to each force and torque. This sensor was calibrated with a **gray-box model** deduced by a **FEM analysis**. This approach has some limitations, in particular that the sensor was able to estimate only the forces and not all the wrench components. The idea of calibration is presented in the following picture.

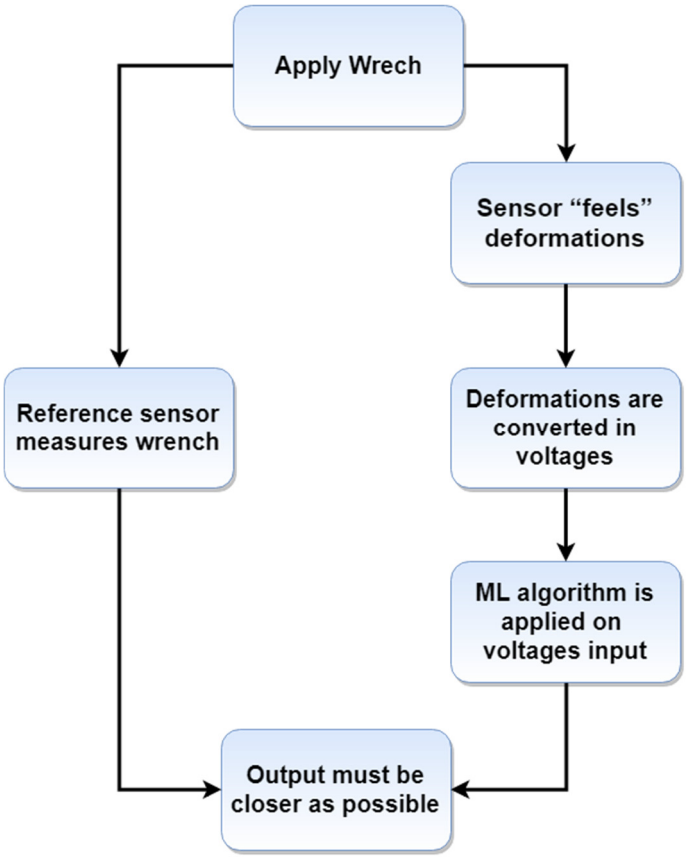


Figure 14 Calibration data flow

A Machine learning-based approach is able to overcome the limitations of the previous procedure. The critical point of the machine learning-based approach is the training data collection. The objective is to estimate the wrench in all possible combinations in a large interval of the contact plane orientation. The dimensionality of the problem is large, so there is a significant risk of missed wrench/orientation combinations in the training set. Moreover, the dimensionality and the correlation among the inputs, whose number (25 for the

sensor) is significantly larger than the dimension of the target set and the large number of samples acquired during the calibration phase can slow down the training phase and can easily cause unnecessary overfitting.

3.1 Training Set Construction

In order to collect the training set data, the sensor is mounted on a reference force/torque sensor, the (Robotous RFT40), as in Fig. 15. Σ_{sph} is the reference frame of both sensors; Σ_{sph} is a frame placed in the center of the undeformed silicone sphere; σ_{CoP} is a frame placed in the center of pressure (CoP) of the contact area with the $z\ axis$ normal to the contact plane.

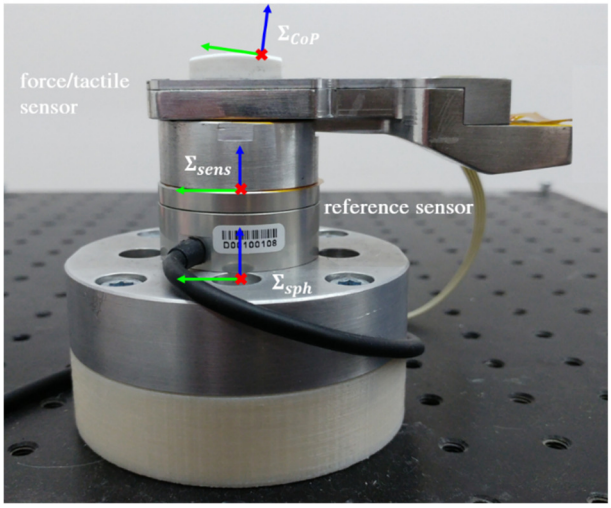


Figure 15 Testbench for sensor calibration.

An operator who applies forces and moments by touching the sensor with an object makes data generation. The target wrench and the input tactile voltages are recorded synchronized through the ROS network. In order to ensure a good training set, the input space (and consequently the target space) has to be properly covered. Furthermore, bad data should be avoided, e.g., samples during slipping of the object on the sensor pad surface or during the relaxing phase of the deformable layer. These issues are tackled by resorting to a dedicated Matlab GUI (Fig. 16).

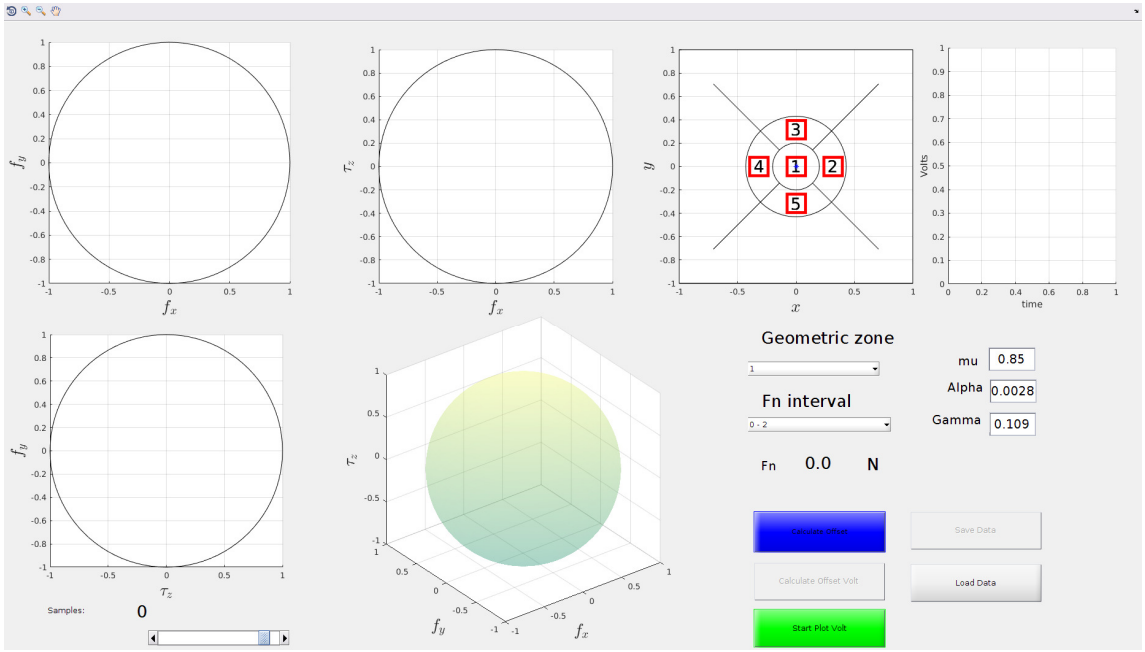


Figure 16 GUI used in the calibration procedure

The user interface displays in real-time the calibration data acquired. The visualization of the samples is carried out using the **limit surface** (LS) theory, which is an extension of the Coulomb friction model to the case of roto-translational slippage. The LS gives information about the maximum force and torsional moment that can be applied before a **slippage** occurs, it is a surface defined in the 3D space of the two tangential force components and the torsional moment (the component of the contact moment along the direction normal to the contact surface). When the wrench is inside this surface no slippage occurs, otherwise, there is relative motion between the two contacting surfaces. The maximum pure tangential force (that is the component of the force tangential to the contact surface) and torsional moment are given by:

$$f_{t_{max}} = \mu f_n;$$

$$\tau_{n_{max}} = \alpha f_n^{\gamma+1};$$

where f_n is the component of force normal to the contact frame, μ is the classical Coulomb friction coefficient, α and γ are parameters of the maximum torque

model. All parameters have been previously estimated experimentally. The GUI visualizes the 3D space of tangential force and torsional moment normalized with respect to $f_{t_{max}}$ and $\tau_{n_{max}}$, respectively, using four different plots: a 3D plot and three separate plots one for each view from each coordinate axis. In this normalized space, the LS is approximated as a unit sphere centered in the origin drawn in the 3D plot. This method is useful to discard samples acquired in any slipping phase, namely, samples outside the LS are not included into the training set. This decision about data inclusion cannot be directly taken based on the measured wrench referred to the sensor frame Σ_{sens}. In fact, the LS is defined based on the wrench referred to the Σ_{CoP} frame. The homogeneous transformation matrix expressing the pose of the CoP frame with respect to the sensor frame is estimated considering the tactile map. First of all, the **centroid** of the tactile map is calculated as:

$$x_C = \frac{\sum_{i=1}^{25} x_i \Delta v_i}{\sum_{i=1}^{25} \Delta v_i}$$

$$y_C = \frac{\sum_{i=1}^{25} y_i \Delta v_i}{\sum_{i=1}^{25} \Delta v_i}$$

where (x_i, y_i) are the coordinates of the ith taxel and Δv_i is the difference between the actual voltage value and the voltage value in rest conditions. The centroid is also plotted in the GUI in a separate plot to aid the user understand where he/she is touching the sensor. The CoP is considered located in the point on the contact surface corresponding neglecting the deformation of the sphere (consider that this computation is simply aimed at helping the operator in the calibration procedure). Hence, the coordinates of the CoP with respect to the Σ_{sph} frame are:

$$\boldsymbol{p}_{CoP}^{sph} = \begin{bmatrix} x_C \\ y_C \\ \sqrt{R^2 - x_c^2 - y_c} \end{bmatrix}$$

where $R = 50\,mm$ is the sphere radius. Given the distance between Σ_{sph} and Σ_{sens} (20 mm) it is trivial to find the coordinates of the CoP with respect to the sensor frame ($\boldsymbol{p}_{CoP}^{sens}$). The orientation of the contact plane is basically given by the normal vector to the contact plane. Since the GUI is just an aid for the operator, the contact plane can be well approximated as tangent to the sphere. So the normal unit vector can be calculated with respect to the sphere frame as:

$$\hat{\boldsymbol{n}}_{CoP}^{sph} = \frac{1}{R}\boldsymbol{p}_{CoP}^{sph}$$

Choosing the sphere frame aligned to the sensor frame, this normal vector has the same components in the sensor frame and it is selected as the $z-axis$ of the contact frame. The x and $y\,axes$ of the contact frame can be trivially choosen as the projection of the same axes of sensor frame on the contact plane (conveniently normalized). The computed axes can be organized into a rotation matrix $\boldsymbol{R}_{CoP}^{sens}$ and, finally, the homogenouse transformation matrix of the contact frame is

$$\boldsymbol{T}_{CoP}^{sens} = \begin{bmatrix} \boldsymbol{R}_{CoP}^{sens} & \boldsymbol{p}_{CoP}^{sens} \\ 0 & 1 \end{bmatrix}$$

Finally, by inverting the last matrix, it is possible to find the force and moment vectors in the contact frame:

$$\boldsymbol{f}^{CoP} = \boldsymbol{R}_{sens}^{CoP} f^{sens}$$

$$\boldsymbol{\tau}^{CoP} = \boldsymbol{R}_{sens}^{CoP}\boldsymbol{\tau}^{sens} + \boldsymbol{p}_{sens}^{CoP} \times \boldsymbol{f}^{CoP}$$

With the LS aid, the operator can visualize only the tangential forces and the torsional moment. It is not possible to see variations in the normal force and in the contact plane orientation. Moreover, it is impossible to include such information in the plot because it is already a 3D plot and plots with higher dimensions are impossible to easily visualize. To overcome this problem, in the GUI the operator can select a target interval for the normal force among a set of

predefined intervals. Given the centroid position, a polar area is uniquely defined. In the same manner, given the normal force value, an interval of forces is defined. In this way it is possible to define various 3D spaces, one for each possible combination of the normal force interval and polar area. The task of the operator is to cover all these 3D spaces with samples, the program will automatically discard bad samples.

3.2 Training set Preprocessing

Data Preprocessing is a huge topic, because the preprocessing techniques vary from data to data. Different kind of data (images, text, sounds, videos, csv files, etc) have different methods for preprocessing. However, there are some common methods for almost any kind of data. The most important methods are:

- **Transformation into vectors**: If one got raw text data and need some mechanism to convert those strings into some meaningful numerical representation. If one got categorical data in a csv file, he might want to apply label encoding, or one-hot-encoding. It's a conversion of the data into float (or in some cases, integers), so that ML model can easily process all that.

- **Normalization**: It's highly recommended that your data is properly scaled, which means that data should not have a very huge deviation for every column (**feature**). For instance, if a column has values between 0–1 and another feature whose values are between 100–1000, then this difference of value ranges can cause large gradient updates by optimizer, and network/model might not converge. So, a good way will be to normalize values which are between 100–1000, scaling them between 0–1. Breaking into steps, following steps should be applied to get maximum benefit out of normalization:

- **Smaller values**: Try to have all the values between 0 and 1, or -1 to 1.
- **Homogeneity**: All the columns should have values in roughly the same range.
- **Mean**: Normalize in a way that data have a mean of 0 for each column independently.
- **Standard deviation**: Normalize in a way that data have a standard deviation of 1 for each column independently.

- **Dealing with the missing values**: Having missing values in a dataset is very common, and an effective way to handle missing values leads to a better model trained. One way is to replace all the missing values with 0. If there are a lot of missing values in the data, and data are replaced with 0, the model will eventually learn that all the 0s aren't playing any role in decision-making process of the model, and will pretty much ignore them by assigning them lower weights. Interpolation of the data is a meaningful option as well. Otherwise, missing values are replaced by mean, or median values of the respective column they're a part of.

3.2.1 Training set decimation

The motivation for a decimation algorithm is that samples are often collected so that there are zones of the training set with a very high density compared to others. This is typical when forces are low, e.g., the operator is at the beginning of a maneuver. So there are a lot of samples that add few new information to the dataset. This can cause a useless increase in the computational load and can encourage the learning algorithm to specialize the model towards the behaviour in these high density zones. Therefore, these samples should be removed. The

number of samples is reduced through a **bubble-based decimation algorithm** described hereafter in a general case. The idea is to fix a maximum density for the samples in the input space. Let N be the total number of samples and the couple (v_i, w_i) the $i-th$ sample with input $v_i \in R^m$ and target $w_i \in R^t$, the training set is:

$$T_s = \{(\boldsymbol{v}_i, \boldsymbol{w}_i), i \in \mathbb{I}_{T_s}\}$$

Where

$$\mathbb{I}_{T_s} = \{1, \dots, N\}$$

Note that in the particular case of study $v_i \in \boldsymbol{R}^{25}$, and $w_i \in \boldsymbol{R}^6$. The main idea is to define a bubble in the space of the inputs such that, centering the bubble in a sample, no other sample is in the bubble. In other words, the objective is to find a subset T_s^* of T_s such that:

$$\mathbb{I}_{T_s^*} = \{j \in \mathbb{I}_{T_s} : \|v_j - v_k\| > r \ \forall k \in \mathbb{I}_{T_s}, k \neq j\}$$

being r the radius of the bubble. In this way the maximum density in the input space will be of 1 sample per bubble. The bubble-based decimation can be applied to an heterogeneous input space too, e.g., made of inputs of different scale. In that case it is necessary a pre-normalization of the input data. Considering that for each normal force interval and polar area the voltage map has to be rather different, this algorithm can be applied separately on the data of each 3D space. In this way the computational load of the decimation is reduced.

3.2.2 PCA

Work with many variables can present different problems. If there are a lot of variables is difficult to understand the relationships between variables. Furthermore, more variables means better chance of overfitting. **Principal**

Components Analysis (PCA) is a way of identifying patterns in data, and expressing the data in such a way as to highlight their similarities and differences. Since patterns in data can be hard to find in a high dimensional space, where the advantage of graphical representation is not available, PCA is a powerful tool for analysing data.

3.2.2.1 Intuitive explanation of PCA

First step of PCA is the computation a matrix that summarizes how our variables all relate to one another. Then this matrix is divided into two separate components: direction and magnitude. It is possible to understand the "directions" of the data and its "magnitude" (or how "important" each direction is). Figure 17, displays the two main directions in this data: the "red direction" and the "green direction." In this case, the "red direction" is the more important one. Later will be discussed in which sense is more important.

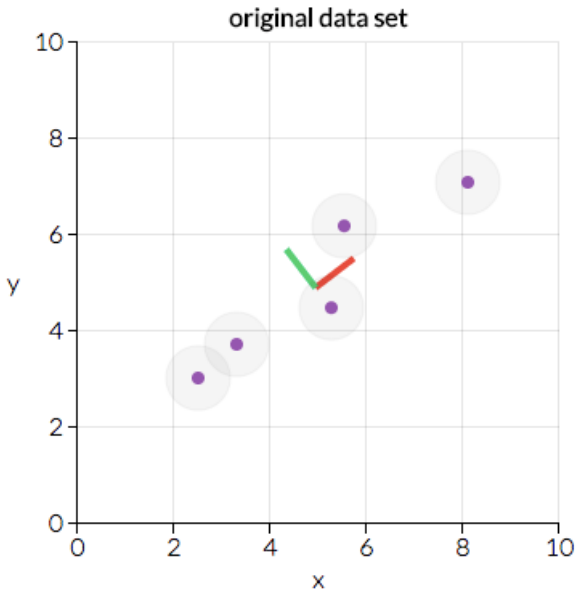


Figure 17 Dataset before PCA

Now a transformation is applied on original data to align with these important directions (which are combinations of our original variables). Figure 18 is the

same exact data as above, but transformed so that the x-and y-axes are now the "red direction" and "green direction."

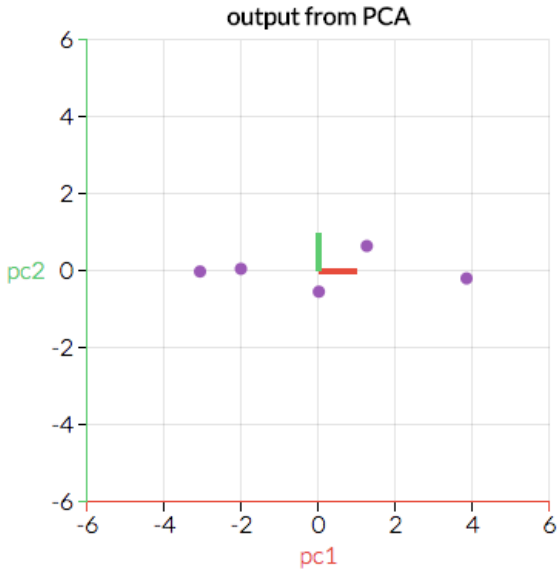


Figure 18 Dataset after PCA

The visual example here is two-dimensional (and thus two "directions"), think about a case where data has more dimensions. By identifying which "directions" are most "important," it is possible to compress or project data into a smaller space by dropping the "directions" that are the "least important". By projecting data into a smaller space, the dimensionality of the feature space is reduced.

3.2.2.2 How PCA works?

Let the data matrix X be of $n \times p$ size, where n is the number of samples and p is the number of features. Let us assume that it is *centered*, i.e. column means have been subtracted and are now equal to zero. Then the $p \times p$ covariance matrix C is given by $C = \frac{X^T X}{n-1}$. It is a symmetric matrix and so it can be diagonalized:

$$C = VLV^T$$

where V is a matrix of eigenvectors (each column is an eigenvector) and L is a diagonal matrix with eigenvalues λ_i in the decreasing order on the diagonal. The eigenvectors are called **principal axes** or **principal directions** of the data. Projections of the data on the principal axes are called **principal components**. Every principal component will always be orthogonal to every other principal component. Because our principal components are orthogonal to one another, they are statistically independent of one another. For this reason columns of projected data are independent of one another. In general, once eigenvectors are found from the covariance matrix, the next step is to order them by eigenvalue, highest to lowest. This gives the components in order of significance. Now if the purpose is dimensionality reduction, one can decide to ignore the components of lesser significance. This choice causes loss of some information, but if the eigenvalues are small, loss is small. If the data originally have dimensions p, and one calculates eigenvectors and eigenvalues, and then only the first k eigenvectors are chosen, then the final data set has only k dimensions. The question becomes "How to choose k?"

Different approaches can be used:

- **Arbitrary selection of the number of dimensions**. If one wants to plot data in two dimensions needs two features. This is use-case dependent and there isn't a hard-and-fast rule to choose how many features pick.
- **Calculate the proportion of variance explained for each feature**, pick a threshold, and add features until a threshold is hit. (For example, if one wants to explain 80% of the total variability possibly explained by the model, add features with the largest explained proportion of variance until proportion of variance explained $\geq 80\%$.)

- **Calculate the proportion of variance explained** for each feature, sort features by proportion of variance explained and plot the cumulative proportion of variance explained as one keeps more features. This plot is called a **scree plot**, it is shown in figure 19. One can pick how many features to include by identifying the point where adding a new feature has a significant drop in variance explained relative to the previous feature, and choosing features up until that point. Because each eigenvalue is roughly the importance of its corresponding eigenvector, the proportion of variance explained is the sum of the eigenvalues of the features kept divided by the sum of the eigenvalues of all features.

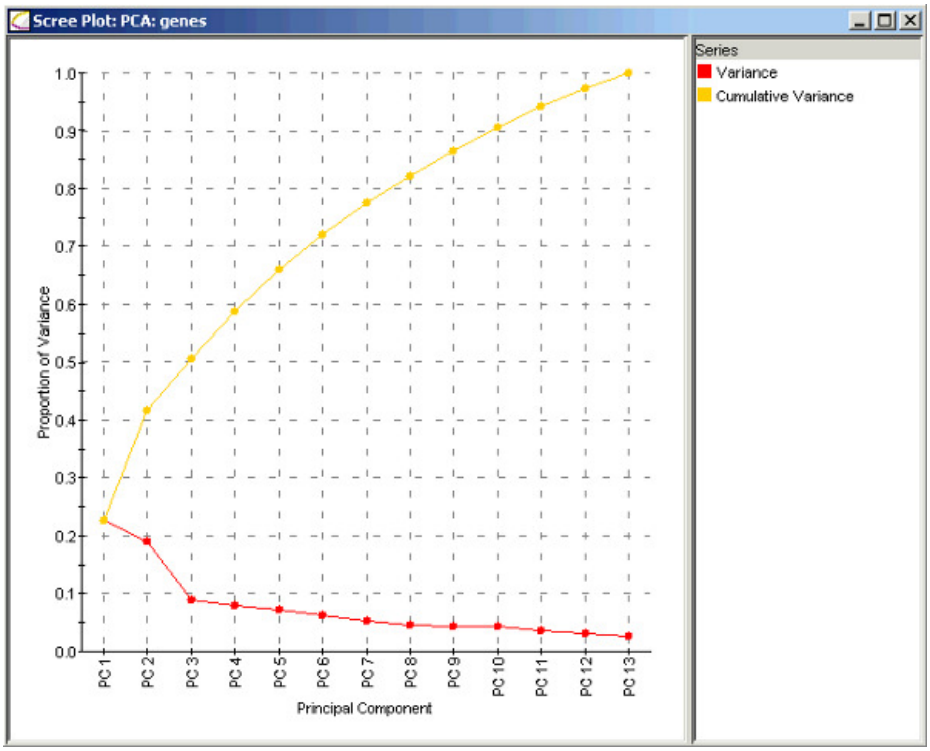


Figure 19 Scree Plot

The red line indicates the proportion of variance explained by each feature, which is calculated by taking that principal component's eigenvalue divided by the sum of all eigenvalues. The proportion of variance explained by including only principal component 1 is $\frac{\lambda_1}{\lambda_1 + \lambda_2 + \ldots + \lambda p}$, which is about 23%. The proportion of

variance explained by including only principal component 2 is $\frac{\lambda_2}{\lambda_1 + \lambda_2 + \ldots + \lambda p}$, or about 19%. The proportion of variance explained by including both principal components 1 and 2 is $\frac{\lambda_1 + \lambda_2}{\lambda_1 + \lambda_2 + \ldots + \lambda p}$which is about 42%. The yellow line indicates the cumulative proportion of variance explained by principal components up to that point.

3.2.2.3 Relationship between SVD and PCA

PCA is essentially based on calculation of eigenvector on covariance matrix. This type of calculation brings with it different problems from a computational point of view (eigenvector decomposition is computationally expensive) and from a memory point of view (having to calculate the covariance matrix). For this reason the most used algorithm for the PCA is the Singular Value Decomposition, known as SVD. This decomposition, differently from eigendecomposition, can be applied also on rectangular matrices. Typically, in a dataset, number of points is $n \gg p$ number of features. Considering dataset X, Singular Value Decomposition of X is:

$$X = USV^T$$

The matrices U and V are column-orthonormal, meaning that as vectors, the columns are orthogonal, and their lengths are 1. The matrix Σ is a diagonal matrix, and the values along its diagonal are called **singular values**. From here, one can easily see that the covariance matrix is:

$$C = \frac{VSU^T}{n-1} = V\frac{S^2}{n-1}V^T$$

meaning that right singular vectors V are principal directions and that singular values are related to the eigenvalues of covariance matrix via $\lambda_i = \frac{s_i^2}{n-1}$. Principal

components are given by $XV = USV^TV = US$. To reduce the dimensionality of the data from p to $k < p$, select k first columns of U and then multiply for the data.

3.2.2.4 Why PCA works and when it fails

PCA is a very technical but relatively intuitive method. First, the covariance matrix is a matrix that contains estimates of how every variable relates to every other variable. Understanding how one variable is associated with another is quite powerful. Second, eigenvalues and eigenvectors are important. Eigenvectors represent directions. One can think of an individual eigenvector as a particular "direction" in scatterplot of data. Eigenvalues represent magnitude, or importance. Bigger eigenvalues correlate with directions that are more important. Finally, the assumption that more variability in a particular direction correlates with explaining the behaviour of the dependent variable. Lots of variability usually indicates signal, whereas little variability usually indicates noise. Thus, the more variability there is in a particular direction is, theoretically, indicative of information. Thus, PCA is a method that brings together:

- A measure of how each variable is associated with one another. (Covariance matrix)
- The directions in which our data are dispersed. (Eigenvectors)
- The relative importance of these different directions. (Eigenvalues)

PCA combines our predictors and allows to drop the eigenvectors that are relatively unimportant. While, the projection method is simple and effective in performing dimensionality reduction. If there are overlapping instances, simply projecting data towards a hyperplane may result in the loss of important information. If the Swiss Roll is projected into 2D plane (*e.g. by dropping x3*), PCA would simply squash the various layers together and lose all the information.

The aim is to unroll the Swiss roll obtaining a 2D data set without much loss of information. This technique is called **Manifold Learning**. For instance, in Swiss Roll example, the objective of the algorithm would be to learn the optimal way to unfold Swiss Roll in enabling us to capture as much of the information as possible.

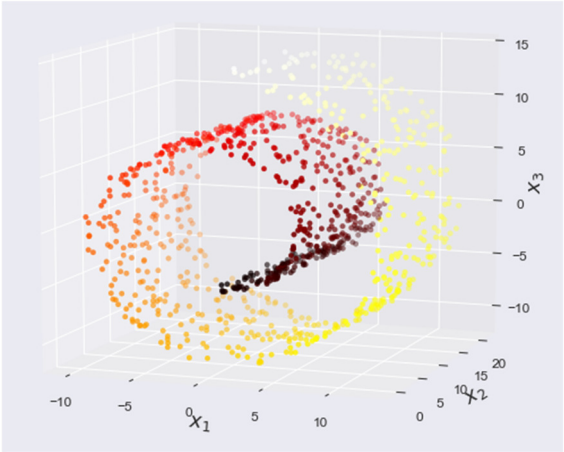


Figure 20 Swiss Roll

3.2.2.5 Example of Application

PCA is widely used in several field: quantitative finance, neuroscience, computer vision, etc... However, the field of computer vision is definitely the one where it finds greater application for different purposes such as pattern recognition, image compression, etc...

- **PCA for pattern recognition**: Images can be considered as a matrix of values N pixels high by N pixels wide. For each image it's possible to create an image vector and insert this victor in a big image-matrix. PCA is applied on this big image-matrix. Then, the problem is, given a new image, whose face from the original set is it? The way this is done is computer vision is to measure the difference between the new image and the original images, but not along the original axes, along the new axes derived from the PCA analysis. It turns out that these axes works much better for

recognising faces, because the PCA analysis has given the original images in terms of the differences and similarities between them. PCA analysis has identified the statistical patterns in the data.

- **PCA for image compression**. The idea of image compression is to create a vector for each pixel of the dataset images. Each item in the vector is from a different image. Then one performs the PCA on this set of data getting N eigenvectors because each vector is N-dimensional. To compress the data some neglects some of the eigenvectors. However, when the original data is reproduced, the images have lost some of the information. This compression technique is said to be **lossly** because the decompressed image is not exactly the same as the original, generally worse.

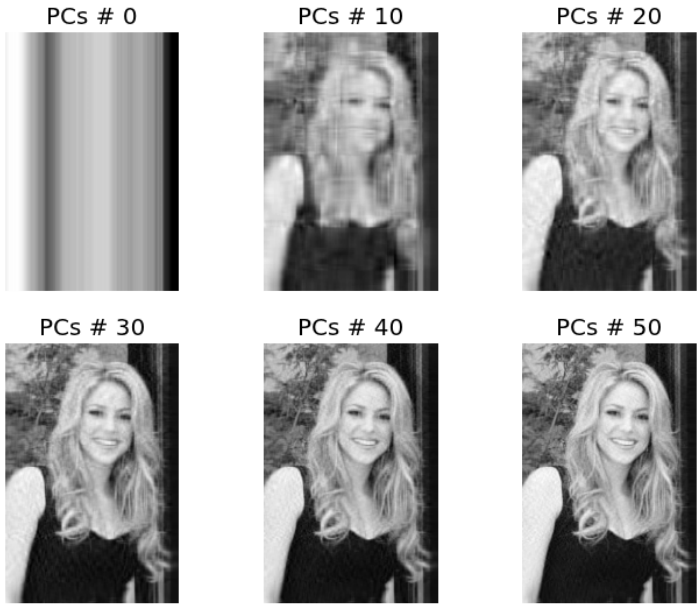


Figure 21 Image compression using PCA

3.2.3 Traning Set Reduction using PCA

After preprocessing using bubbles, the second step was dimensionality reduction using PCA. Figure 11 reports the plot of the cumulative sum of eigenvalues of the input covariance. After 15 components, the cumulative sum is over 0.999. The choice made here is to take into account the first $r = 15$ components. Let $\boldsymbol{U} \in \boldsymbol{R^{m \times r}}$ be the matrix of the first r singular vectors, the ith compressed input will be:

$$\boldsymbol{v_i^* = U^T v_i}$$

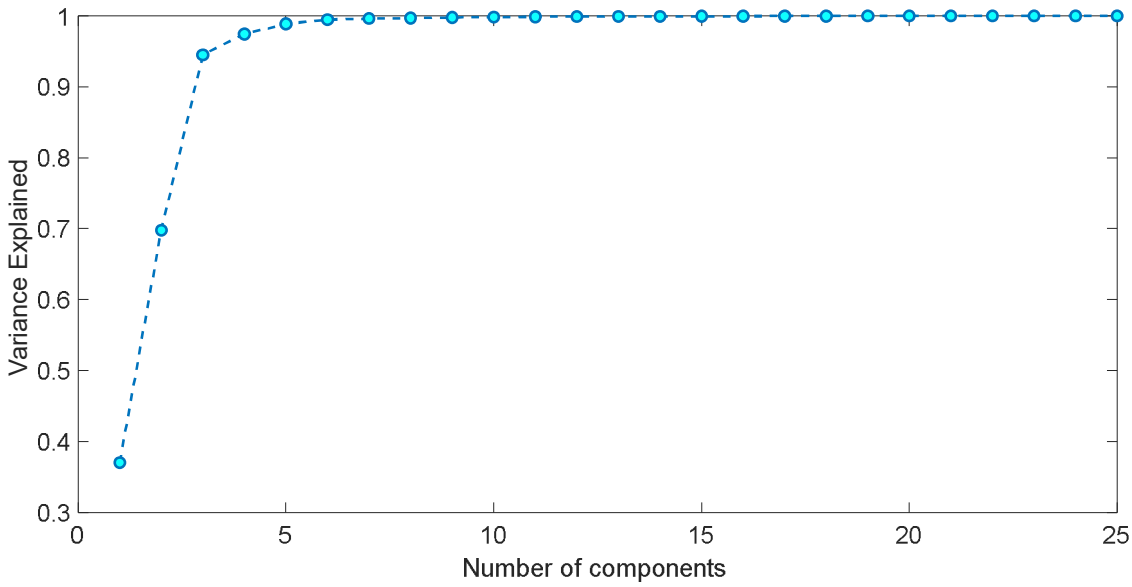


Figure 22 Plot variance explained vs number of components

Chapter 4 Machine Learning Algorithms

4.1 What is Machine Learning?

A lot of people therefore have this misconception that artificial intelligence was designed to replace humans and whatever we do in our daily work or at home. AI was (and is being) developed for the sole purpose of augmenting our lives and amplifying our skills and capabilities in all that we do. We are entering an age where man and machine will collaborate ever more closely. Arthur Samuel in 1959 coined the world Machine Learning

"Machine learning is a branch of data analytics where the machine based on the input Models (Experience) predicts certain behaviours and also learn to adapt without much programming intervention."

Tom M. Mitchell provided a widely quoted, more formal definition of the algorithms studied in the Machine Learning field:

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

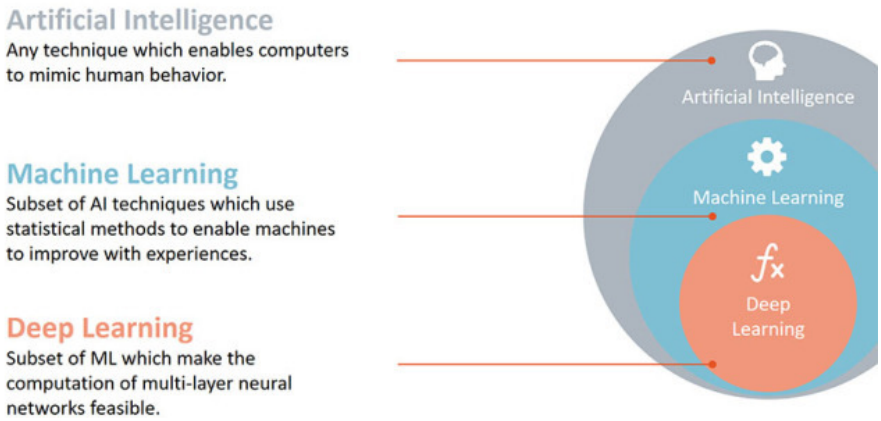


Figure 23 Machine Learning as subfield of Artificial Intelligence

Machine Learning is a key finding in this digital evolution and it's undoubtedly going to shape the future. Our computers are no more used for only simple calculations they are capable of processing petabytes of data in seconds , so ML algorithms when supplemented with right set of data can be a game changer for sectors like manufacturing, healthcare, auto industries, Banking, financial, science...

Figure 24 Machine Learning fields of application

There are three most regularly listed categories of Machine Learning:

- **Supervised Learning**: The group of algorithms that require dataset which consists of example input-output pairs. Each pair consists of data sample used to make prediction and expected outcome called **label**. Word "supervised" comes from a fact that labels need to be assigned to data by the human supervisor. There are two main problems that can be solved with Supervised Learning:
    - **Classification**: process of assigning category to input data sample. Example usages: predicting whether a person is ill or not, detecting fraudulent transactions, face classifier.
    - **Regression**: process of predicting a continuous, numerical value for input data sample. Example usages: assessing the house price, forecasting grocery store food demand, temperature forecasting.

- **Unsupervised Learning**: Group of algorithms that try to draw inferences from **non-labeled data** (without reference to known or labeled outcomes). In Unsupervised Learning, there are no correct answers. Models based on this type of algorithms can be used for discovering **unknown data patterns** and data structure itself.

The most common applications of Unsupervised Learning are:

  - **Pattern recognition and data clustering**: Process of dividing and grouping similar data samples together. Groups are usually called **clusters**. Example usages: segmentation of supermarkets, user base segmentation, signal denoising.

  - **Reducing data dimensionality**: Data dimension is the number of features needed to describe data sample. Dimensionality reduction is a process of compressing features into so-called **principal values** which conveys similar information concisely. By selecting only a few components, the amount of features is reduced and a small part of the data is lost in the process. Example usages: speeding up other Machine Learning algorithms by reducing numbers of calculations, finding a group of most reliable features in data.

  - **Reinforcement Learning**: It is a branch of Machine Learning algorithms which produces so-called **agents**. The agent role is slightly different than classic model. It's to receive information from the environment and react to it by performing an action. The information is fed to an agent in form of numerical data, called **state**, which is stored and then used for choosing right action. As a result, an agent receives a **reward** that can be either positive or

negative. The reward is a feedback that can be used by an agent to update its parameter

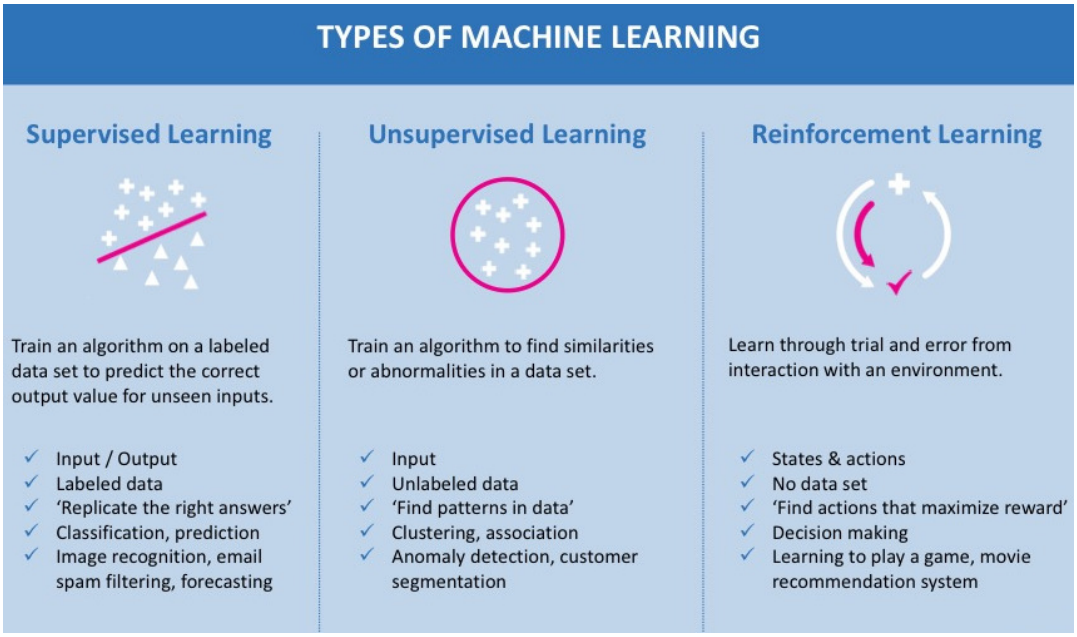


Figure 25 Machine Learning categories

4.2 Machine Learning Training Setup

This paragraph aims to present the approach used to compare the different algorithms presented in the following pages. The block diagram of the training of the Machine Learning algorithms, presented in Figure 26, is essentially the same. Search for best hyperparameters will be explained later.

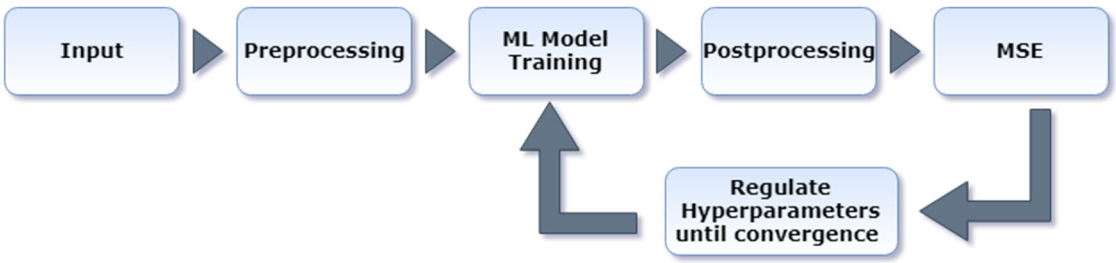
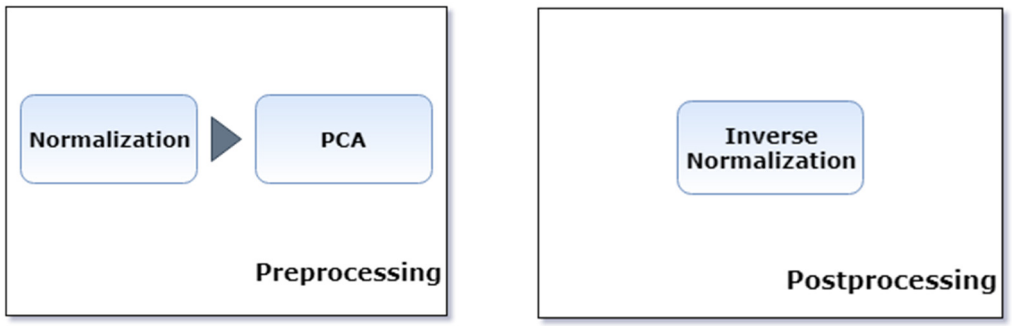


Figure 26 Prediction data flow

In the block of preprocessing (Fig. 27 (a)) data are scaled to values in the range $[-1,1]$ and then, when all the features are in the same scale, PCA is applied, using SVD. The output of the Machine Learning must return in the same scale it had before, so a normalization inverse is applied (Fig. 27(b)). These steps were easily made using Scikit-Library presented in Python.



(a) (b)

Figure 27 (a) Preprocessing block. (b) Postprocessing block.

When training is complete, model can be easily exported in a JSON file and it can be loaded at any time. Prediction data flow is represented in fig. 28. Last step, is the visualization of the results in a plot. Furthermore, MSE is computed on test set, for each of 6 axis.

Figure 28 Prediction data flow

4.3 Decision Tree for Regression

Decision Tree algorithm belongs to the family of supervised learning algorithms. Unlike other supervised learning algorithms, decision tree algorithm can be used

for solving **regression and classification problems** too. The general motive of using Decision Tree is to create a training model which can use to predict class or value of target variables by **learning decision rules** inferred from prior data (training data). A Decision Tree is a hierarchically organized structure, with each node splitting the data space into pieces based on value of a feature. Equivalent to a partition of R_d into K disjoint feature subspaces $\{R_1, \ldots, R_K\}$, where each $R_j \subset R_d$ On each feature subspace R_j, the same decision/prediction is made for all $x \in R_j$. First to understand how a regression tree works, a bit of terminology is necessary:

- **Parent** of a node c is the immediate predecessor node.
- **Children** of a node c are the immediate successors of c, equivalently nodes which have c as a parent.
- **Root node** is the top node of the tree, the only node without parents.
- **Leaf nodes** are nodes which do not have children.
- A **K-ary tree** is a tree where each node (except for leaf nodes) has K children. Usually working with binary trees ($K = 2$).
- **Depth** is the maximal length of a path from the root node to a leaf node.

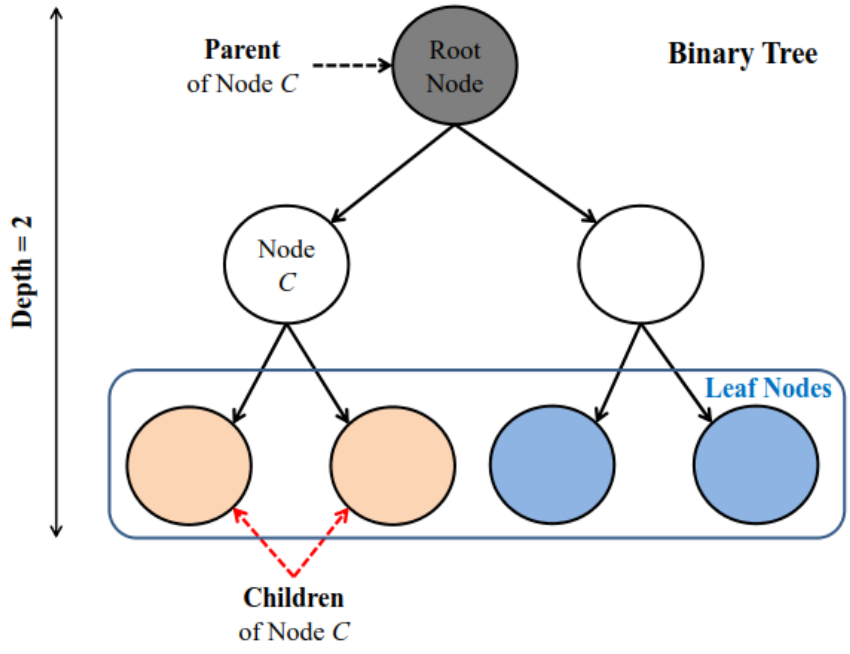


Figure 29 Tree structure and terminology

Differently from Decision Tree, Regression Trees minimize the error (e.g. square loss error) in each leaf. First assume that x is univariate and training samples are arranged so that $x_1 < x_2 < \cdots < x_n$. Given a set of observations of the dependent variable y, what is the one number y_A that best characterizes those N_1 values of y. Minimizing $E\,[y - y_A]^2$, the value that minimizes this is $y_A = E[y]$. Approximation of $E[y]$ is the sample mean, which is $\overline{y_A}$ .The root node becomes a parent node which spawns two children. Each of the children becomes a parent node which in turn spawn two more children. Hence a parent node is always "above" its children. After the root node is constructed, two children node are created in the following manner: Determine a "split" value of $\boldsymbol{x}$, termed $\boldsymbol{x_s}$ so that if $\boldsymbol{x} \leq \boldsymbol{x_s}$, so prediction is $y(p) = \overline{y_L}$ and if $\boldsymbol{x} \geq \boldsymbol{x_s}$, prediction is $y(p) = \overline{y_R}$ (L for "left" and R for "right"). It would be delightful if that split on x simultaneously split y such that all the sample values of y that arrive in the left node are identical (standard deviation is zero) and all the values of y in the right node were a different value of y, but again with a standard deviation of zero. This is probably not going to happen so the algorithm finds the value of x_s such that **the total squared error** (TSE) is minimized:

$$TSE = \sum_{i:x \leq x_s} [y_i - \overline{y_L}]^2 + \sum_{i:x > x_s} [y_i - \overline{y_R}]^2$$

This is equivalent to minimizing $TSE = n_L\sigma_L^2 + n_R\sigma_R^2$ where n_L is the number of training sample that end up in the left node and σ_L is the sample standard deviation (with n_L in the denominator) and σ_R and n_R refer to the right hand node. At this stage, all the training examples sit in one of the two children nodes and the predicted value of the examples in each of the nodes is the mean of the samples in that node. For a new test example, depending on whether the value of the independent variable is less than or greater than the split value of the root node, the predicted value would either $\overline{y_L}$ or $\overline{y_R}$. For univariate x, the algorithm

recursively split at each node, generating two children nodes from each parent node until the variance within each node is zero. This could happen if there is only a single sample in a node or all the samples have the same value of the dependent variable. If $\boldsymbol{x}$ is multivariate, then at each node, we examine each feature of $\boldsymbol{x}$, and determine its best split value. That feature with the minimum TSE becomes the feature to split on. Every time the algorithm determines two new nodes from a parent node, the TSE decreases until it decreases to zero. However, this is done on the training set, improve generalization tree is pruned.

4.3.1 Regression Tree Example

Predict a baseball player's Salary based on Years (the number of years that he has played in the major leagues) and Hits (the number of hits that he made in the previous year). First step is to remove observations that are missing. Salary values, and log-transform Salary so that its distribution has more of a typical bell-shape. Recall that Salary is measured in thousands of dollars.

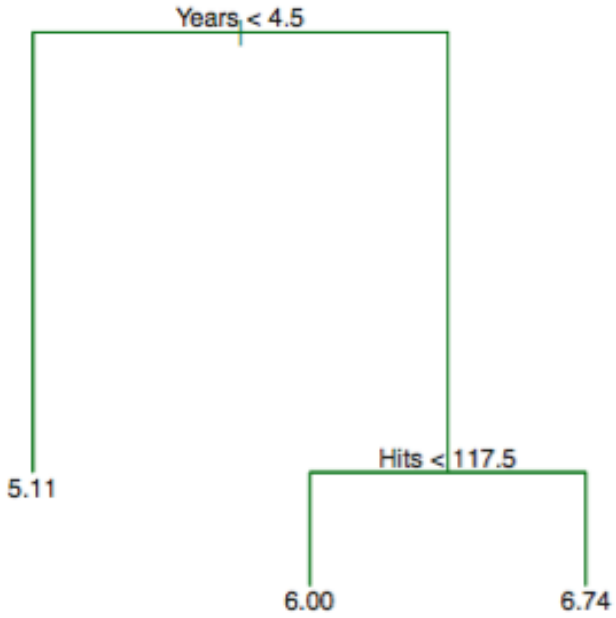


Figure 30 Tree representation of salary

The tree represents a series of splits starting at the top of the tree.

- The top split assigns observations having $Years < 4.5$ to the left branch.
- The predicted salary for these players is given by the mean response value for the players in the data set with $Years < 4.5$.
- For such players, the mean log salary is 5.107, and so we make a prediction of $e^{5.107}$ thousands of dollars, i.e. $165, 174$.

Regions delimited by the classifier can be described with:

1. $R1 = X \mid Years < 4.5$
2. $R2 = X|Years \geq 4.5, Hits < 117.5$
3. $R2 = X|Years \geq 4.5, Hits \geq 117.5$

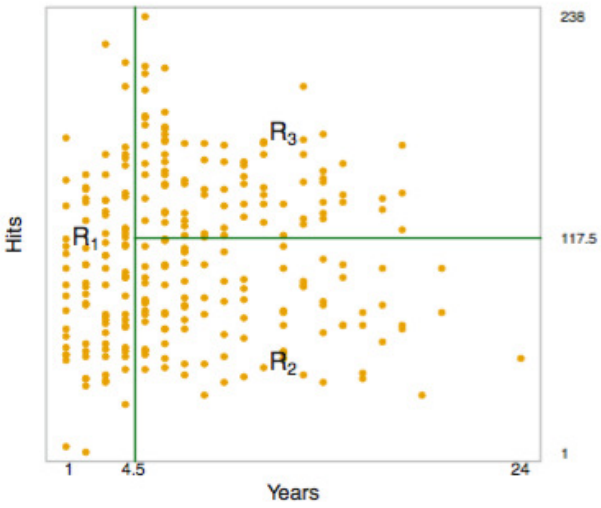


Figure 31 Regions delimited from the tree

In keeping with the tree analogy, the regions $R1$, $R2$, and $R3$ are known as terminal **nodes** or **leaves** of the tree. Decision trees are typically drawn upside down, in the sense that the leaves are at the bottom of the tree. The points along the tree where split happens are referred to as **internal nodes**. The two internal nodes are indicated by the text $Years < 4.5$ and $Hits < 117.5$. The segments of the trees that connect the nodes as **branches**. How it's interpretable this tree? Years is the most important factor in determining Salary, and players with less experience earn lower salaries than more experienced players. Given that a player is less experienced, the number of hits that he made in the previous year seems

to play little role in his salary. But among players who have been in the major leagues for five or more years, the number of hits made in the previous year does affect salary, and players who made more hits last year tend to have higher salaries. The regression tree shown in figure is likely an over-simplification of the true relationship between Hits, Years, and Salary, but it's a very nice easy interpretation over more complicated approaches.

4.3.2 Stopping Criteria

All decision trees need stopping criteria or it would be possible, and undesirable, to grow a tree in which each case occupied its own node. The resulting tree would be computationally expensive, difficult to interpret and would probably not work very well with new data. Essentially 3 stopping rules are adopted:

1. The node depth is equal to the **maxDepth** training parameter.
2. The TSE obtained by generating two children nodes from a parent node does not reduce the TSE (from the parent node) by at least a certain **threshold**, i.e. 5%.
3. No split candidate produces child nodes which have at least **minInstancesPerNode** training instances each.

4.3.3 Overfitting

Overfitting is a practical problem while building a decision tree model. The model is having an issue of overfitting is considered when the algorithm continues to go deeper and deeper in the to reduce the training set error but results with an

increased test set error i.e, accuracy of prediction goes down. It generally happens when it builds many branches due to outliers and irregularities in data.

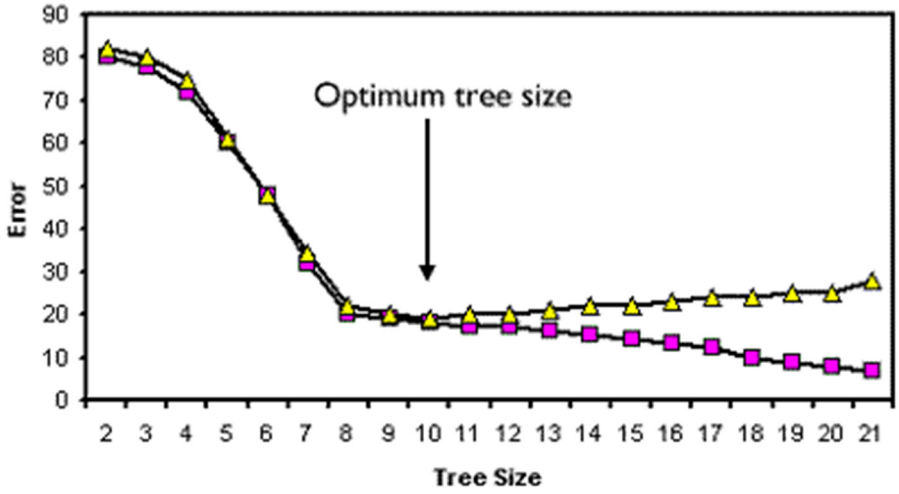


Figure 32 Plot error vs Tree size

Two approaches which we can use to avoid overfitting are:

- **Pre-Pruning**: Pre-pruning is also called forward pruning or online-pruning. Pre-pruning prevent the generation of non-significant branches. Pre-pruning a decision tree involves using a "termination condition" to decide when it is desirable to terminate some of the branches prematurely. When constructing the tree some significant measures can be used to understand the goodness of a split. If partitioning the tuples at a node would result the split that falls below a threshold, then further partitioning of the given subset is halted otherwise it is expanded. High threshold result in oversimplified trees, whereas low threshold result in very little simplification.
- **Post-Pruning**: In post-pruning first, it goes deeper and deeper in the tree to build a complete tree. If the tree shows the overfitting problem then pruning is done as a post-pruning step. One of the most common pruning methods used to improve generalization is K-fold cross validation. This makes use of the existing data to simulate independent test data. First, all

of the data are used and the tree grows as large as possible. This is the reference, **unpruned tree**. Pruning starts at the terminal nodes and proceeds in a stepwise fashion, sequentially removing the least important nodes until the desired size is reached. The end product of this process is a pruned reference tree which should produce the optimum performance with new data.

4.3.4 Regression Tree Pros and Cons

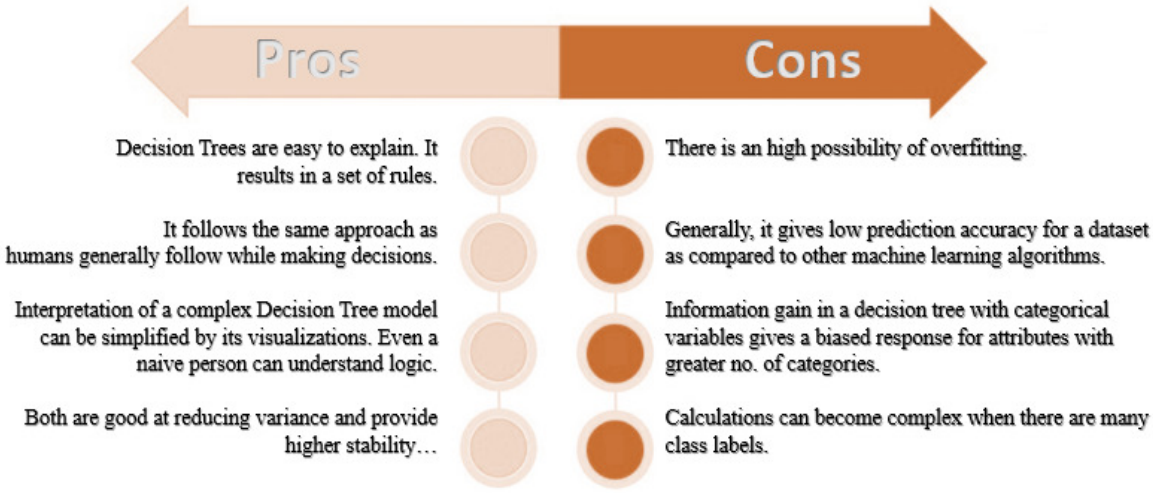


Figure 33 Advantages vs Disadvantages Regression Tree

4.4 K-NN

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

- In k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbours, with the object being assigned to the class most common among its k-nearest neighbours (k is a

positive integer, typically small). If $k = 1$, then the object is simply assigned to the class of that single nearest neighbour.

- In k-NN regression, the output is the property value for the object. This value is the average of the values of its k nearest neighbours.

K-Nearest Neighbour estimation was proposed sixty years ago, but because of the need for large memory and computation, the approach was not popular for a long time. With advances in parallel processing and with memory and computation getting cheaper, such methods have recently become more widely used. Unfortunately, it can still be quite computationally expensive when it comes to large training dataset as we need to compute the distance for each sample. Also, when we consider low-dimensional spaces and we have enough data, NN works very well in terms of accuracy, as we have enough nearby data points to get a good answer. As the number of dimensions increases the algorithm performs worst, because distance measure becomes meaningless when the dimension of the data increases significantly. k-NN is quite robust to noisy training data, especially when a weighted distance is used.

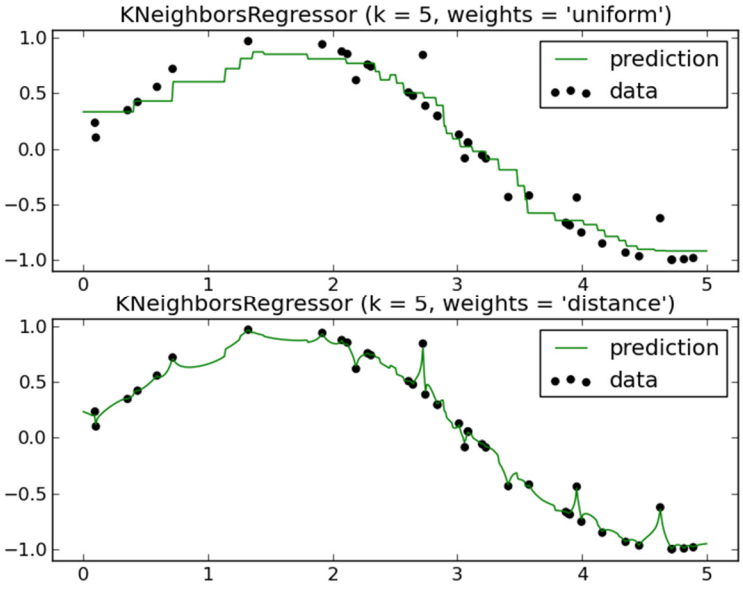


Figure 34 Weights Uniform and Weights Distance

The neighbours are taken from a set of objects for which the class (for K-NN classification) or the object property value (for K-NN regression) is known. This can be thought of as the training set for the algorithm, though no explicit training step is required. A peculiarity of the K-NN algorithm is that it is sensitive to the local structure of the data. A commonly used distance metric for continuous variables is Euclidean distance.

- Euclidean Distance $\sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$
- Manhattan Distance $\sum_{i=1}^{n}|x_i - y_i|$
- Chebyshev Distance $\max(x - y)$
- Minkowski Distance $\sqrt[p]{\sum_{i=1}^{n}(x_i - y_i)^p}$

4.4.1 Knn Pros and Cons

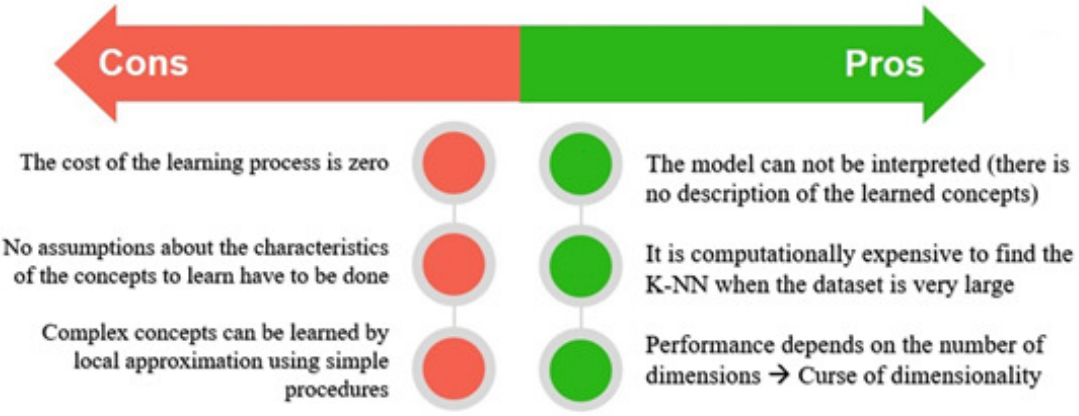


Figure 35 KNN Pros and Cons

4.4.2 K-NN Example

Consider the following table – it consists of the height, age and weight (target) value for 10 people. The weight value of ID11 is missing. The purpose is to predict the weight of this person based on their height and age. For a clearer understanding of this, below is the plot of height versus age from the above table:

ID	Height	Age	Weight
1	5	45	77
2	5.11	26	47
3	5.6	30	55
4	5.9	34	59
5	4.8	40	72
6	5.8	36	60
7	5.3	19	40
8	5.8	28	60
9	5.5	23	45
10	5.6	32	58
11	5.5	38	?

Figure 36 Table of height, age, weight

In the above graph, the y-axis represents the height of a person (in feet) and the x-axis represents the age (in years). The points are numbered according to the ID values. The yellow point (ID 11) is the test point. The algorithm uses "**feature similarity**" to predict values of any new data points. This means that the new point is assigned a value based on how closely it resembles the points in the training set. From the example, we know that ID11 has height and age similar to ID1 and ID5, so the weight would also approximately be the same.

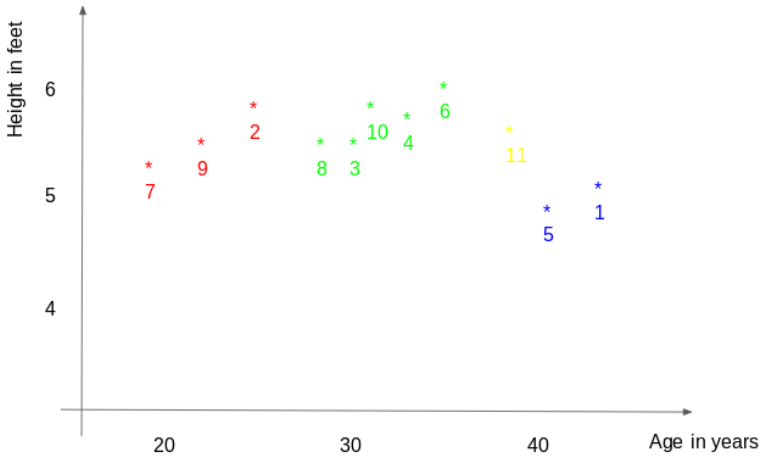


Figure 37 2D Visualization of the table

The average of the values is taken to be the final prediction. Below is a stepwise explanation of the algorithm:

1. First, the distance between the new point and each training point is calculated.

2. The closest k data points are selected (based on the distance). In this example, points 1, 5, 6 will be selected if value of k is 3.

3. The average of these data points is the final prediction for the new point. Here, weight of $ID11 = \frac{77+72+60}{3} = 69.66\,kg$.

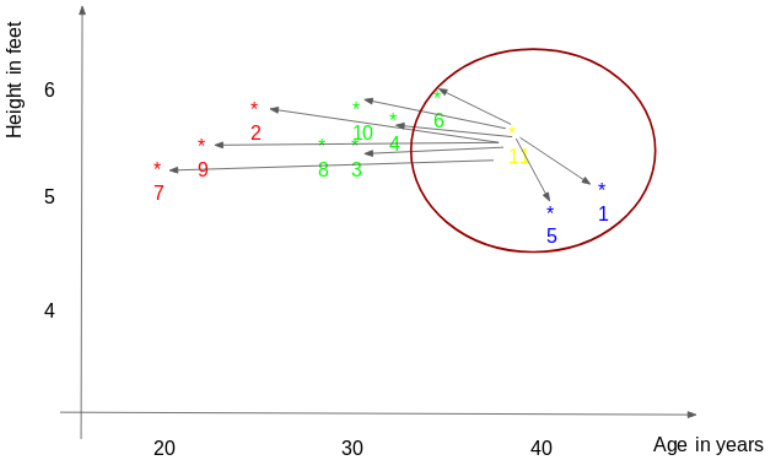


Figure 38 3-NN Regression

The second step is to select the k value. Depending of k, the final result tends to change. Then how it's possible to figure out the optimum value of k? It can be decided on the error calculation for train and validation set. For a very low value of k the model overfits on the training data, which leads to a high error rate on the validation set. On the other hand, for a high value of k, the model performs poorly on both train and validation set. Observing closely Fig.39, the validation error curve reaches a minima at a value of $k = 9$. This value of k is the optimum value of the model (it will vary for different datasets). This curve is known as an "**elbow curve**" and It is usually used to determine k.

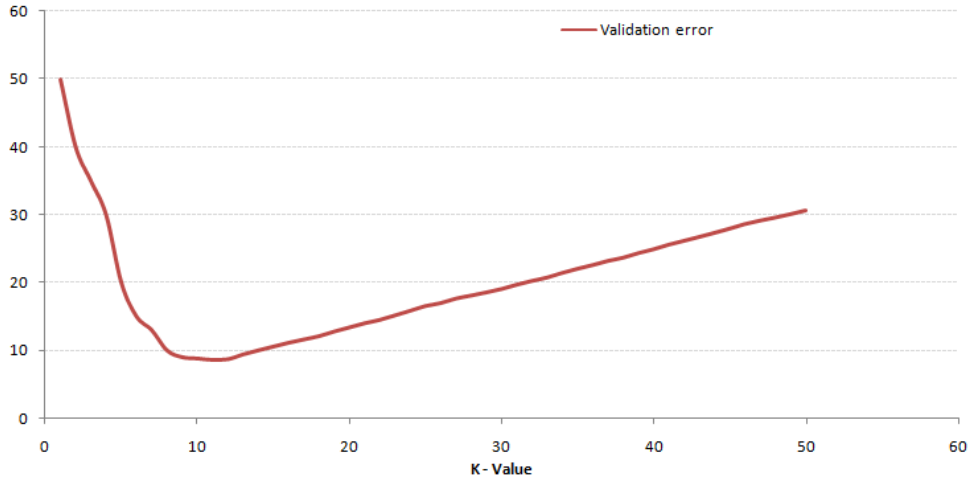


Figure 39 Elbow curve

4.4.3 Nearest Neighbours Algorithm

- **Brute Force**. The most naive neighbor search implementation involves the brute-force computation of distances between all pairs of points in the dataset: this approach scales as $O[DN^2]$. Efficient brute-force neighbors searches can be very competitive for small data samples. However, as the number of samples grows $(n \geq 30)$, the brute-force approach quickly becomes infeasible. Brute force query time is unchanged by data structure and it is largely unaffected by the value of k.

- **K-D Tree**. These structures attempt to reduce the required number of distance calculations by efficiently encoding aggregate distance information for the sample. Using K-D Tree computational cost of a nearest neighbors search can be reduced to $O[DNlogN]$ or better. The construction of a KD tree is very fast: because partitioning is performed only along the data axes, no D-dimensional distances need to be computed. Once constructed, the nearest neighbor of a query point can be determined with only $O[logN]$ distance computations. Though the KD tree approach is very fast for low-dimensional $(D < 20)$ neighbors searches, it becomes inefficient as grows very large: this is one manifestation of the so-called "**curse of**

dimensionality". K-D Tree is sensitive to structure of the data and becomes slower as k increases.

- **Ball Tree**. To address the inefficiencies of KD Trees in higher dimensions, the ball tree data structure was developed. Where KD trees partition data along Cartesian axes, ball trees partition data in a series of nesting hyper-spheres. This makes tree construction more costly than that of the KD tree, but results in a data structure which can be very efficient on highly structured data, even in very high dimensions. Computational complexity is $O[DlogN]$ however performance is highly dependent on the structure of the training data. Ball Tree becomes slower as k increases.

4.4.4 K-NN Calibration Results

After a dimensionality reduction using PCA,data have been divided in training set and test set in a random way. First step was to understand the optimal number of neighbours. For this reason the elbow curve has been drawn in Fig 40.

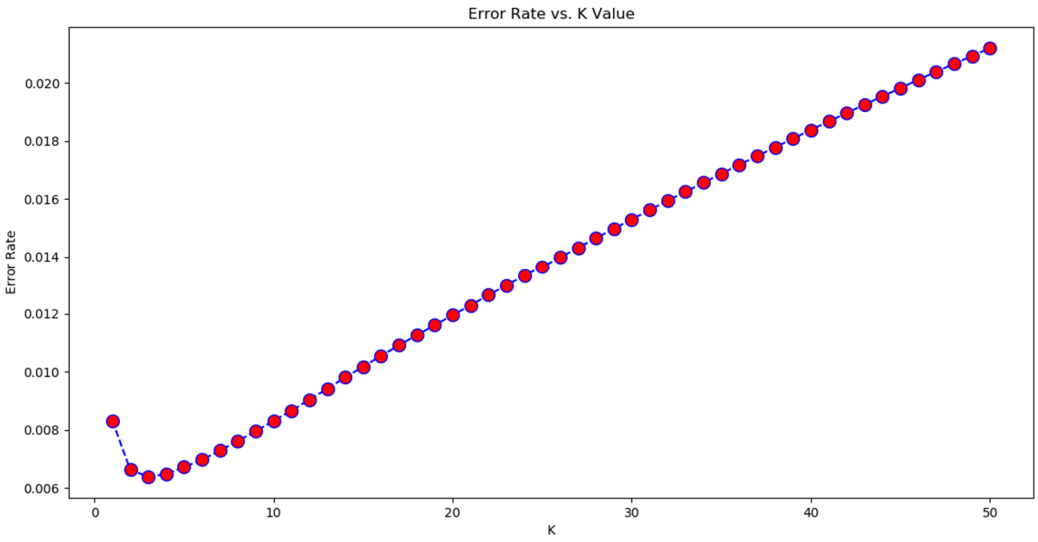


Figure 40 Elbow curve calibration dataset

The value chosen, according to elbow curve, was of 3. Different metrics have been compared, using the same value of neighbours. As show in the image below, when samples are weighted according to distance mean squared error calculated on test set is lower.

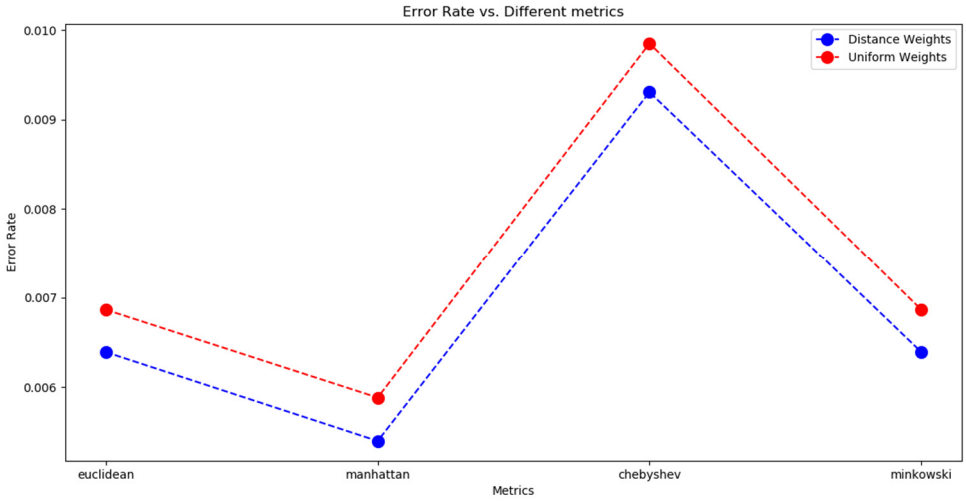


Figure 41 Error Rate vs Metric using different weights

After choosing the model parameters, a test has been conducted. Results on a batch of 3000 samples are shown below.

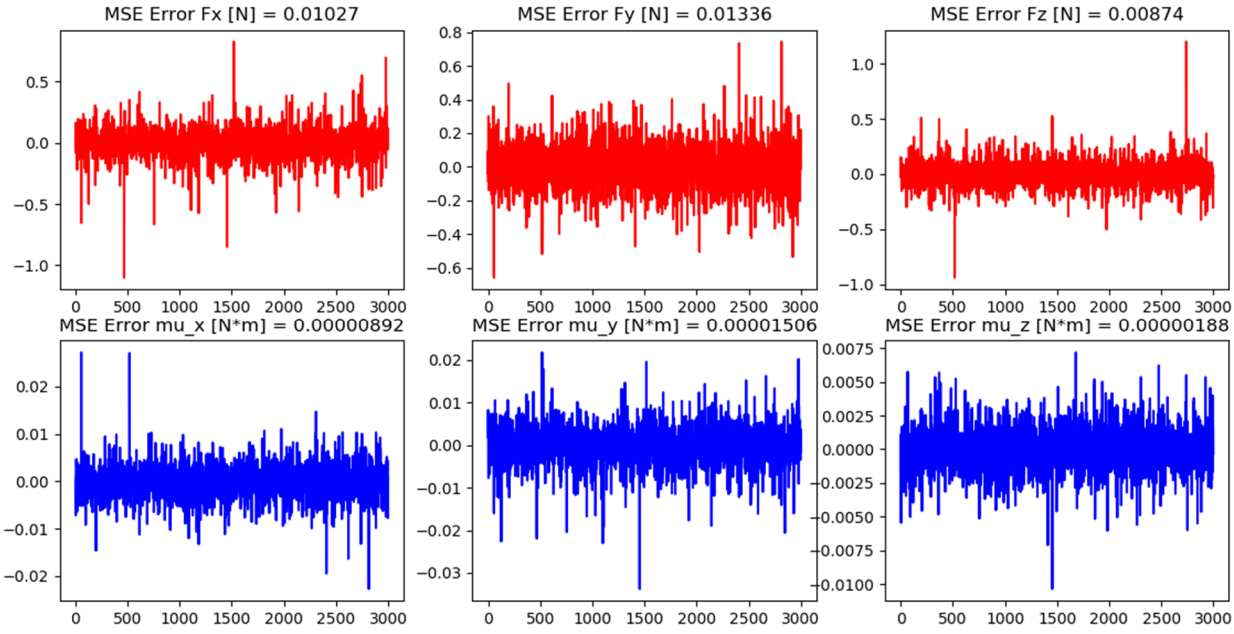


Figure 42 MSE on calibration test set

4.5 Boosting and Bagging

Bagging and Boosting are similar in that they are both ensemble techniques, where a set of weak learners are combined to create a strong learner that obtains better performance than a single one.

4.5.1 What is a weak learner?

Weak learner is a learner that no matter what the distribution over the training data is will always do better than chance, when it tries to label the data. Many possibilities for weak classifiers exist, e.g.:

- **Decision stumps**. It's a 1-Level decision tree whose geometry is a vertical or horizontal line. It is a simple test based on a single feature. Eg: If an email contains the word "money", it is a spam; otherwise, it is a non-spam
- **Decision trees**. They are a more generale version of decision stumps. At every node a decision is made.

4.5.2 What is an ensemble method?

Ensemble is a Machine Learning concept in which the idea is to train **multiple models** using the same learning algorithm. The ensembles take part in a bigger group of methods, called **multiclassifiers**, where a set of hundreds or thousands of learners with a common objective are fused together to solve the problem. The second group of multiclassifiers contain the **hybrid methods**. They use a set of learners too, but they can be trained using different learning techniques.The main causes of error in learning are due to noise, bias and variance. Ensemble helps to minimize these factors. These methods are designed to improve the **stability** and the **accuracy** of Machine Learning algorithms. Combinations of

multiple predictors decrease variance, especially in the case of unstable predictors, and may produce a more reliable prediction than a single predictor. To use Bagging or Boosting you must select a base learner algorithm. For example, if one choose a regression tree, Bagging and Boosting would consist of a pool of trees.

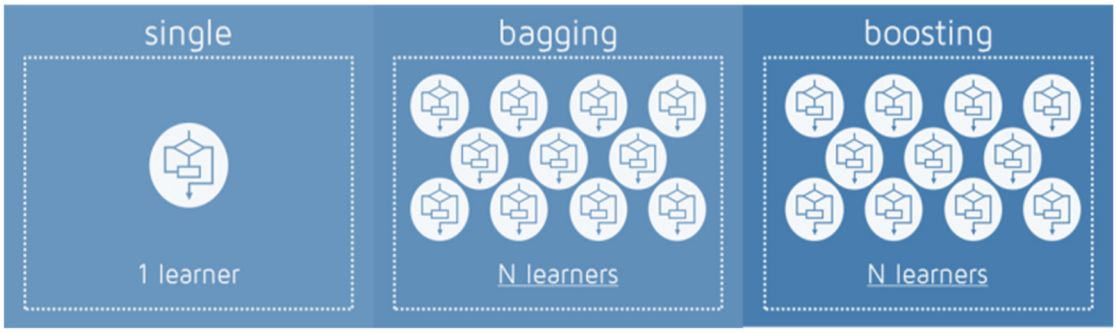


Figure 43 Boosting and bagging

4.5.3 How do Bagging and Boosting get N learners?

Bagging and Boosting get N learners by generating additional data in the training stage. N new training data sets are produced by **random sampling** with replacement from the original set. By sampling with replacement some observations may be repeated in each new training data set.
In the case of Bagging, any element has the same probability to appear in a new data set. However, for Boosting the observations are weighted and therefore some of them will take part in the new sets more often. These multiple sets are used to train the same learner algorithm and therefore different classifiers are produced.

4.5.4 Why are the data elements weighted?

While the training stage is parallel for Bagging (i.e., each model is built independently), Boosting builds the new learner in a sequential way.
In Boosting algorithms each model is trained on data, taking into account the previous predictors' results. After each training step, the weights are redistributed

mispredicted data increases its weights to emphasise the most difficult cases. In this way, subsequent learners will focus on them during their training.

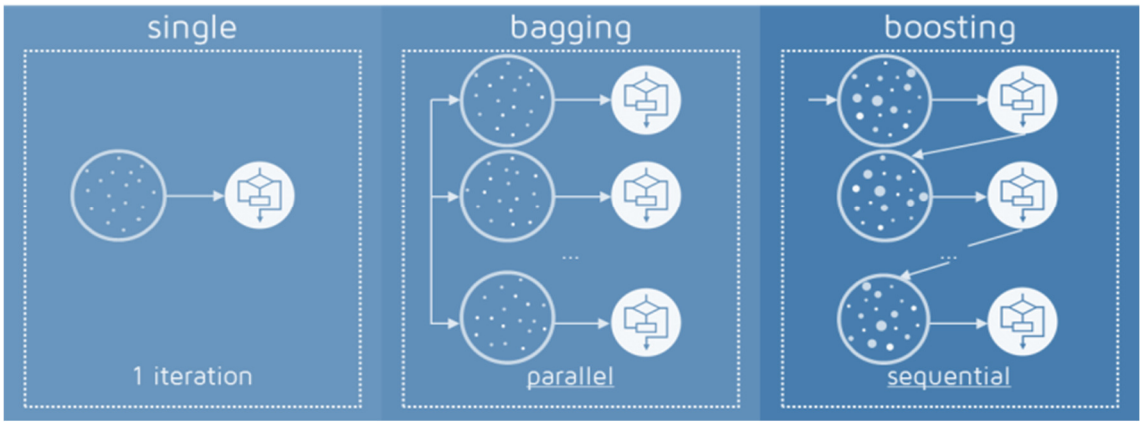


Figure 44 Bagging vs Boosting training

4.5.5 How does the prediction stage work?

Prediction step is made applying N learners to the new observations. In Bagging the result is obtained by averaging the responses of the N learners. However, Boosting assigns a second set of weights, this time for the N classifiers, in order to take a weighted average of their estimates. In the Boosting training stage, the algorithm allocates weights to each resulting model. A learner with good a prediction result on the training data will be assigned a higher weight than a poor one. So when evaluating a new learner, Boosting needs to keep track of learners' errors, too. Let's see the differences in the procedures:

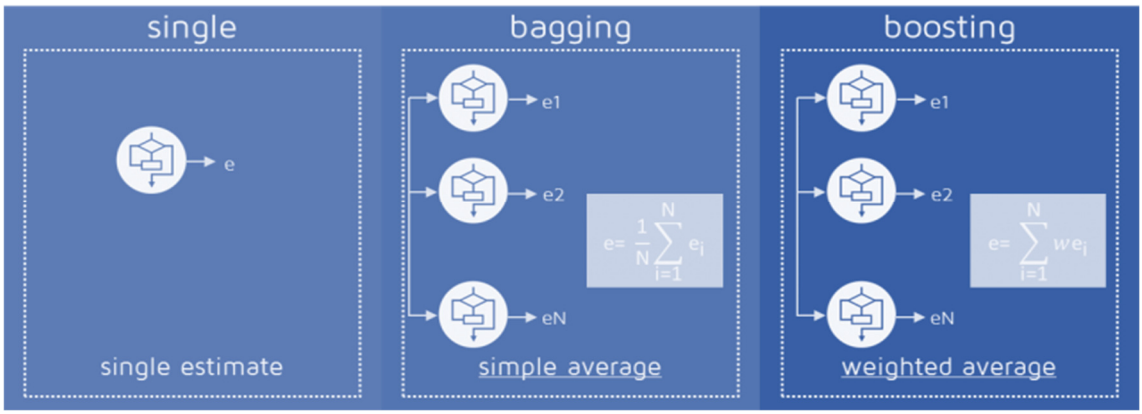


Figure 45 Prediction stage

Some of the Boosting techniques include an extra-condition to keep or discard a single learner. For example, in Adaboost, the most renowned, an error less than 50% is required to maintain the model; otherwise, the iteration is repeated until achieving a learner better than a random guess. The previous image shows the general process of a Boosting method, but several alternatives exist with different ways to determine the weights to use in the next training step and in the prediction stage.

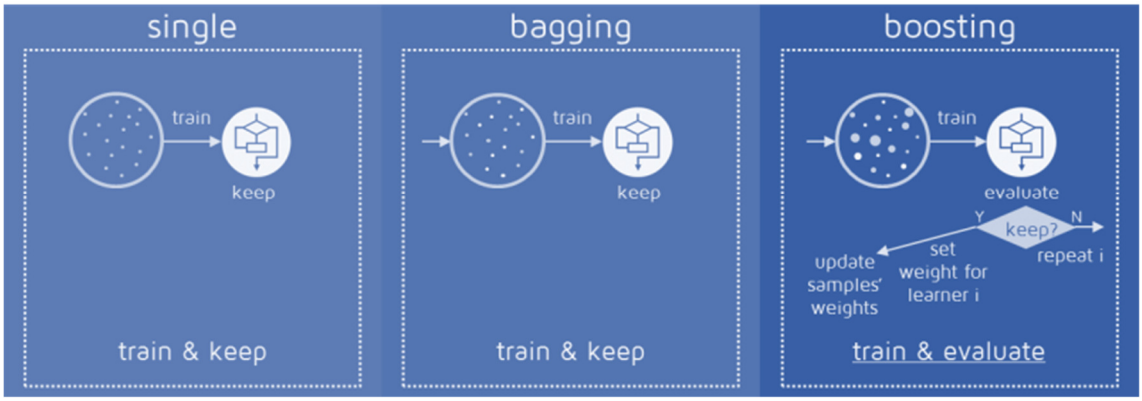


Figure 46 Bagging vs Boosting adaptation to error

4.5.6 Bagging and boosting comparison

There's not an outright winner; it depends on the data, the simulation and the circumstances. Bagging and Boosting decrease the **variance** of your single estimate as they combine several estimations from different models. So the result may be a model with higher stability. If the problem is that the single model gets a very low performance, Bagging will rarely get a better bias. However, Boosting could generate a combined model with lower errors as it optimises the advantages and reduces pitfalls of the single model. By contrast, if the difficulty of the single model is over-fitting, then Bagging is the best option. Boosting for its part doesn't

help to avoid over-fitting; in fact, this technique is faced with this problem itself. For this reason, Bagging is effective more often than Boosting.

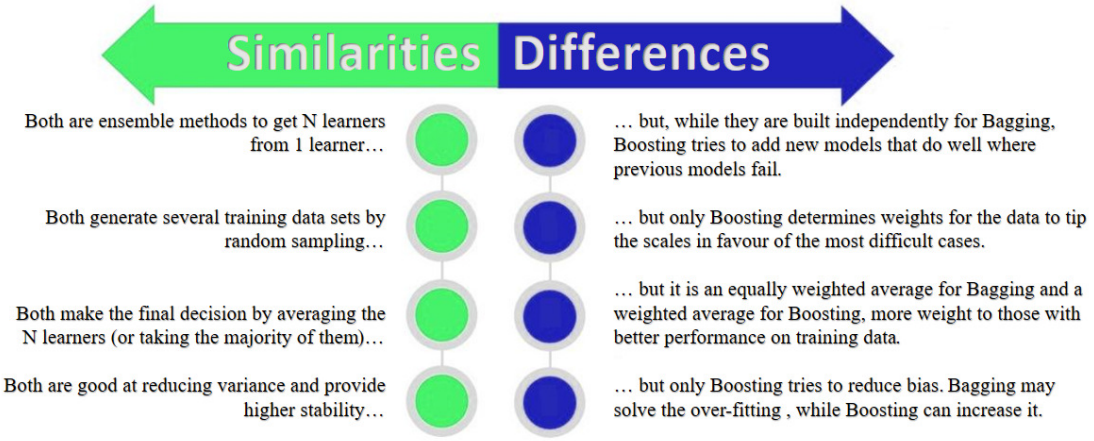


Figure 47 Similarities and differences between Boosting and Bagging

4.6 Random Forest

A Random Forest is an ensemble technique capable of performing both regression and classification tasks with the use of multiple decision trees and a technique called **Bootstrap Aggregation**, commonly known as **bagging**. Bagging, in the Random Forest method, involves training each decision tree on a different data sample where sampling is done with replacement, reducing variance.

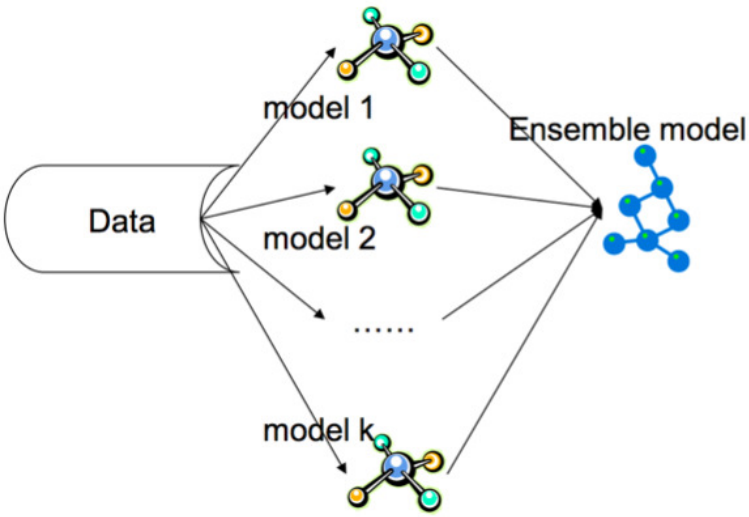


Figure 48 Random forest seen as ensembling models

Random forests for regression are formed by growing trees depending on a random vector θ such that the tree predictor $h(\boldsymbol{x}, \theta)$ takes on numerical values as opposed to labels. The output values are numerical and we assume that the training set is independently drawn from the distribution of the random vector Y,$\mathbf{X}$. The mean-squared generalization error for any numerical predictor $h(\boldsymbol{x})$ is

$$E_{\boldsymbol{X},\boldsymbol{y}} \left(Y - h(\boldsymbol{X})\right)^2$$

Prediction is made by taking the average over k of the trees $\{h(\boldsymbol{x}, \theta_k)\}$. The algorithm can be divided in two stages. In the first step, n Trees are trained selecting a random subset m of total features k, until a stopping criteria occurs. In prediction stage every tree produces its output and final output is computed averaging all predictions.

4.6.1 Out-of-Bag error

Out-of-bag (OOB) error, also called out-of-bag estimate, is a method of measuring the prediction error of random forests, boosted decision trees, and other machine learning models utilizing bagging. OOB is the mean prediction error on each training sample x_i, using only the trees that did not have x_i in their bootstrap sample. In contrast to boosting, trees are independent so they can be trained in parallel.

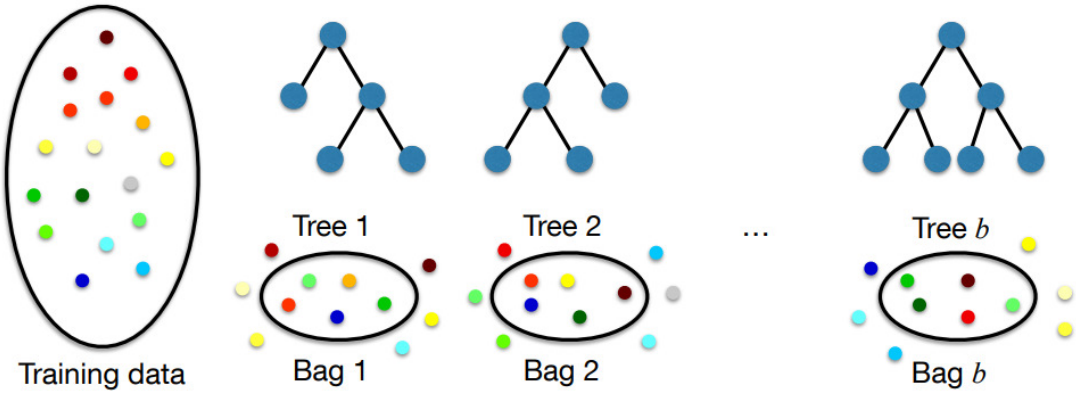


Figure 49 Bag and out of bag data

4.6.2 Important Hyperparameters

Random forests have the reputation of being relatively easy to tune. This is because they only have a few hyperparameters, and aren't overly sensitive to the particular values they take. Tuning the hyperparameters can often increase generalization performance somewhat

- **n_estimators**: the number of trees the algorithm builds before taking the maximum voting or taking averages of predictions. In general, a higher number of trees increases the performance and makes the predictions more stable, but it also slows down the computation.

- **max_features**: increasing max_features generally improves the performance of the model as at each node now we have a higher number of options to be considered. However, this is not necessarily true as this decreases the diversity of individual tree. But, for sure, it decreases the speed of algorithm by increasing the max_features.

- **min_sample_leaf**: leaf is the end node of a decision tree. A smaller leaf makes the model more prone to capturing noise in train data.

- **max_depth**: the maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

4.6.3 Pros and Cons

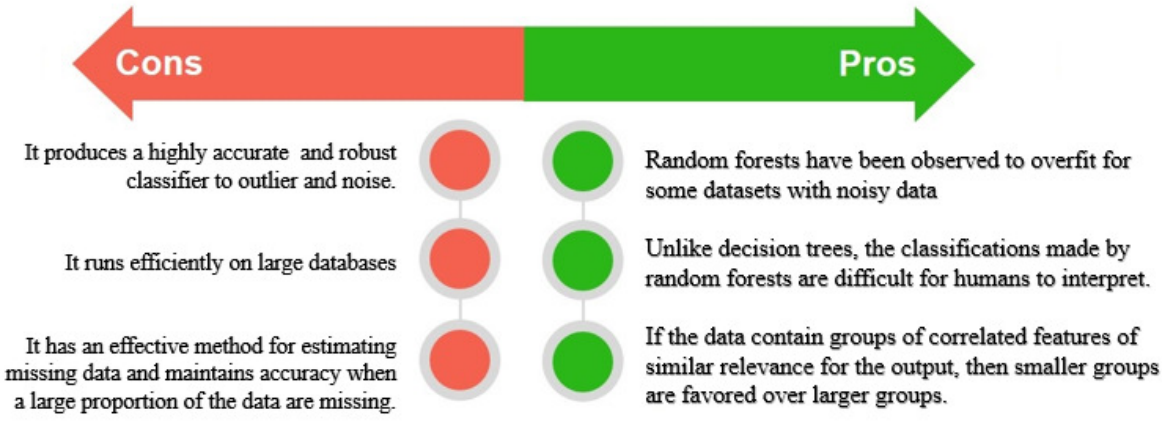


Figure 50 Advantages vs disvantages of Random Forest

4.6.4 Random Forest Calibration Results

After a dimensionality reduction using PCA, data have been divided in training set and test set in a random way. First step is to understand the number of trees along each axis. The curve decreases with a growing number of tree and this mean more stability and smaller variance in predictions. However, computations begins slower and slower, for this reason a trade-off must be chosen. The value chosen is of 100. F_z and μ_z results are in figure 51-52.

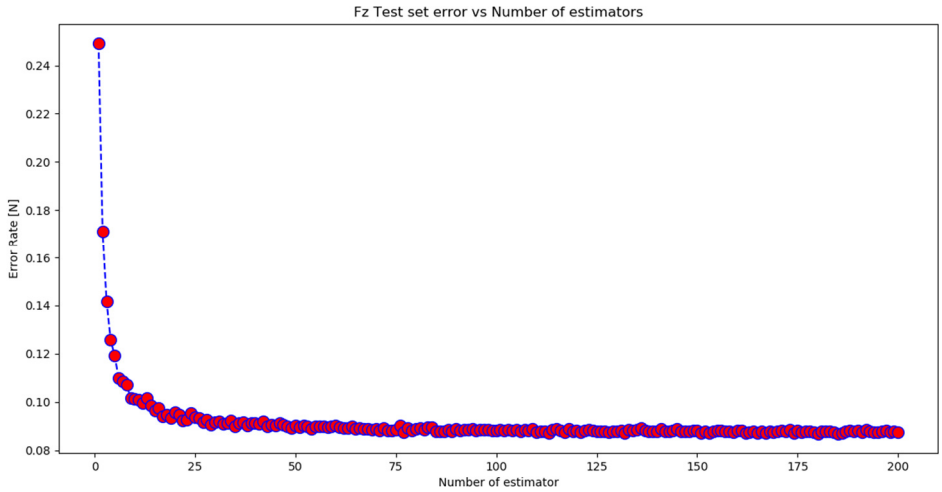


Figure 51 Plot Number of estimators of F_z

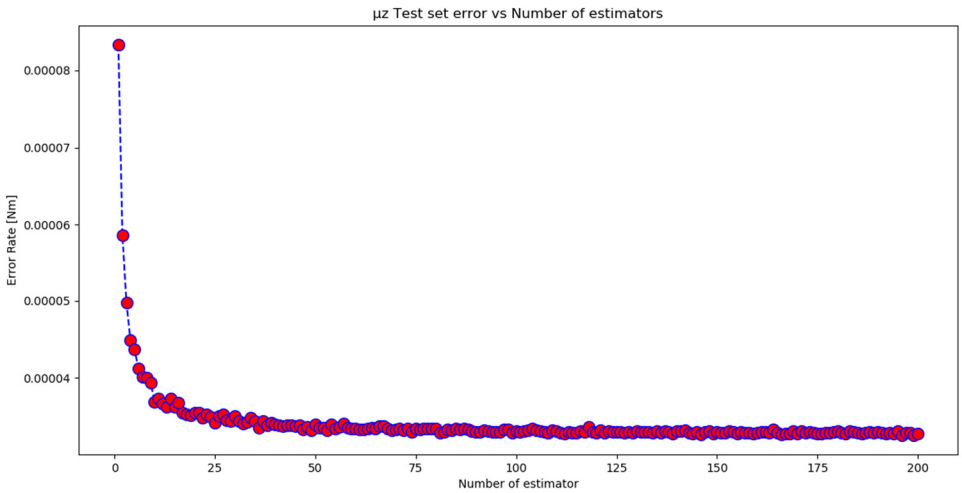


Figure 52 Plot Number of estimators of μ_z

In Random Forest at each node a splitting is made according to the criterion adopted. In regression case, criterion is Mean Squared Error or Mean Absolute Error. Performance are about equal (Figure 53-54), however training is slower in MAE.

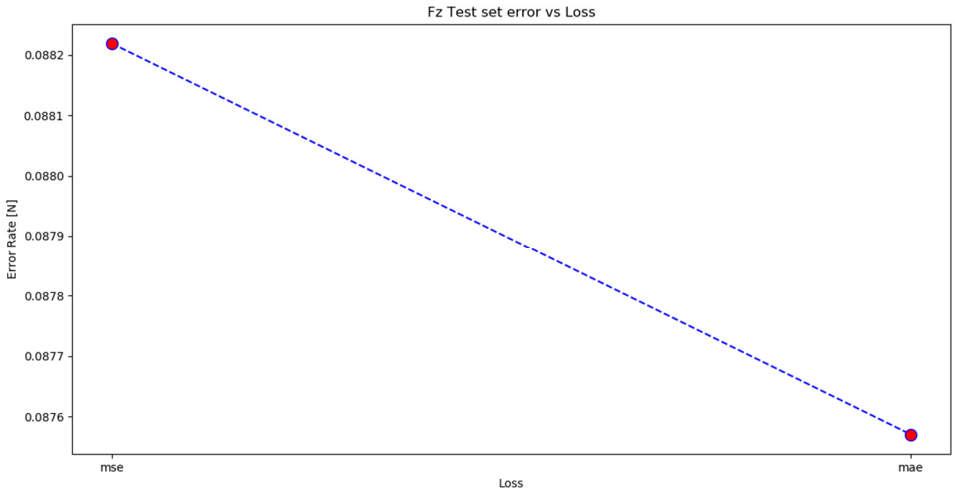


Figure 53 Plot of F_z Losses

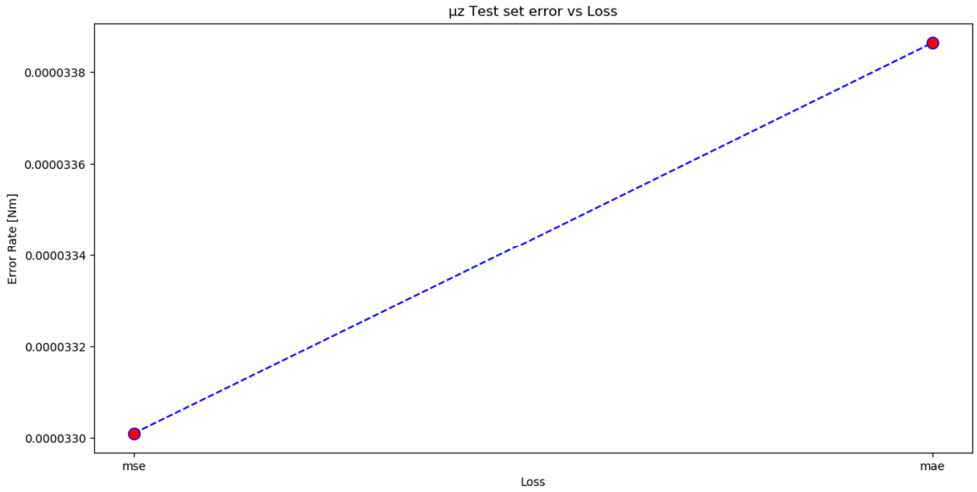


Figure 54 Plot of μ_z Losses

All features in Random Forest are adopted and min_sample_leaf equal to 2 (default value), because performance gets worse when this value increases. Using these parameter and the other parameters previously tuned, a complete simulation on test set has been done. For an easy visualization only 3000 samples are shown in figure 55.

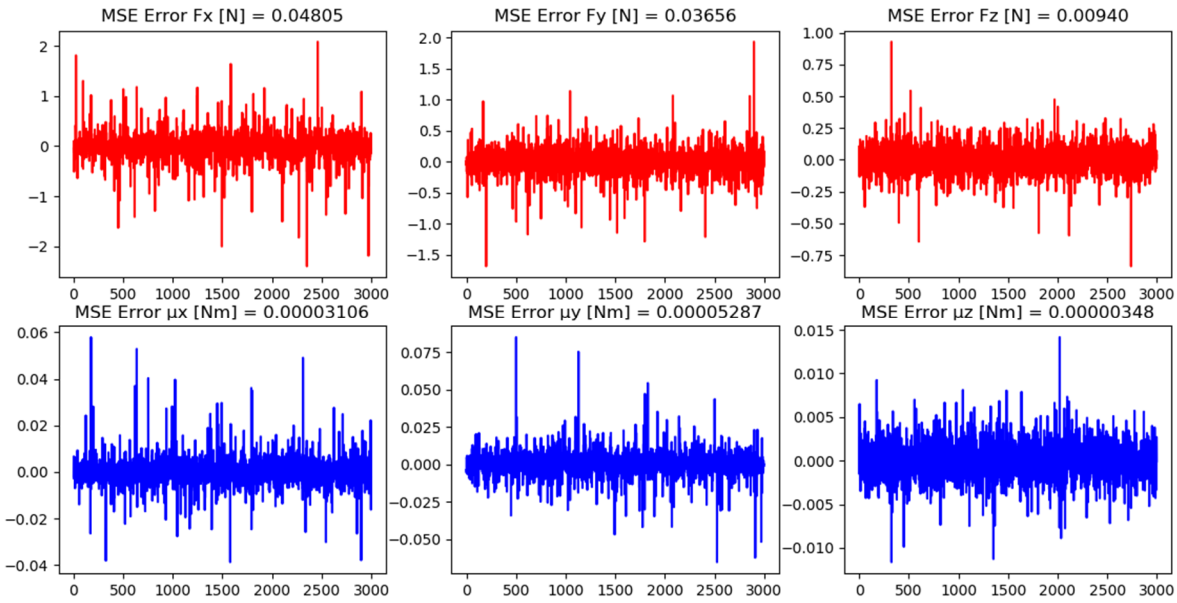


Figure 55 MSE on Test set

4.7 AdaBoost

In the regression context, boosting and bagging are techniques to build a committee of regressors that may be superior to a single regressor. Both bagging and boosting are techniques to obtain smaller prediction errors (in regression) and lower error rates (in classification) using multiple predictors. AdaBoost, short for Adaptive Boosting, is a machine learning meta-algorithm formulated by Yoav Freund and Robert Schapire in which "weak learners" are combined into a weighted sum that represents the final output of the boosted classifier. AdaBoost is adaptive in the sense that subsequent weak learners are tweaked in favor of those instances mispredicted by previous predictors. The individual learners can be weak, but as long as the performance of each one is slightly better than random guessing, the final model can be proven to converge to a strong learner.

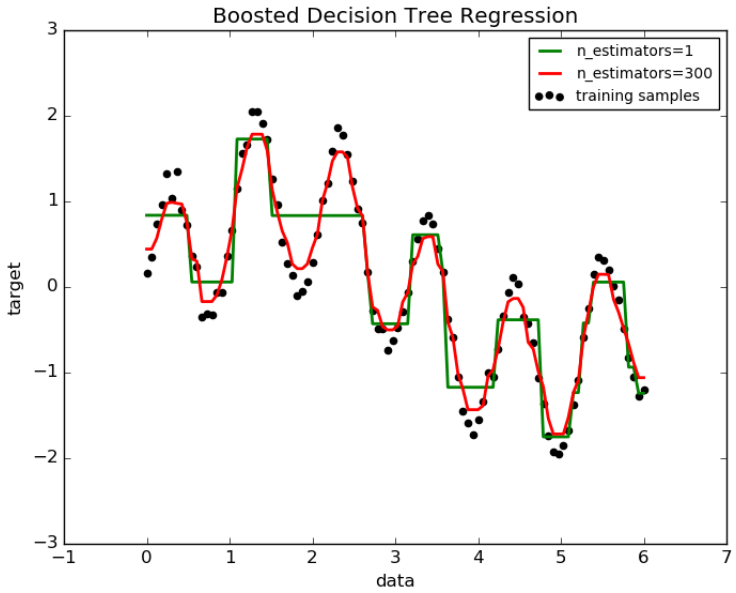


Figure 56 Regression using Adaboost

In AdaBoost, each training instance receives a weight w_i that is used when learning each hypothesis; this weight indicates the relative importance of each instance and is used in computing the error of a hypothesis on the dataset. After each iteration, instances are **reweighted**, according to loss error, receiving larger and proportional weights. Thus, as the process continues, learning focuses on those instances that are most difficult to infer. The key to AdaBoost is the reweighting.

Algorithm 1 `AdaBoost.R2` (Drucker, 1997)

Input the labeled target data set T of size n, the maximum number of iterations N, and a base learning algorithm *Learner*. Unless otherwise specified, set the initial weight vector $\mathbf{w}^1$ such that $w_i^1 = 1/n$ for $1 \leq i \leq n$.

For $t = 1, \ldots, N$:

1. Call *Learner* with the training set T and the distribution $\mathbf{w}^t$, and get a hypothesis $h_t : X \to \mathbb{R}$.
2. Calculate the adjusted error e_i^t for each instance:
   let $D_t = max_{j=1}^{n} |y_j - h_t(x_j)|$
   then $e_i^t = |y_i - h_t(x_i)|/D_t$
3. Calculate the adjusted error of h_t:
   $\epsilon_t = \sum_{i=1}^{n} e_i^t w_i^t$; if $\epsilon_t \geq 0.5$, stop and set $N = t - 1$.
4. Let $\beta_t = \epsilon_t/(1 - \epsilon_t)$.
5. Update the weight vector:
   $w_i^{t+1} = w_i^t \beta_t^{1-e_i^t}/Z_t$ (Z_t is a normalizing constant)

Output the hypothesis:
$h_f(x)$ = the weighted median of $h_t(x)$ for $1 \leq t \leq N$, using $\ln(1/\beta_t)$ as the weight for hypothesis h_t.

Figure 57 Adaboost pseudocode

4.7.1 Adaboost Loss functions

In regression problems, the output given by a hypothesis h_t for an instance x_i is correct or incorrect, but has a real-valued error $e_i = |y_i - h_t(x_i)|$ that may be the method used in AdaBoost.R2 is to express each error in relation to the largest error $D = \max_i |e_i|$. In this way that each adjusted error e_i' is normalized. In particular, one of three possible loss functions is used:

- **Linear** $e_i' = \frac{e_i}{D}$
- **Square** $e_i' = \frac{e_i^2}{D^2}$
- **Exponential** $e_i' = 1 - \exp(-\frac{e_i}{D})$

According to the plot, linear loss puts always the same penalization for different values of the error. This doesn't happen for square loss and exponential loss. Indeed, square loss puts higher and higher emphasis on outliers, differently from exponential loss, which has a smoother shape. For this reason exponential loss is more suitable than square loss in AdaBoost. The degree to which instance xi is reweighted in iteration t thus depends on how large the error of t is on x_i relative to the error on the worst instance.

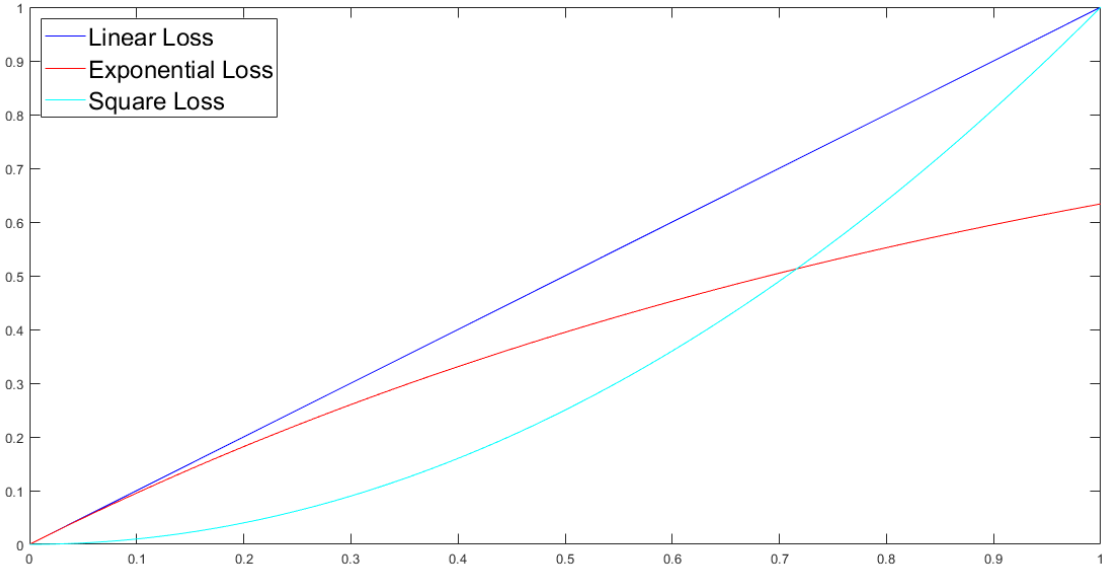


Figure 58 Adaboost Losses

4.7.2 Weighted Median

The choice of the median is arbitrary, probably because the author of the first paper thought that **median-based** approaches because they generalize AdaBoost, not because they are by some metric better.
In statistics, a weighted median of a sample is the 50% weighted percentile. It was first proposed by F. Y. Edgeworth in 1888. Like the median, it is useful as an estimator of central tendency, robust against outliers. It allows for non-uniform statistical weights related to, e.g., varying precision. For n distinct ordered elements $x_1, x_2, \ldots, x_n$ with positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the weighted median is the element x_k satisfying

$$\sum_{i=1}^{k-1} w_i \leq \frac{1}{2} \; and \; \sum_{i=k+1}^{n} w_i \leq \frac{1}{2}$$

This is a generalization of the standard median which is the weighted median of a set of elements with equal weight. If weights of all numbers in the set are equal then median is same as weighted median.

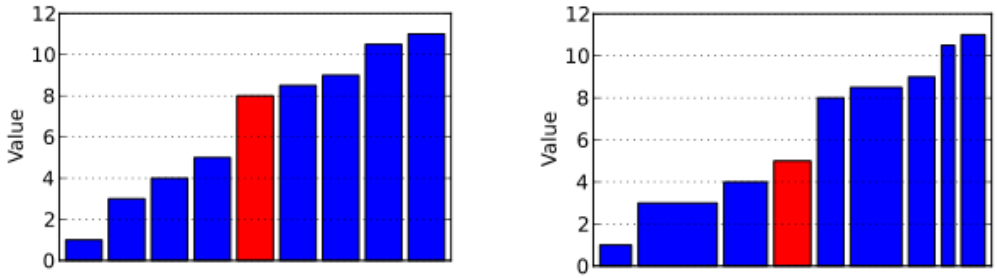


Figure 59 The left chart shows a list of elements with values indicated by height and the median element shown in red. The right chart shows the same elements with weights as indicated by the width of the boxes. The weighted median is shown in red and is different

For simplicity, consider the set of numbers {1;2;3;4;5} with each number having weights $\{0.15; 0.1; 0.2; 0.3; 0.25\}$ respectively. The median is 3 and the weighted median is the element corresponding to the weight 0.3, which is 4. The weights on each side of the pivot add up to 0.45 and 0.25, satisfying the general condition.

4.7.3 Adaboost Pros and Cons

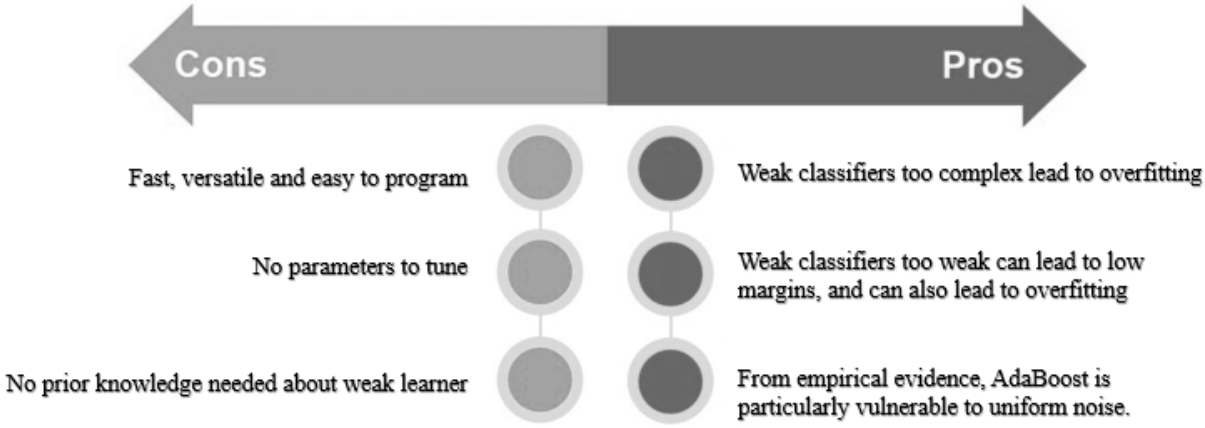


Figure 60 Adaboost advantages and disadvantages

4.7.4 Adaboost Calibration Results

After a dimensionality reduction using PCA,data have been divided in training set and test set in a random way. First step is to understand the optimal number of estimators. For this reason a curve that relates number of estimators and loss is calculated. The curve, obviously, decreases and this mean more stability and accuracy. However, computations begins slower and slower, for this reason a trade-off must be chosen. The value chosen is of 150. Two example are shown below for F_z and μ_z.

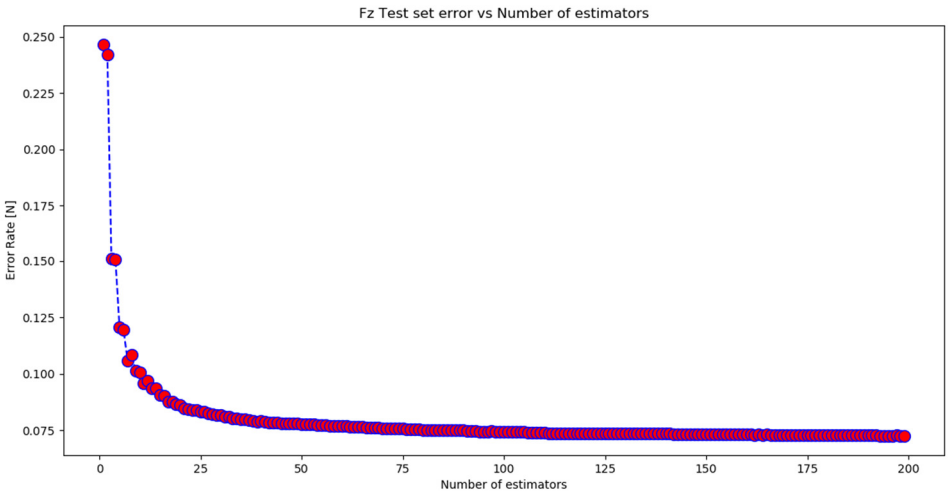


Figure 61 Plot number of estimators F_z

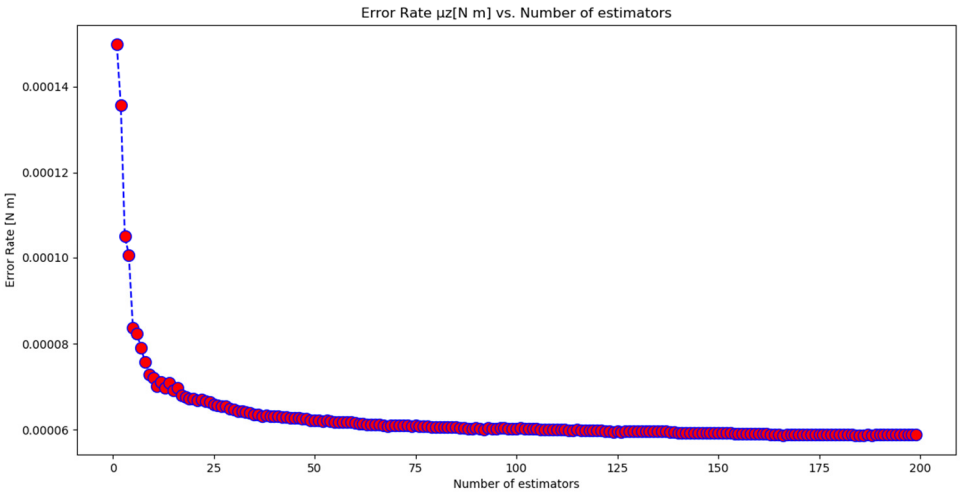


Figure 62 Plot of number of estimators μ_z

After that, learning rate must be tuned. Learning rate regulates the "speed" with which weights are updated. According to the plots learning rate is 1.5. As previous, two example, of the same axes, are shown.

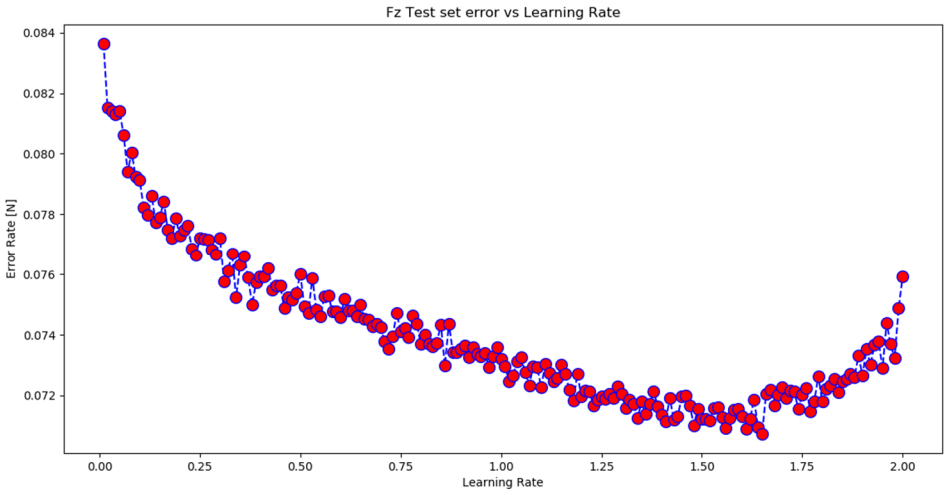


Figure 63 Plot of learning rate F_z

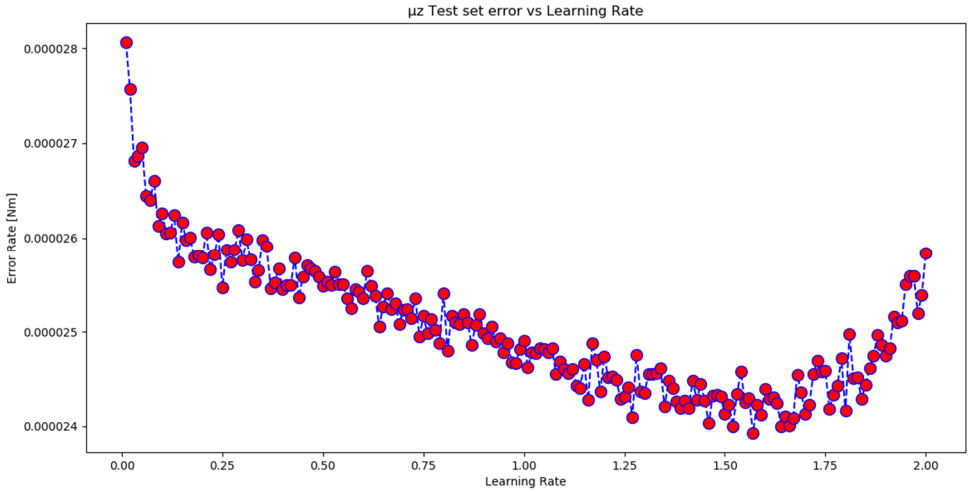


Figure 64 Plot of learning rate μ_z

Finally, last step is the choice of the loss. Usually, linear loss performs better in AdaBoost. Linear loss works better than exponential and square loss, also in this case.

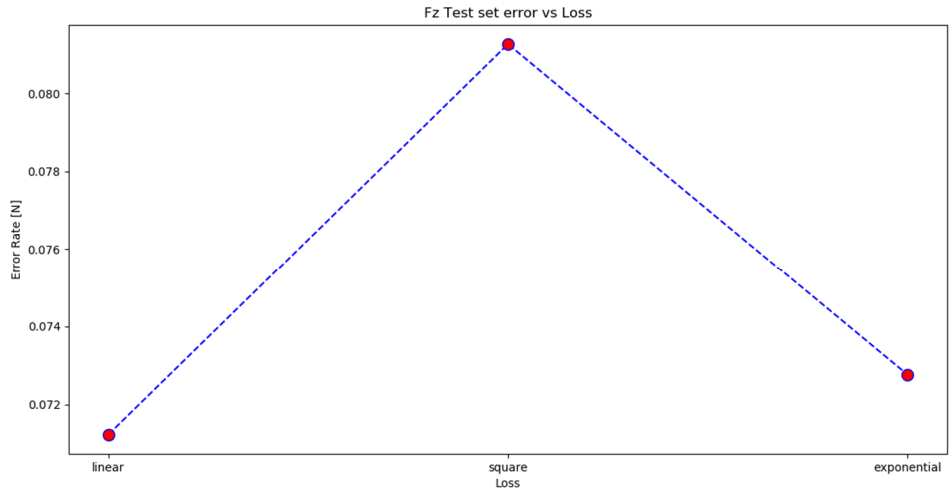


Figure 65 Plot of losses F_z

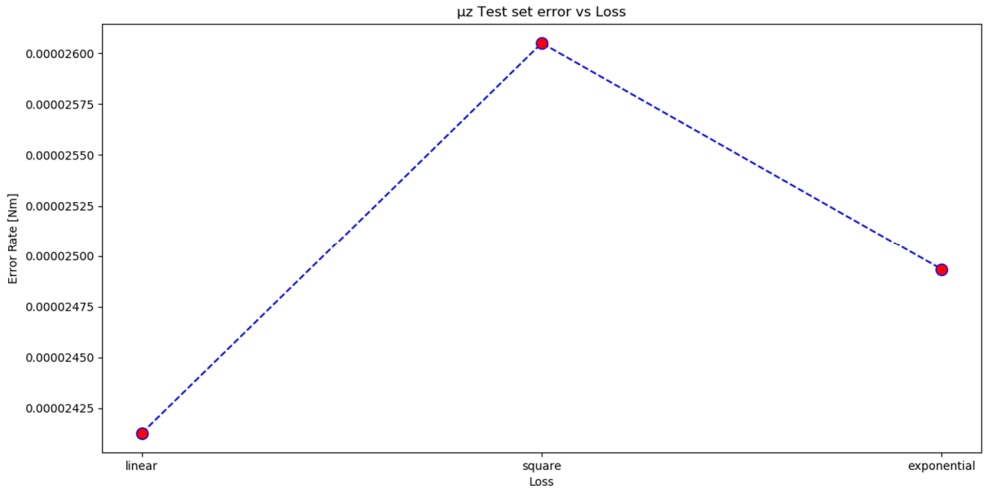


Figure 66 Plot of Losses μ_z

A simulation with all these parameters is shown below. Mean squared error is calculated for every axis using all data in Test set. In figure 67 is plotted error for a batch of 3000 samples, for a better visualization.

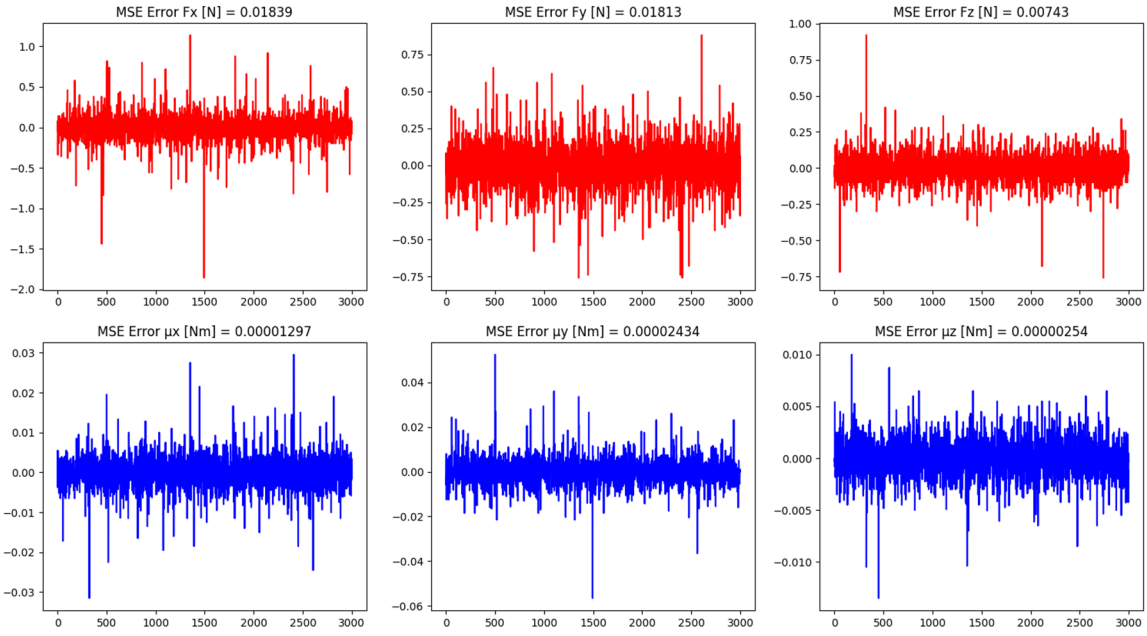


Figure 67 MSE on test set

Chapter 5 Gaussian Process

A **Gaussian process** is a stochastic process (a collection of random variables indexed by time or space), such that every finite collection of those random variables has a multivariate normal distribution, i.e. every finite linear combination of them is normally distributed. The distribution of a Gaussian process is **the joint distribution** of all those (infinitely many) random variables, and as such, it is a **distribution over functions** with a continuous domain, e.g. time or space. A machine-learning algorithm that involves a Gaussian process uses lazy learning and a measure of the **similarity** between points (the *kernel function*) to predict the value for an unseen point from training data. The prediction is not just an estimate for that point, but also has uncertainty information-it is a one-dimensional Gaussian distribution (which is the marginal distribution at that point).When a parameterised kernel is used, optimisation software is typically used to fit a Gaussian process model.

.

5.1 Gaussian Distribution and properties

The Gaussian density is perhaps the most commonly used probability density. It is defined by a mean, μ, and a variance, σ^2. The variance is taken to be the square of the standard deviation, σ.

$$\boldsymbol{p}(\boldsymbol{y}|\mu,\sigma^2) = \frac{1}{\sqrt{2\,\pi\sigma^2}}\exp\left(-\frac{(y-\mu)^2}{2\,\sigma^2}\right) \triangleq \mathcal{N}(y|\mu,\sigma^2)$$

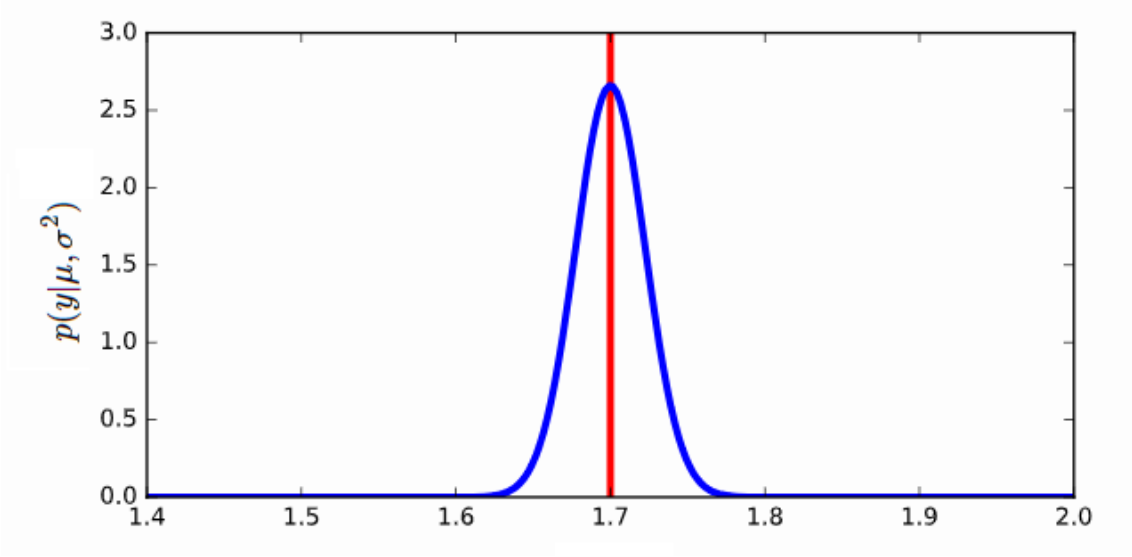


Figure 68 Gaussian Distribution

The Gaussian density has two important properties:

- Sum of Gaussians
- Scaling a Gaussian

5.1.1 Sum of Gaussians

If y_i is sampled from a Gaussian density,

$$y_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$$

It's possible to show that the sum of a set of variables, each drawn independently from this density, is also distributed as Gaussian. The mean of the resulting density is the sum of the means, and the variance is the sum of the variances,

$$\sum_{i=1}^{n} y_i \sim \mathcal{N}\left(\sum_{i=1}^{n} \mu_i, \sum_{i=1}^{n} \sigma_i^2\right)$$

Most random variables, when they are added together, change the family of density they are drawn from. The Gaussian is exceptional in this regard. Indeed, other random variables, if they are independently drawn and summed together tend to a Gaussian density. That is the *central limit theorem* which is a major justification for the use of a Gaussian density.

5.1.2 Scaling a Gaussian

Less unusual is the scaling property of a Gaussian density. If a variable, y, is sampled from a Gaussian

$$y \sim \mathcal{N}(\mu, \sigma^2)$$

and the variable is scaled by a deterministic value, w, then the scaled variable is distributed as:

$$wy \sim \mathcal{N}(w\mu, w^2\sigma^2)$$

Unlike the summing properties, where adding two or more random variables independently sampled from a family of densities typically brings the summed variable outside that family, scaling many densities leaves the distribution of that variable in the same family of densities. Indeed, many densities include a scale parameter (e.g. the Gamma density) which is purely for this purpose. In the Gaussian the standard deviation,σ, is the scale parameter. To see why this makes sense, consider:

$$z \sim \mathcal{N}(0,1)$$

then if scaling by σ, $y = \sigma z$, so y will have a distribution:

$$y = \sigma z \sim \mathcal{N}(0, \sigma^2)$$

5.1.3 Prior Distribution

The tradition in Bayesian inference is to place a probability density over the parameters of interest in your model. This choice is made regardless of whether you generally believe those parameters to be stochastic or deterministic in origin. In other words, to a Bayesian, the modelling treatment does not differentiate between epistemic and aleatoric uncertainty. For linear regression, an example prior could be the following Gaussian prior on the intercept parameter:

$$c \sim \mathcal{N}(0, \alpha_1)$$

where α_1 is the variance of the prior distribution, its mean being zero.

5.1.4 Posterior Distribution

The prior distribution is combined with the likelihood of the data given the parameters $p(y|c)$ to give the posterior via Bayes' rule,

$$p(c|y) = \frac{p(y|c)\, p(c)}{p(y)}$$

where $p(y)$ is the marginal probability of the data, obtained through integration over the joint density, $p(y, c) = p(y|c)p(c)$

$$posterior = \frac{likelihood\ x\ prior}{marginal\ likelihood}$$

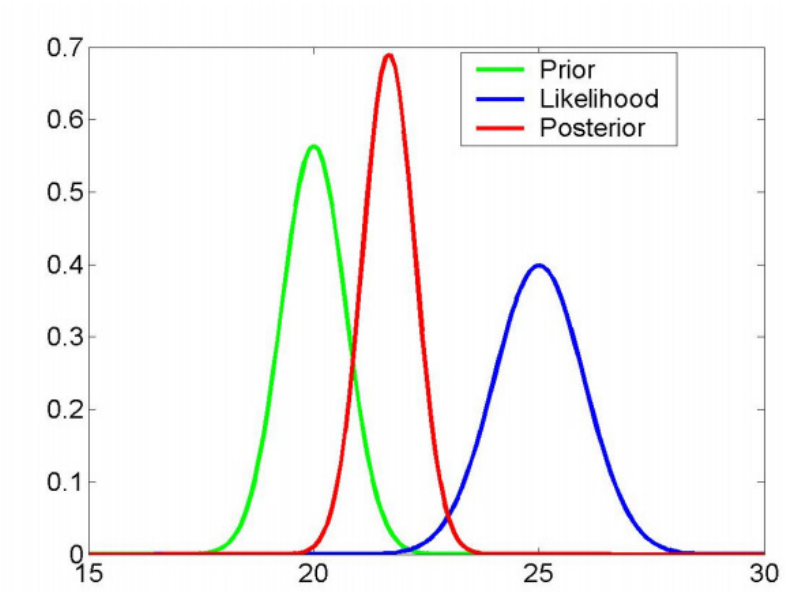


Figure 69 Prior, Likelihood and Posterior Gaussian Distributions

5.1.5 Multivariate Gaussian

The multivariate normal distribution, multivariate Gaussian distribution, or joint normal distribution is a generalization of the one-dimensional (univariate) normal distribution to higher dimensions.

$$p(\boldsymbol{y}) = \frac{1}{\det 2\pi\boldsymbol{\Sigma}^{\frac{1}{2}}}\left(\exp -\frac{1}{2}(\boldsymbol{y}-\boldsymbol{\mu})^T\Sigma^{-1}(\boldsymbol{y}-\boldsymbol{\mu})\right)$$

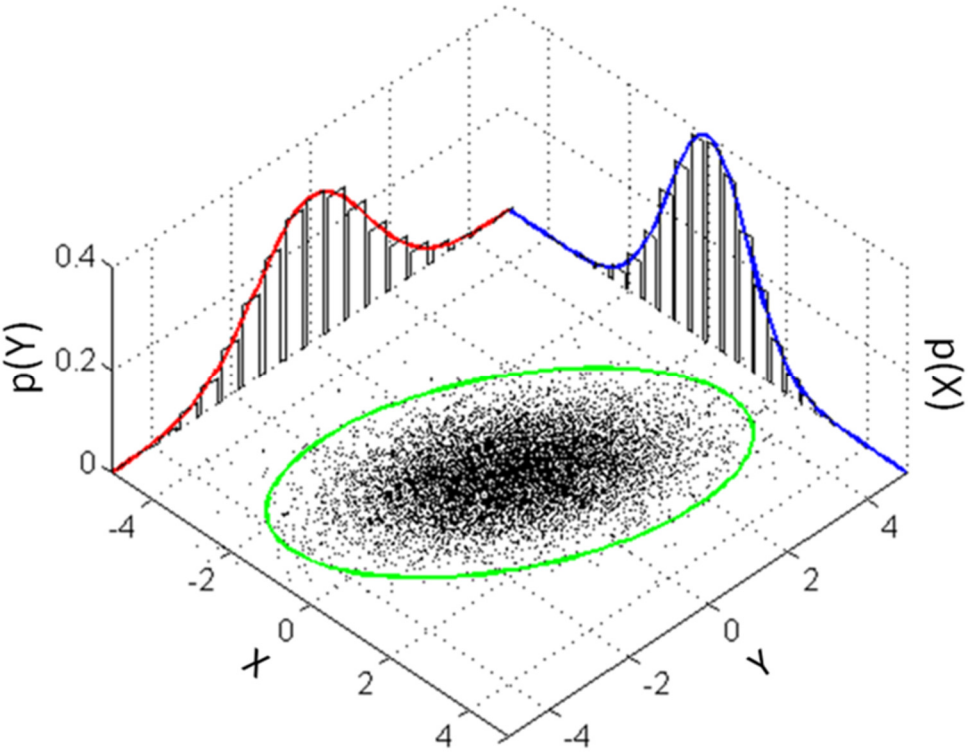


Figure 70 Bidimensional Gaussian distribution

Covariance matrix $\boldsymbol{\Sigma}$, is a matrix whose element in the i, j position is the covariance between the $i-th$ and $j-th$ elements of a random vector. A random vector is a random variable with multiple dimensions. Each element of the vector is a scalar random variable. The equation above reduces to that of the univariate normal distribution if Σ is a 1 x 1 matrix (i.e. a single real number).

5.2 Gaussian Process from different views

There are several ways to interpret Gaussian process (GP) regression models. One can think of a Gaussian process as defining a **distribution over functions**, and inference taking place directly in the space of functions, the function-space view. Although this view is appealing it may initially be difficult to grasp, so we start our exposition in section with the equivalent weight-space view which may be more familiar and accessible to many, and continue in section with the function-space view.

5.2.1 Weight Space view

The simple linear regression model where the output is a linear combination of the inputs has been studied and used extensively. Its main virtues are simplicity of implementation and interpretability. Its main drawback is that it only allows a limited flexibility. If the relationship between input and output can not reasonably be approximated by a linear function, the model will give poor predictions. In the next few lines will be discussed the Bayesian analysis of the standard linear regression model with Gaussian noise

$$f(x) = \boldsymbol{x}^T w\,,\ y = f(x) + \varepsilon$$

where $\boldsymbol{x}$ is the input vector, w is a vector of weights (parameters) of the linear bias, offset model, f is the function value and y is the observed target value. Often a bias weight or offset is included, but as this can be implemented by augmenting the input vector $\boldsymbol{x}$ with an additional element whose value is always one. This under the assumption that the observed values y differ from the function values $f(x)$ by additive noise distributed Gaussian distribution with zero mean and variance σ_n^2:

$$\epsilon \sim \mathcal{N}(0, \sigma_n^2)$$

Likelihood. This noise assumption together with the model directly gives rise to the **likelihood**, the probability density of the observations given the parameters, which is factored over cases in the training set (because of the i.i.d. assumption) to give:

$$\begin{aligned} p(\boldsymbol{y}|X,\boldsymbol{w}) &= \prod_{i=1}^{n} p(y_i|\boldsymbol{x_i},\boldsymbol{w}) \\ &= \prod_{i=1}^{n} \frac{1}{\sqrt{2\pi}\sigma_n} \exp\left(-\frac{(y_i - \boldsymbol{x}_i^t \boldsymbol{w})}{2\sigma_n^2}\right) \\ &= \frac{1}{(2\pi\sigma_n^2)^{n/2}} \exp\left(-\frac{1}{2\sigma_n^2}|\boldsymbol{y} - \boldsymbol{X}^T\boldsymbol{w}|^2\right) = \mathcal{N}(\boldsymbol{X}^T\boldsymbol{w}, \sigma_n^2\, \boldsymbol{I}) \end{aligned}$$

In the Bayesian formalism we need to specify a **prior** over the parameters, expressing our beliefs about the prior parameters before we look at the observations. Considering a zero mean Gaussian prior with covariance matrix Σ_p on the weights:

$$w \sim \mathcal{N}(0, \Sigma_{\mathrm{p}})$$

Inference in the Bayesian linear model is based on the posterior distribution posterior over the weights, computed by Bayes' rule:

$$posterior = \frac{likelihood\ x\ prior}{marginal\ likelihood}, \qquad p(\boldsymbol{w}|\boldsymbol{y},X) = \frac{p(\boldsymbol{y}|X,\boldsymbol{w})p(\boldsymbol{w})}{p(\boldsymbol{y}|X)}$$

Where the normalizing constant, also known as the **marginal likelihood**, is independent of the weights and given by:

$$p(\boldsymbol{y}|X,\boldsymbol{w}) = \int p(\boldsymbol{y}|X,\boldsymbol{w})\, p(\boldsymbol{w})\, d\boldsymbol{w}$$

The posterior combines the likelihood and the prior, and captures everything known about the parameters. Writing only the terms from the likelihood and prior which depend on the weights, and "completing the square" equation becomes:

$$p(\boldsymbol{w}|\boldsymbol{y},X) \propto \exp\left(-\frac{1}{2\sigma_n^2}(\boldsymbol{y}-X^T\boldsymbol{w})^T(\boldsymbol{y}-X^T\boldsymbol{w})\right)\exp\left(-\frac{1}{2}\boldsymbol{w}^T\Sigma_{\mathrm{p}}^{-1}\boldsymbol{w}\right)$$

$$\propto \exp\left(-\frac{1}{2}(\boldsymbol{w}-\overline{\boldsymbol{w}})^T\left(\frac{1}{\sigma_n^2}XX^T+\Sigma_{\mathrm{p}}^{-1}\right)(\boldsymbol{w}-\overline{\boldsymbol{w}})\right)$$

where $\overline{\boldsymbol{w}} = \sigma_n^2\,(\sigma_n^{-2}\,XX^T + \Sigma_{\mathrm{p}}^{-1})^{-1}X\boldsymbol{y}$. This is the form of the posterior distribution as Gaussian with mean $\overline{\boldsymbol{w}}$ and covariance matrix A^{-1}

$$p(\boldsymbol{w}|\boldsymbol{y},X) \sim \mathcal{N}\left(\overline{\boldsymbol{w}} = \frac{1}{\sigma_n^2}A^{-1}X\boldsymbol{y}, A^{-1}\right)$$

Where $A = \sigma_n^{-2}\,XX^T + \Sigma_p^{-1}$.To make predictions for a test case it's made an average over all possible parameter values, weighted by their posterior probability. This is in contrast to non-Bayesian schemes, where a single parameter is typically chosen by some criterion. Thus the **predictive distribution** for $f_* \triangleq f(x_*)$ at x_* is given by averaging the output of all possible linear models w.r.t. the Gaussian posterior:

$$p(f_*|\boldsymbol{x}_*,X,\boldsymbol{y}) = \int p(f_*|x_*,\boldsymbol{w})\,p(\boldsymbol{w}|X,\boldsymbol{y})d\boldsymbol{w} = \mathcal{N}\left(\frac{1}{\sigma_n^2}x_*^TA^{-1}X\boldsymbol{y}, x_*^TA^{-1}x_*\right)$$

The predictive distribution is again Gaussian, with a mean given by the posterior mean of the weights multiplied by the test input, as one would expect from symmetry considerations. The predictive variance is a quadratic form of the test input with the posterior covariance matrix, showing that the predictive uncertainties grow with the magnitude of the test input, as one would expect for a linear model.

5.3 Function Space view

An alternative and equivalent way of reaching identical results to the previous section is possible by considering inference directly in function space. We use a Gaussian process (GP) to describe a distribution over functions. Formally:

Definition: *A* Gaussian process *is a collection of random variables, any finite number of which have a joint Gaussian distribution.*

A Gaussian process is completely specified by its mean function and covariance and variance function. We define **mean function** $m(x)$ and the **covariance function** $k(x, x')$ of a real process $f(x)$ as

$$m(x) = E[f(x)]$$

$$k(x, x') = E\big[\big(f(x) - m(x)\big)\big(f(x') - m(x')\big)\big]$$

and will write the Gaussian process as

$$f(x) \sim GP\,(m(x), k(x, x'))$$

Usually, for notational simplicity mean function will be considered equal to zero, although this need not be done. A Gaussian process is defined as a collection of random variables. Thus, the definition automatically implies the marginalization property. This property simply means marginalization that if the GP e.g. specifies $(y_1, y_2) \sim \mathcal{N}(\mu, \Sigma)$, then it must also specify property $y_1 \sim \mathcal{N}(\mu_{11}, \Sigma_{11})$ where Σ_{11} is the relevant submatrix of Σ. In other words, examination of a larger set of variables does not change the distribution of the smaller set. A simple example of a Gaussian process can be obtained from our Bayesian is a Gaussian process linear regression model $f(x) = \boldsymbol{\varphi}(x)^T \boldsymbol{w}$ with prior $w \sim \mathcal{N}(0, \Sigma_p)$. Mean and covariance are defined as:

$$E[f(x)] = \boldsymbol{\varphi}(x)^T E[\boldsymbol{w}] = 0$$

$$E[f(x)f(x')] = \boldsymbol{\varphi}(x)^T E[\boldsymbol{w}\boldsymbol{w}^T]\boldsymbol{\varphi}(x') = \boldsymbol{\varphi}(x)^T \Sigma \boldsymbol{\varphi}(x')$$

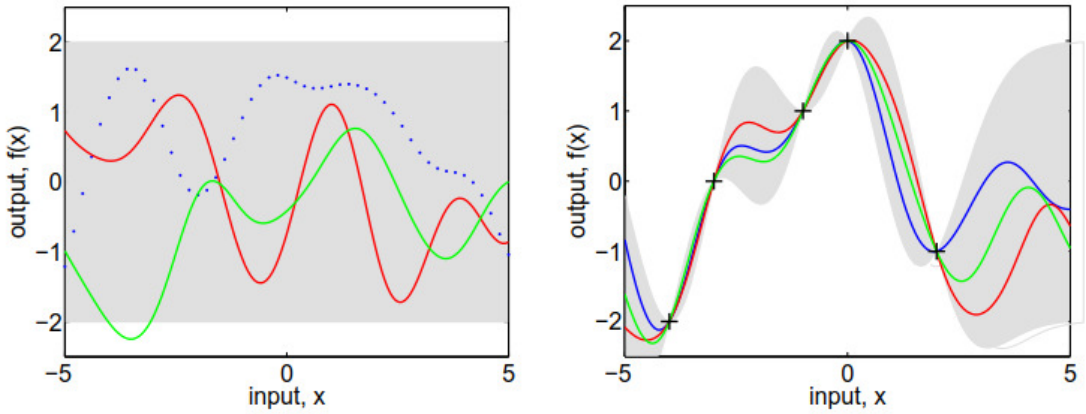
The covariance function specifies the covariance between pairs of random variables, e.g. in the case of *squared exponential* (SE) covariance function formula will be:

$$cov\left(f(\boldsymbol{x}_p), f(\boldsymbol{x}_q)\right) = k(\boldsymbol{x}_p, \boldsymbol{x}_q) = \exp\left(-\frac{1}{2}|x_p - x_q|^2\right)$$

Note, that the covariance between the outputs is written as a function of the inputs. The specification of the covariance function implies a distribution over functions. To see this, one can draw samples from the distribution of functions evaluated at any number of points. This can be done choosing a number of input points X_* and write out the corresponding covariance matrix. A random Gaussian vector is generated with this covariance matrix:

$$f_* \sim \mathcal{N}(0, k(X_*, X_*))$$

and plot the generated values as a function of the inputs. Figure 71(a) shows three such samples. Notice that "informally" the functions look smooth. In fact, the squared exponential covariance function is infinitely differentiable, leading to the process being infinitely mean-square differentiable. The functions seem to have a characteristic length-scale which informally can be thought of as roughly the distance to move in input space before the function value can change significantly. For SE Kernel the characteristic length-scale is around one unit. By replacing $|x_p - x_q|$ with $|x_p - x_q|/\ell$ $(\ell > 0)$ we could change the characteristic length-scale of the process. Also, the overall variance of the magnitude random function can be controlled by a positive pre-factor before the exp.



(a) (b)

Figure 71 (a) Show three functions drawn at random from a GP prior; the dots indicate values of y actually generated. Panel (b) shows three random functions drawn from the posterior. In both plots the shaded area represents the pointwise mean and plus two times the standard deviation for each input value.

5.3.1 Prediction with Noise-free Observations

Drawing random functions from the prior is not of primarily importance. Usually the one wants to incorporate the knowledge that the training data provides about the function. Consider the simple special case where the observations are noise-free, that is we know $\{(\boldsymbol{x}_i, \boldsymbol{f}_i) | \, i = 1, \ldots, n\}$. The joint distribution of the training outputs,$\boldsymbol{f}$, and the test outputs $\boldsymbol{f}_*$ according to the prior is:

$$\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{f}_* \end{bmatrix} \sim \mathcal{N}\left(\boldsymbol{0}, \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right)$$

If there are n training points and n_* test points then $K(X,X_*)$ denotes the $n \times n_*$ matrix of the covariances evaluated at all pairs of training and test points. Similarly for the other entries $K(X,X)$, $K(X_*,X_*)$ and $K(X_*,X)$. To get the posterior distribution over functions the joint prior distribution is restricted in order to contain only those functions which agree with the observed data points.

$$\boldsymbol{f}_* | X_*, X, \boldsymbol{f} \sim \mathcal{N}(K(X_*,X)K(X,X)^{-1}\boldsymbol{f}, K(X_*,X_*) - K(X_*,X)K(X,X)^{-1}K(X,X_*))$$

Function values $\boldsymbol{f}_*$ (corresponding to test inputs X_*) can be sampled from the joint posterior distribution by evaluating the mean and covariance matrix.

5.3.2 Prediction using Noisy Observations

In realistic modelling situations one doesn't have access to function values themselves, but only noisy versions of $y = f(x) + \varepsilon$. Assuming additive independent identically distributed Gaussian noise ε with variance σ_n^2, the prior on the noisy observations becomes:

$$cov(y_p, y_q) = k(x_p, x_q) + \sigma_n^2 \delta_{pq} \text{ or } cov(\boldsymbol{y}) = K(X,X) + \sigma_n^2 I$$

where δ_{pq} is a Kronecker delta which is one iff $p = q$ and zero otherwise. It follows from the independence assumption about the noise, that a diagonal matrix is added, in comparison to the noise-free case. Introducing the noise term, the joint distribution of the observed target values and the function values at the test locations can be written as:

$$\begin{bmatrix} \boldsymbol{f} \\ \boldsymbol{f}_* \end{bmatrix} \sim \mathcal{N}\left(\boldsymbol{0}, \begin{bmatrix} K(X,X) + \sigma_n^2 I & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix}\right)$$

Deriving the conditional distribution, one arrives at the key predictive equations for Gaussian process regression:

$\boldsymbol{f}_* | X_*, X, \boldsymbol{f} \sim \mathcal{N}\left(\overline{\boldsymbol{f}_*}, cov(\boldsymbol{f}_*)\right)$ where

$\overline{\boldsymbol{f}_*} \triangleq E[\boldsymbol{f}_* | X, \boldsymbol{y}, \boldsymbol{X}_*] = K(X_*,X)[K(X,X) + \sigma_n^2 I]^{-1}\boldsymbol{y}$,

$cov(\boldsymbol{f}_*) = K(X_*,X_*) - K(X_*,X)[K(X,X) + \sigma_n^2 I]^{-1}K(X,X_*)$

Introducing a compact form of the notation setting $K = K(X,X)$ and $K_* = K(X,X_*)$. In the case that there is only one test point x_* we write $k(x_*) = k_*$ to denote the vector of covariances between the test point and the n training points. Using this compact notation and for a single test point x_*, it reduces to

$$\overline{f}_* = \boldsymbol{k}_*^T (K + \sigma_n^2 I)^{-1} \boldsymbol{y}$$

$$\mathbb{V}[f_*] = k(\boldsymbol{x}_*, \boldsymbol{x}_*) - \boldsymbol{k}_*^T (K + \sigma_n^2 I)^{-1} \boldsymbol{k}_*$$

The mean prediction is a linear combination of observations y; this is sometimes referred to as a **linear predictor**. Another way to linear predictor look at this equation is to see it as a **linear combination** of n kernel functions, each one centered on a training point, by writing:

$$\overline{f}(x_*) = \sum_{i=1}^{n} \alpha_i \, k(\boldsymbol{x}_i, \boldsymbol{x}_*)$$

where $\boldsymbol{\alpha} = (K + \sigma_n^2)^{-1}\boldsymbol{y}$. The fact that the mean prediction for $f(x_*)$ can be written despite the fact that the GP can be represented in terms of a (possibly infinite) number of basis functions is one manifestation of the *representer theorem*. It will be useful to introduce the **marginal likelyhood** (or evidence) $p(y|X)$ at this point. The marginal likelihood is the **integral of the likelihood** times the prior:

$$p(\boldsymbol{y}|X) = \int p(\boldsymbol{y}|\boldsymbol{f}, X)p(\boldsymbol{f}|X)d\boldsymbol{f}$$

The term marginal likelihood refers to the **marginalization** over the function values f. Under the Gaussian process model the prior is Gaussian $\boldsymbol{f}|X \sim \mathcal{N}(\boldsymbol{0}, K)$, or:

$$\log p(\boldsymbol{f}|X) = -\frac{1}{2}\boldsymbol{f}^T K^{-1}\boldsymbol{f} - \frac{1}{2}\log[K] - \frac{n}{2}\log 2\pi$$

and the likelihood is a factorized Gaussian $y|f \sim \mathcal{N}(\boldsymbol{f}, \sigma_n^2 I)$. Performing integration the log marginal likelihood:

$$\log p(\boldsymbol{y}|X) = -\frac{1}{2}\boldsymbol{y}^T(K + \sigma_n^2)^{-1}\boldsymbol{y} - \frac{1}{2}\log|K + \sigma_n^2 I| - \frac{n}{2}\log 2\pi$$

This result can also be obtained directly by observing that $y \sim \mathcal{N}(0, K + \sigma_n^2 I)$

5.4 Varying the Hyperparameters

Often, the covariance functions adopted have some free parameters. For example, the squared-exponential covariance function in one dimension has the following form:

$$k_y(x_p, x_q) = \sigma_f^2 \exp\left(-\frac{1}{2\ell^2}(x_p - x_q)^2\right) + \sigma_n^2 \delta_{pq}$$

The covariance is denoted k_y as it is for the noisy targets y rather than for the underlying function$f$. The length-scale ℓ, the signal variance σ_f^2 and the noise variance σ_n^2 can be changed. Consider the data shown by "+" signs in Figure 72. This was generated from a GP with the SE kernel with $(\ell, \sigma_f, \sigma_n) = (1, 1, 0.1)$. The figure also shows the 2 standard-deviation error bars for the predictions obtained using these values of the hyperparameters. Error bars get larger for input values that are distant from any training points. Indeed if the $x - axis$ were extended one would see the error bars reflect the prior standard deviation of the process σ_f away from the data. If the length-scale shorter so that $\ell = 0.3$ and kept the other parameters the same, plots will be like those in Figure 72(a) except that the $x - axis$ should be rescaled by a factor of 0.3. Equivalently, if the same $x - axis$ was kept as in Figure 72(a) then a sample function would look much more wiggly. If one makes predictions with a process with $\ell = 0.3$ on the data generated from the $\ell = 1$ process, obtains the result in Figure 72(b). The remaining two parameters were set by optimizing the marginal likelihood. In this case the noise parameter is reduced to $\sigma_n = 0.00005$ as the greater flexibility of the "signal" means that the noise level can be reduced. This can be observed at the two datapoints near $x = 2.5$ in the plots. In Figure 72(a) ($\ell = 1$) these are essentially explained as a similar function value with differing noise. However, in Figure 72(b) ($\ell = 0.3$) the noise level is very low, so these two points have to be explained by a sharp variation in the value of the underlying function f. Notice

also that the short length-scale means that the error bars in Figure 72(b) grow rapidly away from the data. In contrast, if the length-scale is longer, for example to $\ell = 3$, as shown in Figure 72(c). Again the remaining two parameters were set by optimizing the marginal likelihood. In this case the noise level has been increased to $\sigma_n = 0.89$ and the data are now explained by a slowly varying function with a lot of noise.

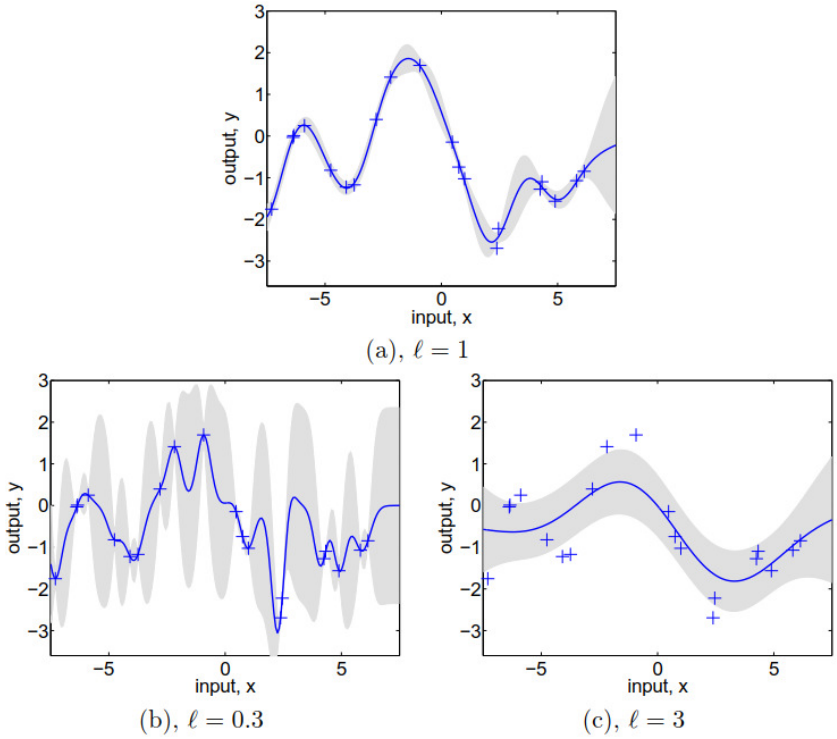


Figure 72 (a) Data is generated from a GP with hyperparameters $(\ell, \sigma_f, \sigma_n) = (1, 1, 0.1)$*, as shown by the + symbols. Using Gaussian process prediction with these hyperparameters we obtain a* 95% *confidence region for the underlying function* f *(shown in grey). Panels (b) and (c) again show the* 95% *confidence region, but this time for hyperparameter values* $(0.3, 1.08, 0.00005)$ *and* $(3.0, 1.16, 0.89)$ *respectively.*

5.5 Covariance Functions

Covariance function is the crucial ingredient in a Gaussian process predictor, as it encodes the assumptions about the predicted. From a slightly different viewpoint it is clear that in supervised learning the notion of similarity between data points is crucial. It is a basic similarity assumption that points with inputs x which are close are likely to have similar target values y, and thus training

points that are near to a test point should be informative about the prediction at that point. The covariance function that defines nearness or similarity. An arbitrary function of input pairs x and x' will not, in general, be a valid covariance function. The purpose of the next lines is to give examples of some functions commonly-used covariance functions and to examine their properties. Most important kernels covariance functions are:

- Squared Exponential
- Matérn
- Exponential
- The γ-exponential
- Rational Quadratic

Squared Exponential Covariance Function

The squared exponential (SE) covariance function has the form

$$k_{se}(r) = \exp\left(-\frac{r^2}{2\ell^2}\right)$$

This covariance function is infinitely differentiable, which means that the GP with this covariance function has mean square derivatives of all orders, and is thus very smooth. Stein argues that such strong smoothness assumptions are unrealistic for modelling many physical processes, and recommends the Matérn class. However, the squared exponential is probably the most widely-used kernel within the kernel machines field.

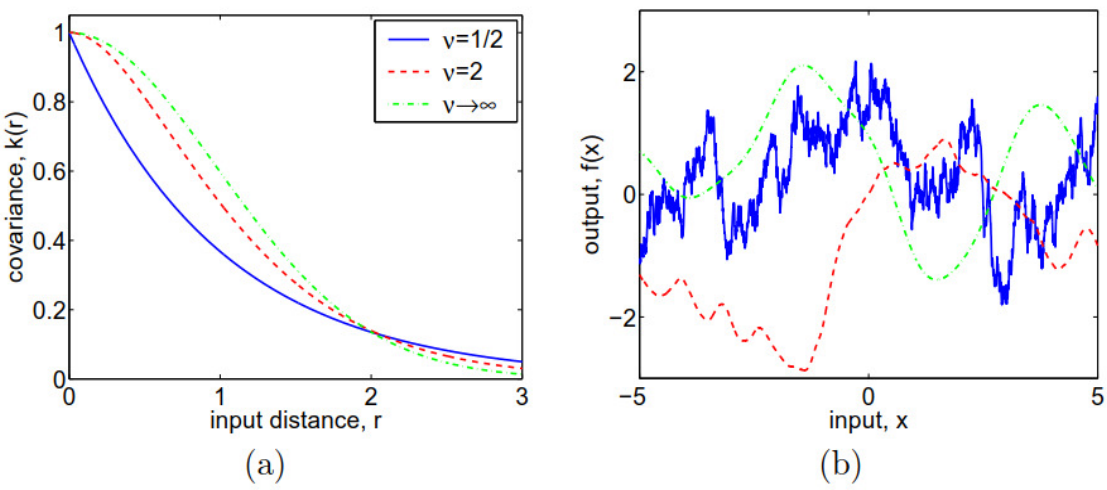


Figure 73 (a): Covariance functions, and (b) random functions drawn from a Gaussian processes with Matérn covariance functions

The Matérn Class of Covariance Functions

Matérn class of covariance functions is given by

$$k_{Matérn}(r) = \frac{2^{1-v}}{\Gamma(v)}\left(\frac{\sqrt{2vr}}{\ell}\right)^{v} K_v\left(\frac{\sqrt{2vr}}{\ell}\right)$$

with positive parameters ν and ℓ, where K_v is a modified Bessel function. Note that the scaling is chosen so that for $\nu \to \infty$ we obtain the SE covariance function $\exp\left(\frac{r^2}{2\ell^2}\right)$

Exponential Covariance Function

The special case obtained by setting $\nu = \frac{1}{2}$ in the Matérn class gives the exponential covariance function $k(r) = \exp\left(-\frac{r}{\ell}\right)$.

The γ-exponential Covariance Function

The γ-exponential family of covariance functions, which includes both the exponential and squared exponential, is given by

$$k(r) = \exp\left(\left(\frac{r}{\ell}\right)^{\gamma}\right) \qquad for\ 0 < \gamma \leq 2.$$

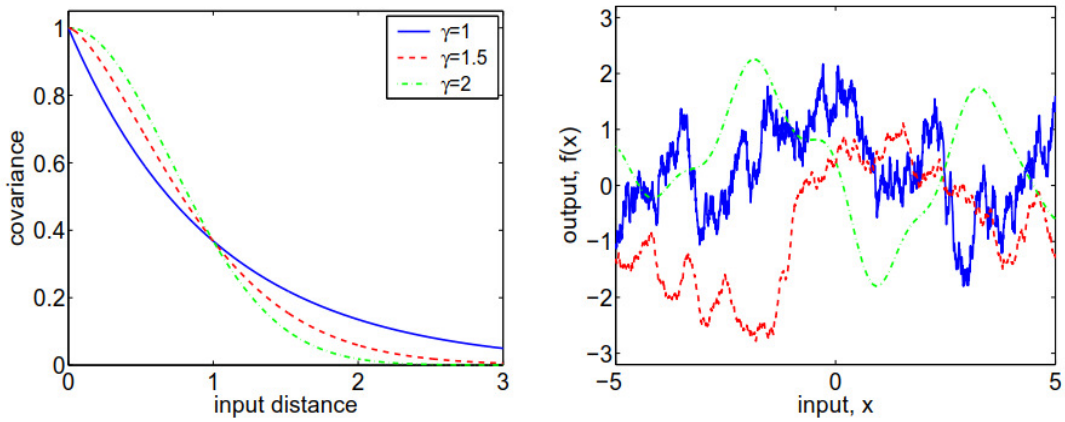


Figure 74 (a) covariance functions, and (b) random functions drawn with a Gaussian process with the γ-exponential covariance function

Rational Quadratic Covariance Function

The rational quadratic (RQ) covariance function

$$k_{RQ}(r) = \left(1 + \frac{r^2}{2\alpha\ell^2}\right)^{-\alpha}$$

with α, $\ell > 0$ can be seen as a scale mixture (an infinite sum) of squared scale mixture exponential (SE) covariance functions with different characteristic length-scales.

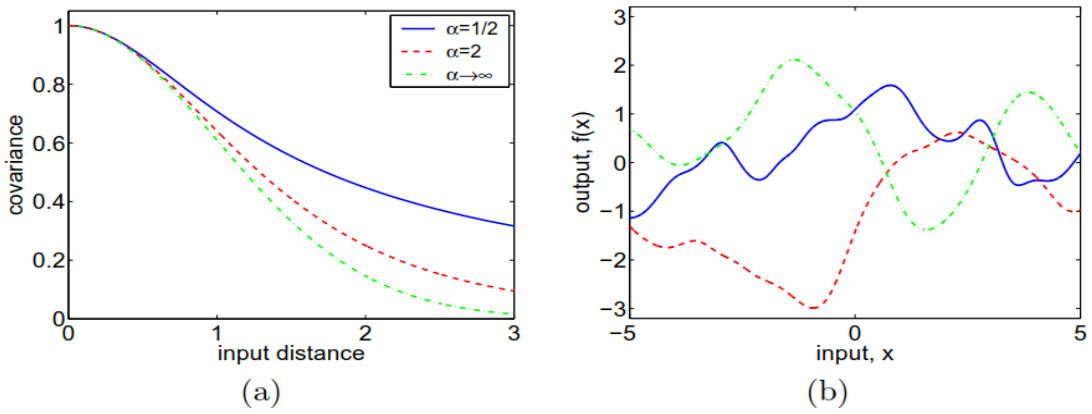


Figure 75 (a) covariance functions, and (b) random functions drawn from Gaussian processes with rational quadratic covariance functions

5.6 Model Selection

Some properties of the model may be easy to specify, while one typically have only vague information available about other aspects. The term **model selection** is adopted to cover both discrete choices and the setting of continuous (hyper-) parameters of the covariance functions. In fact, model selection can help both to refine the predictions of the model, and give interpretation a valuable interpretation to the user about the properties of the data. A multitude of possible families of covariance functions exists, including squared exponential, polynomial, neural network... Each of these families typically have a number of free hyperparameters whose values also need to be determined. Model selection is essentially open ended. Even for the squared exponential covariance function, there is a huge variety of possible distance measures. However, this should not be a cause for despair, rather seen as a possibility to learn. It requires, however, a systematic and practical approach to model selection. For this reason a **Bayesian Model** selection approach is needed. It is common to use a hierarchical specification of models. At the lowest level are the parameters,w. For example, the parameters could be the parameters in a linear model, or the weights in a neural network model. At the second level are hyperparameters θ, which control the distribution of the parameters at the bottom level.

At the bottom level, the posterior over the parameters is given by Bayes' rule:

$$p(\boldsymbol{w}|\boldsymbol{y}, X, \boldsymbol{\theta}, H_i) = \frac{p(\boldsymbol{y}|X, \boldsymbol{w}, H_i)p(\boldsymbol{w}|\boldsymbol{\theta}, H_i)}{p(\boldsymbol{y}|X, \boldsymbol{\theta}, H_i)}$$

where $p(y|X, w, H_i)$ is the likelihood and $p(w|\theta, H_i)$ is the parameter prior. The prior encodes as a probability distribution our knowledge about the parameters prior to seeing the data. If one has only vague prior information about the parameters, then the prior distribution is chosen to be broad to reflect this. The posterior combines the information from the prior and the data (through the likelihood). The normalizing constant in the denominator $p(y|X, \theta, H_i)$ is

independent of the parameters, and called the marginal likelihood (or evidence), and is given by:

$$p(\boldsymbol{y}|X, \boldsymbol{\theta}, H_i) = \int p(\boldsymbol{y}|X, \boldsymbol{w}, H_i) p(\boldsymbol{w}|\boldsymbol{\theta}, H_i) d\boldsymbol{w}$$

At the next level, the posterior over the hyperparameters is given, where the marginal likelihood from the first level plays the role of the likelihood:

$$p(\boldsymbol{\theta}|\boldsymbol{y}, X, H_i) = \frac{p(\boldsymbol{y}|X, \boldsymbol{\theta}, H_i) p(\boldsymbol{\theta}|H_i)}{p(\boldsymbol{y}|X, H_i)}$$

where $p(\boldsymbol{\theta}|H_i)$ is the hyper-prior (the prior for the hyperparameters). The normalizing constant is given by:

$$p(\boldsymbol{y}|X, H_i) = \int p(\boldsymbol{y}|X, \boldsymbol{\theta}, H_i) p(\boldsymbol{\theta}|H_i) d\boldsymbol{\theta}$$

At the top level the posterior for the model is:

$$p(H_i|X, \boldsymbol{y}) = \frac{p(y|X, H_i) p(H_i)}{p(\boldsymbol{y}|X)}$$

Where $p(\boldsymbol{y}|X) = \sum_i p(\boldsymbol{y}|X, H_i) p(H_i)$. The implementation of Bayesian inference evaluates several integrals. Depending on the details of the models, these integrals may or may not be analytically tractable and in general one may have to resort to analytical approximations or **Markov chain Monte Carlo** (MCMC) methods. The integral of $p(\boldsymbol{y}|X, H_i)$ can then be approximated using a local expansion around the maximum (the **Laplace approximation**). This approximation will be good if the posterior for θ is fairly well peaked, which is more often the case for the hyperparameters than for the parameters themselves. The prior over models H_i in $p(H_i|X, \boldsymbol{y})$ is often taken to be flat, so that a priori does not favour one model over another. It is primarily the marginal likelihood from $p(\boldsymbol{y}|X, \theta, H_i)$ involving the integral over the parameter space which distinguishes the Bayesian scheme of inference from other schemes based on

optimization. It is a property of the marginal likelihood that it automatically incorporates a trade-off between model fit and model complexity. This is the reason why the marginal likelihood is valuable in solving the model selection problem.

5.6.1 Marginal Likelihood

Bayesian principles provide a persuasive and consistent framework for inference. Unfortunately, for most interesting models for machine learning, the required computations (integrals over parameter space) are **analytically intractable**, and good approximations are not easily derived. Gaussian process regression models with Gaussian noise are a rare exception: integrals over the parameters are analytically tractable and at the same time the models are very flexible. Since a Gaussian process model is a non-parametric model, it may not be immediately obvious what the parameters of the model are. Generally, one may regard the noise-free latent function values at the training inputs $\boldsymbol{f}$ as the parameters. Rearranging equations the result is:

$$\log p(\boldsymbol{y}|X,\boldsymbol{\theta}) = -\frac{1}{2}\boldsymbol{y}^T K_y^{-1}\boldsymbol{y} - \frac{1}{2}\log|K_y| - \frac{n}{2}\log 2\pi$$

Where $K_y = K_f + \sigma_n^2 I$ is the covariance matrix for the noisy targets $\boldsymbol{y}$ (and K_f is the covariance matrix for the noise-free latent $\boldsymbol{f}$), and we now explicitly write the marginal likelihood conditioned on the hyperparameters (the parameters of the covariance function) θ. The three terms of the marginal likelihood $\log p(\boldsymbol{y}|X,\boldsymbol{\theta})$ have readily interpretable roles: the only term involving the observed targets is the **data-fit** $-\frac{1}{2}y^T K_y^{-1}\boldsymbol{y}$; $\frac{1}{2}\log|K_y|$ is the **complexity penalty** depending only on the covariance function and the inputs and $\frac{n}{2}log(2\pi)$ is a **normalization constant**. In Figure 74(a) is illustrated an example of log marginal likelihood. The data-fit decreases monotonically with the length-scale, since the model

becomes less and less flexible. The negative complexity penalty increases with the length-scale, because the model gets less complex with growing length-scale. The marginal likelihood itself peaks at a value close to 1. For length-scales somewhat longer than 1, the marginal likelihood decreases rapidly due to the poor ability of the model to explain the data.

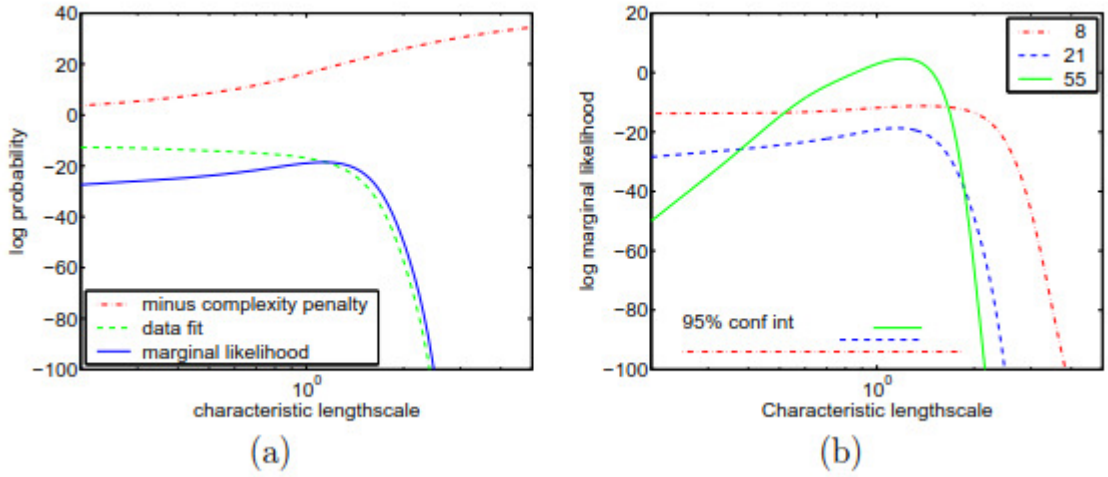


Figure 76 (a) shows a decomposition of the log marginal likelihood and its constituents: data-fit and complexity penalty,as a function of the charateristic length-scale. (b) shows the log marginal likelihood as a function of the charateristic length-scale for different sizes of training sets.

For smaller length-scales the marginal likelihood decreases somewhat more slowly, corresponding to models that do accommodate the data, but waste predictive mass at regions far away from the underlying function. In Figure 76(b) the dependence of the log marginal likelihood on the characteristic length-scale is shown for different numbers of training cases. Generally, the more data, the more peaked the marginal likelihood. For very small numbers of training data points the slope of the log marginal likelihood is very shallow as when only a little data has been observed, both very short and intermediate values of the length-scale are consistent with the data. With more data, the complexity term gets more severe, and discourages too short length-scales. likelihood To set the hyperparameters by maximizing the marginal likelihood, we seek the partial

derivatives of the marginal likelihood w.r.t. the hyperparameters. Deriving expression w.r.t. to θ_j

$$\frac{\partial}{\partial \theta_j} \log p(\boldsymbol{y}|X, \boldsymbol{\theta}) = \frac{1}{2} \boldsymbol{y}^T K^{-1} \frac{\partial K}{\partial \theta_j} K^{-1} \boldsymbol{y} - \frac{1}{2} tr\left(K^{-1} \frac{\partial K}{\partial \theta_j}\right) = \frac{1}{2} tr\left((\alpha\alpha^T - K^{-1}) \frac{\partial K}{\partial \theta_j}\right)$$

Where $\alpha = K^{-1}\boldsymbol{y}$

The complexity of computing the marginal likelihood is dominated by the need to invert the K matrix (the log determinant of K is easily computed as a by-product of the inverse). Standard methods for matrix inversion of positive definite symmetric matrices require time $O(n^3)$ for inversion of an n by n matrix. Once K^{-1} is known, the computation of the derivatives requires only time $O(n^2)$ per hyperparameter. There is no guarantee that the marginal likelihood does not suffer from multiple local optima. Practical experience with simple covariance functions seem to indicate that local maxima are not a devastating problem, but certainly they do exist.

5.7 Gaussian Processes Calibration Results

Calibration using Gaussian Processes has been developed with the help of George, a fast and flexible Python library, for Gaussian Process (GP) Regression. **George** is focused on efficiently evaluating the marginalized likelihood of a dataset under a GP prior, even as this dataset gets Big. It's possible to create a new kernel, combine and use different kernels.

The main strength of the library lies in the possibility of optimizing kernel parameters, minimizing marginal likelihood using L-BFGS algorithm, a method that approximates the BFGS algorithm using a limited amount of computer memory. BFGS algorithm is an iterative method for solving unconstrained nonlinear optimization problems using approximation of the Hessian matrix. A big limitation lies in having to use a limited number of samples ($\leq 20\,000$)

when compared with the size of the dataset n($\geq 500\,000$). To overcome this limit, it was decided to create k small models using $m \ll n$ samples each and combine the mean and variance predicted of the individual models. Calling $\widehat{f_i}$ mean predicted by each Gaussian Process, and σ_i^2 variance associated with the $i-th$ model prediction, final mean and variance is calculated according to the following equation

$$\hat{f} = \frac{1}{k}\sum_{i=1}^{k} \hat{f}_i$$

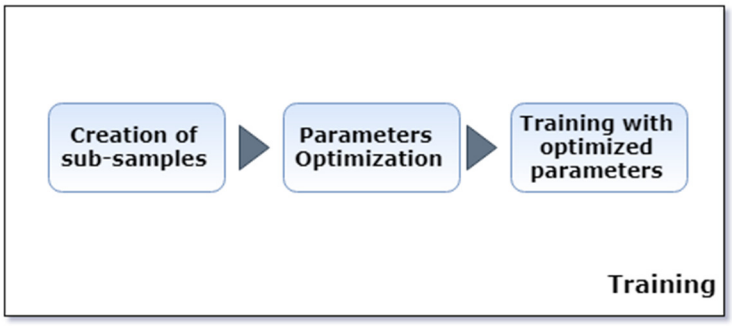
$$var(\hat{f}) = \frac{1}{k^2}\sum_{i=1}^{k} \sigma_i^2$$

However, the implementation turns out to be serial and not parallel. This implementation tries to overcome memory limitation of the PC (framework allocates matrix statically before computation) and numerical stability in inversion of covariance matrix. Different kernels have been used and combined in order to find which one performs better. When the structure of the kernel is fixed a routine optimizes marginal likelihood, finding best parameters for the kernel. Kernel chosen was Matérn52 along each axis, but with different values of length scale (5 for force prediction and 1 for torque prediction).

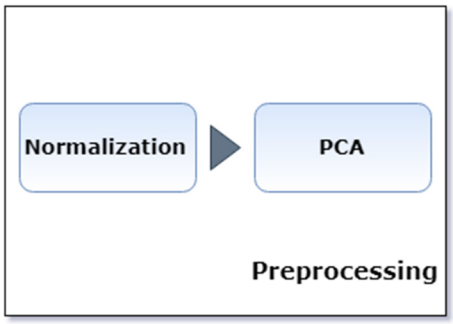
$$k_{tot} = A + B * k_{Matérn}(r)$$

Where A and B are constant kernel. Results on every axis are shown in the following figures. Data flow for Gaussian Processes is made up, as always, of a preprocessing step, then k sub-samples of our dataset with replacement are created. This procedure is repeated along each axis, and then training starts. Finally, according to that previously described, parameters of the kernel chosen, are optimized using a built-in minimization function of Scikit-learn library.

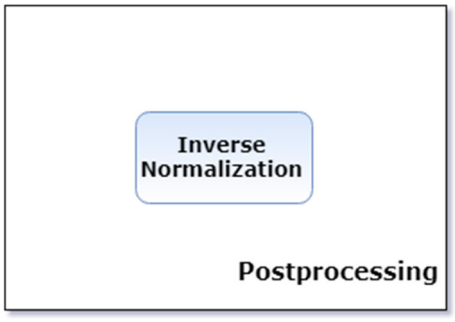
Figure 77 Data flow Gaussian Processes



(a)



(b)



(c)

Figure 78 (a) Training block. (b) Preprocessing block. (c) Postprocessing block

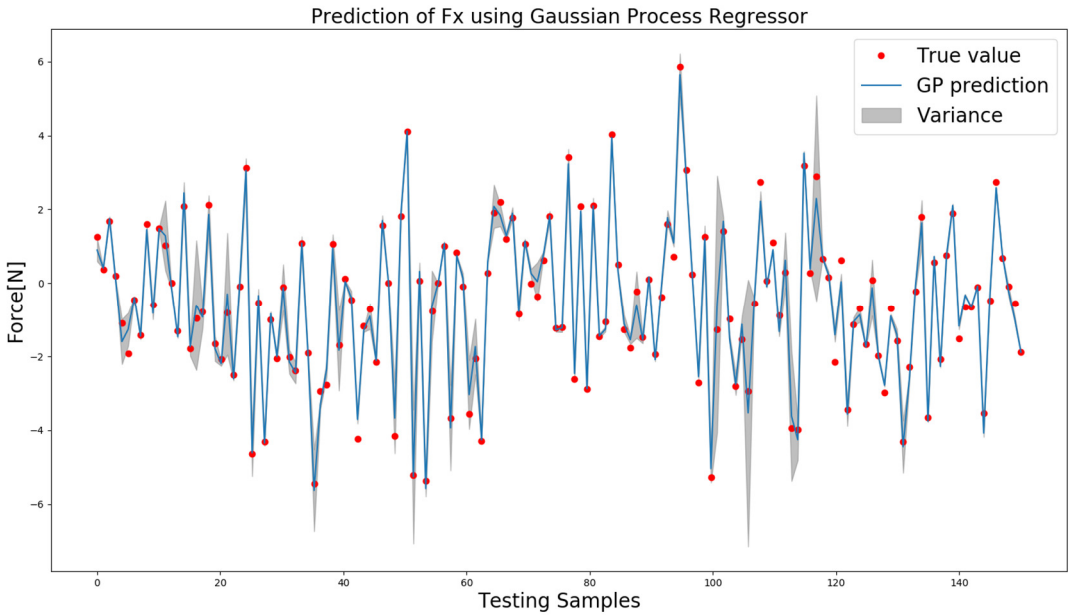


Figure 79 Prediction of F_x

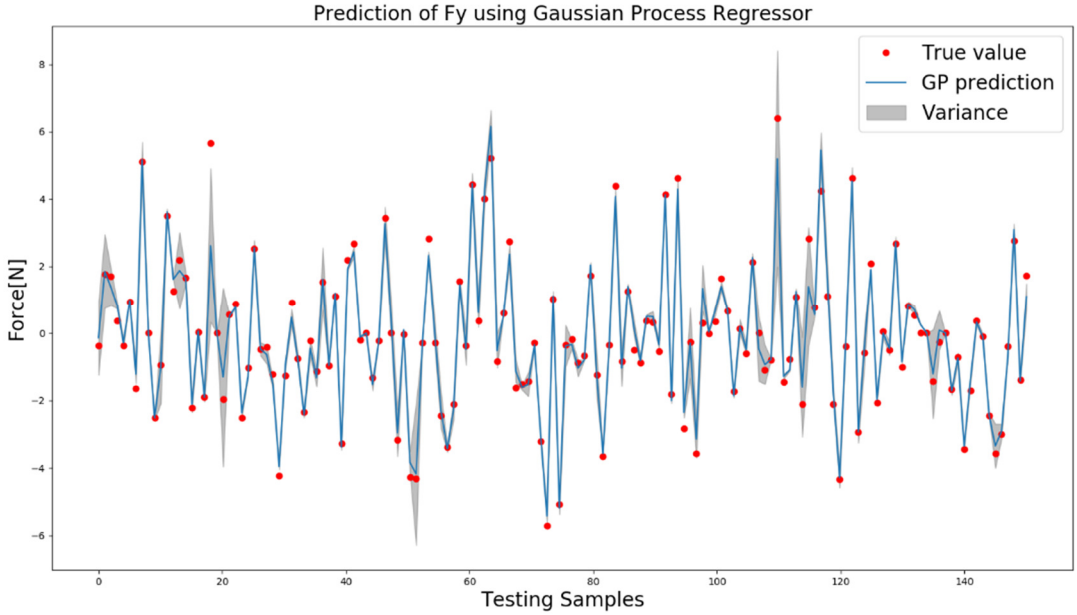


Figure 80 Prediction of F_y

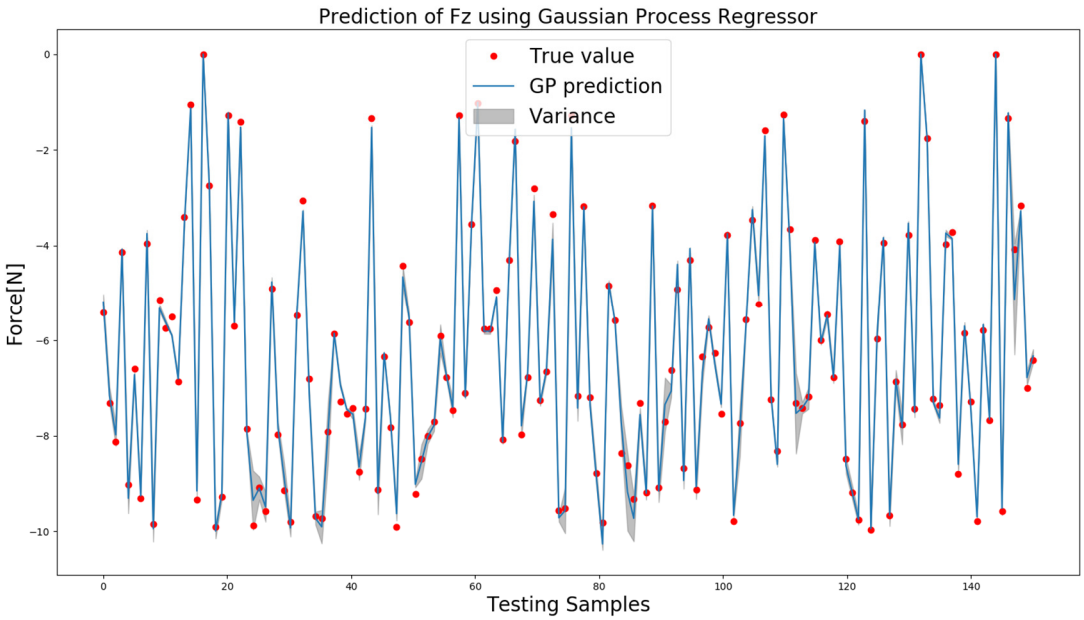


Figure 81 Prediction of F_z

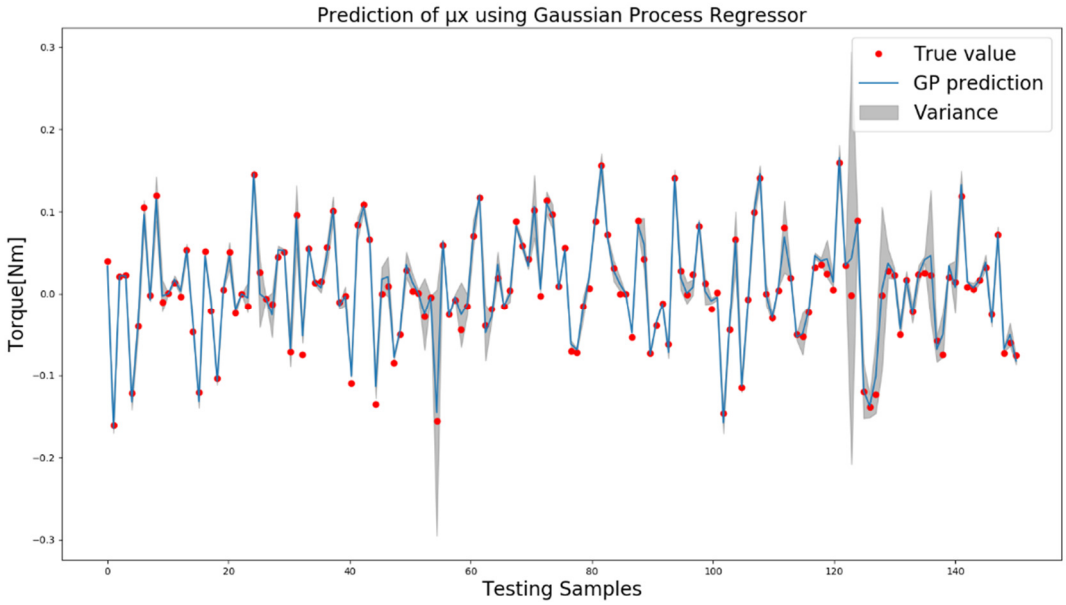


Figure 82 Prediction of μ_x

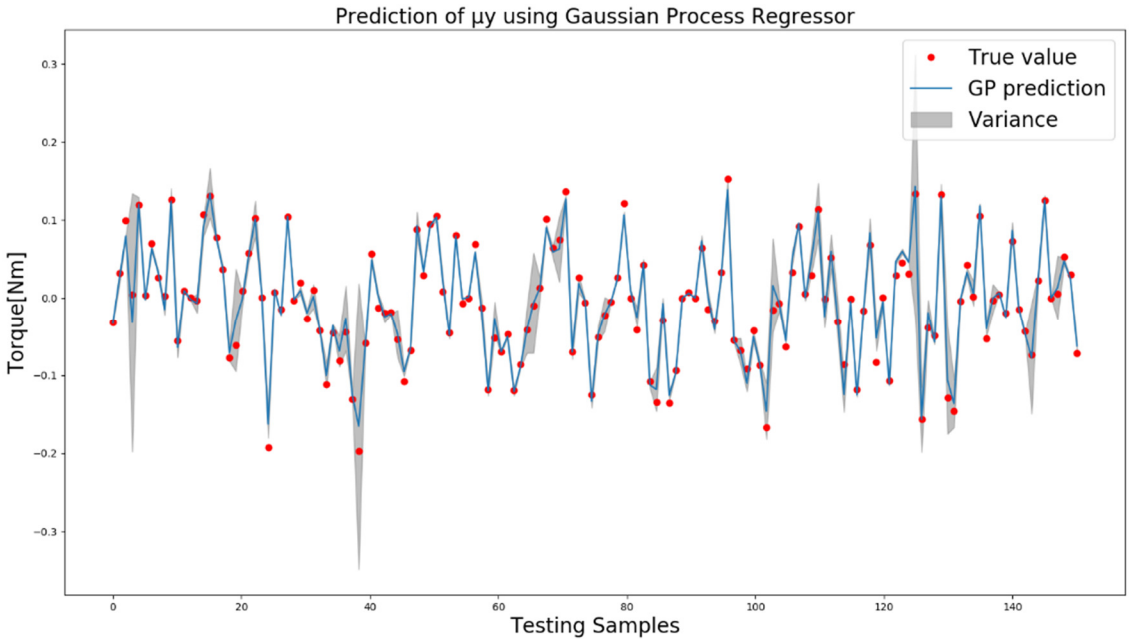


Figure 83 Prediction of μ_y

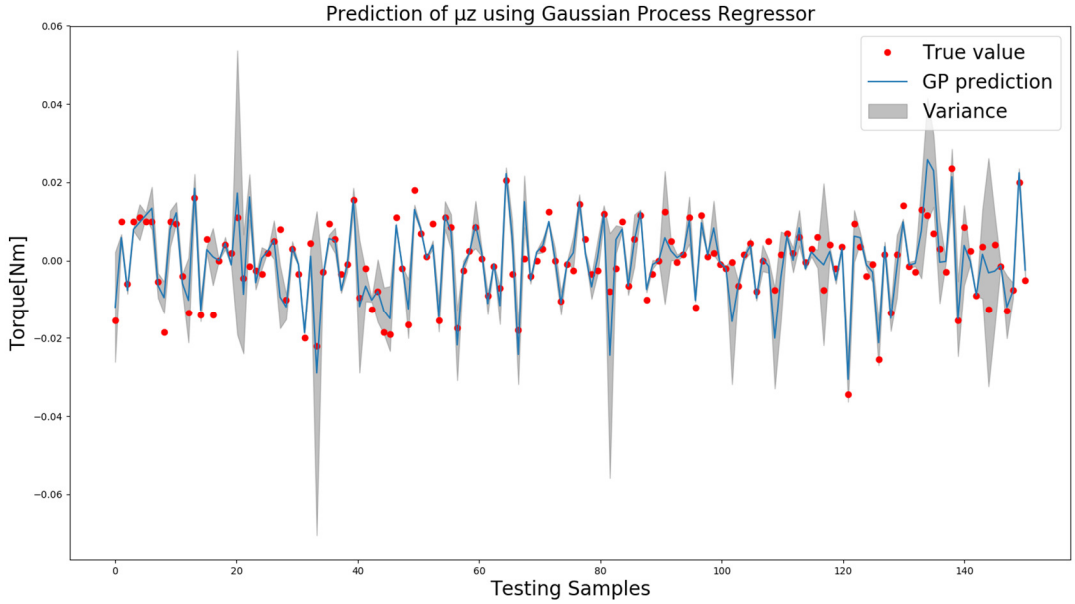


Figure 84 Prediction of μ_z

Chapter 6 Deep Learning

AI is a study of how human brain think, learn, decide and work, when it tries to solve problems. However, two categories of AI are frequently mixed up: Machine Learning and Deep Learning. Both of these refer to statistical modelling of data to extract useful information or make predictions.

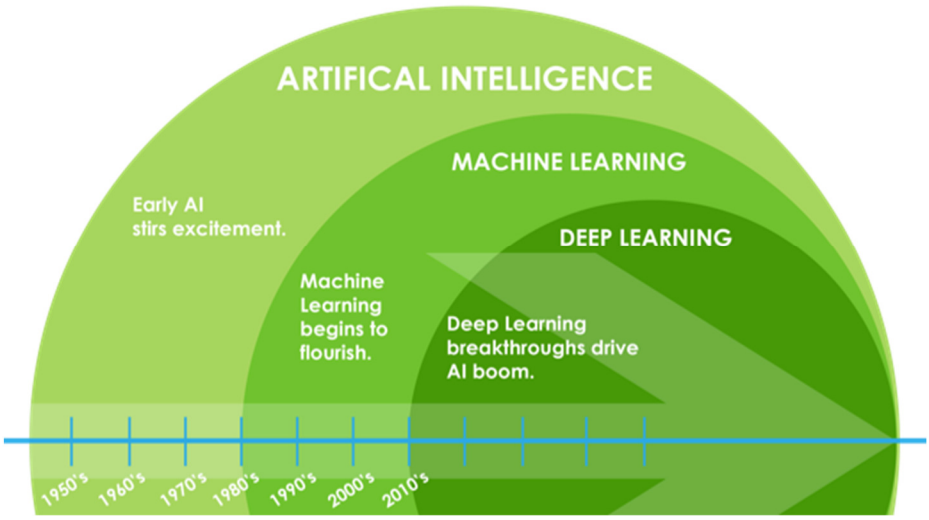


Figure 85 Road to Deep Learning

Machine Learning is a method of statistical learning where each instance in a dataset is described by a set of features or attributes. In contrast, the term "Deep Learning" is a method of statistical learning that extracts features or attributes from raw data. Deep Learning does this by utilizing neural networks with many hidden layers, big data, and powerful computational resources. The terms seem somewhat interchangeable, however, with Deep Learning methods, the algorithm constructs representations of the data automatically. In contrast, data representations are hard-coded as a set of features in machine learning algorithms,

requiring further processes such as feature selection and extraction, (such as PCA).

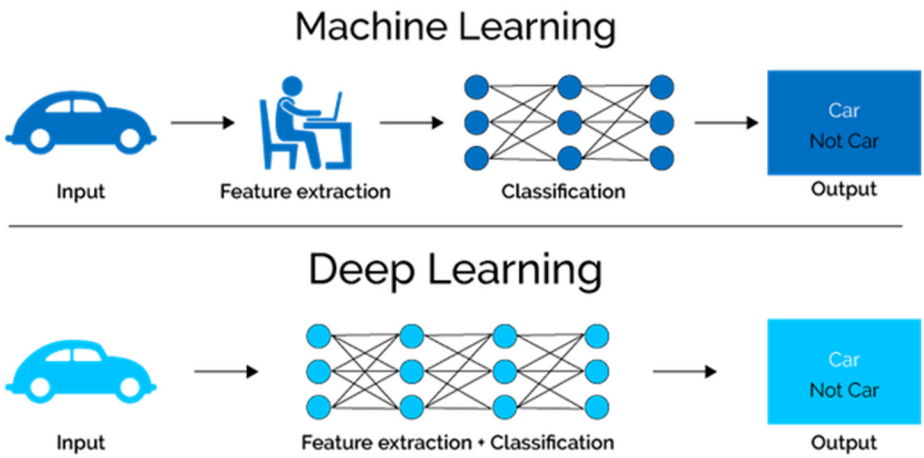


Figure 86 Difference between workflow of Machine Learning vs Deep Learning

6.1 Machine Learning vs Deep Learning

After an overview of Machine Learning and Deep Learning, consider few important points and compare the two techniques.

- **Data Size:** Both Machine Learning and Deep Learning are able to handle massive dataset sizes, however, machine learning methods make much more sense with small datasets. For example, with 100 data points, decision trees, k-nearest neighbours, and other machine learning models will be much more valuable to you than fitting a deep neural network on the data.

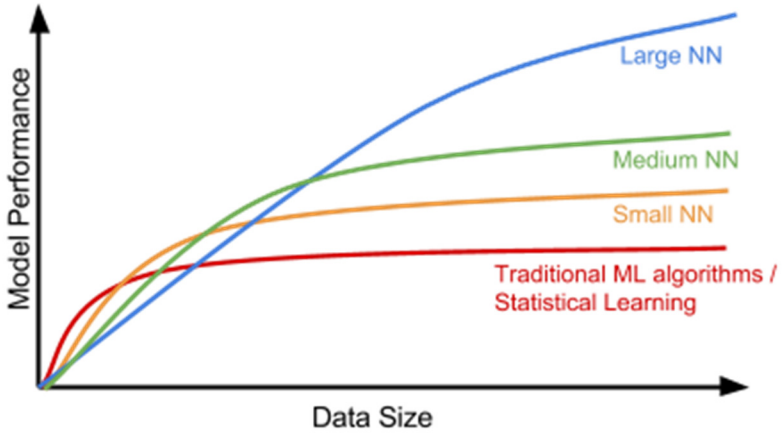


Figure 87 Dependency from the data

- **Hardware dependencies:** generally, deep learning depends on high-end machines. While traditional learning depends on low-end machines. Thus, deep learning requirement includes GPUs.

- **Interpretability**: a lot of the criticism of deep learning methods and machine learning algorithms such as Support Vector Machine or (maybe, because you can at least visualize the constituent probabilities making up the output), Naive Bayes, are due to their difficulty to interpret. For example, when a Convolutional Neural Network outputs 'cat' in a dog vs. cat problem, nobody seems to know why it did that. In contrast, when you are modelling data such as an electronic health record or bank loan dataset with a machine learning technique, it is much easier to understand the reasoning for the model's prediction. One of the best examples of interpretability is decision trees where you follow logical tests down nodes of the tree until you reach a decision.

- **Feature engineering**: feature engineering is the process of transforming raw data into features that represent better the underlying problem to the predictive models, resulting in improved model accuracy on unseen data. Feature engineering turn your inputs into things the algorithm can understand.

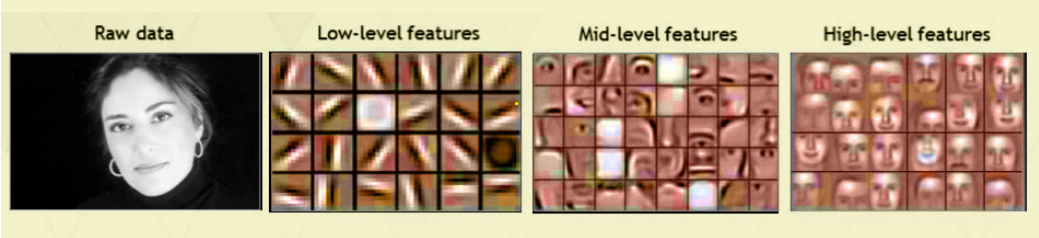


Figure 88 Features extraction example

- **Execution time**: usually, deep learning takes more time as compared to machine learning to train. The main reason behind its long time is that so many parameters in deep learning algorithm. Whereas machine learning takes much less time to train, ranging from a few seconds to a few hours.

6.2 Neural Networks Structure

An **artificial neural network** is a network of simple elements called artificial neurons, which receive input, change their internal state (**activation**) according to that input, and produce output depending on the input and activation.

An artificial neuron mimics the working of a biophysical neuron with inputs and outputs, but is not a biological neuron model.

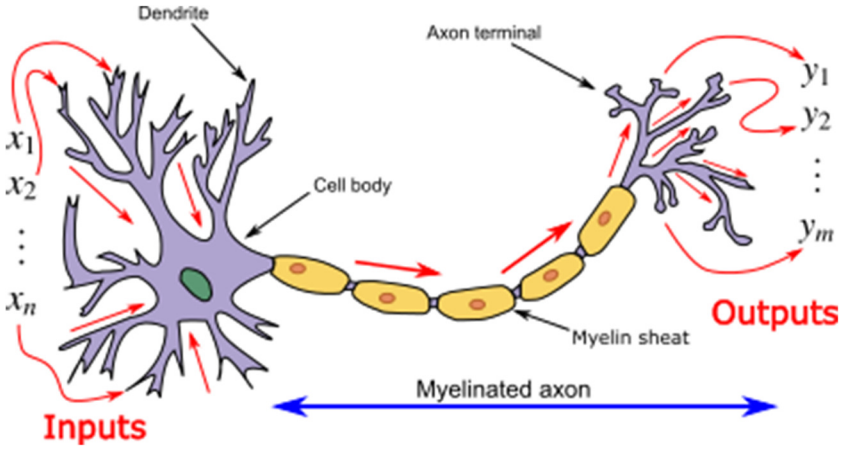


Figure 89 Analogy artificial neuron with biophysical neuron

The basic unit of computation in a neural network is the neuron, often called a node or unit. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated weight (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function f to the weighted sum of its inputs as shown in Figure below:

Linear function: $Wx + b$

Non-linearity activation $f(x)$

Every neuron computes $f(Wx + b)$

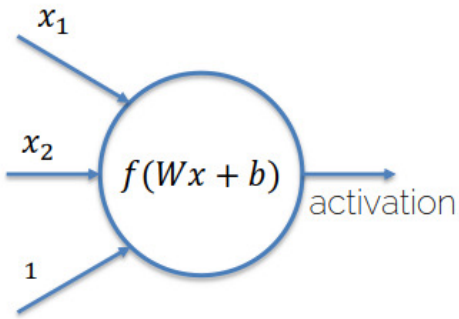


Figure 90 Structure of a neuron

The above network takes numerical inputs X_1 and X_2 and has weight weights w_1 and w_2 associated with those inputs. Additionally, there is another

input 1 with weight b (called the Bias) associated with it. We will learn more details about role of the bias later. The output Y from the neuron is computed as shown in the figure. The function f is non-linear and is called the **Activation Function**. The purpose of the activation function is to introduce non-linearity into the output of a neuron. This is important because most real world data is non linear and we want neurons to learn these non linear representations.

6.3 Activation Function

Every activation function (or *non-linearity*) takes a single number and performs a certain fixed mathematical operation on it. There are several activation functios:

- **Sigmoid**: takes a real-valued input and squashes it to range [0,1].
- **Tanh**: takes a real-valued input and squashes it to the range $[-1, 1]$.
- **ReLU**: ReLU stands for Rectified Linear Unit. It takes a real-valued input and thresholds it at zero (replaces negative values with zero).

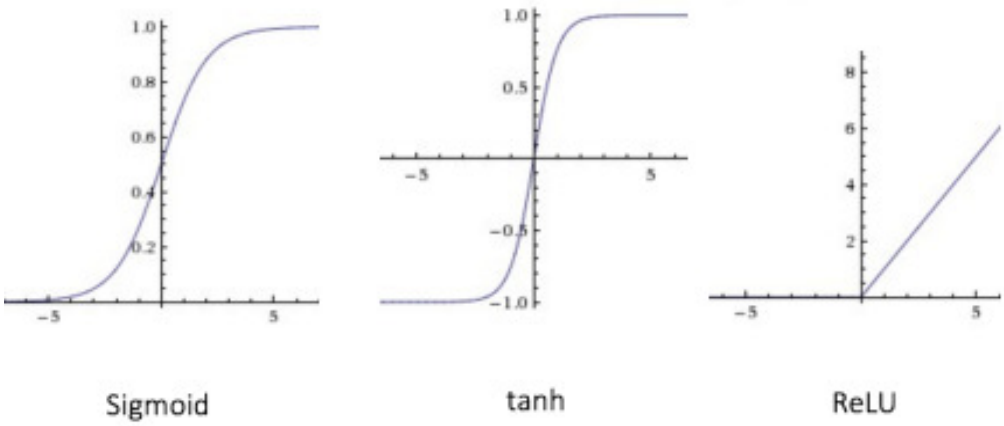


Figure 91 Activation function

6.4 Feedforward Neural Network

The feedforward neural network was the first and simplest type of artificial neural network devised. It contains multiple neurons (nodes) arranged in **layers**. Nodes from adjacent layers have **connections** or **edges** between them. All these connections have **weights** associated with them.

An example of a feedforward neural network is shown in Fig. 92.

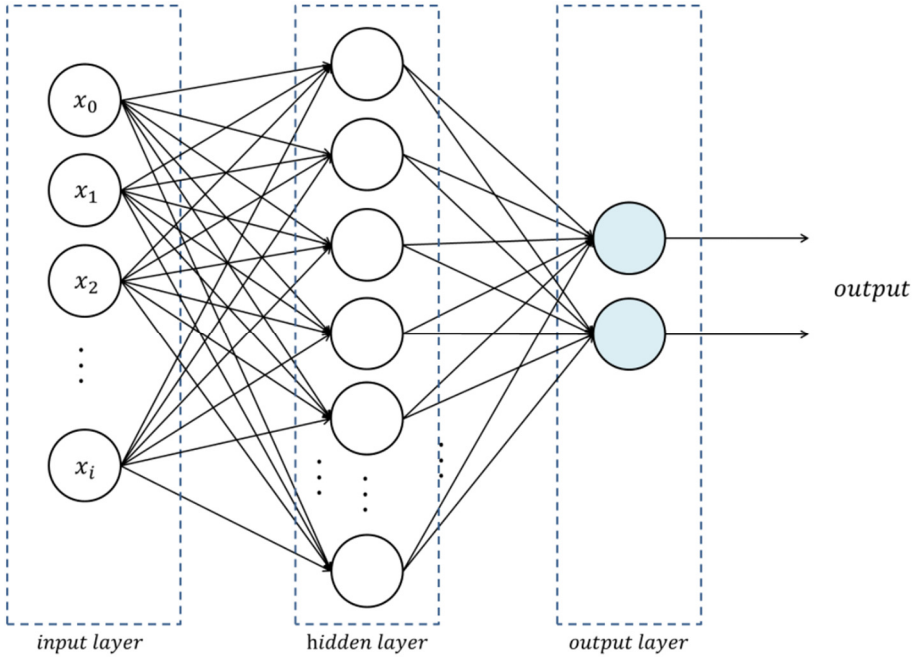


Figure 92 General structure of a feedforward neural network

A feedforward neural network can consist of three types of nodes:

- **Input Nodes** – The Input nodes provide information from the outside world to the network and are together referred to as the "Input Layer". No computation is performed in any of the Input nodes – they just pass on the information to the hidden nodes.
- **Hidden Nodes** – The Hidden nodes have no direct connection with the outside world (hence the name "hidden"). They perform computations and transfer information from the input nodes to the output nodes. A collection

of hidden nodes forms a "Hidden Layer". While a feedforward network will only have a single input layer and a single output layer, it can have zero or multiple Hidden Layers.

- **Output Nodes** – The Output nodes are collectively referred to as the "Output Layer" and are responsible for computations and transferring information from the network to the outside world.

In a feedforward network, the information moves in only one direction – forward – from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network (this property of feed forward networks is different from **Recurrent Neural Networks** in which the connections between the nodes form a cycle).

Two examples of feedforward networks are given below:

- **Single Layer Perceptron** – This is the simplest feedforward neural network and does not contain any hidden layer.
- **Multi Layer Perceptron** – A Multi Layer Perceptron has one or more hidden layers.

6.5 The Back-Propagation Algorithm

Backward Propagation of Errors, often abbreviated as BackProp is a supervised training scheme, which means, it learns from labeled training data (there is a supervisor, to guide its learning).To put in simple terms, BackProp is like "**learning from mistakes**". The supervisor **corrects** the ANN whenever it makes mistakes. An ANN consists of nodes in different layers: input layer, intermediate hidden layers and the output layer. The connections between nodes of adjacent layers have "weights" associated with them. The goal of learning is to

assign correct weights for these edges. Given an input vector, these weights determine what the output vector is. In supervised learning, the training set is labeled. This means, for some given inputs, we know the desired/expected output.

BackProp Algorithm:

Initially all the edge weights are randomly assigned. For every input in the training dataset, the ANN is activated and its output is observed. This output is compared with the desired output that we already know, and the error is "propagated" back to the previous layer using chain-rule. This error is noted and the weights are "adjusted" accordingly using gradient descent. This process is repeated until the output error is below a predetermined threshold.

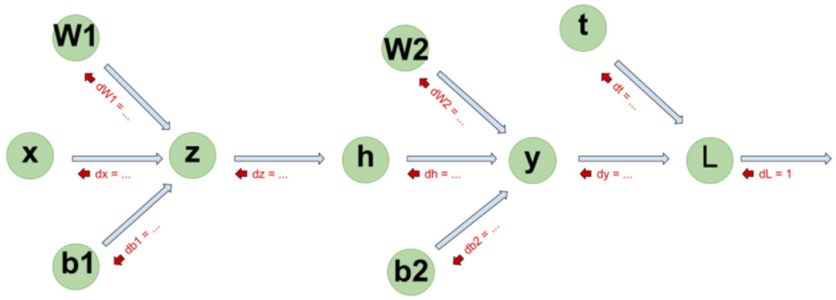


Figure 93 Backpropagation forward pass and backward pass

Once the above algorithm terminates ANN is ready to work with "new" inputs. This ANN is said to have learned from several examples (labeled data) and from its mistakes (error propagation). The heart of backpropagation error is computation of the gradient of the Neural Network.

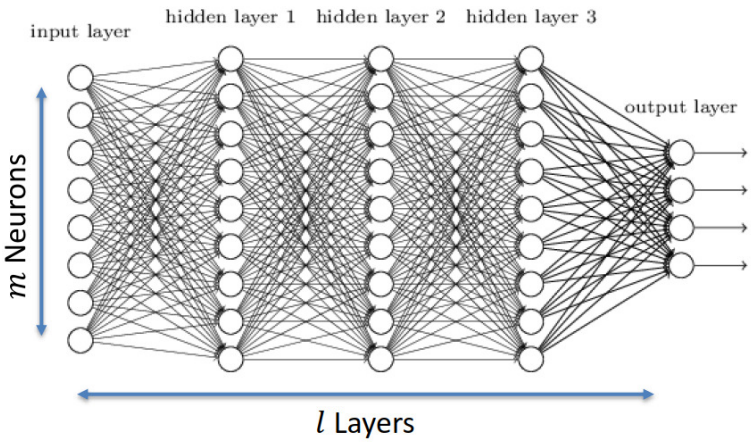


Figure 94 Deep neural network structure

For a given training pair $\{x, y\}$, we want to update all weights; i.e., we need to compute derivatives w.r.t. to all weights.

$$\nabla_w f_{\{x,y\}}(w) = \begin{bmatrix} \frac{\partial f}{\partial w_{0,0,0}} \\ \cdots \\ \frac{\partial f}{\partial w_{l,m,n}} \end{bmatrix} \xrightarrow[\text{DESCENT}]{\text{GRADIENT}} w' = w - \epsilon \nabla_w f_{\{x,y\}}(w)$$

6.6 Optimization Algorithms

Once the analytic gradient is computed with backpropagation, the gradients are used to perform a parameter update. There are several approaches for performing the update. Note that optimization for deep networks is currently a very active area of research. This section highlights some established and common techniques.

6.6.1 Gradient Descent

Gradient descent is a **first-order iterative optimization** algorithm for finding the **minimum** of a function. To find a local minimum of a function using gradient descent, one takes steps proportional to the negative of the gradient (or approximate gradient) of the function at the current point.

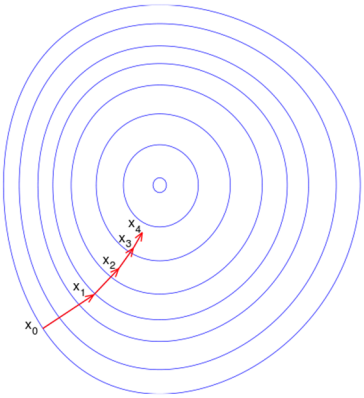


Figure 95 Gradient descent visualization

If, instead, one takes steps proportional to the positive of the gradient, one approaches a local maximum of that function; the procedure is then known as gradient ascent. Gradient descent is also known as steepest descent.

$$w' = w - \epsilon \nabla_x f(x)$$

$\boldsymbol{\epsilon}$ is called **Learning Rate** and it is an hyperparameter of the problem. It determines how fast or slow we will move towards the optimal weights. In order for Gradient Descent to reach the local minimum, we have to set the learning rate to an appropriate value, which is neither too low nor too high. This is because if the steps it takes are too big, it maybe will not reach the local minimum because it just bounces back and forth between the convex function of gradient descent like you can see on the left side of the image below. If you set the learning rate to a very small value, gradient descent will eventually reach the local minimum but it will maybe take too much time like you can see on the right side of the image.

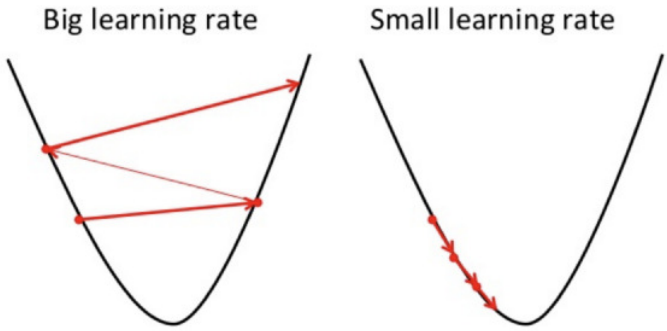


Figure 96 Effects of changing learning rate

This is the reason why the learning rate should be neither too high nor too low.

6.6.2 Stochastic Gradient Descent (SGD)

Gradient is an expectation, and so it can be **approximated** with a small number of samples in a **minibatch**.

Defining a minibatch $B_i = \{\{x_i, y_i\}, \{x_2, y_2\} \dots \{x_m, y_m\}\}$, it's possible to build up to n/m.

$$w^{k+1} = w^k - \alpha \nabla_w L\left(w^k, x_{\{1\dots m\}}, y_{\{1\dots m\}}\right)$$

So parameters at the k-th step are updated using only m samples in the minibatch. However, SGD suffers of some problems:

- Gradient is scaled equally across all dimensions, for this reason learning needs to be chosen conservatively to avoid divergence.
- Slow convergence.
- Finding good learning rate is an art by itself.

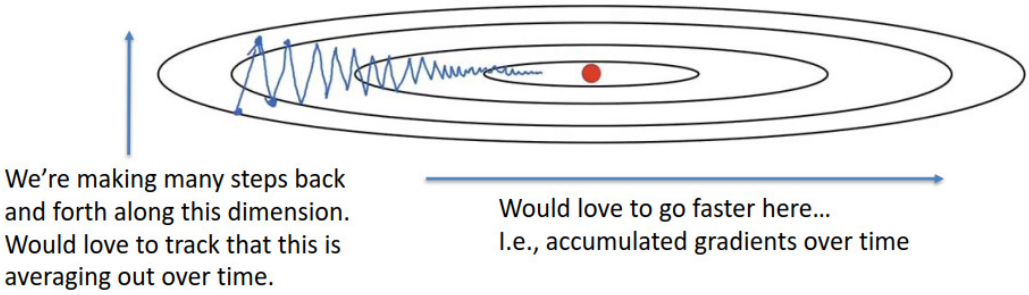


Figure 97 Dampening gradient descent

6.6.3 Gradient Descent with Momentum

Momentum is a method that helps accelerate SGD in the relevant direction and dampens oscillation. It does this by adding a **fraction** β of the update vector of the past time step to the current update vector:

$$v^{k+1} = \beta\, v^k + \nabla_w L(w^k)$$

$$w^{k+1} = w^k - \alpha v^{k+1}$$

Essentially, when using momentum, one pushes a ball down a hill. The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way (until it reaches its terminal velocity if there is air resistance, i.e. $\beta < 1$).

The same thing happens to the parameter. The momentum term increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. As a result, it's possible to achieve faster convergence and reduced oscillation.

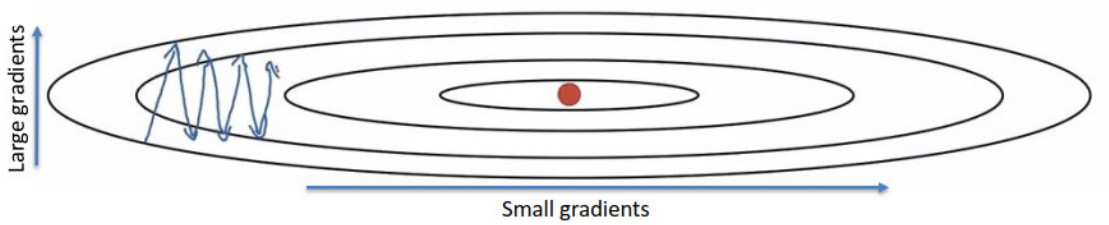


Figure 98 Gradient descent with momentum

6.6.4 Root Mean Squared Prop (RMSProp)

RMSprop divides the learning rate by an exponentially-decaying average of squared gradients. Dividing by square gradients it dampens the oscillations for high-variance directions.

$$s^{k+1} = \beta s^k + (1-\beta)[\nabla_w L \circ \nabla_w L]$$

$$w^{k+1} = w^k - \alpha \frac{\nabla_w L}{\sqrt{s^{k+1}} + \epsilon}$$

ϵ is used to avoid division by zero.

α needs tuning, β often 0.9, ϵ tipically 10^{-8}

Can use faster learning rate because it is less likely to diverge

- Speed up learning speed
- Variance of gradients -> second momentum

6.6.5 Adaptive Moment Estimation (Adam)

Combines Momentum and RMSProp

First momentum $m^{k+1} = \beta_1 m^k + (1-\beta_1)\nabla_w L(w^k)$

Second momentum $v^{k+1} = \beta_2 v^k + (1-\beta_2)[\nabla_w L(w^k) \circ \nabla_w L(w^k)]$

$$w^{k+1} = w^k - \frac{m^{k+1}}{\sqrt{v^{k+1}} + \epsilon}$$

α needs tuning, β_1 often 0.9, β_2 often 0.999, ϵ tipically 10^{-8}.

6.7 Loss Functions

All the algorithms in machine learning rely on minimizing or maximizing a function, called "loss functions". A loss function is a measure of how good a prediction model does in terms of being able to predict the expected outcome. Loss depends on a number of factors including the presence of outliers, machine learning algorithm, ease of finding the derivatives...

- **Mean Square Error (MSE)** is the most commonly used regression loss. MSE is the sum of squared distances between targets and predicted.

$$MSE = \frac{\sum_{i=1}^{n}(y_i - y_i^p)^2}{n}$$

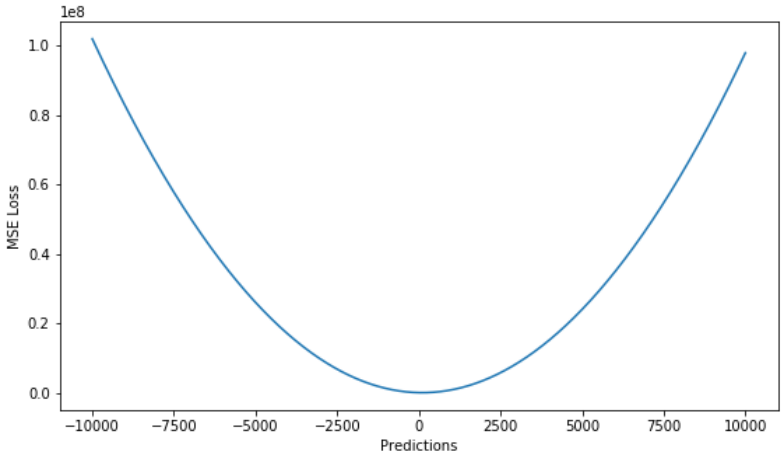


Figure 99 MSE

L2 loss is sensitive to outliers, but gives a more stable and closed form solution (by setting its derivative to 0.)

- **Mean Absolute Error (MAE) or L1 Loss** is another loss function used for regression models. MAE is the sum of absolute differences between our target and predicted variables. So it measures the average magnitude of errors in a set of predictions, without considering their directions.

$$MAE = \frac{\sum_{i=1}^{n} |y_i - y_i^p|}{n}$$

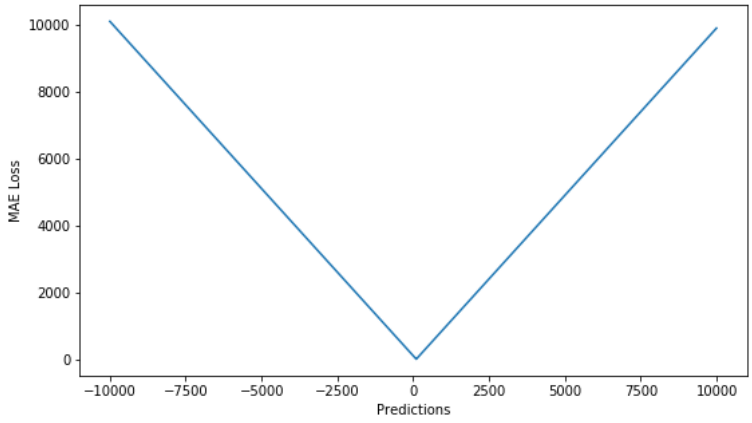


Figure 100 MAE Loss

MAE loss is useful if the training data is corrupted with outliers (i.e. we erroneously receive unrealistically huge negative/positive values in our training environment, but not our testing environment). MAE loss is **more robust** to outliers, but its derivatives are not continuous, making it inefficient to find the solution.

6.8 Regularization Technique

Regularization is a technique which makes slight modifications to the learning algorithm such that the model generalizes better. This in turn improves the model's performance on the unseen data as well. According to the picture, when the point moves towards the right in this image, the model tries to learn too well

the details and the noise from the training data, which ultimately results in poor performance on the unseen data.

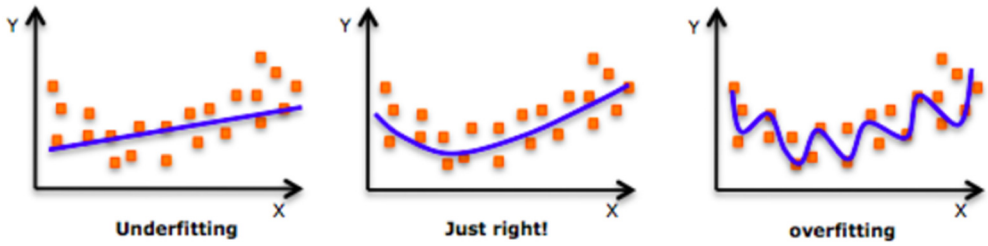


Figure 101 Overfitting and underfitting

Some techniques for regularizations are:

- **L2 & L1 regularization**. They are the most common types of regularization. These update the general cost function by adding another term known as the regularization term.

$$\text{Cost function} = \text{Loss} + \text{Regularization term}$$

Due to the addition of this regularization term, the values of weight matrices decrease because it assumes that a neural network with smaller weight matrices leads to simpler models. Therefore, it will also reduce overfitting to quite an extent. However, this regularization term differs in L1 and L2. In L2, cost function is:

$$\text{Cost function} = \text{Loss} + \frac{\lambda}{2m}\sum |w|^2$$

Here, lambda is the regularization parameter. It is the hyperparameter whose value is optimized for better results. L2 regularization is also known as weight decay as it forces the weights to decay towards zero (but not exactly zero). In L1, we have:

$$\text{Cost function} = \text{Loss} + \frac{\lambda}{2m}\sum |w|$$

In this, we penalize the absolute value of the weights. Unlike L2, the weights may be reduced to zero here. Hence, it is very useful when we are trying to **compress** the model. Otherwise, L2 is usually preferred over L1.

- **Dropout**. This is the one of the most interesting types of regularization techniques. It also produces very good results and is consequently the most frequently used regularization technique in the field of deep learning. To understand dropout, let's say our neural network structure is akin to the one shown below:

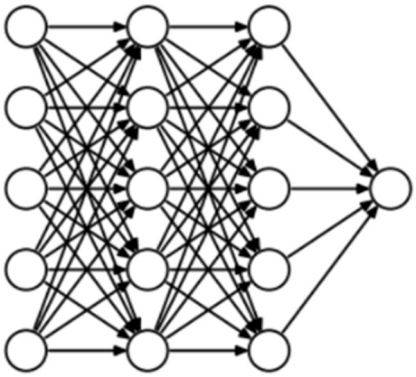


Figure 102 Neural network before dropout

At every iteration, Dropout randomly selects some nodes and removes them along with all of their incoming and outgoing connections as shown below.

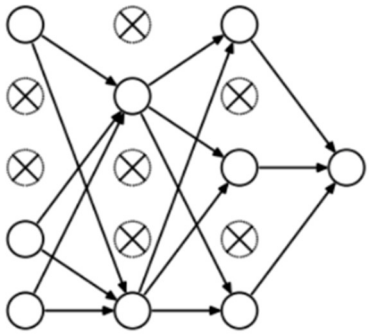


Figure 103 Neural network after dropout

So each iteration has a different set of nodes and this results in a different set of outputs. Dropout can also be thought of as an **ensemble technique** in machine learning. Ensemble models usually perform better than a single

model as they capture more randomness. Similarly, dropout also performs better than a normal neural network model. This probability of choosing how many nodes should be dropped is a hyperparameter of the dropout function. Dropout can be applied to both the hidden layers as well as the input layers. Due to these reasons, dropout is usually preferred in large neural networks structures in order to introduce more randomness.

- **Batch Normalization**: Batch normalization reduces the amount by what the hidden unit values shift around (covariance shift). To increase the stability of a neural network, batch normalization **normalizes** the output of a previous activation layer by subtracting the batch mean and dividing by the batch standard deviation.

$$\mu_{\mathrm{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_{\mathrm{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathrm{B}})^2$$

$$\widehat{x_i} = \frac{x_i - \mu_{\mathrm{B}}}{\sqrt{\sigma_{\mathrm{B}}^2 + \epsilon}}$$

$$y_i = \gamma\widehat{x_i} + \beta \triangleq BN_{\gamma,\beta}(x_i)$$

Batch normalization adds two trainable parameters to each layer, so the normalized output is multiplied by a "standard deviation" parameter (gamma) and add a "mean" parameter (beta). It allows each layer of a network to learn by itself a little bit more independently of other layers. It's possible to use higher learning rates because batch normalization makes sure that there's no activation that's gone really high or really low. It reduces overfitting because it has a slight regularization effects. Similar to dropout, it adds some noise to each hidden layer's activations.

6.9 Hyperparameters optimization

Training Neural Networks can involve many hyperparameter settings. The most common hyperparameters in context of Neural Networks include: the initial learning rate learning rate decay schedule (such as the decay constant) regularization strength (L2 penalty, dropout strength). Larger Neural Networks typically require a long time to train, so performing hyperparameter search can take many days/weeks. It is important to keep this in mind since it influences the design of your code base. Let's see some useful tips.

- **Hyperparameter ranges**. Search for hyperparameters on log scale. The same strategy should be used for the regularization strength. Intuitively, this is because learning rate and regularization strength have multiplicative effects on the training dynamics. For example, a fixed change of adding 0.01 to a learning rate has huge effects on the dynamics if the learning rate is 0.001, but nearly no effect if the learning rate when it is 10. This is because the learning rate multiplies the computed gradient in the update.

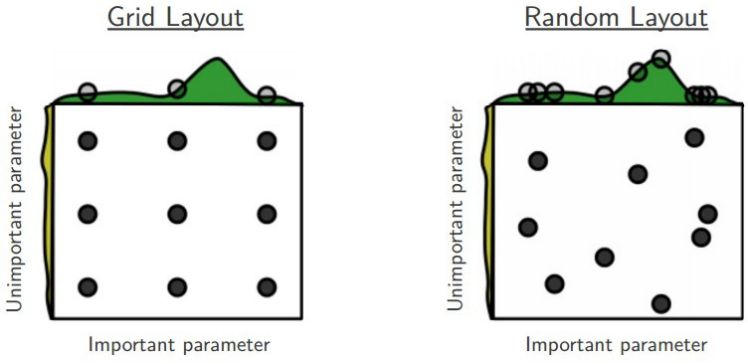


Figure 104 Grid and random search

- **Prefer random search to grid search**. As argued by Bergstra and Bengio "randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid". As it turns out, this is also usually easier to implement.

- **Careful with best values on border**. Sometimes it can happen that you're searching for a hyperparameter (e.g. learning rate) in a bad range. It is important to double check that the final learning rate is not at the edge of this interval, or otherwise you may be missing more optimal hyperparameter setting beyond the interval.

- **Stage your search from coarse to fine**. In practice, it can be helpful to first search in coarse ranges, and then depending on where the best results are turning up, narrow the range.

- **Bayesian Hyperparameter Optimization** is a whole area of research devoted to coming up with algorithms that try to more efficiently navigate the space of hyperparameters. The core idea is to appropriately balance the exploration - exploitation trade-off when querying the performance at different hyperparameters.

6.10 Weights initialization

Before you begin to train the network we have to initialize its parameters.

- **All zero initialization**. This turns out to be a mistake, because if every neuron in the network computes the same output, then they will also all compute the same gradients during backpropagation and undergo the exact same parameter updates. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.

- **Small random numbers**. Therefore, one wants weights to be very close to zero, but as we have argued above, not identically zero. As a solution, it is common to initialize the weights of the neurons to small numbers and refer to doing so as symmetry breaking. The idea is that the neurons are all random and unique in the beginning, so they will compute distinct

updates and integrate themselves as diverse parts of the full network. With this formulation, every neuron's weight vector is initialized as a random vector sampled from a multi-dimensional Gaussian, so the neurons point in **random direction** in the input space. It is also possible to use small numbers drawn from a uniform distribution, but this seems to have relatively little impact on the final performance in practice. A Neural Network layer that has very small weights will during backpropagation compute very small gradients on its data (since this gradient is proportional to the value of the weights). This could greatly diminish the "gradient signal" flowing backward through a network, and could become a concern for deep networks.

- **Variances** $= \frac{1}{\sqrt{n}}$. One problem with the above suggestion is that the distribution of the outputs from a randomly initialized neuron has a variance that grows with the number of inputs. It turns out that data can normalize the variance of each neuron's output to 1 by scaling its weight vector by the square root of its fan-in (i.e. its number of inputs). This ensures that all neurons in the network initially have approximately the same output distribution and empirically improves the rate of convergence. The sketch of the derivation is as follows: Consider the inner product $s = \sum_i^n w_i x_i$ between the weights w and input x, which gives the raw activation of a neuron before the non-linearity.

$$Var(s) = Var\left(\sum_i^n w_i x_i\right)$$

$$= \sum_i^n Var(w_i x_i)$$

$$= \sum_i^n [E(w_i)]^2 Var(x_i) + [E(x_i)]^2 Var(w_i) + Var(x_i)Var(w_i)$$

$$= \sum_i^n Var(x_i)Var(w_i) = (nVar(w))Var(x)$$

It has been assumed zero mean inputs and weights, so $E[x_i] = E[w_i] = 0$. Note that this is not generally the case: For example ReLU units will have a positive mean. An assumption in the last step was made: w_i, x_i are identically distributed. To make variance unitary, the variance of every weight $w = \frac{1}{n}$. Since $Var(aX) = a^2 Var(X)$ for a random variable X and a scalar a, this implies that we should draw from unit gaussian and then scale it by $a = \sqrt{1/n}$, to make its variance $\frac{1}{n}$. A similar analysis is carried out by Glorot et al. The authors end up recommending an initialization of the form $Var(w) = \frac{2}{n_{in}+n_{out}}$ where n_{in}, n_{out} are the number of units in the previous layer and the next layer. This is based on a compromise and an equivalent analysis of the backpropagated gradients. A more recent paper on this topic by He et al., derives an initialization specifically for ReLU neurons, reaching the conclusion that the variance of neurons in the network should be $\frac{2}{n}$. This is the current recommendation for use in practice in the specific case of neural networks with ReLU neurons.

- **Initializing the biases**. It is possible and common to initialize the biases to be zero, since the asymmetry breaking is provided by the small random numbers in the weights. For ReLU non-linearities, some people like to use small constant value such as 0.01 for all biases because this ensures that

all ReLU units fire in the beginning and therefore obtain and propagate some gradient. However, it is not clear if this provides a consistent improvement (in fact some results seem to indicate that this performs worse) and it is more common to simply use 0 bias initialization.

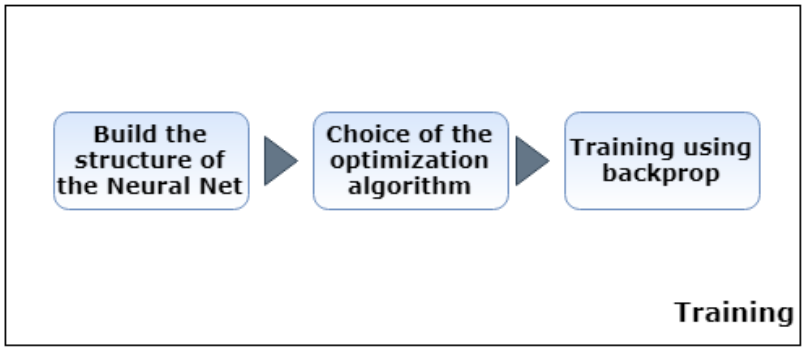
- **Batch Normalization**. It has become a very common practice to use Batch Normalization in neural networks. In practice networks use Batch Normalization are significantly more robust to bad initialization. Additionally, batch normalization can be interpreted as doing preprocessing at every layer of the network, but integrated into the network itself in a differentiable manner.

6.11 Neural Network Calibration

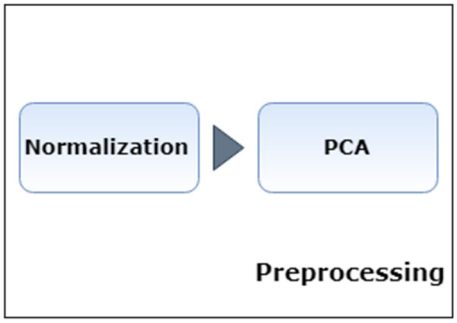
After a preprocessing step, with dimensionality reduction, neural networks were trained. There are **no rules** that indicate how to build a neural network. Using MSE on test set, as a metric for performances, different architectures have been tested. A variable number of hidden layers (from 4 to 8) of 100 neurons each with different types of regularization: Dropout, Batch Normalization, L2. Different combinations were tested, first individually, and then combined together. Regularization L1 was **neglected** because shows worse performance compared to L2. **Tanh** as activation function, **RMSProp** was the optimization algorithm adopted. To condense error values (on Test set) of force and torque, the arithmetic average was used.

All the code was easily developed using **Keras**, an high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation.

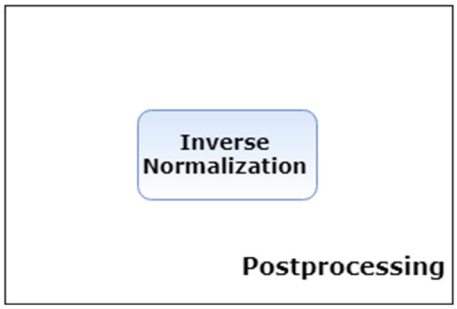
Figure 105 Deep Learning Model Training



(a)



(b)



(c)

Figure 106 (a) Training Model block. (b) Preprocessing block. (c) Postprocessing block.

Results of the experiments are shown in the following tables:

L2 REGULARIZATION		$\lambda = 0.01$	$\lambda = 0.1$	$\lambda = 1$	$\lambda = 10$
4 Layers	F[N]	0.596	0.650	1.101	3.169
	μ[Nm]	5.07e-04	5.38e-04	8.49e-04	0.0021
5 Layers	F[N]	0.581	0.643	1.052	3.139
	μ[Nm]	4.95e-04	5.46e-04	8.08e-04	0.0020
6 Layers	F[N]	0.564	0.619	1.045	3.123
	μ[Nm]	4.79e-04	5.22e-04	8.00e-4	0.0020
7 Layers	F[N]	0.577	0.601	1.055	3.117
	μ[Nm]	5.12e-04	5.12e-04	8.12e-04	0.0020
8 Layers	F[N]	0.565	0.609	1.042	3.111
	μ[Nm]	5.24e-04	5.24e-04	8.05e-04	0.0020

Table 1 Neural network tests with different number of layers and different L2 regularization

BATCH NORMALIZATION		$\lambda = 0.01$	$\lambda = 0.1$	$\lambda = 1$	$\lambda = 10$
4 Layers	F[N]	0.592	0.613	1.083	3.152
	μ[Nm]	4.99e-04	5.11e-04	8.31e-04	0.0021
5 Layers	F[N]	0.571	0.626	1.068	3.137
	μ[Nm]	4.83e-04	5.26e-04	8.2e-04	0.0020
6 Layers	F[N]	0.551	0.599	1.045	3.130
	μ[Nm]	4.70e-04	5.06e-04	8.06e-04	0.0020
7 Layers	F[N]	0.536	0.601	1.039	3.126
	μ[Nm]	4.55e-04	5.02e-04	7.96e-04	0.0020
8 Layers	F[N]	0.532	0.608	1.035	3.117
	μ[Nm]	4.53e-04	5.12e-04	7.94e-04	0.0020

Table 2 Neural Network with different number of layers, Batch normalization before each hidden layer and regularization on last layer

DROPOUT		LAST LAYER	HIDDEN LAYER
4 Layers	F[N]	0.616	1.470
	μ[Nm]	5.28e-04	0.0013
5 Layers	F[N]	0.581	1.674
	μ[Nm]	4.95e-04	0.0015
6 Layers	F[N]	0.615	1.884
	μ[Nm]	5.24e-04	0.0015
7 Layers	F[N]	0.578	2.000
	μ[Nm]	4.92e-04	0.0017
8 Layers	F[N]	0.570	2.087
	μ[Nm]	4.83e-04	0.0017

Table 3 Neural Network with different number of layers and dropout applied on last layer or hidden layer

BATCH NORMALIZATION		LAST LAYER	HIDDEN LAYER
4 Layers	F[N]	0.608	0.562
	μ[Nm]	5.14e-04	4.80e-04
5 Layers	F[N]	0.597	0.538
	μ[Nm]	5.10e-04	4.57e-04
6 Layers	F[N]	0.587	0.541
	μ[Nm]	5.03e-04	4.61e-04
7 Layers	F[N]	0.561	0.540
	μ[Nm]	4.82e-04	4.60e-04
8 Layers	F[N]	0.584	0.542
	μ[Nm]	4.97e-04	4.60e-04

Table 4 Neural Network with different number of layers and batch normalization applied on last layer or hidden layer

In general regularization has great results on the network. Both Dropout and Batch Normalization and L2 performs well, but if they are combined together regularization turns out to be too strong, causing a degradation of the performance. Performance gets worse when L2 regularization strength becomes too high (> 1), as expected. Finally, Dropout performance better when the neural network becomes deeper. The main problem of Dropout and Batch normalization is that make the training slower.

Chapter 7 Conclusions

In this thesis, we considered the problem of the calibration of a tactile sensor. Different algorithms for calibration have been adopted, each of them with its pros and cons. The aim of these few lines is to provide a quick overview on different approaches used.

- **K-NN** shows good performances in terms of accuracy and high rate, indeed for a prediction it needs $0.001s$ (for a small value of k). K-NN has few tunable parameters (distance metric, number of neighbours) and it is easy and fast to train. However, KNN is difficult to interpret and sensitive to outliers. Furthermore, if data are noisy number of neighbours must be increased, causing a degradation of the prediction rate of the algorithm.
- **Random Forest** using regression trees shows very good performances in terms of accuracy, a rate medium ($\approx 42\,Hz$) and medium number of relevant parameters to tune (depth, number of estimators, number of features, criterion). Furthermore, it is intrinsically robust to noisy data because it averages outputs of all the trees. However, Random Forest is difficult to interpret and overfitting can easily occur, but can be limited.
- **AdaBoost** using regression trees shows performance similar to Random Forest in terms of accuracy, a low rate ($\approx 21\,Hz$) and small number of relevant parameters to tune(learning rate and loss). However, it is not very robust to noisy data and outliers.
- **Gaussian process** for regression shows really good performance in terms of accuracy but very low rate of prediction ($\approx 4\,Hz$). Main strength is the ability to provide uncertainty about a prediction. They need an accurate calibration of the kernel and it is not straightforward find a good kernel. Tuning parameters are only kernel parameters (usually length scale and amplitude). Training for big data needs some modifications as previously

presented. Gaussian processes can deal with noisy data, indeed it is possible to incorporate noise in the covariance matrix.

- **Neural Networks** show best performances in terms of accuracy, medium-high rate ($\approx 53\,Hz$). They need a lot of time for training (depending on the chosen architecture) and have an high number of parameters (layers, neurons, different types of regularization ...). Neural networks are robust against outliers and noisy data, but above all, they are a universal approximator.

So, to recap:

	Accuracy	Rate	Training Time	Number of Parameters	Robustness
KNN	Medium	Very High	Low	Low	Low
Random Forest	Medium	Medium	Medium	medium	High
AdaBoost	Medium	Low	Medium	Low	Low
Gaussian Processes	High	Very Low	High	Low	High
Neural Networks	High	Medium High	Very High	Very High	Very High

Table 5 Recap of different algorithms for calibration

Definitely, Adaboost and KNN are not suitable for the calibration purpose, due to their low robustness against noisy data and outliers. Gaussian Process are really slow to predict values and are not suitable for a prediction with a rate of 500 Hz. Random Forest, among Machine Learning algorithms, is the best choice for its properties. However, Neural Networks show best results and robustness so they are the best choice.

7.1 Future Improvements

There are some points that can be improved:

1. Different dimensionality reduction, nonlinear for example.
2. Augmentation of dataset with artificial noisy data or adding new samples to the existing dataset.
3. Parallel implementation of Random Forest.
4. Variants of Random Forest like Boosting trees.
5. Parallel implementation of Bagging of Gaussian Processes.
6. Faster variants of Gaussian Process (Sparse, MVM, KD-Tree Approximation ...).
7. Bayesian optimization of hyperparameters of the Neural Networks.

REFERENCES

[1] Andrej Karpathy, *Convolutional Neural Networks for Visual Recognition.* URL http://cs231n.github.io/

[2] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani, Giuseppe Oriolo. *Robotics: Modelling, Planning and Control*, 2010.

[3] Carl Edward Rasmussen, Christopher K. I. Williams. *Gaussian Processes for Machine Learning*, 2006.

[4] Christopher M Bishop. *Pattern recognition and machine learning*, 2006.

[5] David Pardoe, Peter Stone. Boosting for regression transfer, *Proceedings of the 27th International Conference on ICML*, pages 863-870, 2014.

[6] G. De Maria, C. Natale, S. Pirozzi. Force/tactile sensor for robotic applications, *Sensors and Actuators A: Physical 175*, pages 60–72, 2012.

[7] George documentation. URL https://george.readthedocs.io/en/latest/

[8] Harris Drucker, *Improving Regressors using Boosting Techniques*, 1997.

[9] Hyun Seok Oh, Gitae Kang, Uikyum Kim, Joon Kyue Seo, Won Suk You, Hyouk Ryeol Choi. Force/torque sensor calibration method by using deep-learning in *14th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, 2017.

[10] John Mingers. *Machine Learning 4*, pages 227-243, 1989.

[11] Leo Breiman. *Machine Learning 45*, pages 5–32, 2001.

[12] Raul Rojas, *Neural Networks: A Systematic Introduction,* 2013.

[13] Scikit-learn documentation, *Nearest Neighbours.* URL https://scikit-learn.org/stable/modules/neighbors.html

[14] Sebastian Ruder, *An overview of gradient descent optimization algorithms.* URL http://ruder.io/optimizing-gradient-descent/

[15] Yoav Freund, Robert E. Schapire. A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting, *Journal of Computer and System Sciences 55*, pages 119-139, 1997.

RINGRAZIAMENTI

A conclusione di questo lavoro di tesi, è doveroso dedicare i miei più sentiti ringraziamenti alle persone che ho avuto modo di conoscere in questo importante periodo della mia vita e che mi hanno aiutato a crescere sia dal punto di vista professionale che umano. È difficile in poche righe ricordare tutte le persone che, a vario titolo, hanno contribuito a rendere migliore questo periodo.

Un ringraziamento al mio relatore, il prof. Giuseppe De Maria, per la guida e per gli insegnamenti profusi in questi anni universitari.

Un ringraziamento sentito per la guida competente e solerte va al Prof. Ciro Natale. Un ringraziamento per avermi guidato nell'esperienza in Germania, per avermi fornito tanti consigli utili che mi hanno migliorato umanamente e professionalmente.

Un ringraziamento doveroso ai miei genitori per avermi supportato in qualsiasi modo e incoraggiato nel lungo, e tortuoso, percorso universitario. Devo a voi i miei valori, le mie esperienze e tutto ciò che ora sono. Non esistono parole in grado di descrivere la mia gratitudine e il mio affetto, spesso celati, nei vostri confronti.

Un ringraziamento ai miei cugini, ai miei zii, alle mie nonne per avermi regalato sempre affetto, attenzioni e tante risate durante tutti questi anni. Grazie per avermi fatto sentire membro di un'unica grande famiglia.

Un ringraziamento ai miei nonni, nonostante la vostra assenza fisica, il vostro ricordo nel mio cuore è e sarà sempre un tesoro da custodire gelosamente.

Un ringraziamento ai miei amici per le tante risate insieme, le infinite discussioni calcistiche e i tanti momenti passati insieme.

Un ringraziamento ai miei colleghi universitari, in particolare ai ragazzi del laboratorio di Robotica, per avermi sempre aiutato nelle difficoltà, per aver reso più leggere e divertenti le tante ore passate insieme.