POLITECNICO DI TORINO

Corso di Laurea Magistrale

in Ingegneria Elettrica

Tesi di Laurea Magistrale

Safe Torque Estimation Through Neural Network

Relatore
Gianmario Pellegrino

Candidato
Davide Garibaldi

Anno Accademico 2019/2020

Acknowledgements

To my family, who supported me continuously despite all the problems and all the odds. I will never be grateful enough for all the sacrifices they have done to allow me to be here today. To my mother, my father, my brother and my grandma who have always done everything possible for me to achieve this goal, and this is what matters the most.

To Luca and Oskar, my supervisor and my examiner at KTH, who helped me in carrying out my project, a long work I'm definitely proud of. Thanks to Dr. Pellegrino, my supervisor at Politecnico di Torino, for having accepted me and helped me despite the distance.

To Inmotion, the company I did the thesis for, and Per, my company side supervisor. He has been a great guide during my work and thanks to him I learned many things.

To my friends, who were always here to help me and make me spend a good time, both in Italy and in Sweden.

Particular thanks to Laura and Veronica, who have always been my second family in Italy. They just mean the world to me.

Particular thanks also to Andrea (my university mate), who helped me along this Master's. We shared so many moments of anxiety and/or happiness during the trip that now I feel like he has always been part of my family.

Finally, thanks to the girl who took my hand and guided me along these two years at KTH. Without you everything would have been totally different. Probably I would have not been able to survive in Sweden if you did not offer me a place where to stay when I needed, your continuous and intense support and your love. Thank you Andrea, I hope this two years will be only the beginning of a much longer trip together.

Contents

1 Abstract

Mass electrification of the vehicles is spreading everywhere and new solutions and applications related to the automotive field have to be studied. In particular, a big problem concerning automotive industries is safety, which imposes requirements that must be fulfilled. Considering the control system of a motor, requirements are promptly translated into limitations for the hardware and software, not allowing the use of important measurements for the estimation of the electromagnetic torque produced by the motor. For this reason, it is necessary to investigate the possibility of a new kind of torque estimator.

The purpose of this work is to evaluate whether it is possible to design a new type of safe torque estimator based on a neural network, and implement in the software of the inverter ACH6530 produced by Inmotion. As the name suggests, the neural network can use as inputs only measurements considered safe according to ISO standards and must return an estimation which is able to fulfill all the safety requirements of level ASIL C defined by the document ISO 26262.

Therefore, after a long trial and error process based on different choices related to network structures and parameters, a neural model capable of giving satisfactory results has been designed. Implementation in the system has been carried out after evaluating on a board which neural network structures the system could bear. Finally, the neural network estimator has been tested on the actual motor, giving positive results and showing that this type of application is possible and its accuracy is comparable to the current safe torque estimation implemented in the system.

2 Introduction

Due to the electrification process which is affecting almost every aspect of the human life, the transportation sector is moving towards electrical applications. This generally affects all the different areas related to transportation, such as automotive and railway system for example. Therefore, the development of new technologies in the electrical industrial area is continuously growing, forcing companies like Inmotion to evolve. Inmotion Technologies AB [1] is a Swedish company owned by Zapi Group [2] which produces motor controllers, power converters and auxiliary equipment for the industrial vehicle industry. In particular, Inmotion has multiple costumers within the automotive sector, which is the main area of development of the company.

Since automotive applications are quite a new topic for electrical system suppliers, it is often difficult to design and implement a control system which can perfectly satisfy all the requirements from the costumers. One of the main issues for developers is to carry out a solution which can comply all the safety requirements related to automotive application. Due to the importance of the subject, safety has to be considered, analyzed and verified for each automotive application, considering that it can consist in a bus transporting people, for instance. Standards regarding safety in automotive are given by ISO and other organizations. Issues arise not only because it is usually difficult to translate these definitions into concrete boundaries for the different applications, but also because all the boundaries defined have to be respected to guarantee a safe environment.

When the application is the control of an electrical machine implemented in the powertrain of a vehicle, like in the case of this work, usually safety issues are translated into a quantity to control within given limits. Specifically, the size to consider in this case is the *torque* produced by the electrical motor. The torque is a physical dimension expressed in Nm which can be defined as the rotational equivalent of a linear force that twists an object, forcing its rotation in one direction. Electrical motors are used to produce a torque which can cause the rotation of a load and sustain it, creating movement.

In particular, an induction machine can produce an *electromagnetic torque* which depends on the interaction of two electromagnetic fields at the airgap: the one generated in the stator and the one generated in the rotor. The difference in the angular speed between the two electromagnetic fields is called *slip speed* and it is the base principle which makes the induction machine work. Zero current, which means no electromagnetic field, and/or zero slip speed means zero torque. Equation (1) shows this basic relationship for static working points:

$$T_{em} = \frac{3I_r'^2 R_r'}{\omega_s s} \tag{1}$$

where I_r' is the rotor current considered from the stator side, R_r' is the rotor resistance considered from stator side, ω_s is the expression of the stator electromagnetic field frequency as angular speed and s is the slip speed. When the motor has to be controlled, it is necessary to find a more general equation

which links the currents to the torque for each working point. For this reason, it is necessary to carry out some transformation in order to express the currents according to a new reference system, called $d-q$ plane, which allows simplifications for the vectorial control of the machine. The resulting equation which defines the relationship between torque and currents is the following:

$$T_{em} = \frac{3}{2} p \overline{\psi}_{r,d} \overline{I}_q \tag{2}$$

where p is the number of motor pole pairs, $\overline{\psi}_{r,d}$ is the rotor magnetic flux on the d axis and $\overline{I}_q$ is the q component of the current, both current and flux expressed in $d-q$ plane.

In order to calculate the electromagnetic torque correctly, an estimation of the current $\overline{I}_q$ and an estimation of the rotor flux are needed. The latter can be calculated by implementing a flux estimator which contains the integration of the stator phase voltage over time, plus additional methods. Problems may arise when this calculation has to be carried out in a safe environment. Safety requirements not only cause a direct limitation on the electromagnetic torque, defining the safety area of operation for the application, but they may also limit the measurements available for the calculation of the torque itself. In this case, because of the voltage sensor failure rate in addition to Inmotion control system software and hardware implementation, the phase voltage measurement does not fulfil the requirements necessary to be safe according to ISO standards. Therefore, the magnetic flux can not be used to calculate directly the torque, which has to be estimated from other quantities.

Torque estimation is a crucial element for the correct functioning of Inmotion's converter. Although the control system of the inverter is perfectly able to guarantee the performance requested by the application, it is necessary to supervise the torque generated by the motor in order to assure a correct functioning at any time. The estimation of the electromagnetic torque is involved in the process of supervision and, therefore, its calculation should be carried out with a certain accuracy and it has to respect all the safety requirements.

The torque estimator already implemented inside the system of the inverter is based on a simple equation, but it requires different calculations which can introduce inaccuracies. One of the main problems is the correction to introduce in the estimation of the slip speed and currents due to the variation of temperature in the rotor, quantity which can not be directly measured. It could be thus advantageous to analyze a different estimator, based on quantities which are easier to obtain. A really convenient estimator should use data coming directly from the measurements and reduce the number of calculations which can induce errors or decrease the accuracy of the estimation, like the ones due to temperature dependency.

The purpose of this work is to evaluate whether a safe estimation of the electromagnetic torque produced by the motor controlled by Inmotion's converter can be carried out by a new estimator consisting of a *neural network*. A neural network is an application developed by a branch of the Artificial Intelligence,

called *machine learning*. As the name suggests, machine learning is a field of research that produces algorithms and mathematical models which are supposed to learn how to solve a task after being trained to do it.

The concept behind a neural network is simple: a certain quantity of data is given as input to the algorithm which, through a training process, learns how to solve its task in relation to those inputs. If the network is trained in a correct way, it learns how to solve the problem. This means that, despite the fact it has been trained only over a fixed set of examples, it is able to carry out correct results also when new and different data are given as inputs. This algorithm can therefore generalize, being able to solve in a proper way difficult tasks for new inputs it has never seen before. Due to this amazing property, machine learning algorithms are already widely used for different kinds of purposes, from object and face recognition to classification of massive data. Many studies about learning algorithms have been carried out during these last years, leading to the development of new and powerful algorithms with an incredible versatility and capability, as explained in [3]. For these reasons, neural networks are now used to solve many different kinds of problems and their field of application can only become wider in the future.

In this case, the task of the neural network is to learn how to estimate the electromagnetic torque produced by a motor from available and safe measurements. If the algorithm is able to give a satisfactory estimation using only measures considered safe according to ISO standards, it can be regarded as a good estimator. In order to evaluate the feasibility of this application, the model has first to be designed and trained on available data. Then, it is necessary to implement it within the control system of the inverter and test it on the actual motor. Finally, a comparison between the current estimator and the new implemented model is carried out in order to evaluate the results coming from the neural network.

Chapter 3 will present generally what does safety mean for the studied application and which problems brings with it in the evaluation of the torque. Then, some basic aspects about machine learning will be explained in chapter 4, so that it is possible for the reader to understand the principles and the algorithms on which neural networks are based, presented in chapter 5. How the work has been carried out and which choices have been made to obtain the results is presented in the methodology section, chapter 6. Finally, chapter 7 will present the achieved results and their analysis, while section 9 will give an insight of which future analysis can be done as continuation to this work and improve this type of estimation through neural networks.

3 Safety in automotive

The safety of a system is a requirement that has to be fulfilled in order to guarantee proper operation and the least possible risk for the users or for people which can be affected by the system when it is operating. According to the International Organization of Standardization, *safety* is defined as "absence of unreasonable risk". Since the different types of possible risk are depending on the considered application, it is important to specify that the estimation of the torque with neural network presented in this paper is applied to the control system of a motor for an electric vehicle.

As explained later on, the estimation of the torque is an important component in the system's software of this application because it ensures a good control on the motor of the vehicle if done correctly. On the other hand, a failure of this estimation causes the trip of the power supply of the motor with a consequent loss of control on the torque.

In addition, a fairly good estimation is not enough. In order for the system to be safe, it has to fulfill standard requirements defined by the International Organization of Standardization. The reference document for functional safety regarding automotive applications and road vehicles is ISO 26262 "Road vehicles – Functional safety" [4]. It contains safety standardization for all the phases of an automotive product, from the design to the integration and validation. It also defines functional safety for the entire lifetime cycle of each electrical and electronic automotive equipment. This document is an adaptation of the Functional Safety standard IEC 61508 for Automotive Electric/Electronic Systems.

The meaning of safety applied to automotive field, its evaluation and how this affects the estimation of the torque is explained in this chapter.

3.1 Functional safety and ASIL

Regarding safety in automotive applications, two important definition coming from [4] are very relevant in order to understand how a *component* or *item* can be judged safe or not:

- Hazard: Potential source of harm caused by malfunctioning behaviour of the item.
- Functional Safety: Absence of unreasonable risk due to hazards caused by malfunctioning behaviour of electrical/electronic systems.

If hazards are not considered or controlled by the user of the vehicle, a combinations of vehicle-level hazards can lead to an *hazardous event*. This means that the vehicle is operating in a situation that can possibly lead to an accident. In order to avoid an hazardous event, *safety goals* are assigned to the system. A safety goal is a top-level safety requirement which has the purpose of reducing the eventuality of one or more hazardous events to a tolerable level.

In the automotive field, hazards and safety goals are classified into ASIL levels. According to [4]:

"An Automotive Safety Integrity Level (ASIL) represents an automotive-specific risk-based classification of a safety goal as well as the validation and confirmation measures required by the standard to ensure accomplishment of that goal."

The classification is based on four different levels going from A to D, where for each specific hazard and safety goal, the level A is the least requiring,while the level D has the highest safe request. In other words, in order to prevent a specific hazard, fulfilling the safety level requirement ASIL A requires less risk reduction than fulfilling ASIL B and so on, while ASIL D requires the highest risk reduction.

The first step to define a safety integrity level is to conduct an hazard analysis. This analysis is usually carried out by Vehicle Manufacturers (and System Integrators like Inmotion if necessary) and its goal is to clarify all the possible hazards that can affect the functioning of the vehicle, their consequences and their likelihood. A safety goal or objective is then defined and associated to each possible hazard coming from the analysis. Finally, an ASIL level is assigned for a given hazard and its safety goal according to three factors:

- Rate of occurrence of the hazard.
- Possible consequences of the hazard.
- Possibility of intervention of the user in order to stop/control the hazardous event.

These three factors define how much a hazard can be dangerous for the safety of the people interacting with the vehicle or surrounding it. If a hazard is more dangerous than another one, then the risk of it to happen has to be reduced more, thus it will have assigned a higher level of safety goal.

To further explain these important concepts, some examples related to a road vehicle are given in the following table:

Hazard	Safety Goal	ASIL Level
Motor torque too high	Avoid unintentional acceleration	B
Drive off without driver's request	Avoid unintentional drive off	C
Drive off backwards instead of forward	Ensure that the requested driving direction will be engaged	C
Drive torque increase abruptly	A sudden increase of the motor torque has to be avoided	B

Table 1: Examples of ASIL levels related to safety goals and hazards

For instance, the hazard of having too high torque can lead to the hazardous event of having an unintended acceleration. In order to avoid this, the safety

goal "avoid unintended accelerations" is defined. The associated level ASIL B specifies that the safety goal is fulfilled and the vehicle can be considered safe in relation to this hazard only if the risk of having too high torque is reduced according to the requests defined by the level ASIL B.

On the other hand, the hazard of driving off in the opposite direction has an associated level ASIL C. This is due to the fact that the hazardous event of driving off in the opposite direction in respect of the commanded one is considered more dangerous according to the three factors described before. Therefore, the risk of this happening has to be reduced more and the safety limitations are stricter.

3.2 Vehicle-level and Inverter-level safety goals

Once each safety goal is defined, it is necessary to understand how it is possible to fulfill the safety requirements according to the requested ASIL level. Since the application is an electrical motor for a vehicle, safety goals are defined in order to prevent hazardous events which can cause an accident, injury of the driver and/or injury of all the people surrounding the vehicle. At this point of the analysis, safety objectives are said to be specified at a *vehicle level*. Examples of vehicle level safety goals are the ones written in Table 1.

However, in order to understand how to accomplish these objectives and produce safe equipment, it is necessary to look deeper into the application. What makes the vehicle to move and operate is the torque at the shaft of the motor. Therefore, in order to operate in a safe way a vehicle, it is necessary to control the torque delivered by the electrical motor. Since the control of the torque is executed within the inverter which supplies the motor, it is necessary to translate the safety objectives into *inverter level* goals.

Generally, having too high or too low torque on the shaft in respect to the one requested is the cause of the possible hazards for the vehicle. It is sufficient to think about a vehicle on the road during a normal operation to understand how much a wrong torque supply can be dangerous: too high torque can cause an unintentional and/or uncontrollable acceleration when it is not desired. The consequences of this happening to a vehicle stopped in front of a zebra crossing waiting for the pedestrians to cross can be deadly. On the other side, a lower than necessary supplied torque can be equally risky. For instance, if a vehicle stops accelerating during an overtaking on the highway when other vehicles are behind attempting the same overtaking, the situation can lead to a heavy accident.

All of these unsafe events can be avoided "simply" having the right amount of torque supplied from the motor at any time . Thus, it is possible to translate all the vehicle-level safety goals into one single objective on the inverter level: *prevention of unintended torque*. Whenever the inverter is able to supply to the motor the right amount of power corresponding to the generation of the proper level of torque as requested from the driver, all the safety goals are accomplished.

3.3 Responsibilities for the safety of a vehicle and required ASIL level

Cases like the ones explained in the previous chapter are hazards deriving from a wrong supply of torque in respect to the command torque due to an error in the control system. It is important to mention that companies producer of inverters and control system for electrical motors in automotive applications, like Inmotion, have no responsibility if an hazardous event occurs because of a fault in another element of the vehicle which is not their product (motor and sensors included).

For instance, if the current sensor at the motor breaks and gives a wrong measure as feedback to the control system, the latter will provide the power required according to the measurement received. This is a safety issue for the vehicle, but it is not due to failures in the control system if it is able to provide the requested power. The producer of the sensor and/or the vehicle manufacturer have the responsibility for the possible hazardous events coming from a failure in the sensor. In addition, a perfect power conversion and control is not enough to prevent hazards caused by the driver. If, for instance, the command torque is high enough to be considered unsafe in relation to the operating conditions, the inverter will anyway provide to the motor the required amount of power even if it not safe for the vehicle. The producer of the control system has no responsibility for this issue.

It is also important to consider that the main responsible for the general safety of a vehicle is its manufacturer. Therefore, it is the producer who decides the level of safety that all the items inside the vehicle have to accomplish. Inmotion has to provide a power conversion system which is able to fulfill all the requirements requested from the manufacturer of the vehicle. Thus, the ASIL level to fulfill depends on the requirements Inmotion's costumer. Usually, an ASIL C level is required by almost every manufacturer.

Practically, every hazard of level from A to C imposes a limitation for the system. This limitation specifies how much unintended torque is allowed at the inverter level and how many failures for a certain period of time are allowed in the case the control is not able to provide the proper power. The inverter has to supply power to the motor so that every ASIL C limitation on the torque is accomplished.

3.4 Inmotion's inverter and safety control system

The torque estimation method developed in this work can be applied to any power control system of any motor if needed. Since the project is carried out in Inmotion, it is necessary to have a closer look to the inverter the network is applied on. The device is the high voltage inverter ACH6530 [5], power supply for the motor AVE130 from ZF [6].

Figure 1 shows a simplified block diagram of the parts of the power converter, focusing on the monitoring process: the blue block represent the actual control system, where the voltage to apply to the motor is calculated from feedbacks and

required command torque. How this block works to comply its goal is out of the scope of this work, therefore it will not be explained here. From the safety point of view, the important blocks are in green: they are the blocks which monitor if the inverter is working safely and it is able to comply the ASIL C level of safety. All the orange blocks are items defined outside the converter and therefore Inmotion has no responsibility for their safety requirements. In addition, this block representation does not correspond to reality. All the calculations and tasks for the control system are actually computed by one CPU and supervised by another CPU. There are no different physical places in the inverter where tasks are solved, but this representation is easy and intuitive and therefore it is adopted.

The block affected by this work is the safety block (green) "Torque estimation". Further information about it are given in the next subsection.

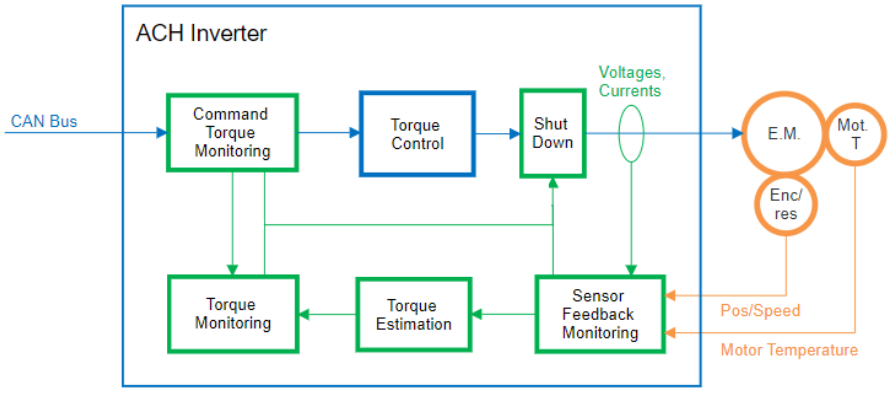

Figure 1: Block diagram of the ACH6530 inverter.

3.4.1 Safe torque estimation and safe torque

It has been explained in the previous sections how important is to have a proper supply of power to the motor, but what tells to the system which is the level of torque provided by the motor in order to control it properly? Most of the vehicle applications do not reckon on torque transducers because they are expensive and often fragile, especially if the vehicle is intended to be off road. Thus, the torque given by the motor is usually estimated and not directly measured.

In addition, it is necessary to specify which torque has to be fed back to the control system. Feeding back the measured torque directly from a transducer would not be a correct option even if there actually was a transducer measuring the torque on the shaft. In fact, the value to compare with the command torque in order to evaluate the safety of the system is the *electromagnetic torque* produced by the motor and not the mechanical torque on the shaft. In any case, the application for which this work is carried out does not have a torque transducer, therefore the torque has to be estimated. This estimation, as well

as all the other monitoring parts of the control system, has to be performed according to the ASIL C level of safety. This imposes limitations on the use of the available measurements on the motor.

The way Inmotion evaluates if a measurement is safe enough to fulfil ASIL C is a protected information, therefore it will not be explained here. The result of this evaluation is that the ACH inverter produced by Inmotion, together with the equipment coming from the costumer, is able to guarantee an ASIL C safety level for the following measurements:

- Phase current measurement, in A_{peak}.
- Rotor speed measurement, in rpm.
- Stator temperature measurement, in °C.
- DC bus voltage measurement, in V.

Furthermore, these measurements allow the software of the inverter to estimated safely the slip speed and the currents in $d-q$ plane I_d and I_q.

Considering what said so far, it is possible now to introduce the concept of *safe torque*. In particular, an estimation of the electromagnetic torque of a motor is considered safe if it is carried out using only data coming from measurements or calculations considered safe according to the required ASIL level. For instance, since the phase current measure complies ASIL C requirement, a torque estimation which relies only on this measurement is considered safe according to ASIL C and lower levels. If more measures are used in the estimation and at least one of them is not able to comply ASIL C, the estimation is not safe according to this level.

The main purpose of this work is to investigate if it is possible to develop a neural network capable of estimating safely the torque in order to substitute the algorithm used today with a more efficient or reliable one. According to what explained previously, the only available inputs for the network to estimate the torque in an ASIL C safe way are:

- Phase current measure.
- Rotor speed measure.
- Stator temperature measure.
- DC bus voltage measure.
- Slip speed estimation coming from the safety module of the system.
- I_d and I_q estimation coming from the safety module of the system, only for PM motors.

How the torque is estimated currently in the converter is a protected information. It is just necessary to specify that it relies on an equation which uses the estimated slip speed and the currents in $d-q$ plane, confirming that it is considered a safe estimation and complies the ASIL C requirements.

3.4.2 Torque monitoring and accomplishment of the ASIL level

As long as the inverter is able to provide the right amount of power to the motor according to which value of torque is required on the shaft, the safety goals required to Inmotion's converters are accomplished and the product can be considered safe for the application. Coming to this point, it is therefore necessary to specify when the power provided by the control system is considered to be "the right amount" in order to meet the ASIL C requirements.

After the safe torque is estimated, independently on how the estimation is done, it is compared to the command torque into the "Torque Monitoring" block. This block is the responsible for the monitoring of the torque: if the estimated torque is outside the safe limits which are based on the command torque, the inverter is tripped. This action is necessary because the estimated safe torque is an estimation of the electromagnetic torque of the motor, which is generated according to the power supplied by the control system. If the estimated torque happens to be outside the limits, according to the safety supervisor the control is giving a power which makes the motor run outside the safe limits. Therefore, tripping is needed.

For this reason, an incorrect estimation of the torque in comparison with the actual torque generated by the motor can lead to different problems:

- If the torque is estimated not to be within the safety limits even if it actually is, the drive stops providing power to the vehicle even if this action is not necessary.

- If the torque is estimated to be within the safety limits but it actually is not, the drive provides power to the vehicle when it should not, increasing the risks and creating safety issues.

Losing the control on the torque of the motor is in any case something which should be avoided and can lead to dangerous situations not only for the user of the vehicle, but also for all the people which can be affected by its operation. Thus, it is necessary a safe estimation able to resemble correctly enough the electromagnetic torque of the motor.

In order to accomplish this task, the estimation should land inside the safe limits, as already said. These limitations are defined by Inmotion's engineers: a risk analysis on the product defines the hazards. After the analysis, safety requirements are translated from goals on the vehicle level to limitations on the generated torque on inverter level. Part of these limitations are function of the command torque and thus they are called *safe functions*. If the estimated torque stays within the boundaries imposed by the safe functions, it can be considered a safe estimation according to ASIL C level. The following list shows all the limits defined by the safe functions that the estimation of electromagnetic torque with neural network has to comply:

- When the command torque is zero, the estimated torque can not overcome, in magnitude, the value of 2% of the maximum torque of the motor.

- When the estimated torque has the opposite sign of the command torque, the estimated torque can not have a value higher than 10% of the maximum torque of the motor for more than 100 ms continuously.
- When the estimated torque has the same sign of the command torque and the command torque is 33% of the maximum torque of the motor or less, the estimated torque can not overcome the value of the command torque plus 10% of the maximum torque of the motor for more than 100 ms continuously.
- When the estimated torque has the same sign of the command torque and the command torque is more than 33% of the maximum torque of the motor, the estimated torque can not overcome the value of the command torque multiplied by a factor 1.3 for more than 100 ms continuously.
- If the estimated torque overcomes the above limits for less than 100 ms, its integral in time can not be higher than the integral in time of the constant maximum torque evaluated during 100 ms.

Apart from the safe functions, the safety requirements define another limitation not dependent on the command torque: the estimated torque can not have sign different from the actual torque of the motor for more than 100 ms continuously.

Furthermore, according to Inmotion internal requirements, the estimated torque can not underestimate in respect to the actual torque of the motor.

Therefore, for the estimation to be reliable and implemented into the software system of the inverter, it is necessary that it fulfils all the limitations here listed, at least. In addition, the estimation should be as accurate as possible: staying within the boundaries is necessary for the application to work, but it does not guarantee a good estimation. In fact, being the maximum torque of the motor equal to 480 Nm, for a command torque of 160 Nm (33% of the maximum torque), a constant error of 48 Nm is acceptable according to safe standards.

The aim of the work is, thus, to understand if it is possible to train a neural network which can estimate the electromagnetic torque complying all the safety requirements and compare its accuracy with the current safe estimation.

4 Introduction to Machine Learning

Machine learning (ML) is a subset of Artificial Intelligence and its aim is to apply mathematical and statistical models or algorithms that the computer can use to solve a specific problem. The strength behind a machine learning algorithm is that it is not directly programmed to solve the specific task it is used for, but it can *learn* from the input data in order to carry out a result for the problem. Therefore Machine Learning can be utilized every time that the solution to a specific problem is too complex for a model/program expressly written from a human in order to solve the required task. In fact, a machine learning algorithm does not achieve results though equations, but it usually relies on patterns found in the input data or inferences in order to learn how to withdraw the right conclusion.

4.1 Definition of a Learning Algorithm

A good definition of what does "learn" mean for a mathematical model is provided by [7]: "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." In other words, if the performance P of the algorithm in respect to a task T improves and this improvement comes with the experience E, the model is able to learn. A huge amount of different tasks, experiences necessary in order to learn and methods to evaluate the performance can be defined. A general definition of these terms is given in the next subsections.

4.1.1 Aim of a Learning Algorithm: the Task, T

As said previously, Machine Learning enables a human to solve tasks which would be too difficult to be solved by a program or model designed from a human specifically for those tasks. The task T for a Machine Learning algorithm is defined as the actual purpose for which the algorithm is used. Thus, it is not the process through which the algorithm find a solution to the problem. Often a task is expressed in an example that the learning model should process. This example is defined as a list of k features or measurements x taken from the event that the algorithm has to process. The task is therefore in this way represented by a vector of characteristics/features of the problem intended to be solved:

$$T = [x_1,\ x_2,\ x_3,\ ...,\ x_k] \tag{3}$$

There are different tasks that a ML model can solve and the most common are:

- Classification: given k different categories/classes to which an input can belong to, the model has to recognize all the inputs from a certain category or to which class an input belongs to. Usually, to solve this problem the

algorithm creates a function $f : \mathbb{R}^n \to \{1, ..., k\}$ and when $y = f(\mathbf{x})$, the input described by the vector $\mathbf{x}$ is assigned to the class y, identified by the model with a number. An application of this task is, for example, object recognition. This means that the model is called to recognize an object (input) among a number of memorized types of objects (categories).

Classification can be performed also with missing inputs. In this case, instead of defining a single function f to classify the inputs, a set of function has to be learned by the algorithm and each of them is used to classify $\mathbf{x}$ with a different subset of its missing inputs. An example of this application can be medical diagnosis.

- Regression: like in Statistics, this task involves the prediction of a numerical value or vector given some inputs. In order to solve this problem, the algorithm usually defines a function $f : \mathbb{R}^n \to \mathbb{R}$ which describes the relationship between inputs and output. Although this process is similar to what expressed in the previous type of task, regression is different from classification because the output of a regression algorithm is only one: a numerical value or a vector of numerical values. Instead, for classification problems more classes can be evaluated at the same time. Regression can have many different applications, like predicting costs, financial tradings and the torque of a motor.

- Transcription: for this task, the model is used to transcribe some information from an unorganized structure of some data into an organized and discrete textual form. An example of the application of this task is speech recognition, where the algorithm is asked to write in sequences of characters and words what it listens to from a recording or a speech.

- Translation: a translation task is when the algorithm is used to translate from a sequence of symbols and characters in a language to the corresponding sequence of symbols and characters in another language.

- Denoising: after corrupting a signal $\mathbf{x}$ into $\tilde{\mathbf{x}}$ with an unknown corruption process, this corrupted signal $\tilde{\mathbf{x}}$ is given as input to the model which have to return as output the clean signal $\mathbf{x}$ or generally the conditional probability distribution $p(\mathbf{x}|\tilde{\mathbf{x}})$.

As seen from this list of tasks, which are important but only a small part of the many possible problems that can be solved with the help of learning algorithms, Machine Learning is a powerful tool which enables humans to solve a great amount of different and difficult tasks.

4.1.2 Evaluation of a Learning Algorithm: the Performance measure, P

The performance measure P describes how well the Machine Learning algorithm can carry out the assigned task. In other words, P evaluate the performance of

the model in respect to the specific task T which is intended to be performed. Therefore, the design of a performance measure P is usually depending on the kind of task requested.

For instance, for a classification problem, a good way of evaluating the performance of the model can be its *accuracy* or its *error rate*: the proportion of examples for which the algorithm carries out respectively the correct and the incorrect output.

Considering instead a regression task, a valuable performance measure can be the *mean squared error* (mse) between a desired output and the actual output of the model, since a regression task tries to predict the output from the inputs.

It is also important to mention that even if it looks straightforward to choose the correct way to evaluate the performance of the model, in practice it is actually difficult to select the option which represents in the best way the desired behavior of the system among different types of possible performance measures. It has to be taken into consideration that the choice of P is relevant for interpreting the performance of the model in respect to the data used to evaluate the performance itself.

In fact, independently on the type of the performed task, it is possible to evaluate the performance of the model in respect to different data sets. Intuitively, it is most useful to evaluate the performance of the algorithm in respect to data that it has never seen before. It is indeed important to evaluate how well a Machine Learning algorithm can perform for sets of data different from the one used to *train* the model in order to understand if the model is able to work properly when implemented and used for its task in an actual application. Thus, its performance is usually evaluated in respect to a set of *test data* which is different and separated from the *training data*.

4.1.3 Different categories of Learning Algorithms: the Experience, E

Usually a Machine Learning algorithm learns while experiencing from a *dataset* during the learning process, also called *training process*. A dataset can be defined as a collection of examples, or data points, that the model has to experience to learn from it. Depending on which kind of experience the algorithm is allowed to have from the dataset, Machine Learning models can be divided into different catergories, the most common of which are:

- Unsupervised Learning algorithms: the algorithm is allowed to experience a dataset which contains some features and it learns properties of the structure of the dataset.

  This approach is widely utilized, for instance, in order to find substructures and useful patterns in the dataset in order to cluster the inputs, which means to divide the dataset in different groups of similar data points. Another application for this type of algorithm is the calculation of the probability distribution and probability density of the entire dataset.

In other words, considering $\mathbf{x}$ a vector of data points corresponding to the dataset, an unsupervised algorithm observes the data points of $\mathbf{x}$ and learns the probability distribution of $\mathbf{x}$ or useful features of it. There is no *teacher* and the algorithm has to learn to solve the task by itself.

- Supervised Learning algorithms: the model is allowed to experience a dataset $\mathbf{x}$ but each data points or sample of $\mathbf{x}$ has an associated target. The targets, collected in the vector $\mathbf{y}$, are provided by a *teacher* who shows the learning algorithm which results it has to achieve. Thus, the algorithm has to learn how to predict $\mathbf{y}$ from $\mathbf{x}$ during the training process.

  Classification and regression models, for instance, are supervised learning algorithms. In the first case, the algorithm has to learn how to predict the class of the input knowing the list of possible categories, while in the second case the model has to predict the output related to the inputs knowing the desired values of the output.

- Reinforcement Learning algorithms: in this case, the algorithm includes feedback loops that allow the interaction between the model and its experience. Hence, the model does not experience only a fixed dataset of examples, but it is able to interacting with its environment during the learning process.

Whichever category a Machine Learning algorithm belongs to, it has to experience a dataset to learn. The dataset is usually expressed as a matrix of examples called *design matrix*. In the design matrix, each column represents a property or feature of the dataset and each row contains a different example of the features. In order for this representation to work, each feature has to be described by a vector of examples and every vector has to have the same length, which means that the system has to have the same number of example for each property.

This is the case of the data experienced in this work, but it is not always possible to have an ordered and systematic dataset. For example, different pictures used for object recognition purpose can have different width and height. This means the the number of pixel through which they can be described can be different for each picture and the vectors representing the features have different length one from each other. In order to solve this problem, various methods exist, but they are out of the purpose of this work and for this reason they will not be explained here.

4.1.4 Fitting data: training, generalization and errors

Machine Learning models are used, as said previously, to solve problems which can not be solved with conventional programs created by the humans. In order to achieve this goal, the model has to be trained.

The training process is a process during which a set of data called *training data* is used to calculate the parameters of the algorithm in order to fit the training samples. During this phase, the performance measure, also called

cost function is minimized in order to try to reduce to zero the error between predicted value (calculated by the model) and desired value.

However, having a good performance compared to the training data is not sufficient to affirm that the model is able to carry out correct solutions in general. In fact, for the model to be implemented in actual applications, it has to be able to perform well on new data that it has never seen before.

This ability of performing well on unobserved data is called *generalization*. Intuitively, the more the model is able to generalize from what it has learnt, the more it will predict good responses from new unseen data, the more it will be useful and applicable to solve the problem it has been trained for.

Therefore, in order to verify how well an algorithm can generalize after being trained, it is tested with another set of data called *test set*, which is collected separately from the training set. From these two types of data, it is possible to define two different types of evaluation on the algorithm:

- Training error: measure of the error of the model compared to the training data. In other words, this is a measure of how well the algorithm can fit the training data.
- Generalization error or Test error: measure of how well the model can generalize. It is defined as the expected value of the error compared to a new set of input data.

A good Machine Learning model should have a low training error, so that it can give a good representation of the data it has been trained on. At the same time, it should also have a low test error to be able to generalize correctly from what it has learnt. In addition, it is important to remember that the algorithm can observe only the training set. Hence, it is necessary to find a way to affect the performance of the model on the test set only through the training set.

Obviously, if training data and test data are collected arbitrarily and independently one from the other, it is not possible to affect the test performance while observing only the training set. For this reason, some assumptions for training and test data have to be considered:

- First, training and test set have to be identically distributed.
- Second, all the examples of each dataset have to be independent from each other.

In order to fulfill the first assumption, typically a larger set is divided sample by sample randomly into the two subsets of training and test data. If the number of samples of the general set is large enough, the two randomly originated subsets will have the same distribution. Thus, while choosing the input data for the algorithm, it is important to consider the amount of examples and their distribution. Usually, it is better to have large and reliable data.

Once the data are divided into the two different subset, it is possible to use them to evaluate the errors of the model. The process starts with the training phase during which training data are sampled in order to calculate parameters

of the learning algorithm. These parameters are called *weights* and they define how a feature x affects the predicted value. In other words, during the training process, the algorithm calculates the correct weights in order to minimize the training error and fit correctly the training data. After this process, the algorithm is evaluated by sampling the test set and the test error is calculated.

Since the test phase is carried out after the training one and its results depend on how well the network has been trained, the expected test error can be equal or larger than the expected training error, but never lower. This latter case would mean that an algorithm can fit better a new (never seen before) set of data rather than the one it was trained on.

Resuming this, it can be said that the lower the training error and the gap between training and test error are, the better the learning algorithm will perform.

4.2 Challenges of a Learning Algorithm: underfitting and overfitting

Underfitting and *overfitting* represent the two direct consequences of having large training and test errors. Whenever the training error due to the training process is not sufficiently low, underfitting occurs. A large training error means that the algorithm is not able to fit enough, or in a proper way, the training data. This is defined as underfitting condition.

On the contrary when the training error is low enough so that the algorithm can fit well the training data, but the test error is large, overfitting occurs. This condition means that, though the algorithm can properly represent the data it was trained on, it does not have the ability of generalizing from what it has learnt.

Overfitting can be explained in an intuitive way, for instance, thinking about a student (algorithm) who learns by heart (overfits) how to solve a mathematical problem (training data) instead of understanding how to solve it. As long as the problem is always the same, the student does not have any issue in solving it. When some characteristics or properties of the problem change, the student is not able to carry out the right solution anymore because he learned by heart how to solve the problem instead of understanding how to solve it. In the same way, if the algorithm overfits the training data, it is able to give correct results when tested with samples included in the training data. However, if it is tested with new samples, it will fail because it has "learnt by heart" the data which belong to the training set.

Since the main purpose of a learning model is to be able to generalize and solve a problem without having a direct knowledge about the problem itself, overfitting behavior has to be absolutely avoided. In the same way, underfitting means that the algorithm is not even able to fit in a correct way the data it has been trained on. Knowing that the test error can not be lower than the training error, it is straightforward to understand that in order to have a proper ability of generalizing, underfitting has to be avoided as well.

Underfitting and overfitting represent the two main challenges to overcome while designing and training a learning algorithm and they affect every type of learning model, from simple linear regression to complex neural networks.

4.2.1 Reducing the training error: capacity and hypothesis space

One way to deal with the problems caused by over- and underfitting is to set the right *capacity* for the algorithm. According to [3], the capacity of a model can be seen as "its ability to fit a wide variety of functions". This parameter is extremely important because it establishes the complexity of the solution the algorithm is allowed to carry out. Generally, setting higher capacity for the model corresponds to allowing more complex solutions.

However, complexity does not necessarily mean more satisfying or correct results. It is actually better to choose the capacity of the algorithm according to the requirements of the problem. A way to achieve this it to modify the *hypothesis space* of the model.

Again according to [3], the *hypothesis space* is "the set of functions that the learning algorithm is allowed to select as being the solution".

By increasing the number of functions available for the algorithm and consequently increasing its capacity, it is most likely possible that the model can resemble better the training samples. In general, increasing the capacity of the algorithm allows more possibilities for the model to fit the data. Therefore, the higher the capacity is, the lower the training error will generally be.

On the other hand, it is not recommended to increase the capacity over what is needed for the task. If this happens, it is possible that the algorithm chooses as solution a function which fits perfectly the training set, but it is too complex in comparison with what is required. This results in a very low training error, but also in a highly poor ability of generalizing because the algorithm is allowed to fit every sample point using complex functions or combinations of them.

In other words, having an insufficient capacity means that the model can not solve complex problems because it is not able to fit the training data (underfitting), while having high capacity may lead to the opposite problem, overfitting the data. An example of this behavior is shown in Fig. 2-4: six random points belonging to a quadratic function are taken as training samples and fit with models of different capacity.

Figure 2 shows the fitting with a polynomial model of the first grade. Intuitively, modelling a quadratic function with a linear one results in a very poor fitting and this corresponds to the condition of underfitting. In fact, none of the training samples is actually fit by the algorithm.

For the model in Figure 3, the capacity is chosen in order to allow the algorithm to have as result a quadratic function. The solution passes through all the training data and is able to predict accurately the other points belonging to the quadratic distribution, showing a good ability of generalizing from the random samples.

Finally, in Figure 4, the model is allowed to fit the six random points of the quadratic distribution with a sixth grade polynomial function. As shown, the solution is able to fit all the samples, but the shape of the curve is not quadratic. This is an example of overestimation: the model is able to fit the training data, but it is not able to generalize because the high capacity allows the result to be uselessly complex.

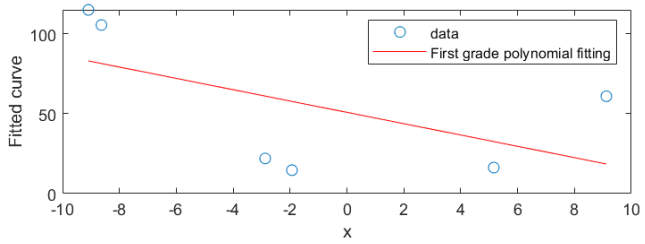

Figure 2: Fitting example with low capacity: underfitting.

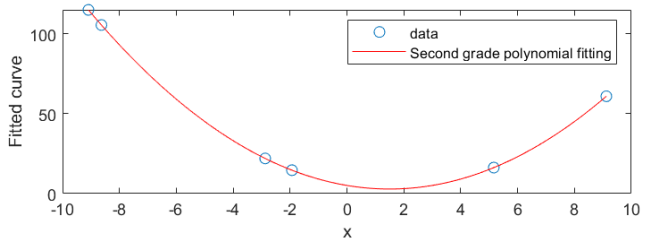

Figure 3: Fitting example with right capacity.

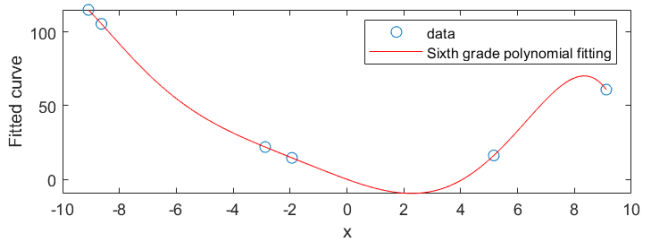

Figure 4: Fitting example with high capacity: overfitting.

The case shown in Figure 4 is a perfect example to understand how complexity can influence the result: it is not necessary to use a sixth grade polynomial function to fit quadratic samples. Increasing the complexity of the solution is not the winning strategy to obtain better results. Instead, it leads to overfitting and a poorer generalization. Therefore, when the capacity or the hypothesis

space of the model is chosen, it is important to remember that simple solutions give higher ability of generalizing.

Although this is true, it is still necessary to choose a function complex enough to result in a low training error in order not to underfit. But how can the optimal capacity be chosen? As previously explained, with the increasing of the complexity, the training error is decreasing, eventually tending asymptotically to zero when the capacity is considered to be infinite. However, the test error is typically decreasing until the capacity reaches its optimal point and then it increases because the algorithm tends carry out too complex solutions. This leads to loss of generalization ability and overfitting.

For lower capacities, training and test errors are both high and the algorithm is in underfitting condition. Moving towards higher capacities, both errors decrease but the gap between them increases. Over a certain capacity, the test error increases, defining a sort of U-shaped curve. The optimal capacity is defined where the training error is sufficiently low and the gap between the two errors is not high enough to overweight the decrease in the training error.

In other words, whenever the negative effect given by the increasing of the gap between generalization and training error is high enough to overweight the benefit given by the decreasing of the training error, the optimal capacity point is crossed and the algorithm enters in overfitting condition.

4.2.2 Reducing the generalization error: regularization and weight decay

As explained so far, it is possible to give a set of preferences in the hypothesis space (choosing how many different functions the model is allowed to fit in the data) in order to adapt the learning algorithm to the characteristics of the problem to solve. In this way the model performs better in solving that problem.

However, this is not the only choice the user has in order to improve the performance of the algorithm. The behavior of the model is, in fact, not only affected by the quantity of the functions allowed, but also by the identity of the chosen functions. For instance, if only linear polynomials are selected as possible choices of solution for the algorithm, the latter will not be able to predict correctly a highly non linear function like the *cosine*. Therefore, not only increasing the quantity of the functions, but also choosing the quality of the functions allowed in the hypothesis space affects the performance of the model.

A general way of controlling the capacity of the algorithm to improve its performance is to express preferences between the allowed functions. Among the possible solutions, all are suitable but one of them is preferred and a different one can be chosen only if it fits the training samples much better than the preferred one. Expressing a strong preference against one function has almost the same effect as removing that function from the hypothesis space, since the learning model will most likely not consider it as solution.

All the ways to express preferences for different solutions are together known as *regularization*. According to [3], "regularization is any modication we make

to a learning algorithm that is intended to reduce its generalization error but not its training error". Intuitively, quantity and quality of the allowed functions are not modified by regularization methods, which express just a preference. Therefore, the training error is not affected, while choosing a preferred function according to the requirements of the task can highly improve the generalization error of the algorithm.

Different regularization methods have been studied and applied. The one used in this work, which is also the most common for regression, is called *weight decay*. Weight decay can be introduced in the algorithm by adding to the *cost function* the preference for the *weights* to have small *squared* L^2 *norm*. In order to understand what does this mean, some definitions are now presented or reminded, referring to the simple case of linear regression. The more complex method of weight decay implemented for neural networks is studied later on, but its principle is the same explained here.

- Weight, w: parameter which defines how much a feature affects the prediction. For linear regression, where the prediction $\hat{\boldsymbol{y}}$ is given by:

$$\hat{\boldsymbol{y}} = \boldsymbol{w}^T \boldsymbol{x} \tag{4}$$

  the weights are simply the coefficients of the $\boldsymbol{x}$.

- Cost or loss function: it is the function defining the performance measurement. In the case of linear regression, the most common cost function is the mean squared error, which is used during the training phase to minimize the training error:

$$C(\boldsymbol{w}) = \frac{1}{m} \sum_{i}^{m} \left(\hat{\boldsymbol{y}}^{(train)} - \boldsymbol{y}^{(train)} \right)_i^2 =$$

$$= \frac{1}{m^{(train)}} \parallel \boldsymbol{x}^{(train)} \boldsymbol{w} - \boldsymbol{y}^{(train)} \parallel_2^2 = MSE_{train}. \tag{5}$$

  After the algorithm is trained, the same cost function is used to evaluate how well the algorithm performs on new data during the test phase:

$$C(\boldsymbol{w}) = \frac{1}{m} \sum_{i}^{m} \left(\hat{\boldsymbol{y}}^{(test)} - \boldsymbol{y}^{(test)} \right)_i^2 =$$

$$= \frac{1}{m^{(test)}} \parallel \boldsymbol{x}^{(test)} \boldsymbol{w} - \boldsymbol{y}^{(test)} \parallel_2^2 = MSE_{test}. \tag{6}$$

- Training process: process during which training data are used to calculate the parameters of the algorithm in order to fit the training samples. During this phase, the cost function is minimized in order to try to reduce to zero the error between predicted value $\hat{\boldsymbol{y}}^{(train)}$ and actual $\boldsymbol{y}^{(train)}$. This corresponds to calculate the values of weights $\boldsymbol{w}$ which minimize MSE_{train}.

- Squared L^2 norm: this is the Euclidean norm, defined as:

$$\| \boldsymbol{x} \|_2 \doteq \sqrt{x_1^2 + \cdots + x_n^2} \tag{7}$$

given a vector $\boldsymbol{x} = (x_1 + \cdots + x_n)$.

Having these definitions clarified, it is possible to understand how weight decay can be implemented for linear regression. In order to introduce the preference for the weights to have small Euclidean norm it is necessary to add a term, defined as *regularizer* $\Omega(\boldsymbol{w})$, in the cost function. The new loss function $J(\boldsymbol{w})$ has this form:

$$J(\boldsymbol{w}) = MSE_{train} + \lambda \cdot \Omega() = MSE_{train} + \lambda \cdot \boldsymbol{w}^T \boldsymbol{w}. \tag{8}$$

where λ is a value which express how strong the preference for small L^2 norm weights is. If $\lambda = 0$, no preference is expressed and weights are calculated only minimizing MSE_{train} during the training process. If instead λ is high, weights are forced to be smaller. Minimizing $J(\boldsymbol{w})$ means indeed to choose weights taking into account that both MSE_{train} and ω have to be small. The larger is the value of λ, the stronger is the preference of having small weights.

Therefore, the act of the regularization is given by the regularizer $\lambda \cdot \boldsymbol{w}^T \boldsymbol{w}$. However, this is not the only way to regularize a learning algorithm. For instance, neural networks can be regularized with weight decay for squared L^2 norm or for absolute value L^1 norm. Other methods do not reckon on modifying the cost function, like *early stopping*. These three different ways of regularizing will be analyzed specifically for neural networks in the correspondent chapter, while all the other methods not presented are out of the aim of this work.

4.3 Control on the Learning Algorithm: hyperparameters and validation

Weights are not the only parameters of a learning algorithm. If this was so, after defining the type of model to use (see next section), the user would not have any kind of control on the algorithm because weights are calculated by the model itself during the training process. Actually, a feature that the algorithm can not modify or calculate has already been discussed: λ.

The λ used for the regularization is a parameter which can be set by the user before the training phase, it is not affected or adapted by the model itself and it gives the possibility to control the behavior of the algorithm. All the parameters which respect these statements are called *hyperparameters*. Another hyperparameter discussed so far, even if indirectly, is the degree of the polynomial of the example in figures (2), (3) and (4) since it modifies the capacity of the model.

Usually a feature is chosen to be a hyperparameter because it is not convenient to let the algorithm adapt it during the training. For instance, if the model could calculate a capacity hyperparameter, it would choose it in order to

have the highest possible capacity, because higher capacity means lower training error and better fitting. But this, as already explained, would surely bring to overfitting.

However, the choice of the values of the hyperparameters for a learning algorithm highly depends on the problem the model has to be trained on and it is not easy for the user to set them correctly.

This issue can be solved by letting the model adapt the hyperparameters using a *validation set*. This set of data is used to estimate the generalization error during the training process so that the model can adjust the hyperparameters accordingly. For this to work, the validation set can not be observed during the training phase (during the actual training, while the algorithm is calculating the weights) and it can not contain any sample of the test set. The reason behind this is easy to understand: the test set has to be used only for testing the algorithm, it is not possible to use data from the test set to adapt parameters of the model, otherwise the generalization ability can not be evaluated.

For these reasons, the validation set is usually extracted from the training data: among all the samples belonging to this set, a subset of usually 75-80 % of the whole data is used to train the algorithm, while the remaining part is not observed from the model and forms the subset of the validation set.

It is necessary now to clarify why it has been said that the validation set can not be observed during the training process but it is used during the same process to estimate the generalization error. Recalling the definition of training phase given in section 4.1.4, it is possible to consider the calculation of the hyperparameters within the "calculation of the parameters in order to fit the training sample". The training phase can be divided according to this consideration into two different processes:

- An actual training phase, when the weights are calculated in order to fit the training samples.
- A validation phase, when the algorithm is tested with the validation set in order to estimate the generalization/validation error and adapt the hyperparameters.

After the hyperparameters are modified according to the validation error, the actual training phase is repeated to calculate the new weights and the validation set is then tested again. This process continues until both hyperparameters and parameters are optimized to fit the training samples. This defines the whole training process if validation is used and further explanation are given in the respective chapter for training process applied to neural networks.

However, the practice of dividing the training data into two disjointed set can represent a problem when the total data set is too small. Having a small validation set, or a small test set in general, leads to statistical uncertainties on the validation/test error. If the estimation of the generalization error is not correct, the hyperparameters can be chosen wrongly by the algorithm and the training process may not be able to succeed. For this reason, all the samples of the training set can be used as validation data if *cross-validation* is applied.

Cross-validation methods are techniques which allow the algorithm to be validated with all the samples by choosing as validation data new random subsets of the whole dataset at each iteration or every n iterations of the training process. There are multiple and different ways to do this and the methodology used in this work is explained in the respective chapter for training process applied to neural networks.

4.4 Evaluation of a Learning Algorithm: maximum likelihood and cross-entropy

It has been explained that during the training process the algorithm can estimate the generalization error in order to adapt the hyperparameters. But how is it this error estimated? In addition, in order to calculate the correct weights during the training phase, how can the model evaluate how much such parameters are wrong at any iteration? The answer, as already presented in the example related to linear regression in section 4.2.2, is the cost function.

In case of regression, the cost function is usually the mean squared error, but for different learning algorithm it is necessary to understand the general principle behind this kind of evaluation so that the right cost function can be chosen according to the required task.

This general criterion is called *maximum likelihood principle* and it is based on statistics and probabilistic theory. Considering n random samples $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)} \cdot \cdot, \boldsymbol{x}^{(n)}\} = X$ taken from an unknown population p, a parametric family of probability distribution $p_{model}(X; \boldsymbol{\theta}) | \boldsymbol{\theta} \in \boldsymbol{\Theta}$ with $\boldsymbol{\theta}$ a vector of parameters which indices the distributions within the family in the Euclidean space $\boldsymbol{\Theta}$, the goal of maximum likelihood estimation is to estimate the parameter values $\hat{\boldsymbol{\theta}}$ in order to define the distribution which has most probably generated the samples X.

This can be done by using the *maximum likelihood function* or *estimator*:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \boldsymbol{\Theta}}{\arg \max} \; p_{model}(X; \boldsymbol{\theta}) \tag{9}$$

where arg max indicates the argument of the maximum of the function $p_{model}(X; \boldsymbol{\theta})$. Since maximizing the probability function leads to a great amount of products between probability values, it is more useful to apply the function natural logarithm to the likelihood, so that all the products become sums without altering the result for the parameters. This gives as likelihood estimator:

$$\hat{\boldsymbol{\theta}} = \underset{\boldsymbol{\theta} \in \boldsymbol{\Theta}}{\arg \max} \; E_{\boldsymbol{x} \sim \hat{p}} \; \ln \left(p_{dist}(\boldsymbol{x}; \boldsymbol{\theta}) \right) \tag{10}$$

where $E_{\boldsymbol{x} \sim \hat{p}}$ is the expected value of $\boldsymbol{x}$ with respect to the empirical distribution $\hat{p}$ defined by the samples X. This function is called *log-likelihood estimator* and maximizing it corresponds to minimize the term $-E_{\boldsymbol{x} \sim \hat{p}} \left[\ln \left(p_{model}(\boldsymbol{x}) \right) \right]$.

This last term is known as *cross-entropy* and it measures how much is the difference between two different distributions over the same samples. Considering, for instance, the same sample set X and the two distributions p and p_{model} defined before. Assuming that p is the *true distribution* of the samples (the

one the learning algorithm is trying to resemble) and p_{model} is an *estimated distribution* of the dataset X (the result of the learning algorithm). The cross entropy function

$$H(p, p_{model}) = -E_{\boldsymbol{x} \sim p_{model}} \left[\ln \left(p_{model}(\boldsymbol{x}) \right) \right] \tag{11}$$

between p and p_{model} evaluates how well the estimated function p_{model} is close to the true distribution p over the same dataset X.

Therefore, maximizing the log-likelihood estimator applied to the algorithm is equivalent to minimize the cross-entropy function between the true distribution of X and the distribution of X estimated by the model. This means to calculate the parameters of the resulting distribution of the model so that the difference between the results and true values that the results should equalize is minimized.

According to statistic theory, minimizing the mean squared error corresponds somehow to maximize the likelihood estimator. This is not demonstrated because out of the scope of the work, but it is crucial because it affirms that the MSE cost function has an important property which belongs to the log-likelihood estimator: under certain circumstances the function is *consistent*.

Consistency is an important characteristic which assures convergence between two models with the increasing of the number of samples n of the dataset. This property is expressed by the following equation:

$$P\Big(\lim_{n \to \text{inf}} \hat{\boldsymbol{\theta}} \Big) = \boldsymbol{\theta} \tag{12}$$

which states that the probability of $\hat{\boldsymbol{\theta}}$ for infinite samples $\boldsymbol{y}$ is equal to $\boldsymbol{\theta}$. Therefore, using MSE as a cost function, assures that, eventually, the result of the learning algorithm will be perfect if the number of samples is enough. The higher is n, the lower the error between model result and actual value can be. In order for the mean squared error to be a consistent cost function, or in general to have consistency for a maximum likelihood estimator, these two conditions have to be verified:

- The true distribution p must belong to the family of distributions $p_{model}(X; \boldsymbol{\theta})$.
- The true distribution p must correspond to one and only one value of $\boldsymbol{\theta}$.

Under these conditions, using the principle of maximum likelihood to evaluate the generalization error of the algorithm will lead to a satisfactory result for the model. For this reason, almost every learning algorithm is evaluated with the cross entropy function or an equivalent, like mean squared error.

4.5 Optimization of a Learning Algorithm: gradient descent

After explaining how to design a cost function for the learning model by using the maximum likelihood principle, it is necessary to explain how it is possible to use this function to optimize the algorithm so that it results in the best solution.

The process of optimization is usually related to maximizing or minimizing a function $f(\boldsymbol{x})$ by varying $\boldsymbol{x}$ itself. Most of the times, optimization means to minimize $f(\boldsymbol{x})$. When it is instead necessary to maximize the function, it is still possible to do it by minimizing $-f(\boldsymbol{x})$. In the case of a learning algorithm, as explained in the previous section, the aim is to minimize the cost function so that the error between prediction and actual value becomes the least possible.

A good property which can be used to minimize a function $f(x)$ is the *derivative* $f'(x)$. The derivative gives an important information about how the function is varying for a change in x and, therefore, it can be used to evaluate for which value of x the function reaches its minimum.

Knowing that for a positive derivative the function is increasing, it is possible to head towards the minimum of $f(x)$ simply by reducing progressively the value of x while monitoring the value of the derivative. On the other hand, a negative derivative indicates a decreasing function, so the minimum can be reached by increasing x until the derivative reaches zero. As known from calculus theory in fact, a null derivative defines a stationary point, which can be either a maximum, a minimum or a saddle point. By moving x in little steps of the opposite sign of the derivative, it is possible to reach a minimum.

In the case of multiple inputs, which is the likeliest possibility while solving a problem with a learning algorithm, the concept of derivative can be generalized into the concept of *gradient* $\nabla_{\boldsymbol{x}} f(\boldsymbol{x})$. Calculus theory shows that the gradient of a function $f(\boldsymbol{x})$ points towards the direction of maximum increasing for the function, while negative gradient points directly "downhill", where $f(\boldsymbol{x})$ is decreasing the most. Therefore, exactly like for the simple derivative, varying step by step $\boldsymbol{x}$ in the direction of the negative gradient will lead to the minimum of the function.

Each step can be written mathematically as:

$$\boldsymbol{x}' = \boldsymbol{x} - \epsilon \nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \tag{13}$$

where $\boldsymbol{x}'$ is the new value of $\boldsymbol{x}$ after the step and ϵ is a parameter known as *learning rate*, which defines how large the step is. Usually, the learning rate is set to a small non zero value. Many iterations of this step will vary $\boldsymbol{x}$ so that the value of the function moves "downhill" towards its minimum. For this reason, this gradient-based process of minimizing the cost function is called *gradient descent* or *steepest descent*.

Gradient descend method is the most used process to train learning algorithms. Its application to a learning model is straightforward: considering the general $f(\boldsymbol{x})$ as specifically $J(\boldsymbol{\theta})$, it is possible to obtain:

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) \tag{14}$$

where $J(\boldsymbol{\theta})$ is a general cost function of the model based on cross-entropy:

$$C(\boldsymbol{\theta}) = E_{\boldsymbol{x} \sim p_{model}} L(\boldsymbol{x}, y, \boldsymbol{\theta}) \tag{15}$$

with $L(\boldsymbol{x}, y, \boldsymbol{\theta}) = -\ln\big(p(y \mid \boldsymbol{x}; \boldsymbol{\theta})\big)$.

Over n different samples for the training data, equation (15) can be written as:

$$C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}) \tag{16}$$

so that the gradient is calculated as:

$$\nabla_{\boldsymbol{\theta}} C(\boldsymbol{\theta}) = \frac{1}{n}\sum_{i=1}^{n} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \tag{17}$$

Since the calculation of the gradient has to be repeated for each sample of the training data at every iteration, this approach may be computationally too expensive, especially for training set with a huge amount of data. Usually, since the cost function as expressed in equation (15) is the calculation of an expected value, it is possible to approximate this calculation over a group of samples instead of calculating the gradient for each example. Under the constraint that the group, called *minibatch*, is a group of n' points uniformly drawn from the original training set, it is possible to estimate the gradient with the minibatch instead of calculating it precisely with the whole set of data. This leads to:

$$\boldsymbol{\theta}' = \boldsymbol{\theta} - \epsilon \boldsymbol{g} \tag{18}$$

with

$$\boldsymbol{g} = \frac{1}{n'}\sum_{i=1}^{n'} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}^{(i)}, y^{(i)}, \boldsymbol{\theta}). \tag{19}$$

If the distribution of the minibatch resembles well the distribution of the total training set, the result will be much less computationally expensive but yet the process will be successful.

However, some clarification have to be made. The process, as described so far, is able to converge under some assumption made on the cost function. First, the cost function has to be differentiable, otherwise it is impossible to calculate the gradient. Secondarily, if the gradient is Lipschitz continuous with constant $L > 0$ [8] and the function is convex, then convergence is guaranteed. Intuitively, a purely convex function has only a minimum which is absolute. The gradient descent algorithm will then tend to reach that minimum of the cost function.

In the case a function is not convex, applying gradient descent may be unsuccessful for multiple reasons: first, convergence in a reasonable time is not guaranteed. Second, even if convergence is reached in a useful time, the result may not be accurate enough because the convergence can lead to a local minimum which gives a poor accuracy in respect with the result the user wants from the model. In addition, the solution is reached when the gradient of the function becomes zero and this can happen also for a saddle point in which the algorithm can be "trapped".

One more consideration is that, since the method will converge to the closest local minimum, gradient descent is sensitive to the values of the initial parameters. Initial values define to which minimum the process will converge to, thus

it is good practice to set them all to random small non zero values. The choice of initial parameters is in this case crucial.

For all these reasons, gradient descent has been considered slow and unreliable, but if implemented with other optimization methods, it can be fast and accurate enough to obtain a correct solution for the model. In general, it is a process which works well with the algorithm used nowadays, like neural networks, and it is for this reason that it will be used in this work.

4.6 Different types of Learning Algorithms

It may be interesting to conclude the section on introduction of machine learning with some examples of different basic types of learning algorithm and their properties, depending on which category they belong to. None of these model will be explained in details, since the aim of the work is to design a neural network. The characteristics of the latter are explained in the next chapter.

4.6.1 Unsupervised Learning Algorithms

Since unsupervised learning algorithms are used to learn features and structure of a dataset in order to extract information, they are often used to estimate properties of the samples, like density, to denoise some data from a distribution or to cluster the dataset into groups of similar samples. All of these tasks are usually achieved by trying to represent the data in a simpler way without losing much information.

Two of the most common methods to simplify the representation of a dataset are *lower-dimensional representation* and *sparse representation*. The first one refers to concentrate as much information as possible about the dataset $\boldsymbol{x}$ in a smaller representation, while the second one consists in representing $\boldsymbol{x}$ with a system which has almost all the entries equal to zero for every feature. For the second method, the system usually requires higher dimensionality than the original dataset $\boldsymbol{x}$, since it is necessary to maintain almost the same information even though the great majority of the entries are zero.

The two different learning algorithms now presented are based on one of these two representations:

- k-means clustering: it divides the training set into k different groups of samples which are close to each other. It works by choosing randomly k *centroids* $\{c_1, c_2, \cdots, c_k\}$ and assigning each observation to the closest centroid. A sample is thus represented by a vector of k elements with the i_{th} one, corresponding to the assigned centroid, equal to 1 and all the others equal to zero. After all the examples are expressed as vectors, each centroid is updated to the mean of all the observations assigned to that centroid. This procedure is then repeated until convergence.

  In order to optimize the final clustering, it is possible to start with different randomly selected sets of centroids and choose as solution the one with lowest total distance of the samples from the centers.

This is an example of application of the concept of sparse representation: each sample is represented by a vector containing $k-1$ zero elements and only one different from zero.

- Principal Component Analysis: it represent a dataset with a lower-dimensional space of independents orthogonal coordinates losing as little precision as possible.

  Supposing that the dataset is formed by q examples for p features, it can be represented by $\{\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)} \cdot \cdot, \boldsymbol{x}^{(q)}\}$ in $\mathbb{R}^p$, where $\boldsymbol{x}$ is the vector of the features and $\boldsymbol{X}$ is the design matrix. The same dataset can be represented in a t dimensional space $\mathbb{R}^t$ by transforming it multiplying the design matrix by a transformation matrix $\boldsymbol{D}^T$ of dimensions n x t. The matrix $\boldsymbol{D}$, according to linear algebra, is chosen from the analysis of the eigenvalues of $\boldsymbol{X}^T \cdot \boldsymbol{X}$. In particular, $\boldsymbol{D}$ is the matrix composed by the eigenvectors corresponding to the t largest eigenvalues of $\boldsymbol{X}^T \cdot \boldsymbol{X}$ in decreasing order.

  The largest eigenvalue defines the *principal component*, which contains the greatest amount of information about $\boldsymbol{x}$, while very small eigenvalue contains very much poor information about the dataset and they can be neglected without having a high loss. Therefore, it is possible to select which eigenvectors to neglect while defining $\boldsymbol{D}$ according to which precision is required and how much it is intended to simplify the representation of the dataset.

4.6.2 Supervised Learning Algorithms

In the case of supervised learning, the algorithm has to learn the association between some inputs $\boldsymbol{x}$ and some outputs $\boldsymbol{y}$ by deducing a relationship between them. Some learning models belonging to this category can be based on probability distributions, others on different methods like the *kernel trick* which is not explained because beyond the scope of this work.

- k-nearest neighbors: algorithm which can be applied to both classification or regression, it is simply based on the evaluation of the k nearest neighbors of the sample that has to be predicted/classified. If it is a classification problem, the most present category between the k closest neighbors of the sample i_{th} is assigned to x_i. If it is a regression problem, the output y for a new test input $\boldsymbol{x}$ is calculated as the average of the y values corresponding to the k nearest neighbors of $\boldsymbol{x}$ in the training data.

- Support Vector Machines: it is a binary classifier which belongs to the *kernel machines* category, since it is based on the kernel trick.

  Basically, it defines a linear boundary between two classes using the two closest observation which belongs one to a class and the other to the other class. These two observation are called *support vectors* and the boundary is chosen as the line which maximize the distance between both samples.

If the samples are not linearly separable, they can be transformed into a space where they are in order to use SVM classifier. For multiclass problems, many SVM can be combined to separate the dataset in multiple categories.

- Decision trees: decision trees can be used for both classification and regression. Like the SVM algorithm, it divides the dataset into different regions, but instead of based on the two closest samples, it is based on a sequence of "1 or 0" questions. Each time a question is asked, a *node* is formed and each node defines two regions in the space of the dataset. An internal node (a subsequent question) divides each region previously created into two sub-regions and so on. Depending on the required precision, a tree can be as complex as possible, but too complex trees may overfit if there is noise in the training data. A possible structure of a tree is shown in the following figure.

5 Introduction to Artificial Neural Networks

Artificial Neural Networks (ANNs), or simply Neural Networks (NNs) are the most important and crucial machine learning. They can belong to both supervised and unsupervised learning category and they are an incredibly powerful tool. In fact, a neural network allows the user to solve basically any kind of problem defined so far, like clustering, classification and regression, and many others more. Nowadays, for instance, some of the uses of the neural networks include face recognition, financial and business prediction and security systems control.

Different architectures and types of ANN are optimal for solving a different kinds of problem. For instance, Self-Organizing Maps suits perfectly for data clustering and belong to unsupervised learning approach. On the contrary, feedforward networks are usually used to solve classification and regression problems, therefore they belong to the supervised learning method, as well as recurrent networks.

In the following sections, the principles of the design, functioning and optimization of a particular type of neural network, called *feedforward neural network*, are explained.

5.1 Basics principles and architecture of a feedforward neural network

In this section, feedforward neural networks are analyzed in their most important features. Most of the general information explained here can be however considered valid also for more complex networks, like recurrent or recursive NN, but the purpose is to explain and understand the case of feedforward networks because it is the kind of algorithm used in this work.

First of all, the goal of this model is to approximate a function f in order to link an input $\boldsymbol{x}$ to an output $\boldsymbol{y}$. This is done by training the network, so that the algorithm can calculate the right set of parameters which give as solution the function that best approximates the relationship between $\boldsymbol{x}$ and $\boldsymbol{y}$.

In addition, this model is called *feedforward* because it only allows the flow of information from the input $\boldsymbol{x}$ to the output $\boldsymbol{y}$: inputs are given to the algorithm, the algorithm performs all the necessary intermediate calculations to learn the function f, the output is given according to the learnt function. There is no feedback, the output of the model is not connected with the model itself and it can not be used to modify or improve f and consequently the solution. When this happens because feedback connections are included in the model, the algorithm is called *recurrent neural network*.

Furthermore, this model is called *neural* because it tries to resemble the structure of a human brain: it is composed by different *layers* made by a vector of elements. These elements are called *neurons* or *units* because exactly like a human neuron, they receive a value from the inputs, they elaborate it with a calculation and they give the result as output. This output is the input of the neurons of the next layer.

Finally, this algorithm is called *network* because it is based on composition of more functions in order to determine f. Each function, called *activation function*, corresponds to a layer of the network and f is formed by the composition of the functions of all the layers. If for instance the network has 3 layers, $f_{(1)}$, $f_{(2)}$ and $f_{(3)}$ are respectively the functions of the first, second and third layer and the output is given by:

$$\boldsymbol{y} = f(\boldsymbol{x}) = f_{(3)}\big(f_{(2)}\big(f_{(1)}\big(\boldsymbol{x}\big)\big)\big) \tag{20}$$

This characteristic gives to feedforward neural networks an incredibly wide field of application and it is the main reason why networks can solve problems that humans can not solve through the application of "simple" equations. On the other side, their complexity requires the user to be very knowledgeable about features, parameters and functioning of the processes involved in the application of a neural network. Otherwise, it is really difficult to drive the solution towards the correct point.

5.1.1 Architecture of a feedforward neural network

A neural network consists, as previously said, in a set of layers which contain a certain number of interconnected units. The first layer of a NN is the *input layer*: each neuron of this layer contain the values of one of the inputs and therefore the number of neurons is equal to the number of different inputs.

The last layer of a network is called *output layer* and its number of neurons depends on the task of the network: if the network, for instance, is used for classification, the number of outputs is the number of predicted classes. For example, if the algorithm is used to distinguish all the images which contain dogs and/or cats among a set of pictures, the number of the outputs may be three: one related to the class "dog", one related to the class "cat" and one for the images which contain both elements belonging to the category "dog" and "cat". Each analyzed image have his set of different outputs and these can be for instance one (true) if the image contains the class correlated to that output, or zero (false) if not. For regression problems on the contrary, since the aim of the model is to estimate the relationships between n independent continuous variables (inputs) and one dependent continuous variable, there are n input neurons and only one output neuron corresponding to that dependent variable.

Between input and output layer, there are *hidden layers* containing *hidden neurons*. The number of both hidden layers and neurons can be chosen freely. They are called hidden because the outputs of the neurons of these layers are not directly observable. The total number of layers of the network defines its *depth*, while the dimension of the hidden layers (i.e. how many neurons they contain) defines the *width* of the model.

As last, it is crucial to understand how layers are connected in order to move the information. Usually, the i_{th} neuron of the j_{th} layer is connected to all the neurons of the previous layer and thus it receives as inputs all the outputs of the neurons of the previous layer. This is valid for each unit of each layer. The result is a structure similar to the one shown in figure 5: all the neurons can

receive as input the outputs of all the neurons of the previous layer and give their output as input to all the neurons of the next layer.

Connections between units are a parameter called *weight* and their values define how strongly two units are connected, playing an important role in the solution carried out by the network. The purpose of the training phase is in fact the adaptation of these parameters, plus some other learnable ones, in order to obtain the most correct result possible.

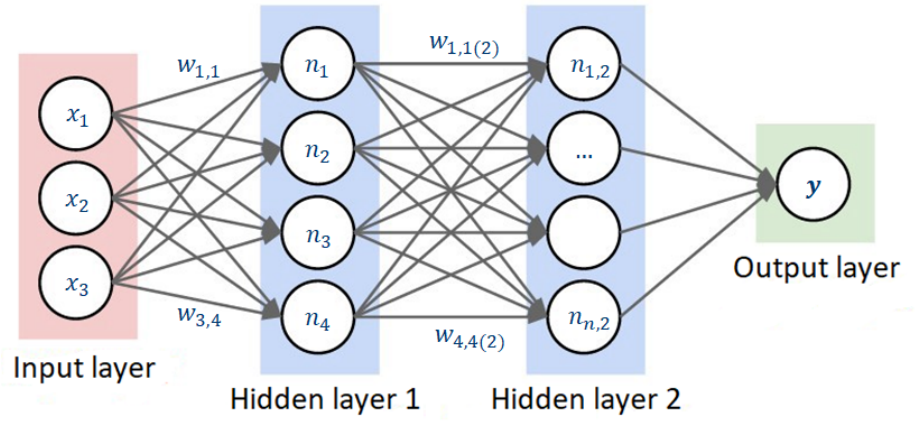

Figure 5: Example of neural network architecture.

5.1.2 Output units and cost function

One of the different types of layers of a network is the output layer. Its purpose is to add a final transformation to the function defined by the hidden units $\boldsymbol{h}(\boldsymbol{x})$ in order to carry out an algorithm capable of solving correctly the given problem. Depending on the task of the network, different types of output units can be used and the choice of the cost function of the model is strictly related to the choice of the output neurons.

In general, the model of the neural network defines a distribution of probability $p(\boldsymbol{y} \mid \boldsymbol{x}; \boldsymbol{\theta})$ and cross-entropy between training data and predicted result is used as cost function. In addition, regularization methods like weight decay are usually combined and the total cost function used to train the network is thus formed by an estimator deriving by maximum likelihood principle and a regularizing term.

Different distributions can be chosen by the model as possible solution and evaluated by using cross-entropy. Each different output unit corresponds to one of them and the task defines which one to use. Some of the most important cases are analyzed in the following subsections.

It is also important to consider that, since the training process of the neural networks is based on a gradient descent method, it is crucial to maintain the gradient of the cost function as large as possible. This property will be explained

in the relative chapter.

5.1.2.1 Linear units

Linear units simply apply a linear transformation. Considering a function $\boldsymbol{h}(\boldsymbol{x})$ as output of the hidden layers, a liner unit gives as result:

$$\hat{\boldsymbol{y}} = \boldsymbol{\omega}^T \boldsymbol{h} + \boldsymbol{b} \tag{21}$$

.

If a linear unit is used to produce an output using a conditional Gaussian distribution $p(\boldsymbol{y} \mid \boldsymbol{x}) = N(\boldsymbol{y}; \hat{\boldsymbol{y}}, \boldsymbol{I})$, the result will be the mean value of the distribution. Therefore, applying maximum likelihood means in this case to minimize the mean squared error.

Since linear functions do not saturate, using linear units during the training process does not give gradient problems and for this reason they can be widely used if combined with optimization methods.

5.1.2.2 Sigmoid units

Whenever the task of the network is to predict the value of a binary variable y, like for example in classification problems with two classes, it is easy and advantageous to use a Bernoulli distribution over y conditioned by $\boldsymbol{x}$. The prediction of this kind of problems is, in fact, just $P(y = 1 \mid \boldsymbol{x})$ and in order for the result to be in the interval $[0, 1]$, a sigmoid unit together with maximum likelihood principle can be used.

A sigmoid unit applies two functions to $\boldsymbol{h}$ in order to return a probability as output: it uses a linear function to calculate $z = \boldsymbol{\omega}^T \boldsymbol{h} + b$ and then it applies the sigmoid function $\sigma(z)$ to transform the result of the linearization into a probability. The sigmoid function is defined by the following equation:

$$\sigma(z) = \frac{1}{1 + \exp(-z)} \tag{22}$$

and it works perfectly if combined with cross-entropy.

Since the log-likelihood cost function can be expressed as

$$C(\boldsymbol{\theta}) = -\ln(P(y \mid \boldsymbol{x})) = -\ln(\sigma(z)), \tag{23}$$

it applies a natural logarithm to the model and the exponential present in the sigmoid function simplifies with it. This simplification allows the sigmoid function to be used efficiently for gradient-based processes, avoiding the function to saturate rapidly, even though it still saturates for really high positive values of x.

The sigmoid function is shown in the following figure.

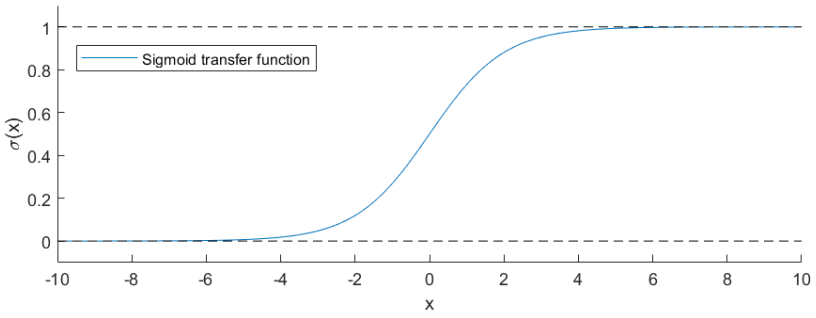

Figure 6: Sigmoid transfer function.

5.1.2.3 Softmax units

The softmax function can be considered as a generalization of the sigmoid function described in the previous paragraph. As such, it can be used every time that the aim of the network is to represent a probability distribution over a discrete variable y with n possible and different values. For instance, the softmax unit can be used if the network is intended to be a classifier of n different classes.

As in the case of the Bernoulli distribution, the prediction should be between zero and one to represent a probability, but in this case the predicted variable is a vector $\hat{\boldsymbol{y}}$ of n elements with $\hat{y}_i = P(y = i \mid \boldsymbol{x})$. Each element $\hat{y}_i$ must be a value in the interval $[0, 1]$ and, for the distribution to be valid, in addition the sum of all $\hat{y}_i$ must be 1.

Exactly like the sigmoid unit, the first function applied to $\boldsymbol{h}$ is a linear function which gives a vector $\boldsymbol{z}$: $\boldsymbol{z} = \boldsymbol{\omega}^T \boldsymbol{h} + \boldsymbol{b}$, then the softmax function is applied. This last is given as definition in the following form:

$$\text{softmax}(\boldsymbol{z})_i = \frac{\exp(z_i)}{\sum_j^n \exp(z_j)}. \tag{24}$$

As well as the previous case of the sigmoid, the exponential nature of this function couples well with the training process of the network if cross-entropy is used. In this case, applying the logarithm to the softmax function leads to:

$$\ln\left(\text{softmax}(\boldsymbol{z})_i\right) = z_i - \ln\left(\sum_j^n \exp(z_j)\right). \tag{25}$$

The term z_i has a direct contribution to the cost function and it cannot saturate since it is linear. For this reason, the training process will continue even if the second term tries to push down $\boldsymbol{z}$. Thanks to this property, softmax units are widely used as output units of neural networks for the related tasks.

5.1.3 Hidden units and activation functions

After discussing about output units and their different usage according to the task the network has to solve, it is necessary to define the core of the network itself: the hidden units.

Nowadays, many studies about the design of hidden units are carried on and there is not a theoretical guideline about which unit to prefer depending on the problem. It is usually impossible to understand or predict which are the best units to use a priori. The only current way to choose correctly the type of hidden neurons is to proceed with a trial and error process based on intuition and test the network on a validation set to see which units give the best result.

A good choice as starting point can be the use of *rectified linear units* or *pure linear units*, but many others are available and need to be tried to compare the results. Some of the most used and their properties are listed in the following paragraphs.

5.1.3.1 Pure linear and rectified linear units

One of the possible hidden units is simply the linear unit as expressed in equation (21). Usually the value of the bias is initialized to a really small value and the activation function applied to the neuron is approximately the line $g(z_i) = z_i$. The advantage of using this function is that for high values of z, the gradient is large and penalizes more the error during the training process.

More often, instead of a linear unit, a *rectified linear unit*, also called *ReLU* is used. The activation function of this type of neuron is given by:

$$g(z) = \max\{0, z_i\}. \tag{26}$$

The difference between the two is that a rectified unit gives zero as output every time that $z < 0$, while the linear one keeps giving as result the value of z_i for any z_i. Therefore, it can be said that the ReLU is *disabled* while z is negative (since it is outputting zero) and it is *active* when z is positive. As in the case of the linear unit, when the rectified linear neuron is active, the derivative will be nonzero and the gradient based training can proceed.

Since it may not be convenient to have the unit disabled every time that z is negative because the neuron is not able to learn while disabled, many generalization of the ReLU have been purposed. One of them is the *leaky ReLU*, which outputs:

$$g(z_i) = \max(0, z_i) + \alpha_i \min(0, z_i). \tag{27}$$

Usually α_i is set to a really small value, so that the derivative is low but not equal to zero, allowing the unit to learn for negative z. Another variation of the ReLU is the *parametric ReLU*, also called $PReLU$, for which the coefficient α_i is a learnable parameter.

5.1.3.2 Maxout units

Maxout units are the extreme generalization of a rectified linear unit. This

activation function divides $\boldsymbol{z}$ into groups G of k different values and output the maximum element of each group:

$$g(\boldsymbol{z})_i = \max_{j \in G^{(i)}} (z_j) \tag{28}$$

where (i) indicate the i_{th} group of k elements the element z_j belongs to.

The advantage given by dividing $\boldsymbol{z}$ in different groups and outputting the maximum for each group is that the transfer function is by extent learned by the model itself. In fact, depending on the maxima, the maxout unit can implement any sort of convex function, including ReLU or PReLU, if groups have enough elements. For this reason, maxout units are often considered a good choice as hidden neurons of a neural network.

On the other side, the implementation of this function during the training process is more complex because each unit has to be parametrized by k weight vectors instead of just by one. For this reason maxout activation functions usually needs more regularization if k is not kept low.

5.1.3.3 Sigmoid and hyperbolic tangent units

The same sigmoid unit $g(z) = \sigma(z)$ studied as output can be used in the hidden layers as hidden unit. A possible variation of this function is the *hyperbolic tangent* activation function, which is related to the sigmoid in the following way:

$$g(z) = \tanh(z) = 2\sigma(2z) - 1. \tag{29}$$

Although sigmoid units are extremely useful for binary classification as outputs, for high values of z, in both positive and negative direction, $\sigma(z)$ saturates giving a very low value of the derivative. This saturation can thus bring some problems during the gradient-based phase of the training, but these units can still be used if optimized in the correct way.

5.1.4 Further architectural considerations

Before starting the analysis of the training process of a neural network, it is important to add some considerations. First, according to [9], a feedforward neural network formed by a linear output layer and at least one hidden layer with any activation function can approximate any measurable function which defines the relationship between any set of inputs and outputs of any dimension if it contains the right number of hidden neurons. This is known as the *universal approximation theorem*. Therefore, theoretically, a feedforward network could solve any problem with just one hidden layer if it is wide enough.

In addition, similarly to the case of the choice of hidden units, there is no a guideline which tells how many layers and/or units a network should have to be able to solve a problem. The choice of the optimal number of layers and neurons depends on the task and it has to be done according to intuition, experience and trial and error procedure. Empirically, it is said that deeper networks (more layers) tend to generalize better for certain tasks like transcription. This is in

general believed true and the trend is to build deeper networks rather than wider ones.

In some cases an external element, like the CPU the network has to be implemented on, can limit the width and/or the depth of the network. This is, for instance, the case of this work. It is indeed important to remember that bigger networks will require more computational effort, both to be trained and to be run after the training.

Finally, according to the universal approximation theorem, given any function, a network big enough to represent that function can be designed. However, this does not guarantee that the network will be able to learn that function even if it is complex enough to do it. This depends in fact on the training process of the model. If the training phase is not optimized in a correct way, the model may never be able to find the parameters correspondent to the desired function. Furthermore, if there is overfitting during the training, the network could directly choose the wrong function as solution.

Therefore, it may not be easy or fast to find the optimal architecture for the neural network in order to solve a problem. Considering that each type of network has its own architectural considerations and the design of the structure of the model also depends on the task, normal practice is just to try and test different configurations until a good architecture is found. Experience can help, but neural networks remain nowadays a sort of "black box" and more studies have to be carried out in order to give more indications regarding the architecture.

5.2 Gradient-based training process

As for many others learning algorithms, the training process of neural networks is based on gradient descent method. Because of this, the total process can be thought as the composition of two main calculation steps: *forward propagation* and *back propagation*. Activation functions determine the results of forward propagation, while the cost function is the starting point for the back propagation calculation. The alternation of the these steps defines the training phase. Both processes and their application will be explained in the next chapters.

Generally, neural networks are used to learn difficult relationships between inputs and outputs, having as result a really complex and nonlinear model. This is a huge problem for gradient descent: most of the cost functions become nonconvex if applied to a nonlinear model and this is, for example, the case of the mean squared error cost function, which would be convex otherwise.

Problems related to nonconvex cost function for gradient descent have been already pointed out in chapter 4.5, but they are recalled here for more clarity:

- Convergence in a reasonable time is not guaranteed: the process will eventually arrive to a solution but it is not possible to forecast how much it will take.
- Convergence to the absolute minimum is not guaranteed: it is instead more probable to converge to a local minimum that can give an accurate enough solution or not.
- Sensitivity to initial parameters value: the initial value of the parameters can define to which minimum the process will lead to.
- Presence of critical points: the process can reach as solution a saddle point which is not a minimum.

In order to cope with all these issues, some optimization strategies have to be applied to the gradient descent approach used during back propagation in the training process of a neural network. These techniques will be described in chapter 5.4, related to optimization of the training.

5.2.1 Forward propagation in a feedforward neural network

As previously explained, the flow of information in a feedforward neural network is heading from the inputs to the outputs. There is, however, an important clarification to mention: this means that the output is not directly used as feedback for the neurons of the model. In other words, no unit of any layer will receive as input the final output of the algorithm at any time during the training process.

There is though a feedback calculation based on gradient descent which allows the model to adapt the parameters according to the cost function and this is known as back propagation. Since this is only an adaptation of the weights of the network according to the output, it is completely different from inputting

the output directly into the units of the network. The model derived from this action would be far more complicated because the output would directly affect itself creating a *state* inside the model. This is the case of *convolutional* and *recurrent networks*, which are not studied here.

Coming back to forward propagation, it is known that the purpose of the training process is to adapt the parameters in order to learn the function $f(\boldsymbol{x})$ which identifies in the best way the relationship between inputs and outputs. But supposing that weights and other learnable parameters are calculated in the previous step of the training process, how does the information travel from the inputs to the outputs?

As explained in chapter 5.1.1, each neuron receives multiple inputs, equal to the number of the units of the previous layer, but it has to output a single value. In addition, each layer has its own activation function $f_{(i)}$, which has to be applied to all its units. Considering this, the calculation of the values of the neurons in a feedforward network is carried out in the following way:

- First, each value entering in the i_{th} neuron of the j_{th} layer is multiplied by the correspondent weight defined by the connections.
- Then, all the input values obtained in this way are summed.
- A constant b_j, called *bias* is added to the sum. This constant is the same for all the neurons in a layer, so it is a layer parameter.
- The total sum obtained is the input of a transfer function which is the activation function of the layer the neuron belongs to.
- The output of the activation function is the output of the neuron.

All these steps can be resumed in the following equation:

$$x_i = f_{(j)}\Big(b_j + \sum_{k=1}^{n} \big(w_k \cdot x_k\big)\Big) \tag{30}$$

where j identifies the layer of the i_{th} neuron, k identifies a neuron of the previous layer, n is the number of the units of the previous layer and x indicates in this case the value of the neurons.

Weights are calculated during the previous step of the training, the value of the neurons are calculated with the criteria just explained starting from the first hidden layer which receives as input the values of the inputs of the network with relatives weighted connections. A "cascade" process is carried out until the neurons of the last hidden layer are calculated. Those are the inputs for the final output layer which, following the same calculation scheme just described, applies its transfer function and gives the result of the algorithm. This result is then evaluated with the cost function in order to adapt again the parameters during the back propagation calculation. This process it repeated during each iteration of the training phase: forward propagation calculates the output of

the network and back propagation adapts the parameters according to the new results until the minimum of the cost function is reached.

The presence of the added bias b_j is important: it can be considered like adding an intercept to a linear equation. It adds a parameter that the network can adapt in order to learn the correct function and it can by itself directly determine if the activation function is triggered or not. Supposing, for example, that a rectified linear activation function is set for a layer of the neural network. If the value of the sum term of equation 30 is small and positive, like 0.2, and the value of the bias is larger and negative, for instance -0.3, the function will not be triggered (because the argument is negative) even though the unit receives inputs from the previous layer. It is thus recommended to set the initial value of the biases to positive small numbers, so that the training process can start.

5.2.2 Back propagation in a feedforward neural network

Even though forward propagation is reduced to basic math operations and the application of some transfer functions, the reverse process used to calculate the parameters of the network is not as easy to analyze nor as fast to implement from the computational point of view. Back propagation, or simply backprop, is a general algorithm which can be used not only for this scope, but also for many other purposes. As it is applied to the neural network's training process in this work, it is implemented together with gradient descent, but it can also power other different methods of upgrading parameters if used for other problems.

Since gradient descent implies the calculation of gradients through the layers of the network, a crucial notion to have in order to understand this process is the *chain rule of calculus*. The chain rule simply states that, considering a scalar case and supposing to have $y = g(x)$ and $z = f(g(x)) = f(y)$:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{31}$$

Generalizing for vector calculations, considering $\boldsymbol{x} \in \mathbb{R}^q$, $\boldsymbol{y} \in \mathbb{R}^p$, g a function from $\mathbb{R}^q$ to $\mathbb{R}^p$, f a function from $\mathbb{R}^p$ to $\mathbb{R}$ and $\boldsymbol{y} = g(\boldsymbol{x})$, $z = f(g(\boldsymbol{x})) = f(\boldsymbol{y})$, the chain rule can be written as:

$$\frac{\partial z}{\partial x_i} = \sum_j^n \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}, \tag{32}$$

which in vectorial notation leads to:

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}\right)^T \nabla_{\boldsymbol{y}} z \tag{33}$$

where $\dfrac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$ is the p x q Jacobian matrix of g. The chain rule can also be extended to calculations with tensors.

Once this property is known, it is easy to understand that it can be used recursively to compute all the gradients from the cost function back to the first

layer of the network. This is basically the main core of the back propagation algorithm. Once all the gradients are calculated, they can be used to update the parameters of the network by applying the gradient descent equation. However, here it comes the first consideration to be aware of.

When the chain rule is implemented in a computer, it can carry out the calculation in two different ways. Supposing to have 3 layers with the same transfer function f, building them up as $x = f(w)$, $y = f(x)$ and $z = f(y)$. The computation of $\dfrac{\partial z}{\partial w}$ can be done in both the following ways:

$$\frac{\partial z}{\partial w} = f'(y) \cdot f'(x) \cdot f'(w) \tag{34}$$

$$\frac{\partial z}{\partial w} = f'(f(f(w))) \cdot f'(f(w)) \cdot f'(w). \tag{35}$$

Equation 34 represent the approach used by backprop: the value of $f(w)$ is calculated only once and stored into the variable x. For the second method, the naive approach, the expression $f(w)$ appears multiple times and it has to be computed each time, without requiring additional memory to store it. Both approaches are valid and leads to the same result, but the one used by backprop has much higher efficiency from the computational point of view, at the cost of additional memory occupied. The naive approach can be used if the system has limited memory and store a large matrix would be a problem.

Now that the logic behind the backprop algorithm has been defined, it is necessary to have a closer look on how to apply it to the neural network's training process. Following Algorithm 6.4 in chapter 6 in [3], it is possible to define the steps to go through in order to apply backprop to the training process:

1. First, after the forward propagation, it is necessary to compute the gradient of the cost function at the output layer and store it in a variable $\boldsymbol{g}$:
$$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} C = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$$

2. Start a ***for cycle*** for k from the network depth l to 1 in which the following actions have to be computed.

3. Considering $\boldsymbol{a}^{(k)}$ the activation function of layer k, back propagate the gradient calculation to $\boldsymbol{a}^{(k)}$ through the chain rule:
$$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} C = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$$
where $\odot$ indicates the element-wise multiplication.

4. Compute the gradients on weights $\boldsymbol{w}^{(k)}$ and bias $b^{(k)}$ of the k_{th} layer. In the really likely case of the presence of regularization with parameter λ and function, include the regularizer $\Omega(\boldsymbol{\theta})$:
$$\nabla_{\boldsymbol{b}^{(k)}} C = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\boldsymbol{\theta})$$
$$\nabla_{\boldsymbol{w}^{(k)}} C = \boldsymbol{g}\ \boldsymbol{h}^{(k-1)} + \lambda \nabla_{\boldsymbol{w}^{(k)}} \Omega(\boldsymbol{\theta})$$

5. Finally, back propagate the gradient to the previous layer $k-1$ activation function $\boldsymbol{h}^{(k-1)}$:
$$\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} C = \boldsymbol{w}^{(k)T} \boldsymbol{g}$$

6. End the ***for cycle***.

This algorithm calculates the gradients of the activation functions of each layer (step 3) starting from the output (step 1) and going back until the first hidden layer. Each gradient computed in this way represents how much each layer's output should change in order to reduce the error on the final output. It is then possible to use these gradients to calculate how much single weights and biases should vary in order to have the reduction in the error (step 4).

Finally, gradient descent can be applied to update all the weights and biases of all layers:
$$\boldsymbol{W}^{(i+1)} = \boldsymbol{W}^{(i)} - \epsilon \nabla_{\boldsymbol{W}^{(i)}} C \tag{36}$$
$$\boldsymbol{b}^{(i+1)} = \boldsymbol{b}^{(i)} - \epsilon \nabla_{\boldsymbol{b}^{(i)}} C \tag{37}$$
where i indicates the current iteration. $\boldsymbol{W}$ and $\boldsymbol{b}$ represent respectively the matrices of all the weights and the vector of all the biases of the current training iteration.

Despite the efficiency of the algorithm, its real-world implementation may be difficult for many factors, like the different types of data it has to handle or memory's bottlenecks. It is though still an efficient way to compute the gradients needed by gradient descent algorithms and for this reason it is used nowadays for the training process of different types of neural networks.

5.3 Regularization of a feedforward neural network

The aim of a learning algorithm, as specified many times, is to be able to be able to generalize from a data set and handle in a correct way new unseen examples. A general way to improve this ability of a model is presented in chapter 4.2.2, where regularization is introduced as "express preferences for the algorithm in order to improve the generalization error". It is now necessary to specify how different methods of regularizing can affect a neural network model.

There are different types of regularization strategies: some of them rely on adding constraints or penalization factors to the cost function, like weight decay. Others present a totally different approach, like *early stopping*. In this chapter, the concept behind some common regularizing methods are presented, paying more attention on weight decay, which is the technique used in this work.

5.3.1 Norm penalties regularizers

A common approach to regularize a neural network is to limit the capacity of model. This principle is the same expressed in chapter 4.2.2, but it is here applied to a feedforward network. As previously said, the limitation on the algorithm can be introduced by adding a norm penalty regularizer $\Omega(\boldsymbol{\theta})$ to the

cost function $C(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})$. The new objective function $J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y})$ has thus the following form:

$$J(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) = C(\boldsymbol{\theta}; \boldsymbol{X}, \boldsymbol{y}) + \alpha\Omega(\boldsymbol{\theta}), \tag{38}$$

where α is a hyperparameter which weights the contribution of Ω to J with respect to the single cost function C. The higher α is, the more the regularization is.

It is possible to choose different norms in order to penalize the size of the model's parameters. The main two are L^2 regularization, which is the case of *Ridge regression* also known as weight decay, and L^1 norm regularization.

In any case, in general it is profitable to regularize only the weight parameter $\boldsymbol{w}$ instead the whole parameter matrix $\boldsymbol{\theta}$. This is due to the presence of biases: they do not need to be regularize because they do not introduce a large variance in the system, being a layer parameter instead of connecting two different layers. Also, it has been studied that regularizing the biases may bring to massive underfitting.

In the next section, L^2 norm regularization will be analyze specifically as application for neural networks, while the following section will present generally the concept of L^1 regularization and its differences with weight decay strategy.

5.3.1.1 L^2 norm regularization: weight decay

Ridge regression applied to neural network training is implemented in the same way as described in chapter 4.2.2: the regularizer to add to the cost function is the term $\Omega(\boldsymbol{\theta}) = \frac{1}{2}||\boldsymbol{w}||_2^2$. As explained previously, the only parameters included in the L^2 norm function are the weights. Therefore, the cost function of the model can be written as:

$$J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \frac{\alpha}{2}\boldsymbol{w}^T\boldsymbol{w} + C(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) \tag{39}$$

and its gradient is:

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}} C(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}). \tag{40}$$

Applying the update given by gradient descent:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} - \epsilon\big(\alpha\boldsymbol{w} + \nabla_{\boldsymbol{w}} C(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})\big) \tag{41}$$

which written in another way is:

$$\boldsymbol{w} \leftarrow (1 - \epsilon\alpha)\boldsymbol{w} - \nabla_{\boldsymbol{w}} C(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}). \tag{42}$$

Equation 42 shows that at each step of the training process, weights are both updated according to the gradient and lowered by a multiplicative factor $\epsilon\alpha$. Which are the consequences of this action over the whole training process?

In order to evaluate this aspect, it is necessary to consider a quadratic approximation of the cross-entropy cost function in the neighborhood of the

weights which minimize the cost function itself. If the cost function is then mean squared error, the approximation is perfect and no error is introduced. The approximation $\hat{J}$ is given by:

$$\hat{J}(\boldsymbol{\theta}) = J(\boldsymbol{w}^*) + \frac{1}{2}(\boldsymbol{w} - \boldsymbol{w}^*)^T \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) \tag{43}$$

with $\boldsymbol{H}$ equal to the Hessian matrix of J with respect to $\boldsymbol{w}$ evaluated in $\boldsymbol{w}^*$. Furthermore, since $\boldsymbol{w}^*$ is the location of the minimum values of J, $\boldsymbol{H}$ is positive semidefinite. The gradient of $\hat{J}$ is:

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*) \tag{44}$$

and the minimum of the function occurs when this equals zero. Adding weight decay brings to an equation similar to 40 and solving for the minimum of $\hat{J}$, considering $\tilde{\boldsymbol{w}}$ as values of $\boldsymbol{w}$ at the minimum:

$$\alpha\tilde{\boldsymbol{w}} + \boldsymbol{H}(\tilde{\boldsymbol{w}} - \boldsymbol{w}^*) = 0 \tag{45}$$

$$(\boldsymbol{H} + \alpha\boldsymbol{I})\tilde{\boldsymbol{w}} = \boldsymbol{H}\boldsymbol{w}^* \tag{46}$$

$$\tilde{\boldsymbol{w}} = (\boldsymbol{H} + \alpha\boldsymbol{I})^{-1}\boldsymbol{H}\boldsymbol{w}^* \tag{47}$$

Equation 47 shows that if the regularizing coefficient α tend to zero, the regularized solution approaches the standard solution $\boldsymbol{w}^*$. In order to evaluate what happens if α is a positive non-zero number, it is necessary to decompose the Hessian matrix $\boldsymbol{H}$. Since it is real and symmetric, $\boldsymbol{H}$ can be split into a diagonal matrix $\boldsymbol{\Lambda}$ with eigenvalue elements λ_i and its orthonormal basis of eigenvectors $\boldsymbol{Q}$ according to equation 48:

$$\boldsymbol{H} = \boldsymbol{Q}\boldsymbol{\Lambda}\boldsymbol{Q}^T. \tag{48}$$

Substituting $\boldsymbol{H}$ with its decomposition in equation 47 and solving for $\tilde{\boldsymbol{w}}$, it is obtained:

$$\tilde{\boldsymbol{w}} = \boldsymbol{Q}(\boldsymbol{\Lambda} + \alpha\boldsymbol{I})^{-1}\boldsymbol{\Lambda}\boldsymbol{Q}^T\boldsymbol{w}^*. \tag{49}$$

Equation 49 shows that the L^2 norm regularization of the weights has a rescaling effect on $\boldsymbol{w}^*$. In particular, the component of $\boldsymbol{w}^*$ corresponding to the i_{th} eigenvector and eigenvalue λ_i of $\boldsymbol{H}$ is rescaled by a factor $\dfrac{\lambda_i}{\lambda_i + \alpha}$ along the direction of the i_{th} eigenvector.

If for the direction of an eigenvector of $\boldsymbol{H}$ the eigenvalue is large, so that $\lambda_i \gg \alpha$, regularization has a small effect on the solution. On the other hand, along directions which have a small eigenvalue, so that $\lambda_i \ll \alpha$, the components of $\boldsymbol{w}^*$ are shrunk and their magnitude approaches zero. Therefore, it is possible to conclude the following:

- Relevant directions for the reduction of the cost function during the training process, which correspond to large eigenvalues by definition, are nearly completely not affected by weight decay regularization.
- Directions which do not contribute a lot in the increase of the gradient, corresponding to small eigenvalues, are shrunk away during the training process by the regularizer and therefore neglected.

5.3.1.2 L^1 norm regularization

Another possible way of penalizing the size of the parameters of the model is to apply L^1 norm regularization. This method is based on the following regularizer:

$$\Omega(\boldsymbol{\theta}) = \| \boldsymbol{w} \|_1 = \sum_i |w_i| \tag{50}$$

which is as the sum of the absolute values of all the parameters, yet again considering only the weights.

Adding this regularizer to the cost function of the model brings to the equation:

$$J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \| \boldsymbol{w} \|_1 + C(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) \tag{51}$$

where the gradient of $J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y})$ is calculated as:

$$\nabla_{\boldsymbol{w}} J(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = \alpha \text{sign}(\boldsymbol{w}) + \nabla_{\boldsymbol{w}} C(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) \tag{52}$$

with $\text{sign}(\boldsymbol{w})$ is the sign of $\boldsymbol{w}$ applied element-wise. It is possible to see from the equation of the gradient that L^1 regularization is different from weight decay: the contribution to the gradient brought by the regularization term is a constant with sign equal to the sign of the correspondent weight it is applied on, while for L^2 regularization it scaled linearly with each weight w_i.

Once again, to analyze the total effect on the training process, it is necessary to consider a quadratic approximation of the cost function in the neighborhood of the solution which minimizes it, but this time there is not a perfect solution since the regularizer is not quadratic. Through a truncated Taylor polynomial expansion of the cost function, the gradient of the approximation is given by:

$$\nabla_{\boldsymbol{w}} \hat{J}(\boldsymbol{w}) = \boldsymbol{H}(\boldsymbol{w} - \boldsymbol{w}^*). \tag{53}$$

Since L^1 does not have perfect solution if a quadratic approximation is applied and the Hessian matrix is full, it is necessary to assume a diagonal $\boldsymbol{H}$. This assumption is valid if, for example, training data has been pre-processed with an algorithm which removes correlations between the different input features, like Principal Component Analysis.

If this assumption holds, the approximation can be written as:

$$\hat{J}(\boldsymbol{w}; \boldsymbol{X}, \boldsymbol{y}) = J(\boldsymbol{w}^*; \boldsymbol{X}, \boldsymbol{y}) + \sum_i \left(\frac{1}{2} H_{i,i} (\boldsymbol{w}_i - \boldsymbol{w}_i^*)^2 + \alpha |w_i| \right) \tag{54}$$

and minimizing this cost function has an analytical solution in:

$$\tilde{w}_i = \text{sign}(w_i^*) \max \left\{ |w_i^*| - \frac{\alpha}{H_{i,i}}, 0 \right\}. \tag{55}$$

Considering w_i^* positive, equation 55 can give two solutions:

- If $w_i^* \leq \dfrac{\alpha}{H_{i,i}}$: in this case, the outcome will simply be zero because L^1 regularization in direction i overcomes the term given by the cost function C and the results of the subtraction between the two is negative.

- If $w_i^* \geq \frac{\alpha}{H_{i,i}}$: in this case, the outcome is not zero since the regularization just shifts w_i^* by a value equal to $\frac{\alpha}{H_{i,i}}$ in direction i.

When instead w_i^* is negative, for the first case L^1 regularization makes the outcome to be less negative, while it will be zero for the second case.

Comparing this result with the one obtained for weight decay, it is possible to affirm that this type of regularization gives a more sparse solution. This occurs because for L^2 norm penalty the solution is given by:

$$\tilde{w}_i = \frac{H_{i,i}}{H_{i,i} + \alpha} w_i^* \tag{56}$$

which is nonzero every time that w_i^* is different from zero. For the case of L^1 regularization, solutions can be zero also for large enough values of α. Therefore, the solution carried out through L^1 norm penalty is surely more sparse then the one derived with weight decay.

This sparsity property is often exploited in feature selection mechanisms, like the one carried out by the model called LASSO (Least Absolute Shrinkage and Selection Operator), in which L^1 regularization is used to reduce to zero some weights corresponding to certain features. This indicates that those features can be safely neglected.

5.3.2 Early stopping

Early stopping is probably the most common regularizing method used for the training of neural networks. This is due to the fact that it is based on a simple yet effective process. As explained in chapter 4.2.1, usually the training error is constantly decreasing to smaller values, while the generalization error presents a U-shaped curved.

This means that the best possible set of parameters are not found at the end of the training, but somewhere at a certain iteration during the process. Early stopping is based on the simple idea of storing a copy of the parameters corresponding to the lowest generalization error in time and return these parameters at the end of the training process. The stored parameters are updated every time that an iteration improves the generalization error, while they remain the same if during a training iteration the error is not lowered. If the generalization error does not improve for a certain number of iterations set by the user, it is possible to assume that the best solution has already been found. Therefore, it is directly possible to stop the training process.

Since the generalization error has to be calculated during the training phase, early stopping needs a validation set which is not used to train the network. If input data have a huge number of samples, this is not a problem. If it is not so, it is possible to train the network with early stopping until the the validation error makes it stop and then retrain the algorithm over the whole dataset only until the iteration corresponding to the best set of parameters found previously.

Early stopping can thus be seen as a regularizer because it treats the number of training iterations as hyperparameter. After the first training in which the lowest generalization error and correspondent parameters are found, the optimal number of iterations is deducted. This is now a fixed value which can be used for a second training. Stopping the process at a set number of iterations is equivalent to limit the capacity of the model because the steps the algorithm has to fit the training data is fixed. This is, for instance, a way to prevent the model from overfitting.

When the model is not retrained for the second time over the whole dataset, the principle of early stopping can still be considered as a regularizer for the same reason. Independently on how long the training will continue, the result will be the one corresponding to the best generalization error. Therefore, even though the model continues its training increasing its capacity, the final algorithm will be limited to the best parameters.

It is true that early stopping has the disadvantage of storing a copy of the best parameters, but this is almost in all cases not a problem, since the cost in term of memory is often negligible and storing a variable does not require much time for the CPU. The total training time is not really affected under this point of view, while it is importantly affected by the advantage of stopping the training some iterations after the best parameters are found. This is an incredible benefit and the amount of time saved during the training session is valuable. In addition, early stopping gives regularization without adding a penalty term to the cost function, avoiding a more complex calculation of the gradient. For all these reasons, early stopping is the most common regularization method used in deep learning community.

5.4 Optimization methods for training a feedforward neural network

The training process of a neural network has been presented so far as a gradient-based optimization problem. A way to proceed is to perform gradient descent applied to the cost function of the algorithm in order to find the best weights and biases. However, this procedure alone may not lead the satisfactory results. Optimization of the training phase of a neural network is the most difficult task in deep learning and it can take from days to months to find the best way to perform it.

This problem deals with trying to find the parameters $\boldsymbol{\theta}$ of a neural network which cause a significant reduction in the cost function $J(\boldsymbol{\theta})$ which includes both evaluation of the performance on the training set $C(\boldsymbol{\theta})$ and regularization term $\Omega(\boldsymbol{\theta})$.

The quantity to reduce is the expected generalization error over a true distribution of data p, but it is not possible to optimize it directly. Under this point of view, this is not a conventional optimization problem for two reasons:

- first, the goal of a normal optimization problem would be to reduce J itself, why in the case of the neural network J is optimized in order to

reduce another quantity, the generalization error.

- second, the cost function reduced is the objective function over a distribution of training data p_{model} and not over the true distribution p. Therefore, the process minimizes the expected loss on the training set, called *empirical risk*, hoping that this will improve the general expected generalization error, also called *risk*.

The method expressed by the second motivation is known as *empirical risk minimization* and it is the base of the training process of a neural network. However, it only relies on the minimization of the cost function and its application is not straightforward. If backprop with gradient descent is applied, the derivative of the objective function has to be defined, otherwise it is impossible to calculate the gradient. Furthermore, if the model has high enough capacity, it may overfit and memorize the training set reducing the empirical risk, but not the risk.

For these reasons, empirical risk minimization is not always feasible and usually more optimization methods are added to the training process. Thus, the optimized quantity is not only the cost function, but something slightly different. In the following sections, first an additional important feature about training optimization will be analyzed. Then, the principal issues and challenges related to the training process are presented. In conclusion, some of the most common and applied optimization methods will be discussed.

5.4.1 Batch and minibatch

As explained previously, the training process of a neural network is most likely based on backpropagation and gradient descent. The quantity used during backprop is the cost function J, the gradient of which has to be calculated. In the case of maximum likelihood estimation problems, $J(\boldsymbol{\theta})$ and its gradient $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ are expressed respectively as:

$$J(\boldsymbol{\theta}) = E_{\boldsymbol{x},y\sim p} \ \ln(p_{model}(\boldsymbol{x}, y; \boldsymbol{\theta})) \tag{57}$$

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = E_{\boldsymbol{x},y\sim p} \ \nabla_{\boldsymbol{\theta}} \ln(p_{model}(\boldsymbol{x}, y; \boldsymbol{\theta})). \tag{58}$$

Computing the gradient of the expected value over the whole training dataset $\boldsymbol{x}$ can be really expensive from the computational point of view, especially if the set contains a huge total number of examples n because the calculation has to be done for each one of them. In order to solve this problem, what may be done in practice is to sample randomly a small amount of examples from the dataset and calculate the average of the expectations only for that small group of samples. This leads to an approximation of the expected value, but the algorithm may converge much faster in terms of total computation.

An algorithm which performs the optimization over the entire training dataset, calculating the exact gradient, is called *batch* or *deterministic* gradient method. The extreme case on the opposite side, which is when the algorithm uses only

a single example at a time to calculate the expectation, is called *stochastic* method.

Most of the optimization models land actually in the middle, sampling a number $1 < i < n$ of random examples. These algorithms are called *minibatch* methods and the size i of the number of samples to consider during the training process depends on different considerations:

- In general, larger minibatches give a more accurate estimation of the gradient.
- If all the examples of a minibatch are computed in parallel in order to give the average of the expectations, the memory of the processor occupied scales with the size of the minibatch.
- For some hardware, like GPUs, the runtime is optimized for sizes which are powers of 2, usually from 32 to 256.
- Generalization error is often best with stochastic methods, but the computational time increases because it takes more steps in order to evaluate the entire training dataset one example by one.

In addition, it is important that the examples for each minibatch are chosen randomly: in order to have a good estimation of the expected value for the entire dataset, the evaluated examples for each minibatch have to be independent and selected randomly from the set. Furthermore, two subsequent calculations of the gradient should be independent one from the other as well. This means that new minibatches should be chosen during the iteration process.

Usually, the entire dataset is divided evenly into a certain number of minibatches which contain fixed samples and the order of the minibatches is shuffled every x iteration during the training process. Even though this method does not correspond to have complete random selection of examples or batches, experimentally it does not show a harmful effect on the calculation of the expected value of the cost function. However, if the order of the dataset is not shuffled, the performance of the algorithm may significantly fall.

5.4.2 Challenges in the optimization process of a neural network

Although optimization is usually reduced to the minimization a function, it is actually a really difficult task to perform. As already explained, if the process has to be applied to a convex function, gradient descent assures the convergence to the absolute minimum in a useful time if the cost function is conditioned in the right way. Neural networks introduce a highly complex model, which will leads most likely to non convexity for the objective function. As consequence, the optimization results more complicated and challenging for many aspects.

5.4.2.1 Local minima

One of the difficulties introduced by the deep non convex model is the presence of local minima which do not correspond to the global minimum of the model.

Any deep algorithm is actually almost guaranteed to introduce a huge number of local minima which can be sometimes nearly infinite. This is due to the fact that the neural network model is not *identifiable*.

Identifiability means that a model is able, with a sufficiently large training set, to identify one and only one set of parameters as correct solution. Neural networks are nonidentifiable models because they present *weight space symmetry*.

Considering a neural network with a hidden layers of j hidden units each one, the following transformations will give the equivalent outputs:

- For each neuron of a hidden layer, change simultaneously sign to both weight in input and weight in output. This gives 2^j possible equivalent transformations for each layer.
- For any pair of neurons of a hidden layer, simultaneously switch both input weights and output weights of the two neurons. This is equivalent to rearrange the neurons of the hidden layer and $j!$ orderings are possible.

Therefore, considering only a single layer, $2^j j!$ equivalent transformations exist. This mean that for a network which has k hidden layers, $\prod_{i=1}^{k} 2^{j_i} j_i!$ sets of parameters will give the same neural network. These two properties together are known as weight space symmetry.

The possibility of having so many different configurations for the same solution introduces a large amount of local minima which can be even noncountable. Despite this great number, the minima introduced because of weight space symmetry are fortunately equivalent in cost function value. Therefore, if they all have low enough cost, they don't represent a problem for the training process.

Problems arise when local minima present high cost and the solution of the algorithm "gets stuck" in one of them. What actually happens when a local minimum is encountered is not predictable and this problem is an active area of research. In general experts tend to say that if the network is sufficiently large, most of the local minima have a low cost function value abd therefore they do not represent a problem during the training process if a solution with low instead of zero error is accepted.

5.4.2.2 Saddle points

A major problem for the training process can be saddle points. For high dimensional models, for gradient equivalent to zero it is highly probable to find a saddle point instead of a local minimum. A saddle point is a point in which the Hessian matrix of the function has both positive and negative eigenvalues, while for a minimum the eigenvalues are only positive. This is translated in a region of points for which the function has convex behavior on one side (points with grater cost than the saddle point) and nonconvex behavior on the other side (points with lower cost than the saddle point).

A really good property of many functions is that their Hessian matrix tends to have positive eigenvalues for regions at low cost. This means that local

minima tend to have low cost, while saddle points can be found at higher cost values.

This has different implications on the training process of an algorithm, depending on which method is used to train the model. For first order methods the situation is not clear, but gradient descent look like able to escape from saddle points. In fact, this algorithm is designed in order to move the solution downhill along the function. Even though the gradient around the saddle point is very small, if the model manages to overcome the critical point updating the parameters, the training process can continue without problems. Empirically, gradient descent is proofed to escape from saddle points in many cases.

However, the situation is different for second order methods, like Newton-Raphson's. This algorithm is in fact designed to solve for a point where the gradient is zero, resulting in getting stuck in a solution with high cost if a saddle point is encountered. It is possible to modify the algorithm so that it can escape from such critical points, but the implementation is difficult. It is mainly for this reason that second order methods are still not highly utilized to train neural networks, while gradient descent remains solid.

5.4.2.3 Incorrect local structures

Even though only critical points have been considered as an issue for the training process so far, it is actually possible to have major complications and perform poorly also when the solution is heading far from local minima or flat regions. This is, for example, the case when the local region of the cost function that the algorithm is running across does not resemble correctly the global structure.

Considering the global structure as the region of the global minimum, if the region the algorithm is working in is not similar to the global structure, the training process can lead to a solution with high cost or head to a correct result but taking an unreasonable amount of time.

In order to understand how this can occur, figure 7 shows a case in which the region of application of the optimizing algorithm, on the right, has a horizontal asymptote to a high cost value, while the global minimum is on the left side of the "mountain". Assuming the solution to be carried out with gradient descent and following the light blue path towards the right, it is possible to see that the problem is not due to any critical point. Instead, this region of the cost function, which does not resemble the shape of the global minimum, will bring the solution towards a horizontal asymptote. Clearly, this is the wrong path to follow because the solution will have a high cost value with really low gradient.

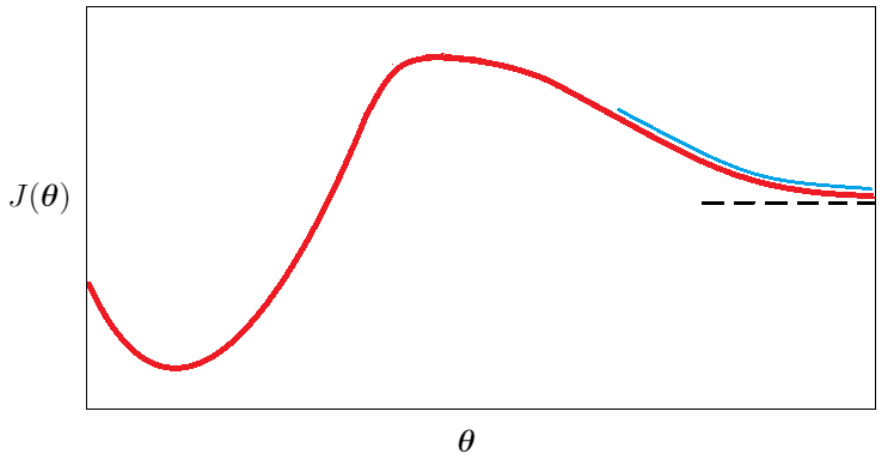

Figure 7: Example of an objective function in which a local region, like the one on the right, does not resemble the shape of the global minimum, on the left. This region can represent a large difficulty for the training process.

For higher-dimensional spaces, gradient descent can be able to circumnavigate the mountain, heading to a correct solution. This often happens, but with a trade off: the computational time increases dramatically. In fact, according to [3], most of the training time is due to the trajectory that the algorithm is following in order to carry out the solution. In the case of figure 7, gradient descent can not give a correct solution because it can not climb the mountain. Imagining now the mountain in 3D, it is possible for gradient descent to reach the global minimum from the same starting point by circumnavigating the mountain itself. However, this can only be done through a much wider trajectory than a direct one, which highly increases the number of steps to perform. Furthermore, if the solution gets close to the "base" of the mountain or to a flat region during the process, the steps are performed with much lower gradient. It is not rare that the increasing in time due to a wider and/or flatter trajectory of the solution makes the algorithm to be useless because unable to perform in a reasonable time.

Intuitively, if the model was initialized to start from the other side of the mountain, the trajectory leading to the minimum would have been direct. Improving the initialization in order to let the algorithm start in a region which is pointing directly to a correct solution is the best way to solve the problem of the wide trajectory due to poor correspondence between the local and the general structure. It is also for this reason that parameters are usually initialized to small random values, but there are no clear rules or guidelines on how to set the initial values in the best possible way depending on the task that has to be solved.

The cases explained in these chapters are only some of the problems which

can be found during the training process. Regardless of which one of them is found or which can cause major issues, some improvements have to be done on the model in order to be able to perform correctly in general.

5.4.3 Optimization algorithms

In the following chapters, some feature to add to the basic gradient descent algorithm are presented. Their aim is to help the gradient descent model to perform in a faster way and/or better in general. Through their application, the training process has more possibilities to be faster and to lead to a correct solution bypassing the problems previously explained.

5.4.3.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is probably the most used optimization algorithm in deep learning. It is a variation of gradient descent in which minibatches of one element are used to calculate the empirical risk.

The most important parameter for this algorithm is the learning rate. So far, the learning rate has been considered constant, but since SGD introduces a noise which does not fade at the minimum by random sampling the training samples, in this case the learning rate needs to be decreased. This is usually done linearly until a certain iteration τ, after which the learning rate is left constant, according to the following equation:

$$\epsilon_k = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau \tag{59}$$

where ϵ_k is the learning rate at iteration k and $\alpha = \dfrac{k}{\tau}$. τ can be set to the number of iterations necessary to make some hundred passes through the entire dataset, while ϵ_τ may be set to around 1% of ϵ_0. This last can be chosen by trial and error, but it is usually recommended to observe plots of the objective function over time to try to understand which is its optimal value. Unfortunately, there is not an actual method to choose the initial learning rate apart from trial and error. When the learning rate is too large, the learning curve has violent oscillations with an great increase in the cost function with consequent instability. When it is instead too small, the training will proceed really slowly (small steps) and the solution may get stuck at a high cost value.

The advantage of using SGD is that, since a single random sample is processed at each time instead of the whole training set, the computational time does not depend on the number of samples of the training data. It is even possible that the algorithm reaches convergence with a certain tolerance before processing the entire dataset if this last is large enough.

5.4.3.2 Momentum

The method of momentum is design to reduce the computational time accelerating the learning process, especially when the gradient is low or noisy. This algorithm introduces a new parameter $\boldsymbol{v}$ which resembles the physical concept of

velocity. This parameter is set to an exponentially decaying average of the past negative gradients and forces the solution to keep moving along that direction.

The updating algorithm for the parameters according to the momentum model is given by:

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left(\frac{1}{m} \sum_{i=1}^{m} L\big(\boldsymbol{f}\big(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}\big), \boldsymbol{y}^{(i)}\big) \right) \tag{60}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}, \tag{61}$$

where α is an hyperparameter of value $\alpha \in [0, 1)$ which defines how fast the contributions of the previous gradients exponentially decay. For higher values for α, the previous gradients affect more the current direction of the solution.

With this type of updating, each step of the training process depends not only on the norm of the gradient, but also on how aligned the sequence of the previous gradients is. If multiple subsequent gradients point at the same direction, the term $\boldsymbol{v}$ accumulates the gradient values becoming more negative. This has as result bigger steps during the updating of the parameters and consequently they are more shrunk along the direction of the current gradient. This accelerates the parameters reduction and the training process. In the case that a great number of the gradients is aligned in the same direction, the velocity eventually reaches a terminal value which gives the following step size:

$$\frac{\epsilon \parallel \boldsymbol{g} \parallel}{1 - \alpha}. \tag{62}$$

According to this equation, when the algorithm reaches the terminal velocity, a parameter α of value 0.9 can speed up ten times the entire training of the neural network. Even if the algorithm does not reach full speed, the improvement introduced by adding momentum to gradient descent is not negligible in comparison to the computational cost added by the calculation of the velocity. Usually the hyperparameter α is set to values of 0.5, 0.9 or 0.99, but it can also be adapted over time like the learning rate, starting as low value and increased later on during the training process.

5.4.3.3 Adaptive learning rate and Adam

Since the choice of the learning rate heavily affects the result of the training of the neural network and its performance, it is very important to choose an optimal value for this parameter. Unfortunately, it is also the most difficult parameter to set. For this reason, one of the main fields of study during the past years has been related to try to let the algorithm adapt the learning rate by itself.

An effective solution based on a simple principle was adopted by [10]: if the partial derivative of the cost function with respect to a parameter maintain the same sign of the previous iteration, it means that the direction of the solution can be followed. Thus, the learning rate can be increased to have a larger step.

If the signs of the partial derivatives are instead different, the learning rate can be lowered.

This is valid only if a deterministic or batch approach is used, because the gradient is calculated over the entire training set, while more considerations have to be taken into account if the process is stochastic or based on minibatches. Many adaptive algorithms for minibatches strategy have been developed relatively recently and one of them is the *Adam* model. Its name comes from "Adaptive momentum" and the algorithm combines the adaptive learning method to the momentum method presented in the previous chapter.

Since, as it will be explained in the next chapter, a batch strategy is adopted in this work, the algorithm of this training method shown as applied to the batch case. Considering two subsequent iterations $k-1$ and k of the training process and c_k the generalization error at iteration k:

if $c_{(k)} < c_{(k-1)}$

$$\epsilon_{(k)} = f_{inc}\epsilon_{(k-1)} \tag{63}$$

elseif $c_{(k)} > f_{threshold}c_{(k-1)}$

$$\epsilon_{(k)} = f_{dec}\epsilon_{(k-1)} \tag{64}$$

elseif $c_{(k)} > c_{(k-1)}$ *and* $c_{(k)} < f_{threshold}c_{(k-1)}$

$$\epsilon_{(k)} = \epsilon_{(k-1)} \tag{65}$$

end

Velocity and parameters are then updated according to the momentum equations:

$$\boldsymbol{v}_{(k)} \leftarrow \alpha\boldsymbol{v}_{(k-1)} - \epsilon_{(k)}\nabla_{\boldsymbol{\theta}}\left(\frac{1}{m}\sum_{i=1}^{m} L\big(\boldsymbol{f}\big(\boldsymbol{x}^{(i)};\boldsymbol{\theta}\big),\boldsymbol{y}^{(i)}\big)\right) \tag{66}$$

$$\boldsymbol{\theta}_{(k)} \leftarrow \boldsymbol{\theta}_{(k-1)} + \boldsymbol{v}_{(k)}. \tag{67}$$

In equations (63) and (64), f_{inc} and f_{dec} are respectively the increasing and the decreasing factor for the learning rate, while $f_{threshold}$ is a threshold value set to decide when to decrease the learning rate. Usually $f_{threshold} = 1.04$, while typical values for f_{dec} and f_{inc} are 0.7 and 1.05 respectively.

Thanks to the adaptation of the learning rate and the application of the momentum, which allow the process to have steps of the optimal size and "inertia" if the solution is heading to the correct direction, this training algorithm is really popular and used to train not only feedforward networks, but also more complicated and deeper neural networks.

However there is a drawback due to the concept of this Adam model: according to equation 63, after a certain number of iterations following the right

trajectory, the learning rate will be large. If the optimal trajectory changes direction, a large step will cause the solution to finish out of it. This brings instability to the solution with a consequent increasing of the error. The learning rate is then decreased according to equation 64 until the steps are small enough to find a new optimal trajectory from the point that the solution has reached. Therefore, this process allows the model to adapt the learning rate for better performance during the optimal trajectory, but intrinsically drives the solution out of it when the learning rate increases too much.

6 Methodology

In this section, all the procedures and the considerations followed to reach the results are explained.

As it can be deducted from the previous chapters, when it comes to neural networks it is not easy to understand how to proceed for unskilled users: some concepts behind the algorithm may be difficult to deal with, setting parameters is usually a long trial and error process and limitations given by the hardware, if not considered, can fail an entire work of months.

A solid knowledge about the algorithms and experience about neural networks are surely helpful in saving time when choosing the type of learning models and its parameters. For this reason, a first recommendation for every beginner in neural networks and machine learning is to study about the topic, both before and during the application. This is not only useful to understand how to set the parameters and start the procedure, but also to be able to realize whether a model can be good or not according to the trend of the training process.

All the knowledge about machine learning necessary to carry out this and other works can be found in *Deep Learning* by Ian Goodfellow, Yoshua Bengio and Aaron Courville [3]. This book is the main source of information used to write almost entirely chapters 4 and 5. In addition, this knowledge is the base for the methodological procedures explained here, through which the results presented in chapter 7 can be achieved.

It is finally worth it to remember that the aim of this work is to verify if it is possible to estimate the electromagnetic torque produced by the motor AVE 130 with a neural network. Inputs to the network are limited for safety reasons, as said in chapter 3.4.1.

6.1 Procedure to train the neural network

As starting point, it is necessary to clarify that the work has been carried out using MATLAB R2019b [11] and its toolbox "Deep Learning Toolbox", which provides a pretty wide set of possibilities for neural networks.

The procedure followed during the working hours in order to be able to train an algorithm which can give satisfactory results for the aim of the work is shown in the flowcharts in figures 8 and 9. First, it is necessary to distinguish between two types of settings for the network, depending on how they affect the learning process. The possible settings to choose defined in figure 8, which are the type of network, the choice of the inputs and the choice of the training algorithm, define the "basic structure" of the network: which network is used, how it is learning and on which data. A change in one of these settings while following the trial and error procedure described in the flowchart would cause the entire procedure to restart from zero. Fortunately, they are the easiest settings to arrange because they are almost entirely defined by the task of the network. If not, just a small amount of experience or understanding of the problem is sufficient to decide at least the type of network and the training process to follow. Different inputs can be tested.

On the other side, the settings in figure 9 do not change the entire logic behind the learning process or the network main properties. Therefore, a change in these choices does not require the trial and error procedure to restart with the needing of redefining all the other hyperparameters. These settings are the architecture of the network, the learning rate and all the other hyperparameters required by the training algorithm so that it can work properly. For instance, adding a layer to the network may not require to modify the learning rate, but changing the learning algorithm from gradient descent to another method, would definitely mean to reset all the parameters previously defined.

In addition, the settings are ordered from the top to the bottom of the flowchart according to the importance they have in varying the final results. For instance, this means that modifying the architecture is usually more impacting than modifying the learning rate or the other learning parameters.

The flowchart defines a trial and error process which allows to tune all the hyperparameters in order to train a neural network capable of giving satisfactory results over the training dataset and ready to be tested with different data. After defining the basic structure with the settings in figure 8 and initializing all the other hyperparameters, the network is trained and then tested over the training data (chapter 6.2). If the results are satisfactory, a test over different unseen data is carried out. If they are not, the algorithm needs to be modified and the trial and error procedure starts.

For beginner users and considering an adaptive learning algorithm, following the logic of the flowchart can be a good option. Starting from a small architecture, the inner hyperparameters can be changed, starting from the one which set the regularization. The latter usually affects the results much more than the others. If a satisfactory solution can not be found, the configuration of the network can be enlarged by adding layers or neurons. Then, all the inner hyperparameters should be tested again. This is a long procedure which can take dozens of training trials for a single architecture configuration. Fortunately, after some attempts it may be clear which changes can affect more the result and how much they can do it, so that it becomes recognizable in just few hundreds of iterations if the training process is leading to a promising solution or not. Experience speeds up the procedure, letting the user understand which hyperparameters to ignore, which to test and how much to change them. This can reduce the trial process for each architecture to only some attempts. After the user has this kind of experience, it is actually recommended to follow the opposite path, according to the importance of the settings. Therefore, the process can start with varying directly the architecture from a small configuration to bigger ones until the training process can reach a quite low error after the first few hundreds epochs/iterations. How low the error should be to proceed, is judged by the user according to previous results. After choosing a promising architecture, the regularization hyperparameter is again the most important parameter to evaluate and tune. Finally, all the most inner hyperparameters should be tried and changed until they are tuned. Being able to understand how the training process could evolve from the first hundreds of iterations is essential to save time, since it is not necessary anymore to wait until a late phase of the

training to decide if it is worth it to continue.

If it is not possible to achieve satisfactory results following this procedure, it means that major changes have to be considered and settings of the basic logic and structure of the network have to be modified. Whenever this happens, the method just described has to be repeated from the beginning and all the hyperparameters, including the architecture, have to be redefined. Major changes can be made up to changing completely the type of network used to solve the problem if necessary. If none of the neural networks tried in this way is able to give satisfactory results on the training dataset, it means either that the user has not tried enough configurations or that it is impossible to solve the problem with a neural network.

The process described so far is the method followed during this work in order to reach the results presented in chapter 7. In the following paragraphs, each different setting chosen during the trial and error process is explained and justified.

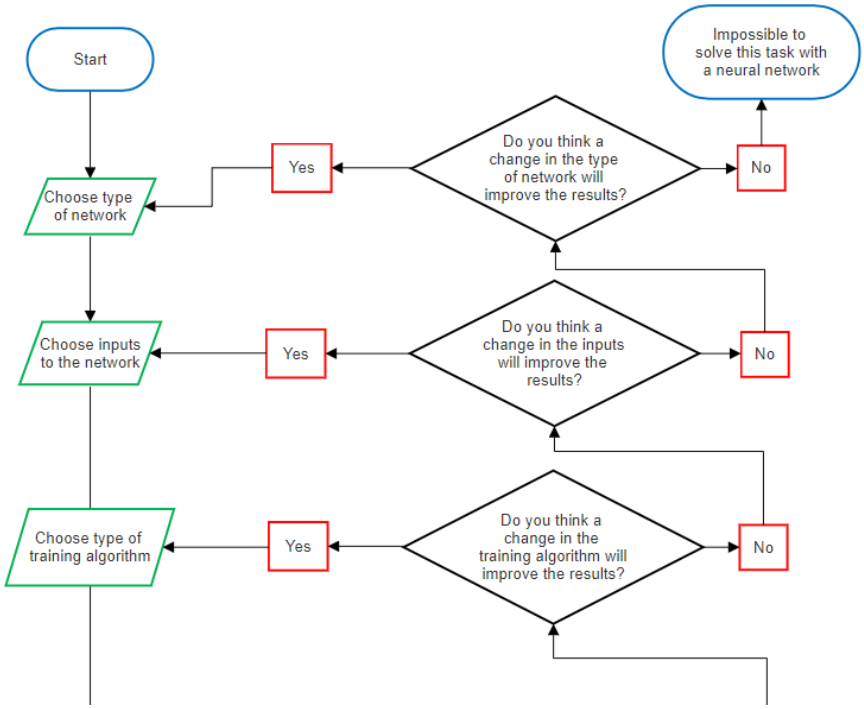

Figure 8: Flowchart followed to train the network, part 1.

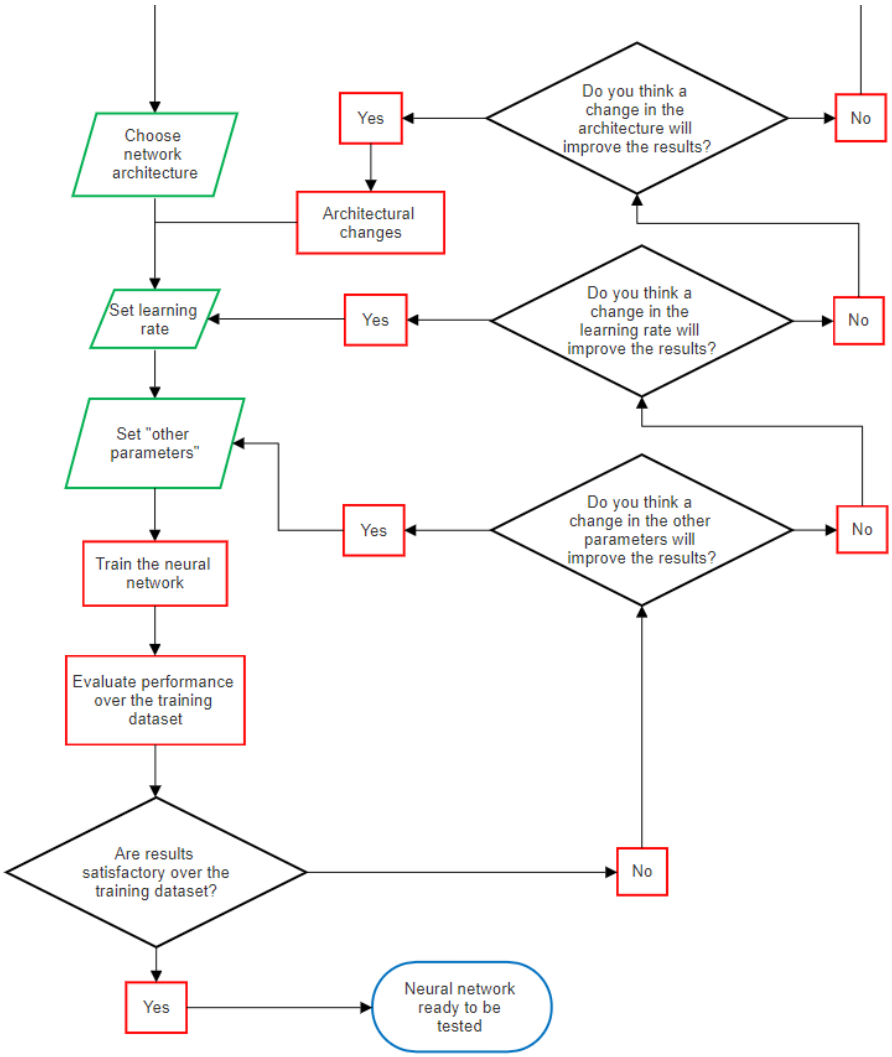

Figure 9: Flowchart followed to train the network, part 2.

6.1.1 Choice of the type of network

Regarding the choice of the type of neural network to use for the torque estimation, there are two consideration to take into account:

- The network, having to resemble a vector of torque values $\boldsymbol{y}$ from a dataset of inputs $\boldsymbol{x}$, has to solve a regression problem. Thus, among all the possible choices given by the tool, a regressive network has to be chosen or built.
- The network has to be implemented in a CPU, therefore it can not be too complex due to limitation in the running time of the system and in the

memory occupied.

Considering these two aspects, recursive or convolutional neural networks were avoided because they are complex solutions. The tool, apart from the structure of a general feedforward algorithm, presents directly the possibility to define a regressive shallow neural network with the command *fitnet*. Feature of the network built by this command is to have as output layer a regression layer with pure linear activation function. Furthermore, this structure allows to choose freely the number of hidden layers, hidden units, activation functions for each layer and training algorithm.

Since this choice allows the user to have both a simple and a regressive solution, with the possibility to choose every other parameter, this is the type of neural network which is used in this work.

6.1.2 Choice of the inputs to the network

Regarding the choice of the inputs to feed the neural network with, as said before, only measurements considered as safe can be used. The possible choices are:

- Phase peak current, in A_{peak}.
- Rotor speed, in rpm.
- Stator temperature, in C°.
- DC bus voltage, in V.
- Estimation of slip speed from the module of the control system, in rpm.
- Estimation of I_d current in the $d-q$ plane from the control system, in A_{peak}.
- Estimation of I_q current in the $d-q$ plane from the control system, in A_{peak}.

An important consideration to take into account is that all the inputs should be expressed in SI units while given to the network or used to train it. Therefore, if this is not the case from the direct measurements, the inputs are transformed into SI units by using the relative scaling factor. The reason for this is explained in chapter 6.4.

Since there is no direct equation which allows to calculate the electromagnetic torque from these inputs because of the lack of the phase voltage measurement, the torque has to be estimated. This is done in Inmotion's system through the estimation of I_d, I_q from the peak of the phase current and estimated slip speed.

The aim of the neural network would be to replace the current module with a new way to estimate the torque to remove calculations which could introduce errors in the estimation. It would be therefore completely contradictory to use

the estimated I_d and I_q to train the network because they would be required inputs when the network is used.

However, even if it is estimated from the system and it would be an improvement to remove it from the calculation, totally different considerations have to be done for the slip speed. From the physical point of view for an induction machine, the sign of the slip gives the sign of the torque. It would not be possible for the network to understand which is the right sign the resultant torque has to have from the other inputs. Therefore, the estimation of the slip speed is required as input to the neural network.

Furthermore, easy choices are the peak phase current, without which the network would not be able to resemble the module of the torque, and the rotor speed. This last is needed for two reason: first, the network should be able to understand when the machine is used as motor (same sign between speed and torque) and when it is used as generator, for example during a regenerative breaking (different sign between speed and torque). Second, without the indication of the speed, the network would not be able to understand the concept of field weakening.

The DC bus voltage measurement, on the other side, is an almost constant value which does not give any indication about the flux of the machine, since it is not the voltage fed to the motor. For these reason, it can be disregarded.

As last input to analyse, the temperature deserves its own study: a critical measurement for the motor, since it influences its parameters and consequently current, slip speed and torque, is the rotor temperature measurement. Since there is no practical way for this to be directly done, parameters are corrected according to an estimation based on the stator temperature variance, but errors are inevitable. One of the most interesting points of this work is to evaluate if the neural network is in some way able to bypass the lack of this information and for this reason, the stator temperature has not been used as input. The results given by this choice can be seen in section 7. More considerations are given in chapter 9.

Finally, one more choice related to the inputs has to be done: the choice of the teacher. In order to resemble the electromagnetic torque of the motor, two different possibilities can be investigated. The first one is the torque on the shaft coming from the torque transducer. This signal lifts up two problems for the estimation: the shaft torque is not equal to the electromagnetic torque and the signal itself is heavily filtered because of the ripple on the measurement. The first issue is due to two factors:

- The measured torque at the shaft takes into account friction and windage losses (mechanical losses) which constitutes the main difference between electromagnetic and mechanical torque.

- The measured torque at the shaft, depending on the stiffness of the shaft, may take into account the torque generated by the inertia during a speed change, while the electromagnetic torque is free from this component.

Therefore, unless corrections for both traits are implemented before using the

signal as teacher, this measurement is not a completely correct representation of the air-gap torque. In addition, the measure requires an additional filter due to the poor performance of the torque transducer. This introduces a difference in the transient behavior of the torque in respect to the inputs used to estimate the torque itself. Therefore, the measured torque is not an optimal choice as teacher. Further considerations about the filtering of the input are handled in chapter 6.4.

For this reason, the chosen target for the training process is the torque calculated by the control system, which is a direct calculation of the electromagnetic torque of the motor.

In practice, the inputs are curves of the peak value of the current, rotor speed, slip speed and electromagnetic torque calculated by the control system derived from an accuracy test of the motor. The test was run for temperatures included in the range 80-95 degrees, in order to evaluate the performance of the control for "constant" temperature. A fixed set of different steps in torque, both in positive and negative direction, has been given to the motor for each and different value of speed, both positive and negative, after the pre-heating phase. The sequence of the torque steps expressed in factor to multiply to the maximum torque of the motor available for that speed, is the following:

$$\text{from zero,} \quad 0.5 \quad 1.0 \quad 0.3 \quad 0.7 \quad 0.1 \quad 0.6 \quad 0.9 \quad 0.2 \quad 0.4 \quad 0.8 \tag{68}$$

These torque steps are given first in the positive direction and then in the negative one at each different speed defined by the rotor speed sequence. The latter, expressed in rpm, is the following:

$$500 \quad 1000 \quad 1500 \quad 2500 \quad 3000 \quad 4000 \quad 5000 \quad 6000 \quad 7000 \quad 8000 \quad 9000. \tag{69}$$

After the speed sequence has been tested completely, it is repeated in the opposite direction (negative speed) for the same values of speed, with the same values of torque steps, both in positive and negative direction.

The choice of these curves as inputs, represented in the following pictures, is due to the fact that they cover a great amount of working points of the motor. In fact the motor is controlled to produce different percentage of the maximum torque for each speed, including field weakening working point. Furthermore, the test gives also points in which the machine is used both as motor and as generator. The curves of the inputs are shown in the following figures.

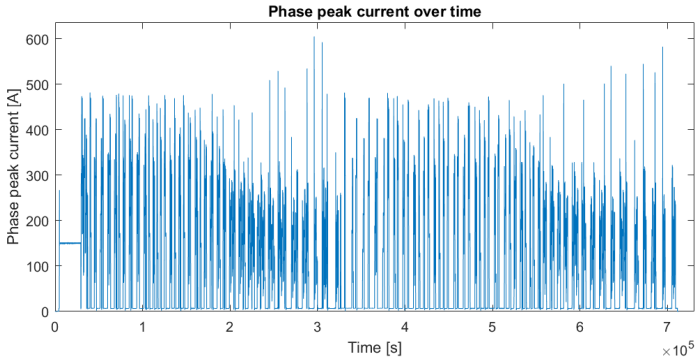

Figure 10: Curve of phase peak current (A) used as input to the neural network.

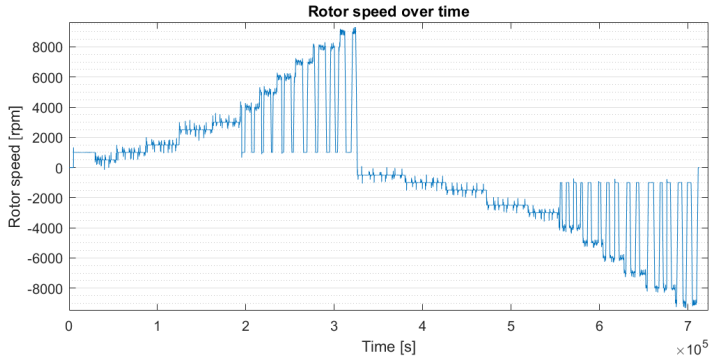

Figure 11: Curve of rotor speed (rpm) used as input to the neural network.

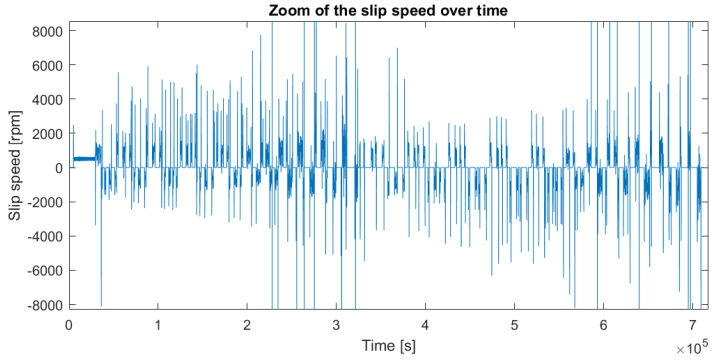

Figure 12: Curve of slip speed (rpm, zoomed) used as input to the neural network.

6.1.3 Choice of the type of training algorithm

Despite MATLAB Deep Learning Toolbox allows different choices of algorithms to train the network, gradient descent is definitely the one to start with, as explained in the chapters related to the theory of learning models and neural networks. After starting with the basic gradient descent algorithm, given by the MATLAB command *traingd*, it was straightforward to understand that an optimized algorithm was needed.

Training with gradient descent means to tune the learning rate, a long trial and error process. For this reason, after a few attempts, the training algorithm chosen to continue the work was an adaptive learning algorithm with momentum, a variation of the Adam model presented in chapter 5.4.3.3. The command used to implement it in MATLAB is *traingdx*. This choice allows the user to avoid the tuning of the learning rate, but it introduces other hyperparameters which are analyzed in chapter 6.1.6. In addition, a feature of this model is the implementation of early stopping, present during the training process.

The equation used for backpropagation by this algorithm is the following:

$$dX = \alpha \cdot dX_{prev} + \epsilon\alpha \cdot \frac{dJ}{dX}, \tag{70}$$

where X is a parameter, α is the momentum hyperparameter, dX_{prev} is the variation of the parameter at the previous iteration and ϵ is the learning rate of the current iteration, which changes in respect to ϵ_{prev} as explained in chapter 5.4.3.3 according to equations (63), (64) and (65). The parameters X considered for this calculation and consequently updated are only weights and biases. The chosen cost function J is mean squared error, for the benefits explained in chapter 4.4, and a regularization term with hyperparameter β corresponding to weight decay is added.

Other types of training algorithms were not tried, since the chosen one was able to give satisfactory results. Nothing can be said about their effectiveness.

6.1.4 Choice of the network architecture

As explained previously, from this setting on, changing something would mean to modify the result while maintaining the same logic behind the algorithm. This means that after setting up the basic structure of the network with the selections made so far, the main part of the work to try to achieve correct results is the choice of the architecture and parameters related to the training model.

Taking into consideration that the learning rate does not have to be tuned thanks to the adaptive algorithm, defining the right architecture of the network is by far the hardest and longest task. A recommendation, considering that small models should generalize better, occupy less memory in terms of parameters and have a faster runtime, is to start with a small network with simple activation functions, like the linear ones. The structure can be increased in size and capacity if necessary.

The means to modify the architecture (red block “Architectural changes” in the flowchart) in order to improve the results are explained in chapter 6.1.7.

This process occupied the great majority of the working time, until the following architecture gave satisfactory results for the aim of the work:

- Input layer with three units, corresponding to the three inputs chosen as in chapter 6.1.2.
- Six hidden layers, with 10 units each one.
- Pure linear transfer function for the first three hidden layers, hyperbolic tangent activation function for all the remaining three hidden layers.
- Regressive output layer with one unit and pure linear transfer function.

This final choice derives from many different considerations related to the necessity of having good enough results from the evaluation and testing (chapter 6.2) while satisfying hardware constraints (chapter 6.3). It is thus justified in the relative sections.

6.1.5 Choice of the learning rate

As already mentioned, the adaptive learning algorithm is able to increase or decrease the learning rate at each iteration of the training process according to the value of the cost function of the previous reiteration. Therefore, tuning ϵ is not necessary anymore. Its initial value is left equal to the default value from MATLAB, which is 0.01.

6.1.6 Choice of the "other parameters"

The term "other parameters" is referred to all the other hyperparameters which depends on the learning algorithm chosen, plus the regularization hyperparameter. For instance, if basic gradient descent is used, there is no α, as well as adaptive factors. Since Adam is used, a lot of hyperparameters are added to the regularization term. Here there is a list with all of them and their respective chosen values.

- Maximum number of epochs/iterations: maximum number of iterations after which the training process is stopped independently from the result. Value: 200000.
  This value, which can be considered high, is chosen in order to let the algorithm train until the training error over time becomes almost horizontal, indicating that it is not possible to improve the results significantly if the training process continues.
- Performance goal: indicator of the goal of the training process, zero means that the solution is perfectly coincident to a minimum of the function. Value: 0, for obvious reasons.
- Momentum hyperparameter: α, as described in section 5.4.3.2. Value: 0.9 (default value).

This hyperparameter is left as default value because it is one of the recommended value from the theory.

- Learning rate increasing factor: f_{inc}, as described in chapter 5.4.3.3. Value: 1.05 (default value).
  This value is not increased from the default of 1.05 because a large increasing in the learning rate causes larger steps for the solution driving to the drawback described in section 5.4.3.3 for a lower number of consecutive iterations.

- Learning rate decreasing factor: f_{dec}, as described in chapter 5.4.3.3. Value: 0.1.
  The choice of setting this parameter the low value of 0.1 (decreasing the learning rate ten times at each epoch) is due to the fact that when the instability due to a high learning rate comes to the solution, it is necessary to return to low values of learning rate to be able to have a small step. Smaller steps are more prone to follow the new optimal direction instead of "drifting" away. For this reason, a low decreasing learning rate factor can help the algorithm to recover the optimal direction in the least number of iterations.

- Learning rate increasing threshold: $f_{threshold}$, as described in chapter 5.4.3.3. Value: 1.04 (default value).

- Maximum validation failures: hyperparameter which defines after how many consecutive iterations that the performance is not decreased the training algorithm has to stop. This is the threshold for early stopping. Value: 80.
  This value is chosen to be reasonably high so that the training process can continue until it finds a deep minimum. In addition, a high value allows the algorithm to stabilize again before stopping the process because of the drawback described in section 5.4.3.3.

The regularization hyperparameter β deserves its own spot: the regularization method implemented through the *traingdx* command is weight decay. As already described in chapter 5.3.1.1, adding a regularization term to the cost function means to shrink the parameters. Decreasing parameters correspondent to directions irrelevant to the reduction of the cost function is not a problem, it lowers their influence on the results improving the performance. However, when the hyperparameter of regularization is too high, it can cause the shrinking of the parameters related to the optimal direction of the training. This can cause problems in finding and reaching the optimal solution for the training process. For this reason, this hyperparameter can not be set to a large value, but it has to be tuned.

MATLAB intrinsically normalizes parameters and cost function during the calculations, so that the regularization hyperparameter can be chosen as a value between 0 and 1. Larger values avoid overfitting improving the generalization error, but penalize the network for large weight and training error as drawback.

In order to have satisfactory results, this parameter has to be set to a value higher than zero considering all the previous choices. The training process showed a drastic improvement in runtime and results achieved for values larger than 0.7. The chosen value is eventually 0.95.

Higher values than 0.95 may lead to a drawback. The process can obtain the same results in less number of iteration in comparison to the same algorithm, same architecture and hyperparameters, but regularization set to a lower value. This either saves a high amount of time or allows to get better result for the same runtime. However, for some cases, the training stopped before because of validation failures. During the training process, if gradient and performance are monitored, it is possible to notice that sometimes the gradient can reach low values while the cost value is still really high. This is due to the fact that the solution can find a saddle point, a flat region or a badly placed local minimum on its way down. This happens for almost all the network with 5 hidden layers or more, especially at the beginning of the training process when the error is still high. For β equal to 0.95, the gradient usually becomes lower than for lower values of β, causing the steps in the updating of the parameters to be smaller. Sometimes, because of the small steps, the algorithm looks like not able to escape those critical points before the validation failures reaches the threshold of 80 iterations, causing the training process to stop prematurely. In addition, it seems that the destabilization effect of finishing "out of the rail" due to the increasing of the learning rate for adaptation is stronger for higher values of β. This sometimes causes the stop of the training process because the number of iterations necessary to stabilize the solution to a new optimal path are more than 80.

In order to avoid to have relevant problems during the training of a big amount of different networks, the regularization hyperparameter β was set to 0.95, value which led to satisfactory results and still a solid training for most of the cases.

6.1.7 Architectural modifications

The configuration of the neural network highly affects the result of the training process and brings some considerations to be aware of from the hardware point of view. Figure 13 shows the steps to follow internally to the red block "Architectural changes" in figure 9.

A recommendation is to change and test at first the activation functions of the layers before modifying the entire configuration of the architecture. Some functions could be better than others depending on the task to solve, but there is no way to be able to distinguish which to use in which occasion and in which order to set them along the layers. Trial and error is the only way. If none of the different layouts of activation functions tried is leading to a promising training, the network should be made deeper. As explained in chapter 5.1.4, adding layers improves the generalization error more than adding hidden units. Only as final step in the trail and error procedure, the number of neurons should be changed. This way of proceed derives also from some considerations about

the implementation of the network in the hardware, which are postponed to chapter 6.3.

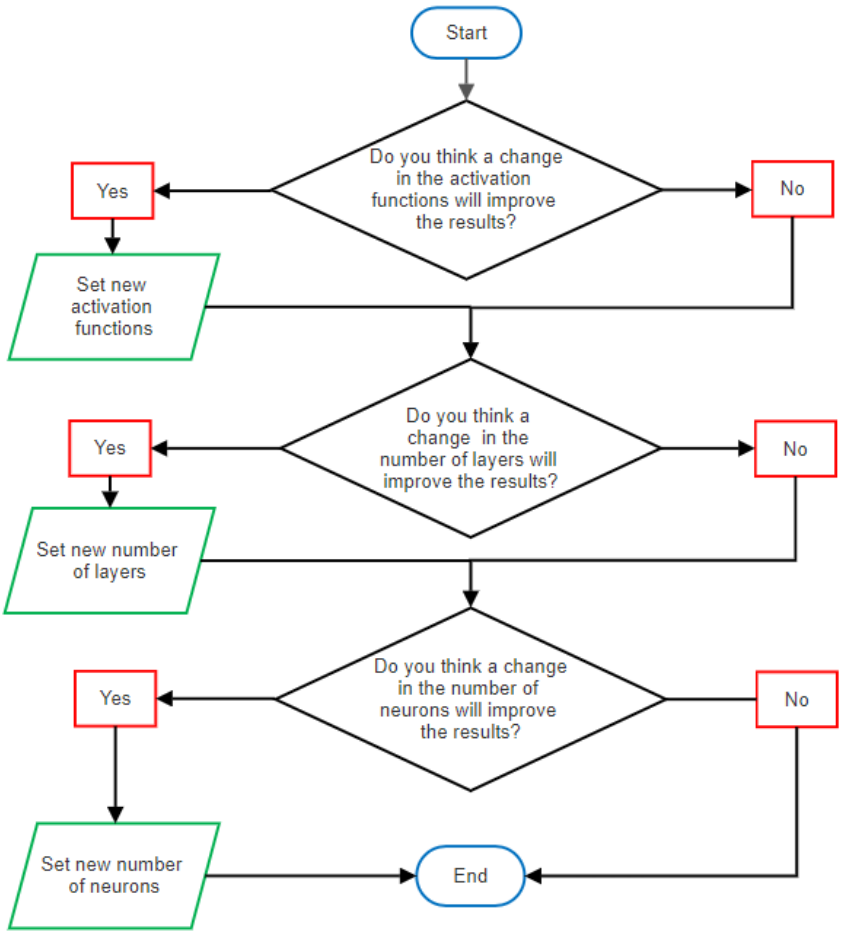

Figure 13: Recommended trial and error procedure to change the architecture of the network.

6.2 Evaluation and testing of the trained network

After all the settings and hyperparameters are chosen and the network is trained, it is necessary to test the algorithm over the training dataset. Since the generalization error is always higher than the training one, if the algorithm can not even fit the training dataset in a proper way, it is impossible that it will work in the application.

However, before testing the network, some modifications to its results can

be done. It is important to remember the physical implementation of the torque estimation: the system has to control the motor of an electric vehicle. Due to the weight and the inertia of the vehicle, it will not start moving for low values of torque. The safety limit related to this behavior is defined by ISO 26262 and it depends on the type of the vehicle. For any specific vehicle, there is a value of torque provided by the motor below which the vehicle does not move and above which the vehicle starts to move. In the case of this motor, this limit value is translated into a boundary in current. Therefore, whenever the current is below that limit, the estimated torque can be set to zero because it will never be able to move the vehicle even if different from zero.

Another consideration to take into account is that, for physical constraints of the motor, the produced torque on the shaft is limited by the value of the peak current. The motor at any moment can not provide a torque which is higher than the peak current multiplied by the motor torque constant K_T. Therefore, it is possible to saturate the value of the estimated torque to the peak value of the current (input to the network) multiplied by the torque constant at any time.

These two modifications on the estimated torque are possible only because of the physical application of the network and they have to be applied right after the training process. It would be detrimental and erroneous to estimate a value of torque which is not even possible according to the physics of the problem, or to let the estimation have a value different from zero when that value does not affect the physical system. This is implemented for every network of the trail and error process right after the training as indicated by the red block "Train the neural network and consider physical constraints" in figure 9.

After applying these modifications to the result of the neural network, it is possible to start the testing process shown in figure 14 which corresponds to the red block "Evaluate performance over the training dataset" in figure 9. The different testings presented here derive from limitations due to safety requirements according to ISO 26262. How safety requirements are translated into torque restrictions is explained in chapter 3.4.2. These limitations are recalled here for simplicity:

1. The estimated torque, output of the neural network, can not overcome 2% of the maximum torque of the motor when the command torque is zero.

2. The estimated torque can not have a different sign from the actual measured torque of the motor for more than 100 ms.

3. The estimated torque can not overcome the following limits, defines in the system as "safe functions" for more than 100 ms, in both positive and negative direction:

    - 10% of the maximum torque of the motor in the opposite direction of the command torque when the command torque is less than 33% of the maximum torque of the motor.

- Value of the command torque plus 10% of the command torque with the same sign of the command torque when the command torque is below 33% of the maximum torque of the motor.
- 1.3 times the command torque in the same direction of the command torque when the command torque is higher than 33% of the maximum torque of the motor.
- If the estimated torque overcomes the above limits for less than 100 ms, its integral in time can not be higher than the integral in time of the constant maximum torque evaluated for 100 ms.

Furthermore, according to internal requirements, the estimated torque can not underestimate continuously for more than 100 ms in respect to the actual measured torque.

The testing process, even if not indicated, obviously starts with giving the training dataset as input to the trained network, which returns the estimated torque as result. This results is then analyzed with an appropriate MATLAB script in order to understand if it can fulfil all the requirements, starting from the first in the above list. Since this evaluation is done for points where the command torque is zero, if the control system is working properly, the current of the motor has to be close to zero, or anyway below the limit in order to produce a torque which can move the vehicle. For this reason, for all these points, the estimated torque should be set to zero and this requirement should not cause problems. Not fulfilling this requirement means to have a wrong functioning in the control of the motor.

The second important check is related to the sign of the estimated torque, which can not be different from the sign of the measured torque for more than 100 ms. If this happens, the supervising system causes the tripping of the inverter even if the control is working properly, allowing the motor to produce the right amount of torque in the right direction.

Before investigating the safe functions, it is necessary to check if the result is underestimating the actual torque, because if that is the case, it is necessary to add a factor and/or an offset to the estimated torque in order to prevent from underestimation. The underestimation should be avoided for periods of time longer than 100 ms according to internal requirements, so new values of factor and/or offset to multiply/add to the estimated torque should be tried until the requirement is fulfilled. Factor and offset have to be applied equally to each sample of the result and they have to affect each testing point in the same way. Applying different factors for different torque levels, for example, would fail the aim of the work, since it is equivalent to modify the estimation in order to fulfil the requirements according to the necessity. An estimation which has to supervise the control system resembling the actual torque of the motor can not be modified in such a way because it would fail its purpose. However, even if underestimation should be completely avoided, since a comparison between the neural network and the current estimation has to be done, the factor and offset chosen for this work are exactly the ones set for the current torque estimation in Inmotion's system. Because of this choice, the result of the network is not able

to prevent completely from underestimation, as shown in chapter 7. It is anyway possible to adjust the coefficient so that underestimation is fully avoided.

Finally, after the estimated torque is increased with underestimation's factors, the safe functions can be evaluated. If results are not considered satisfactory according to one of these testings, the network has to be retrained with different hyperparameters/settings in order to have better results. If the results are considered satisfactory over the training dataset, the real testing of the network on unseen data can proceed.

The procedure to test the network in this case is the same as explained so far, since it is based on requirements that the result has to fulfil when implemented, but with two main differences. First, the inputs for testing the network have to be new unseen data the network has not been trained on. Second, the underestimation factor and/or offset that have been chosen during the evaluation process can not be changed during the testing over unseen data. If the network can give positive results for many different testings over unseen data in this way, it means that the generalization error is low enough so that the algorithm is able to fulfil fully its task. If this does not happen, even dough the training error is low enough, the network has to be retrained with other hyperparameters/settings in order to decrease more the generalization error.

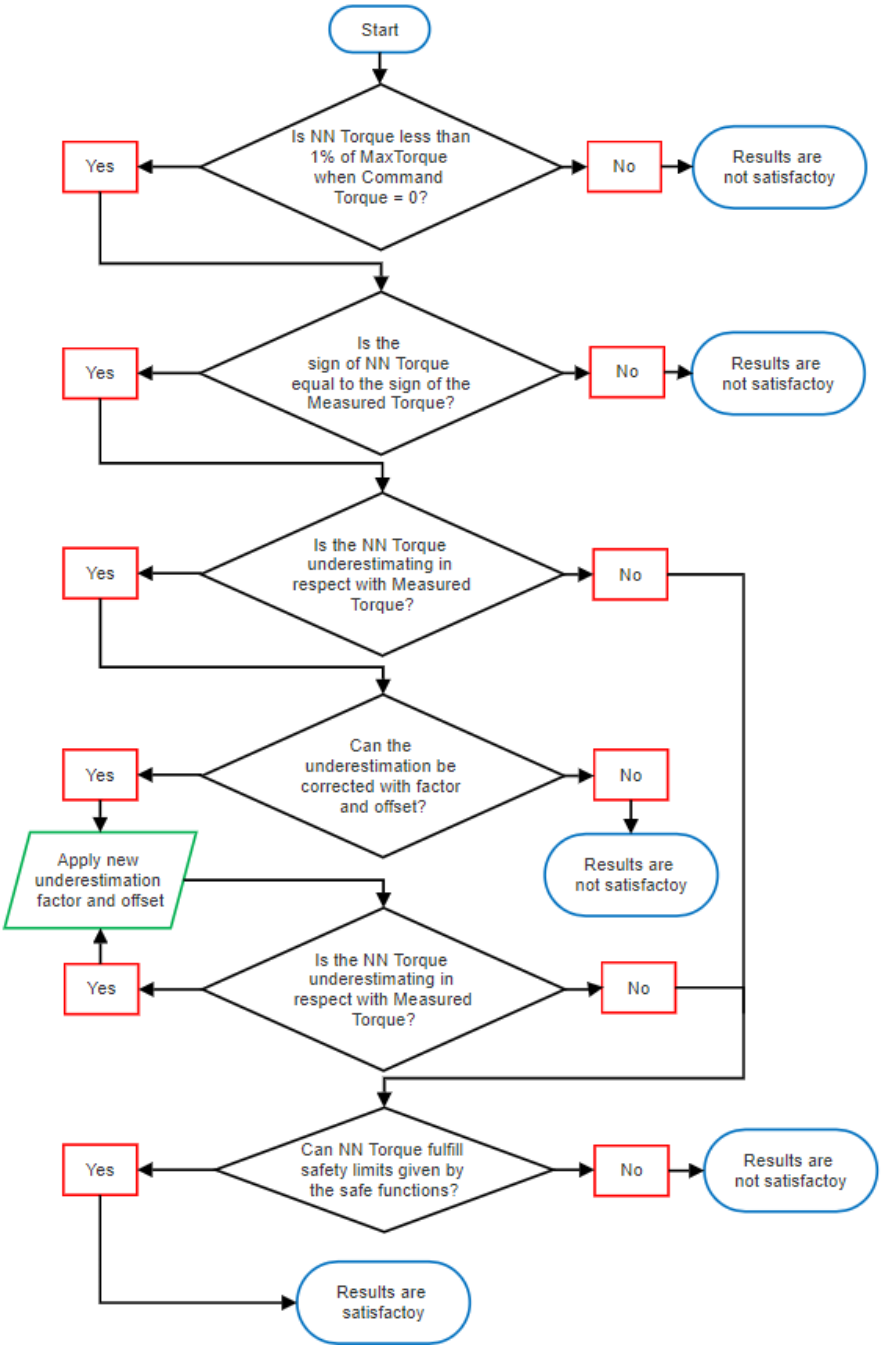

Figure 14: Testing procedure for the trained neural network.

6.3 Implementation in the hardware

After the network has been trained and tested, giving a satisfactory response, it is necessary to implement it in the system of the hardware to test it on the actual application. The system of control and monitoring of the inverter ACH6530 has already been described, through a convenient block diagram representation, in chapter 3.4. It is now necessary to explain how the software is actually structured in order to carry out all the calculations and solve all the tasks necessary for a proper control. After this, it is finally possible to explain how the network can be implemented inside this system.

6.3.1 Software structure of the application

Inmotion's software platform is a modular C based system managed in Subversion [12]. Each project developed by Inmotion is based on the same software platform, which defines the basic structure for each application and it is modified according to the requirements of the project. Thanks to Subversion's scalability, it is indeed possible to create new *branches* from the original platform *trunk*. These branches corresponds to a new version of the entire platform which can be modified locally according to the requirements of the current project and then uploaded back to the original trunk as a new application for the costumer.

Therefore, in order to implement the neural network in the platform, a new branch was created and all the necessary modifications have been done locally offline. Then, the trunk was updated with the final version of the branch so that the network could be tested on the actual application, which is the control system of the motor AVE130.

As said, the platform is modular: this means that the different tasks that the software has to carry out are divided into modules. For each important task there is a module which contains a main function in charge of computing that task, plus all the definition of parameters and secondary functions necessary for the function to work properly. For instance, module one may require data from module two in order to solve its task. In this case, module one has a specific function, apart from the main, which gets the data from module two. When working, the system calls all the modules one on one in the right order so that all the tasks which allow the functioning of the application are solved.

Finally, the language of the software platform is C. This means that the algorithm corresponding to the neural network trained on MATLAB has to be translated in C in order to be implemented in the system.

6.3.2 Generated C code and modular implementation

After the network has been trained and tested on MATLAB to be sure that it was possible to achieve results, it is necessary to derive the corresponding C code used for the implementation. In order to translate a piece of code from MATLAB to C language it is possible to use the application *MATLAB Coder*. The application requires the MATLAB script of a function and returns the equivalent function in generated C code. However, the neural network is a

structure datatype in MATLAB and it is not possible to translate it directly. For this reason, the command *genFunction* allows the user to translate the structure into a function which can be used in the coder application.

The function of the network consists in a list of vectors and matrices containing numbers, which correspond to the weights and biases of each layer coming from the training process, plus the calculation corresponding to forward propagation (chapter 5.2.1) for each layer, from the inputs to the output.

Although the MATLAB function corresponding to the network is easy to obtain directly through the command, it is not straightforward to translate it in C with the coder, since some commands or datatype created within the function are not translatable. Furthermore, as explained in chapter 6.2, the final version of the network is not the simple trained algorithm which does not take into account physical limitations. These limitations have to be added in the function before it is translated, so that the generated C code represent the final version of the network, ready to be implemented. It is therefore necessary to do some changes in the algorithm function:

- Remove all the datatype which can not be translated by the coder, like *cell* data, and substitute them with vectors or matrices.
- Remove all the function which can not be translated and replace them with translatable versions of the function.
- Add the statement regarding the limit current below which the torque can be set to zero because the vehicle is not moved by such a low torque:
  if input current $<$ limit current

$$Neuralnetworkestimatedtorque = 0 \tag{71}$$

  end

After the function is modified, if modifications are not done correctly, it may return a different value from the one given directly by the neural network structure. Thus, it is necessary to test the function and be sure that it returns the same value of the original network structure before using the coder to translate it.

Once the function is ready to be translated, the coder can be run and the C code generated. It is though necessary to decide for some settings about the generated code. The CPU of the control system allows to perform both floating and fixed point calculation with single precision. The entire logic of the MATLAB function is based on floating point calculation with double precision. It is not possible to achieve double precision on the CPU, but it is surely advantageous to maintain floating point calculation instead of switching to fixed point while translating the code. This is due to two reasons: fixed points introduces more approximation and the generated C code results much heavier and bulky for the CPU, with a substantial increment in the computational resources used to run it. Thus, the generated code has to be based on floating point calculations with single precision.

Furthermore, the future and final purpose of the application of the neural network to the platform would be to replace the entire module which is in charge of the estimation of the safe torque, called *toesca*. In order to do this it is necessary, as first, to have an algorithm which responds completely correctly in each point in relation to all the safety and internal limitation. Then it is necessary to test the network for a huge number of test cases to proof that the model is working properly for each point and it can be used in the actual application. Projects like this require a lot of testing capability and time. Instead, this study is carried out to evaluate the possibility of the application. The function of the neural network is therefore only added to the module toesca, in order to evaluate whether this approach is possible if applied to the real application and to compare this new way of estimating the safe torque with the current one.

This affects another possible option for the coder: it is possible to choose if to optimize the generated code for the building time or the running time. Due to internal requirements, the module toesca is run every 500 Hz. This means that the entire flow of calculation performed by the module, including the one executed by the neural function which should be only a small part of it, has to be faster than 2 ms. The natural choice of optimization for the generated code is therefore made in order to have faster runs. Further explanations are given in chapter 6.3.4.

6.3.3 Flow of calculation of the module Toesca

The choice of the inputs to the network has been explained and it has been said that the network is added as function to toesca module, which is in charge of estimating the safe torque of the motor. But how the inputs come to the neural function inside toesca? In order to understand this, it is necessary to follow the flow of calculation of the main function of the module.

It is necessary to specify that toesca module's task is only and exclusively to calculate the estimation of the safe torque. Safe limits are checked by another module called *satosu*. Therefore the calculation flow present in toesca starts with acquiring the data necessary for the torque estimation from other modules and finishes with the computation of the estimated torque.

At the beginning of the main function, the phase peak value of the current of the motor is acquired from the module *currme*, which has the task of calculating the peak value of the phase currents from the safe current measurements and calculating the current angle and its increasing, corresponding to the stator angular speed. This value of current, expressed in $A_{peak} \cdot 10$ for internal representation, is the value given as first input to the network, rescaled to A_{peak}. From the peak current, the maximum value possible of torque due to physical limitations of the motor is calculated. Then, the increasing in the current angle is obtained from currme, while the increasing in the rotor angle is acquired from another module called *encse*. The latter is in charge of getting the measure of the rotor position from the encoder and calculating the increasing in the rotor angle at each sample and the rotor speed in *rpm*. Once both variation of current angle and rotor angle are acquired, the second is subtracted by the first,

giving as result the estimated slip speed. If the current does not overcome the limit in order to produce a torque which can move the vehicle, the slip speed is directly set to zero. This is useful for the other estimation of the safe torque, the one done without the neural model. This value of slip speed is, according to the order used while training the network, the third input to the trained algorithm. Since it is expressed in toesca with an internal representation which is different from rad/s, it is necessary to scale it back to rad/s before giving it to the network. The second and last input, which is the rotor speed in rad/s, is directly acquired from encse in rpm and scaled back to SI unit before feeding it into the network. After having all the entries, they are collected in a vector and given to the neural network function which performs the calculation and gives a result. This result is multiplied by the underestimation coefficient, underestimation offset is added in the correct direction (positive if the torque is positive and negative if the torque is negative) and finally the estimated torque is saturated for the limit value calculated from the peak current. The calculation then proceeds with the estimation of I_d and I_q from peak phase current and slip speed. Finally, currents in $d-q$ plane are use to estimate the torque in the way the estimation is implemented currently.

6.3.4 Constraints introduced by the implementation on the CPU

After explaining how the network is implemented in the system, it is necessary to evaluate how it affects the software. The neural function introduced in the system consists in:

- Definitions of parameters corresponding to the weights and biases coming from the training process, expressed as vectors or matrices. The total number of matrices corresponding to the weights and vectors corresponding to the biases depends on the number of layers. The size of these vectors and matrices depends on the number of units for each layer.
- Calculations derived from the application of equation (30) (forward propagation) for the hidden layers. For each layer, the internal sum is given by two nested for cycles with number of iterations equal to the number of units for that layer. The corresponding activation function is then applied to the resultant matrix coming from the cycles.
- Application of the final output layer with linear transfer function.
- Application of the statement expressed by (71).

The quantity and size of vectors and matrices affect the total memory needed for the CPU to save the network and perform its calculation. In addition, the computational time of the forward calculation also depends on the type of activation function implemented. A linear layer, for example, introduces only a linear calculation, which is performed by a multiplication and the sum of a bias. A tansig unit, on the other side, introduces an exponential in the calculation. According to the mathematical library of the software system, an

exponential is resolved in an approximation given by different multiplications. Therefore, tansig layers are more expensive than linear layers for the CPU from the computational point of view.

The software platform is built so that, according to internal requirements, some modules have to run at 500 Hz and others at 4000 Hz. Toesca is one of the modules which has to run at 500 Hz, which means that every 2 milliseconds the main function is recalled, restarting the calculations from the beginning. This introduces a strict constraint for the neural network: it has to be run within a time which allows toesca to complete every task in less than 2 ms. Inevitably, this is reflected on the architecture of the network, since runtime and memory occupied depends on the size of the parameters matrices.

Moreover, the total time necessary to run the whole safety evaluation, made up by many different modules including toesca, can not overcome the 15% of the total available time for the CPU to solve every task implemented in the software. If this happens, the CPU does not have enough computational resources to run all the tasks of the application, causing the system to fail. This is independent from the 2 ms runtime of toesca.

The first solution which has been tested on the back to back bench is actually a network with a different architecture from the one presented in chapter 6.1.4. It is in fact consisting of 5 layers and 12 neurons per layer with only tansig activation functions. It resulted in a failure because the total runtime for the safety task was higher than 15%.

Due to this problem, a different approach from before has to be adopted. It is now necessary to understand if new configurations allow the CPU to run without causing a failure. This should be done before training completely the network, because it can result in a great waste of time if then the trained algorithm can not be implemented. For this reason, new configurations has been trained only for the first hundreds of iterations (a process of 3-5 minutes) and then run on a board to evaluate the load on the CPU. The board is the same implemented in the actual inverter. Thanks to CAN environment and a software for testing the drives developed by Inmotion, it is possible to have a list of parameters in which the total CPU runtime for each main task can be seen.

Therefore, more steps in the training procedure are now added: a configuration of the architecture is trained for the first few hundreds iterations and it is evaluated on the board to understand if it is possible for the CPU to run it. In case of negative response, new architectures are partially trained and tested on the board until a compliant configuration is found. When this happens, that configuration is trained completely and then tested with the procedure explained in chapter 6.2. If the network gives satisfactory results, it can be implemented and tested on the back to back bench, otherwise the whole procedure described so far is restarted with a different architecture. Results coming from the testing on the board are shown in chapter 7.1.

During the evaluation of the load on the CPU carried out for different architectures, it was possible to derive some conclusions which allowed the trial and error procedure used to change the architecture of the network to be more clear, as shown in chapter 7.1. These considerations results in the method presented

in chapter 6.1.7, expressed by the flowchart shown in figure 13:

- Activation functions highly affects the result in term of runtime: as already explained in this chapter, different activation functions introduce different calculations. Before modifying the size of the network, it is worth it to understand if simpler calculations, like the case of the linear layer in comparison to the case of the tansig layer, give equally good results lowering at the same time the runtime of the software.

- Introducing a new layer should be less expensive from the computational point of view than adding the same number of neurons to all the current layers. For instance, evaluations on the board showed that a network of 6 layers with 10 neurons each has lower running time than a network of 5 layers with 12 units each, considering the same activation functions for both. This can not be said for all the cases, since it depends on the number of neurons added and on the activation functions adopted for the layers, but from the trends of the evaluations carried out on the board, it is recommended to add a layer instead of adding the same number of neurons for all the layers. Apparently, adding two nested for cycles to the function is better than adding some more iterations for each other for cycle already present, from runtime and computational resources point of view. This becomes more true for a high number of layers/for cycles in the function. In addition, deeper algorithm should theoretically give lower generalization error than wider ones.

The whole procedure described in this chapter and in chapter 6.1.7 has as result the network defined in chapter 6.1.4. From the beginning of the trial and error process to the end of it, more than 80 different neural networks has been trained, entirely or partly, in order to evaluated their capability of solving the task and being implemented.

6.4 Further considerations on the training process

Despite the process followed in order to train the network has already been explained in chapter 6.1, some more considerations have to be taken into account for a proper training, especially regarding the inputs. Even if the right inputs are chosen, the training process may fail due to wrong correspondence between them or between some of them and the teacher.

Two important aspects to evaluate are the units and the filtering of the inputs. In chapter 6.1.2 it was specified that the chosen inputs used to train the network are expressed in SI units. This choice is coming from both logical reasoning and experience due to trial and error process. First, since the result is expressed in according to SI units (torque in Nm), there is no reason to use the rotor angular speed and the slip speed in rpm instead of rad/s. Even though this is introducing a scaling factor in the pre-calculation, it is logic to treat all the inputs with the same units standard. This choice was confirmed by the trial and error process, which showed that the training process with rotor speed and

slip speed expressed in rpm is possible, but less effective for the same network configuration. Therefore, the inputs for the network are expressed in SI units.

A second important consideration regards the filtering of the inputs and/or teacher. Since the network has to give correct response for all the operating point of the motor, including transients, having different filtering between inputs and teacher is harmful. The inputs for the implemented neural network comes from toesca module and they present an internal filtering. Same for the torque coming from the control system and used as teacher. If an offline filter is added to all the inputs, but not to the torque, the relationship between inputs and torque is altered. Even if adding a filter does not change the response of the static points, apart from delaying it, the transient response would be different from the actual one. The same happens, obviously, when the torque is filtered but inputs are not. In this case, the training process may be able to achieve somehow reasonable results after a high number of iterations, but the outcome is useless. The relationship learned by the network does not correspond to the real one, therefore it is useless to implement such an algorithm. This is one of the two reasons why the torque coming from the transducer is not used as teacher: the measurement is heavily filtered by the system because of the high ripple introduced by the transducer.

Furthermore, implementing offline the same filter to both inputs and teacher is not an option either. Even though the transient relationship between training dataset and targets is theoretically preserved, it is slowed down in comparison to the actual one. The network in this case may give correct results on testings with the same filtering for the inputs, but not for the real implementation with inputs coming from toesca and the other modules, where that filter is not present.

A final case has to be considered, that is when only one of the inputs is filtered. The analysis of this eventuality is more difficult: even though the relationship between inputs and teacher in not completely preserved, if the filtering of the input is not heavy, the training process is able to reach satisfactory performance. If the generalization error is low enough, it may not be impossible for the network to give satisfactory results on the actual implementation. The only way to have a feedback about this behavior is to test the application with such a kind of trained neural network. However, since it is possible to train the algorithm with the same input filtering as the one which will be used in the implementation, there is no points in introducing a new filter without having the certainty that this will produce advantages on the results.

7 Tests, results and discussion

In this chapter, all the results achieved during the work are presented. First, a table containing the total time of usage of the CPU for the safety task for different architectures is shown. Then, the tests carried out on the motor in order to evaluate the performance of the implemented neural network as estimator are presented. Then, the results coming from these testings are shown and analyzed.

7.1 Tests on the board

In this sections, the results achieved from the testing on the board, as anticipated in chapter 6.3.4, are shown. Tests on the board have been carried out in order to understand which architecture of network could suit the implementation and how different changes of the architecture would affect the total CPU runtime for safety task. The following table shows which different layouts have been tested and their effect on the CPU. The letter L stands for layers, while N indicates the number of neuron present in each layer. Under the column named as "activation functions", when different types of functions are used for the layers, they are indicated in order of application from inputs to output.

Structure	Activation functions	CPU usage [%]
No neural network	-	11.03
2L, 10N	All tansig	12.46
3L, 12N	All tansig	13.64
3L, 15N	All tansig	14.41
4L, 10N	All tansig	13.9
4L, 12N	All tansig	14.52
4L, 15N	All tansig	15.97
4L, 15N	2 pure linear - 2 tansig	13.95
5L, 10N	All tansig	14.74
5L, 10N	2 pure linear - 3 tansig	13.73
5L, 10N	3 pure linear - 2 tansig	13.05
5L, 10N	All pure linear	11.77
5L, 10N	All pure linear, fixed point calculation	12.7
5L, 12N	All tansig	15.7
6L, 10N	3 pure linear - 3 tansig	13.83
6L, 10N	4 pure linear - 2 tansig	13.4
6L, 10N	5 pure linear - 1 tansig	12.66
6L, 12N	All pure linear	12.37
7L, 10N	3 pure linear - 3 tansig - 1 pure linear	14.06

Table 2: Percentage of usage of the CPU runtime by the total safety task for different network configurations.

As seen from the table, the solution with five layers of ten neurons each one, all tansig activation functions, is not actually overcoming the limit of 15% of the total CPU usage as said before. However, the value of 14.74 % is close enough to the boundary to cause problems when the entire software and communication system is set up on the motor. The board was not connected to sensors during these testings. This means that the functions in charge of getting data from the measurement on the motor were not run. Once the full communication system plus sensor is connected, this configuration overcomes the boundaries, leading to the impossibility of its implementation.

Furthermore, the results here presented confirm the consideration explained in chapter 6.3.4.

7.2 Tests on the motor

In order to evaluate the performance of the network in the real system, two similar tests to the one from which the data used as input were collected have been run. This time, a torque sequence has been given for different speeds for two different temperature ranges: the first one, denominated as "low temperature" range and indicated with LT in the figures, goes from 50 to 65 degrees. The second one, called "high temperature" range and indicated with HT, is between 90 and 105 degrees. The sequence of torque steps, applied both temperature ranges, is:

$$\text{from zero, } 0.5,\ 0.3,\ 0.7,\ 0.1,\ 0.9 \cdot \text{Maximum torque.} \tag{72}$$

This sequence is applied for each different speed, first in positive direction and then in negative direction. The speed sequence, in rpm, is the following:

$$500,\ 2000,\ 4000,\ 8000,\ -500,\ -2000,\ -4000,\ -8000. \tag{73}$$

For both tests, there is a pre-heating phase carried out at 500 rpm for a constant torque of 250 Nm.

The choice of testing the network with this working points derives from the fact that the network should be able to give correct results over the total range of operation of the motor. Speeds have been tested from the low value of 500 rpm to the high value of 8000 rpm, comprehensive of the value 2000 rpm, which is not present in the training data. Different steps in torque in comparison to the ones used during the training phase have been used for each speed, in order to evaluate different transients and different working points. Temperatures are different as well, giving slightly different values of current and electromagnetic torque, even if a working point evaluated with this test is the same as a point used to train the network.

In the following chapters, the results coming from both tests are shown and commented.

7.2.1 Low temperature test

In this section, the results for the low temperature test are shown. The following figures show the sequence of steps in torque applied, first in the positive direction and then in the negative one, for each speed.

Figure 15 shows the pre-heating phase and the legend. Colours are maintained without variations for all the other plots:

- Black: electromagnetic torque calculated by the control system, denominated as "measured torque".
- Blue: estimated torque through equation estimator, called "software estimated torque".
- Red: estimated torque through neural network.
- Green: boundaries defined by the safety requirements.

In this figure, the torque estimated by the software is highly underestimating. This is due to the fact that the points of the pre-heating are taken at the beginning of the measurement and the software, at the beginning, applies a slow filter to the torque estimation so that overestimation is avoided.

Each couple of figures shows positive and negative torque steps for one speed. It is possible to notice that for static points, independently on the value of speed, the network is able to give values inside the boundaries defined by the safe limits. Underestimation is not completely avoided, especially in some points at low speed (500 rpm, negative torque steps) as shown in Figure 16 on the right. However, this problem can be solved by tuning the underestimation factor and offset to different values from the one used by the software estimation.

The estimation is satisfactory for the speed of 2000 rpm (Figure 17), apart from some underestimating points around second 560 on the right.

Different is the case for 4000 rpm, Figure 18. Despite the fact that during static working points the network is able to give an estimation which is inside the safe limits, during transients from zero torque to 0.5·maximum available torque the estimation can not fulfil the safety requirements because the values are out of the limits for more than 100 ms. The same occurs for a speed of -4000 rpm (Figure 22), for the transient of the torque step from zero to -0.5·maximum available torque. A zoom of this behavior is shown in Figure 24.

Regarding the high speed of 8000 rpm in both positive and negative direction, the estimation fulfil completely the requirements of safety without underestimating for more than 100 ms. However, a problem may arise if factor and offset for underestimation are modified in order to avoid underestimation in low speed points. Figure 19 and 23 show in fact that for the highest step in torque, the estimation for static points is closer to the safe limit than for each other speed. It is still possible to avoid underestimation and maintain the solution inside the limits by tuning correctly the underestimation offset.

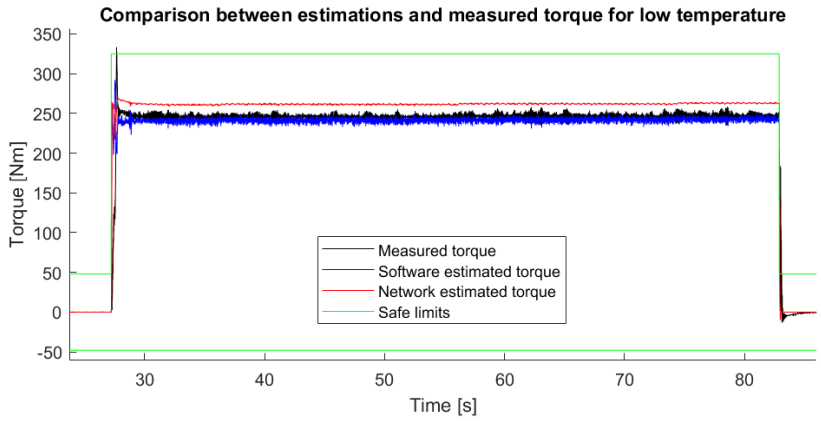

Figure 15: Pre-heating phase of the motor for the low temperature test. The legend shows the colour adopted in all the following pictures related to the low temperature test.

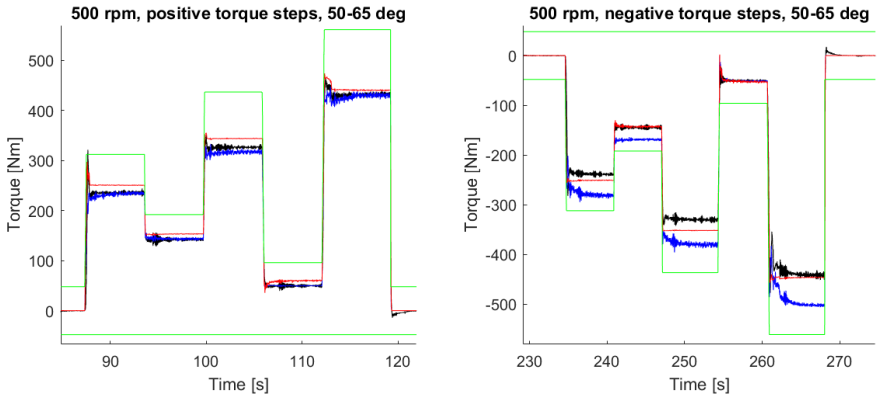

Figure 16: Torque steps in both directions for 500 rpm at the range of temperature 50-65 degrees.

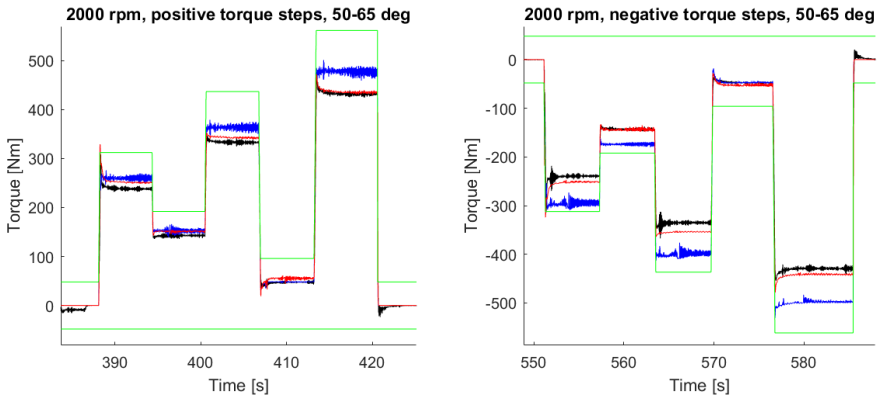

Figure 17: Torque steps in both directions for 2000 rpm at the range of temperature 50-65 degrees.

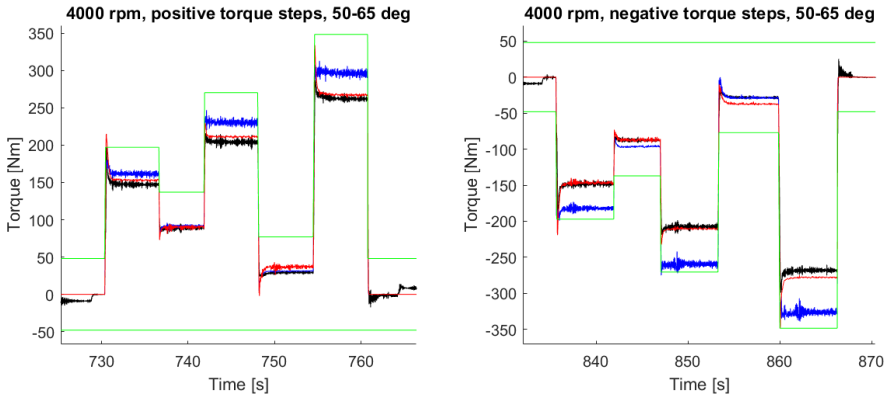

Figure 18: Torque steps in both directions for 4000 rpm at the range of temperature 50-65 degrees.

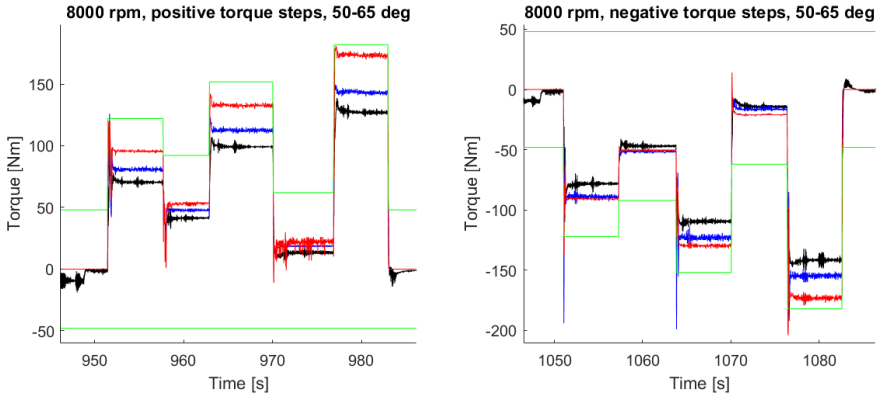

Figure 19: Torque steps in both directions for 8000 rpm at the range of temperature 50-65 degrees.

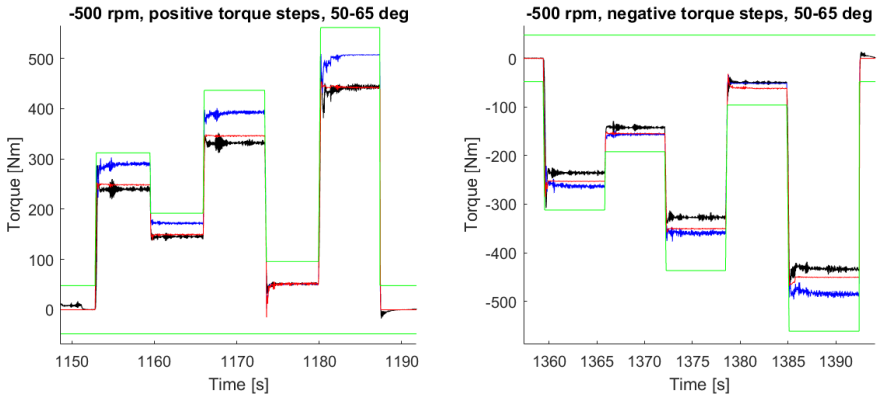

Figure 20: Torque steps in both directions for -500 rpm at the range of temperature 50-65 degrees.

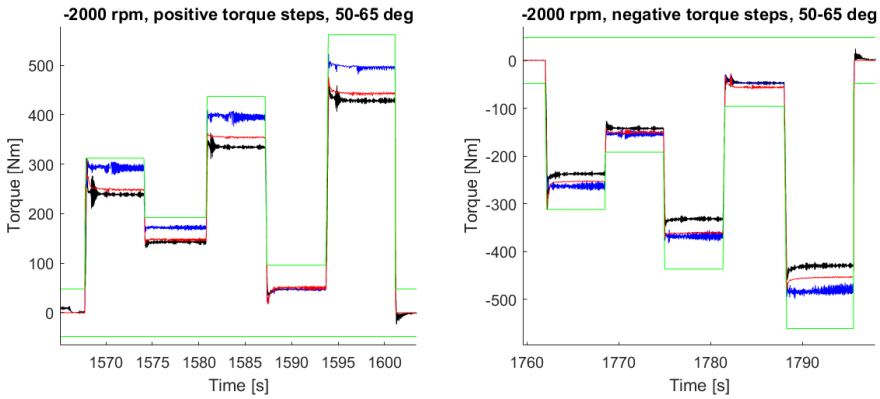

Figure 21: Torque steps in both directions for -2000 rpm at the range of temperature 50-65 degrees.

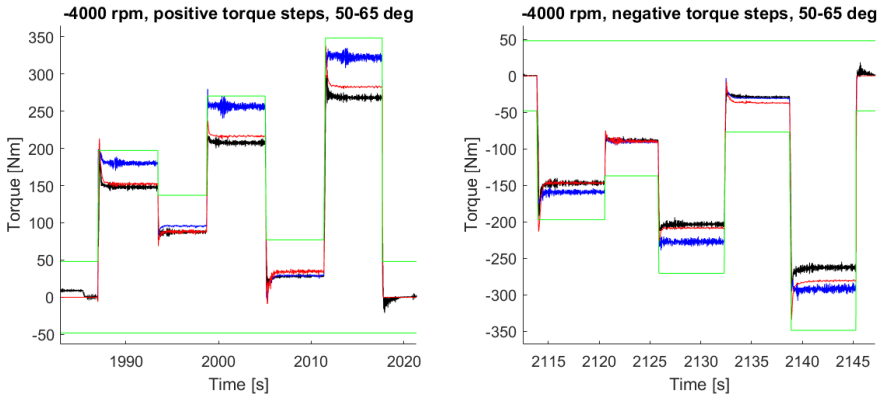

Figure 22: Torque steps in both directions for -4000 rpm at the range of temperature 50-65 degrees.

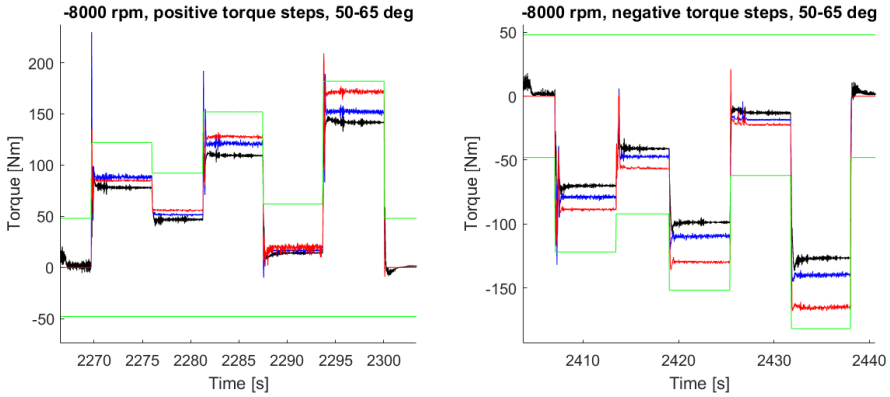

Figure 23: Torque steps in both directions for -8000 rpm at the range of temperature 50-65 degrees.

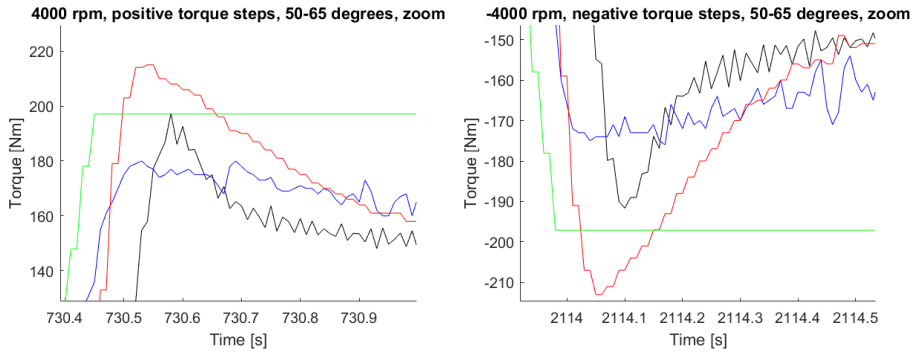

Figure 24: Zoom for the torque steps from 0 to 0.5·maximum available torque in both positive and negative direction.

Finally, the Figures from 25 to 32 show the static points of software and network estimated torque over the command torque at each speed. As expressed in the legends, blue pluses represent static points for the software torque, while green asterisks represent the network estimation.

As possible to deduce from the figures, the network estimator is able to give a result which is comparable with the current estimator. For low value of the torque at any speed, the results between the two estimators are very similar and some points are overlying. For high torques and speeds up to 4000 rpm, the network estimation is able to carry out slightly better results. At the speed of 8000 rpm, both positive and negative, the software estimation becomes better, especially for high values of torque. The following table shows the comparison between the two estimators over the whole data coming from the test. Most

of the points have the same accuracy because since the temperature is low, the motor spent the great majority of time to cool down between some torque steps and others. Therefore, for a great amount of time (more than 70 % of the test) the command torque was zero. These points have been cut and not shown in the previous plots, since they have no use.

Comparison in accuracy	Percentage [%]
Neural estimation more accurate than software estimation	11.8
Neural estimation less accurate than software estimation	10.9
Neural estimation as accurate as software estimation	77.3

Table 3: Comparison in accuracy between software estimation and neural network estimation over the whole dataset from the test at low temperature.

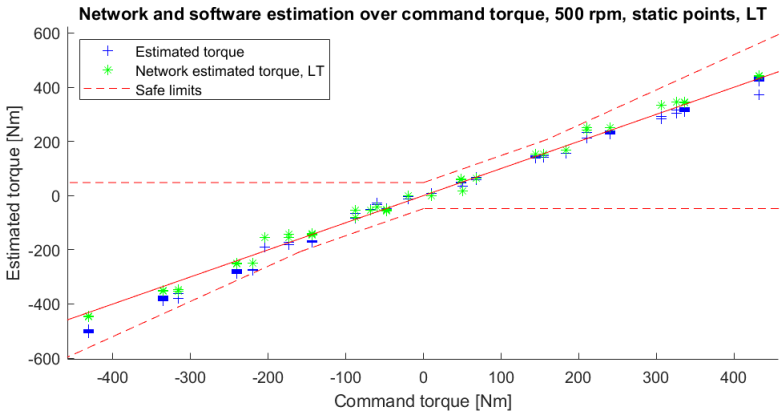

Figure 25: Comparison between estimators over command torque for static points for the range of temperature 50-65 degrees at a speed of 500 rpm.

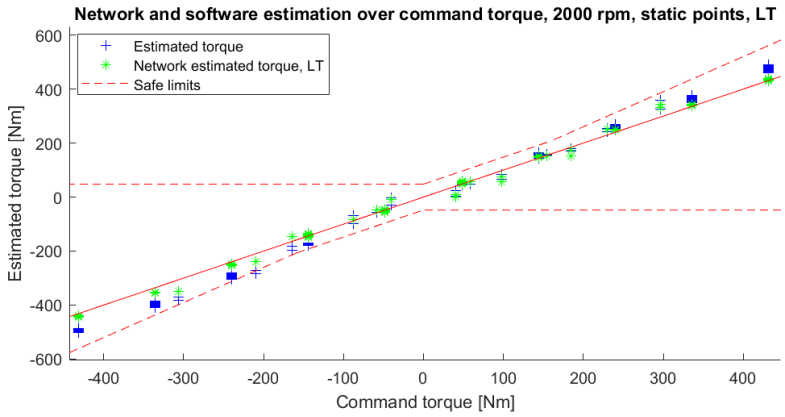

Figure 26: Comparison between estimators over command torque for static points for the range of temperature 50-65 degrees at a speed of 2000 rpm.

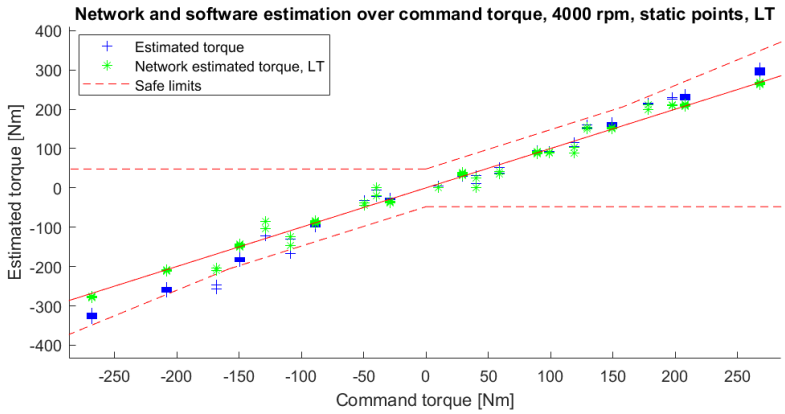

Figure 27: Comparison between estimators over command torque for static points for the range of temperature 50-65 degrees at a speed of 4000 rpm.

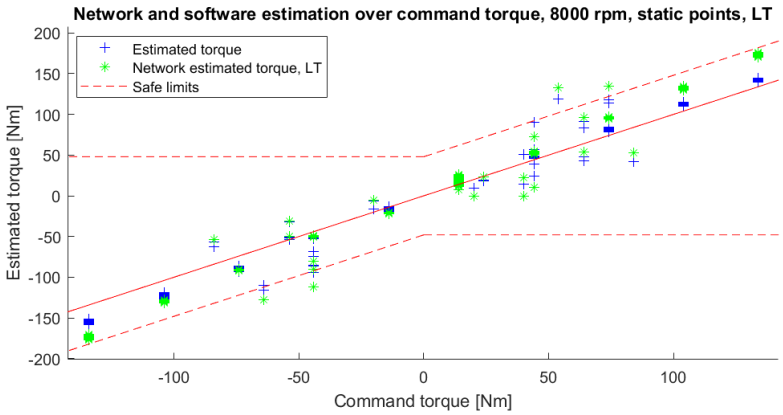

Figure 28: Comparison between estimators over command torque for static points for the range of temperature 50-65 degrees at a speed of 8000 rpm.

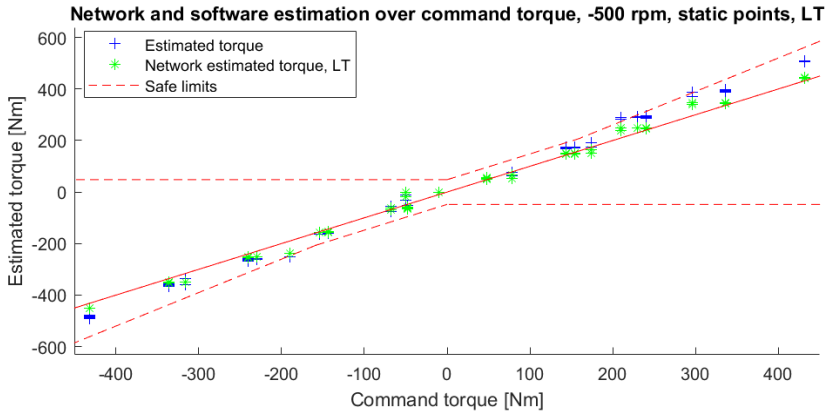

Figure 29: Comparison between estimators over command torque for static points for the range of temperature 50-65 degrees at a speed of -500 rpm.

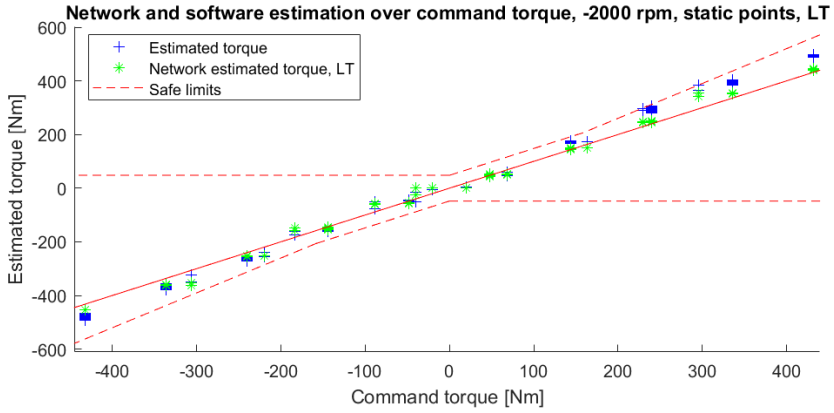

Figure 30: Comparison between estimators over command torque for static points for the range of temperature 50-65 degrees at a speed of -2000 rpm.

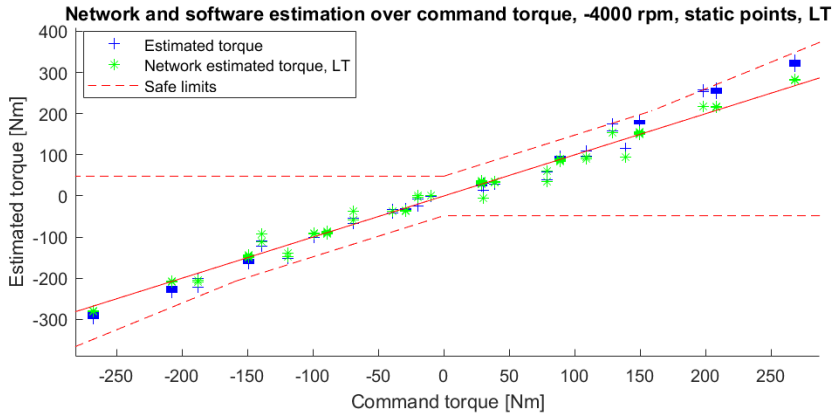

Figure 31: Comparison between estimators over command torque for static points for the range of temperature 50-65 degrees at a speed of -4000 rpm.

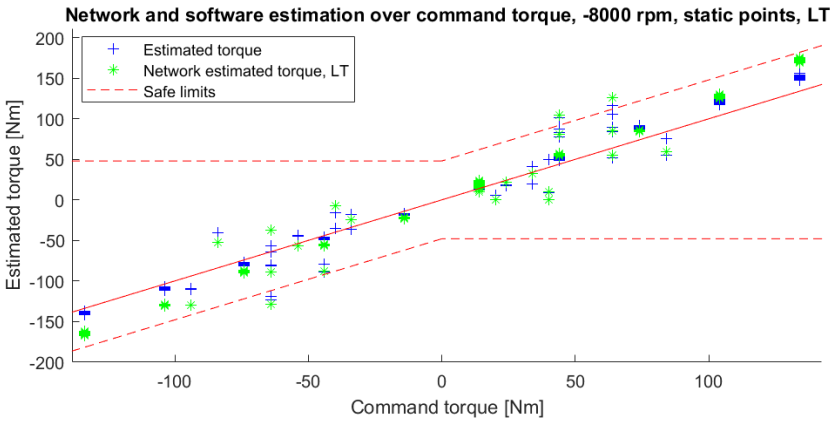

Figure 32: Comparison between estimators over command torque for static points for the range of temperature 50-65 degrees at a speed of -8000 rpm.

7.2.2 High temperature test

In this section, the results for the high temperature test are shown. The following figures show the sequence of steps in torque applied, first in the positive direction and then in the negative one, for each speed.

Figure 33 shows the pre-heating phase and the legend. As for the low temperature test, the colours are maintained without variations for all the other plots:

- Black: electromagnetic torque calculated by the control system, denominated as "measured torque".
- Blue: estimated torque through equation estimator, called "software estimated torque".
- Red: estimated torque through neural network.
- Green: boundaries defined by the safety requirements.

In this figure, the torque estimated by the software is underestimating because of the effect of the same filter explained for the low temperature case.

Although underestimation is still not completely avoided for the case of the high temperature test, in this case no points are outside the safe limits for more than 100 ms.

Underestimation can be found for low speed and high torque (figures 34 and 38) similarly to the low temperature case. Apart from this, the neural network estimation for all the static points at every speed is satisfactory. Transients do not overcome the safety limits for too long in any point. The closest point to be outside the boundaries are once again the first transients (0.5 and -0.5 of the maximum available torque from zero torque) at 4000 and -4000 rpm, but they do not represent a problem. Again, the underestimation issue can be solved tuning

the offset for a value which allows good results for both high temperature and low temperature tests.

Considering high speed points (8000 rpm and -8000 rpm), the neural network is overestimating more for torque and speeds of the same signs than for working points when speed and torque have opposite direction. This behavior is also present for the test at low temperature, but the difference looks like less marked because for both torque directions and both speed directions the neural network is highly overestimating. This behavior is due to the different characteristic of the induction machine for motor or generator operation at high speeds.

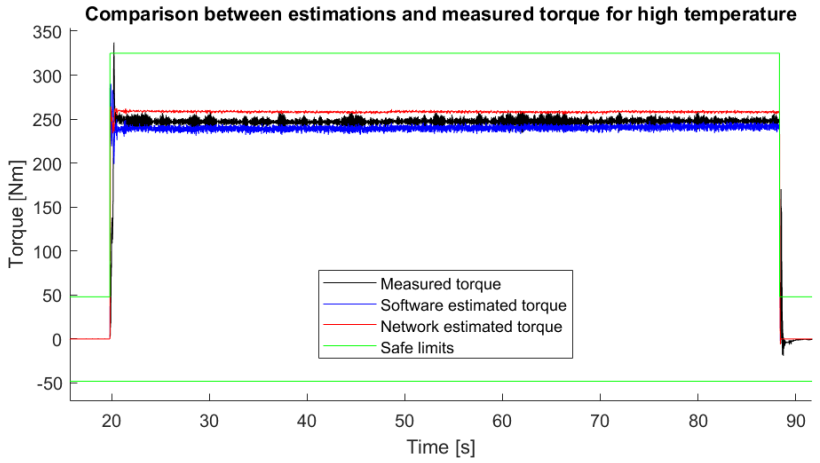

Figure 33: Pre-heating phase of the motor for the high temperature test. The legend shows the colour adopted in all the following pictures related to the high temperature test.

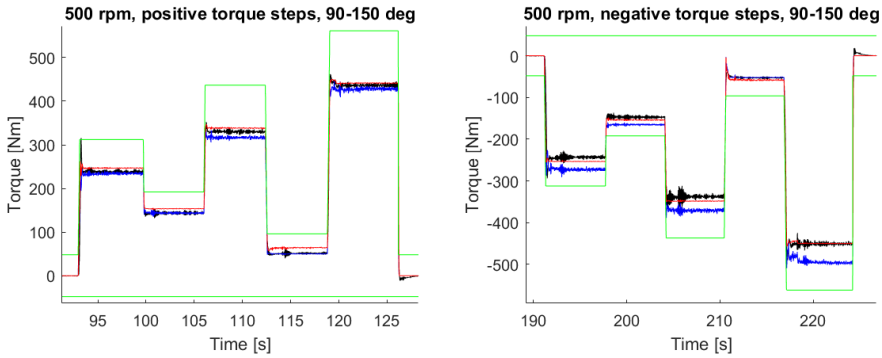

Figure 34: Torque steps in both directions for 500 rpm at the range of temperature 90-105 degrees.

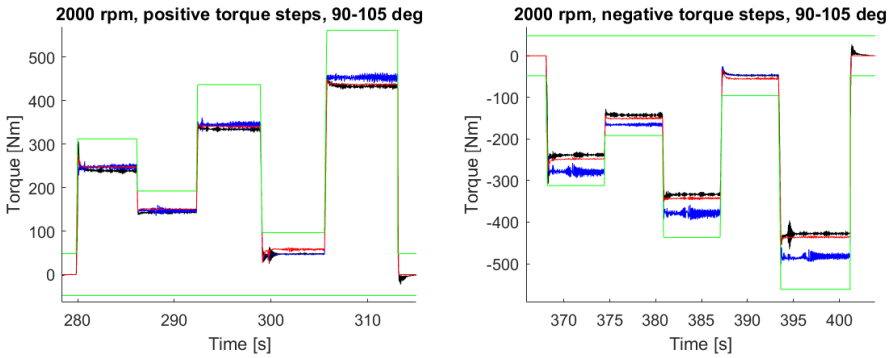

Figure 35: Torque steps in both directions for 2000 rpm at the range of temperature 90-105 degrees.

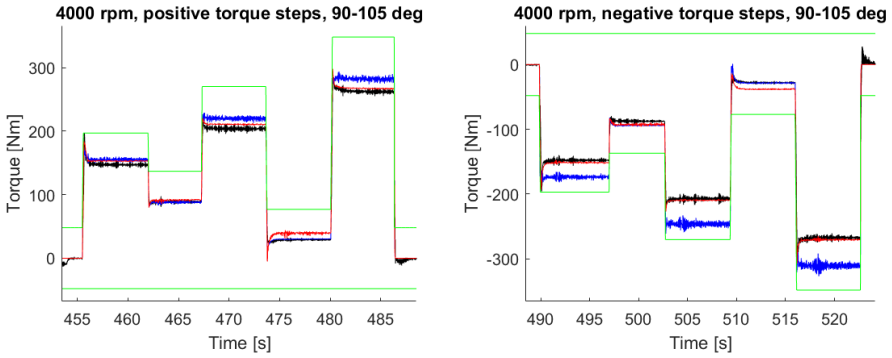

Figure 36: Torque steps in both directions for 4000 rpm at the range of temperature 90-105 degrees.

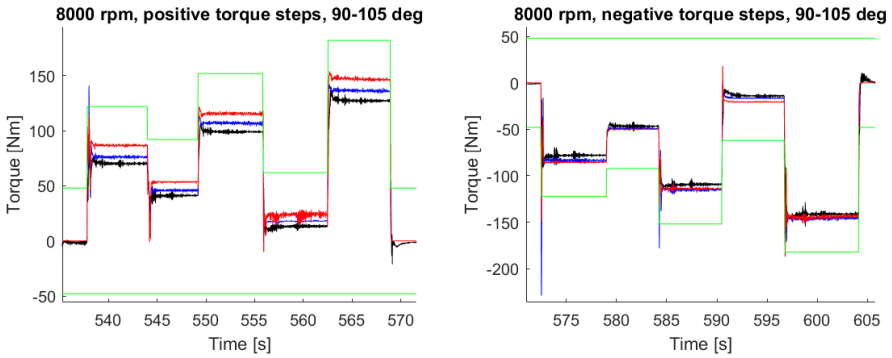

Figure 37: Torque steps in both directions for 8000 rpm at the range of temperature 90-105 degrees.

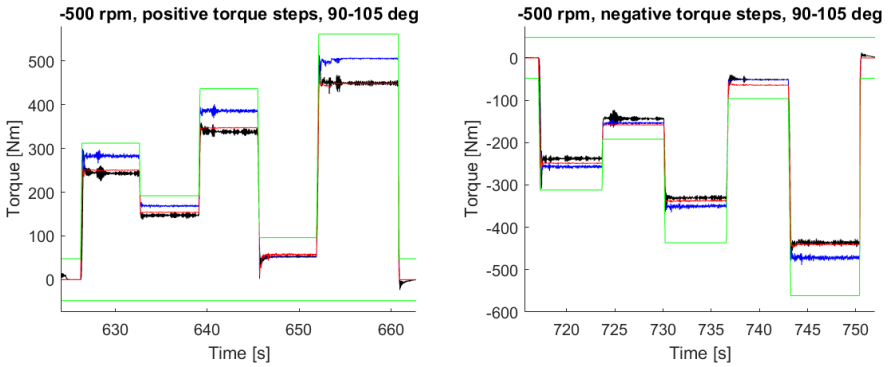

Figure 38: Torque steps in both directions for -500 rpm at the range of temperature 90-105 degrees.

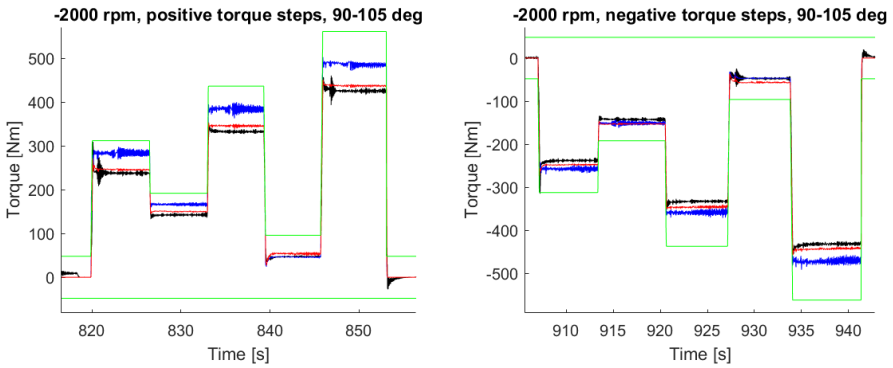

Figure 39: Torque steps in both directions for -2000 rpm at the range of temperature 90-105 degrees.

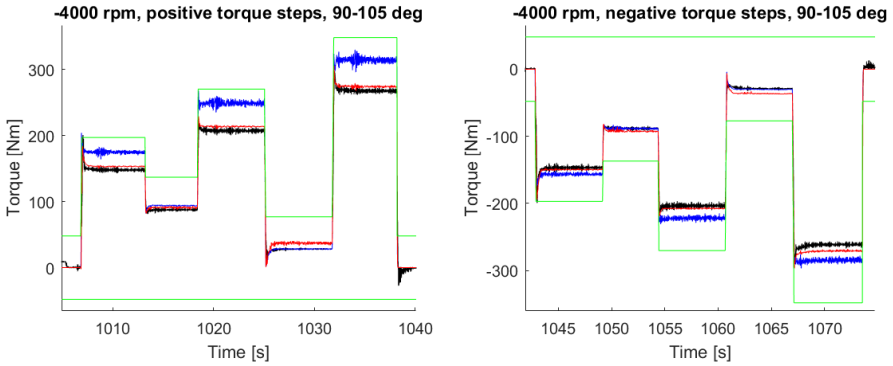

Figure 40: Torque steps in both directions for -4000 rpm at the range of temperature 90-105 degrees.

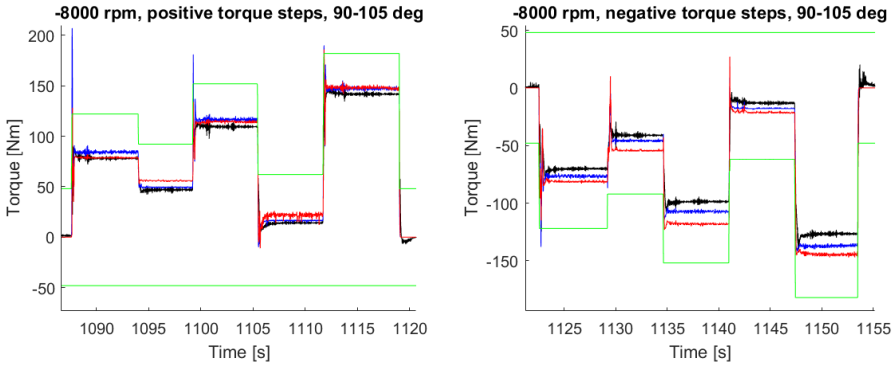

Figure 41: Torque steps in both directions for -8000 rpm at the range of temperature 90-105 degrees.

As for the case of the low temperature test, the following figures represent a comparison between the two different estimators at each speed for static points. Cyan circles indicate the result of the network estimation while blue pluses once again indicate the software estimation.

From Figures 42 - 49 it is possible to affirm that the two estimators have comparable results over the whole set of different torque steps. For low torques, results are more similar, while for high torque the result depends on the speed. Like in the case of low temperature, low speeds show a good accuracy for the network estimation, but this time the software estimation is equally good. For high torque and high speed, as said previously, the result coming from the neural network is overestimating more than the the software estimation.

It is possible to see in table 4 a comparison of the general accuracy of the two estimators over the totality of the test set. For high temperatures, it is easier for the motor to cool down, therefore the points in which the machine has left been resting are less than the case for low temperature.

Comparison in accuracy	Percentage [%]
Neural estimation more accurate than software estimation	23.41
Neural estimation less accurate than software estimation	25.11
Neural estimation as accurate as software estimation	51.48

Table 4: Comparison in accuracy between software estimation and neural network estimation over the whole dataset from the test at high temperature.

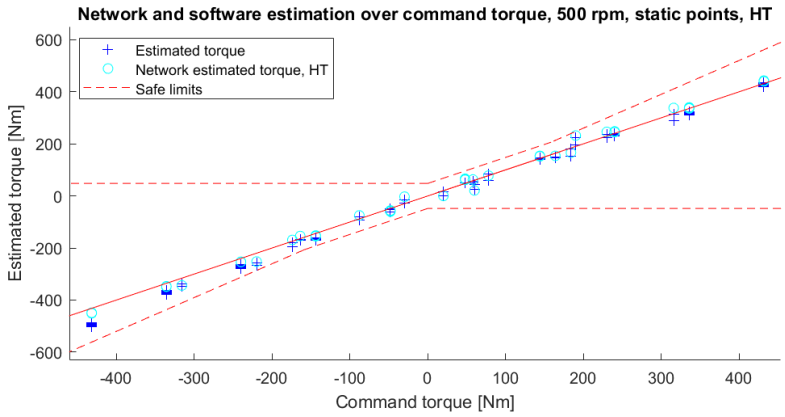

Figure 42: Comparison between estimators over command torque for static points for the range of temperature 90-105 degrees at a speed of 500 rpm.

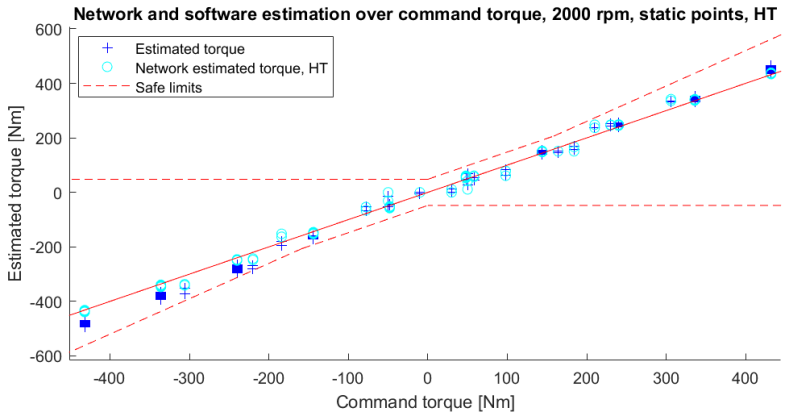

Figure 43: Comparison between estimators over command torque for static points for the range of temperature 90-105 degrees at a speed of 2000 rpm.

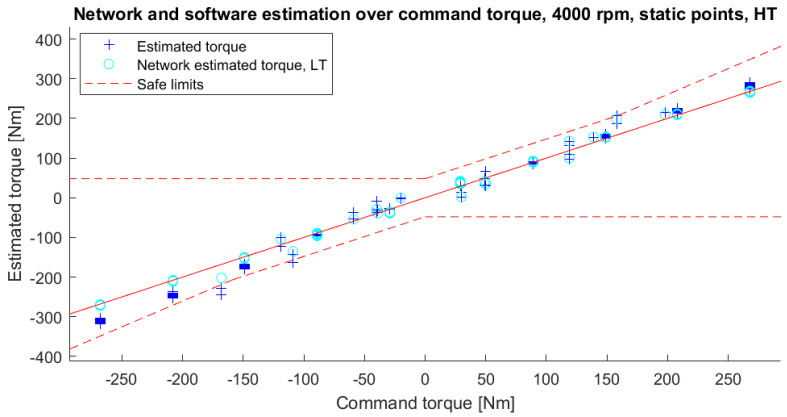

Figure 44: Comparison between estimators over command torque for static points for the range of temperature 90-105 degrees at a speed of 4000 rpm.

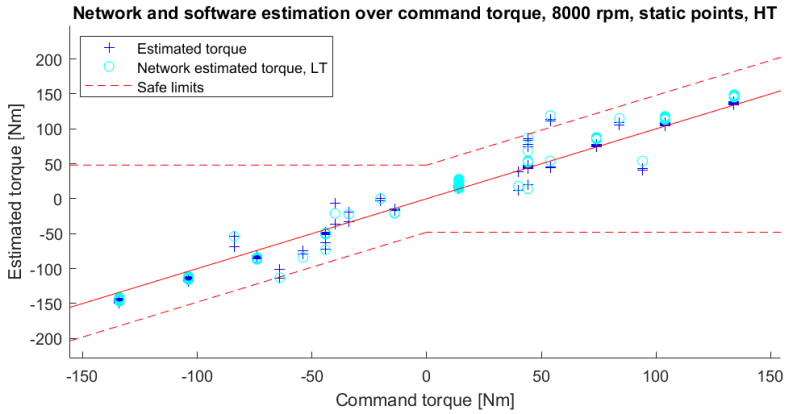

Figure 45: Comparison between estimators over command torque for static points for the range of temperature 90-105 degrees at a speed of 8000 rpm.

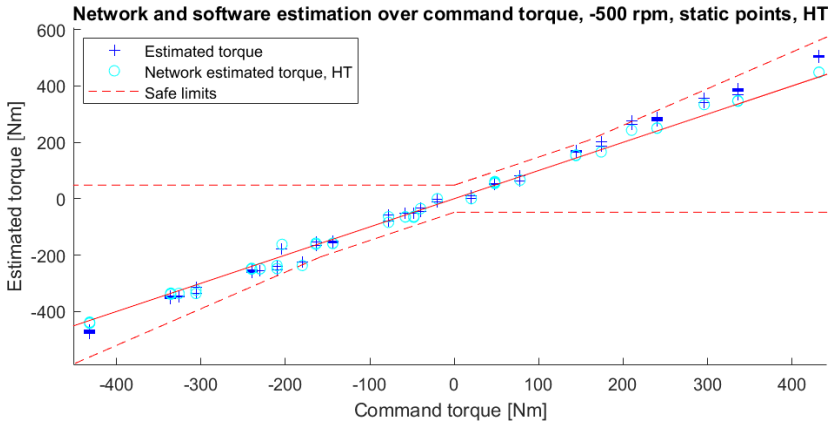

Figure 46: Comparison between estimators over command torque for static points for the range of temperature 90-105 degrees at a speed of -500 rpm.

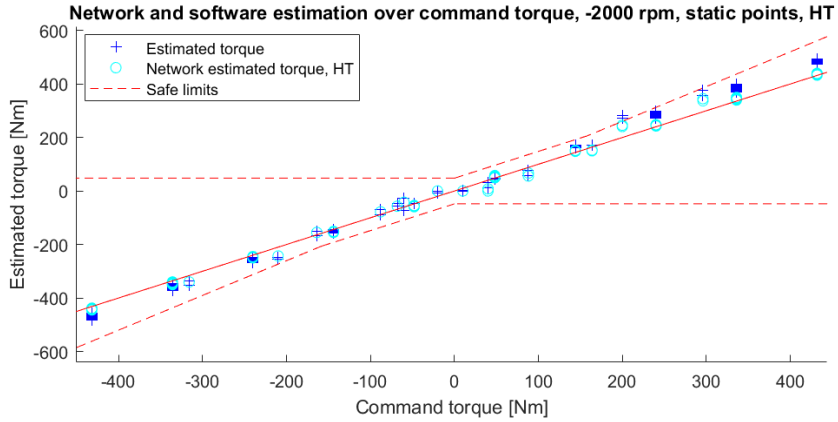

Figure 47: Comparison between estimators over command torque for static points for the range of temperature 90-105 degrees at a speed of -2000 rpm.

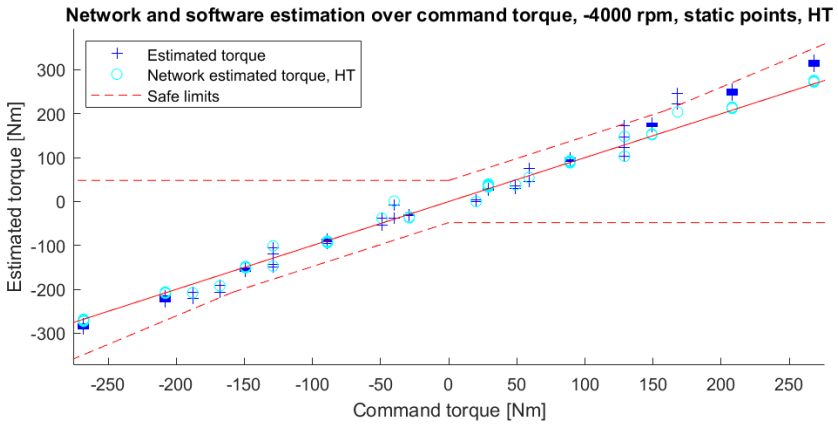

Figure 48: Comparison between estimators over command torque for static points for the range of temperature 90-105 degrees at a speed of -4000 rpm.

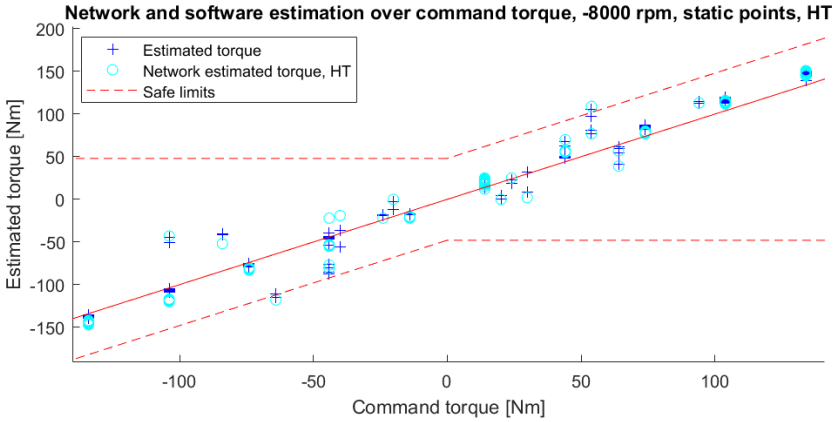

Figure 49: Comparison between estimators over command torque for static points for the range of temperature 90-105 degrees at a speed of -8000 rpm.

7.2.3 Comparison between low and high temperature tests

The following figures show a comparison between software estimation (blue pluses), network estimation for the temperature range 50-65 degrees (green asterisks) and network estimation for the temperature range of 90-105 degrees (cyan circles) for only static points at all the different speeds.

The figures do not show a particular difference between the two network estimation at the two different temperatures for low to medium torque and speed. For high speed, Figures 53 and 57 show that the estimation is generally better for the high temperature test, especially for values of torque close to the

maximum available from the motor. In fact, overestimation at those working points is much larger for the estimation at low temperature.

Even if according to tables 3 and 4 the solution at lower temperature presents generally a slightly better accuracy, it is not capable of fulfilling the safety requirements for some transients. However, the network is able to carry out a satisfactory estimation for both static points and transients for a temperature range of 90-105 degrees. This range is close to the one of the data used to train the network, which was 80-95 degrees.

Comparing Figures 19 and 37 for instance, it is possible to realize that for lower temperatures, the estimation coming from the network shows an increasing behavior. This means that the result is higher for lower temperatures than for higher temperatures, considering the exact same static working point for both cases. This explains why some transients, for the same type of test, overcome the safe limits imposed by safety requirements only for one test and not for the other. Therefore, in order to avoid problems of this kind, a temperature measurement should be used as input to the network.

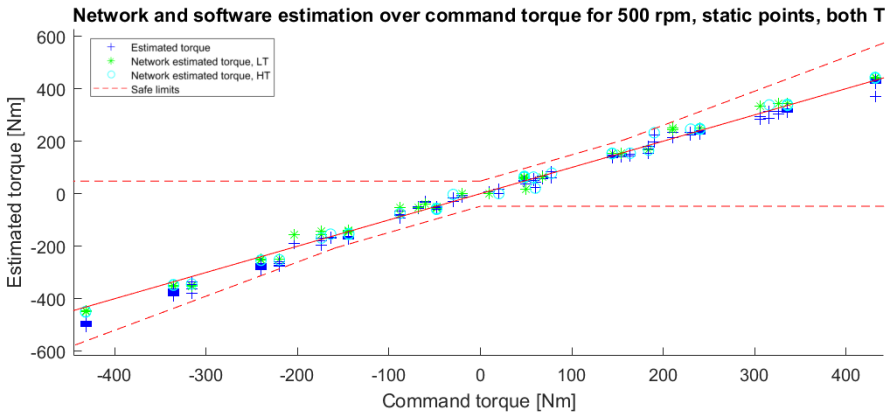

Figure 50: Comparison between estimators over command torque for static points for the two ranges of temperature at a speed of 500 rpm.

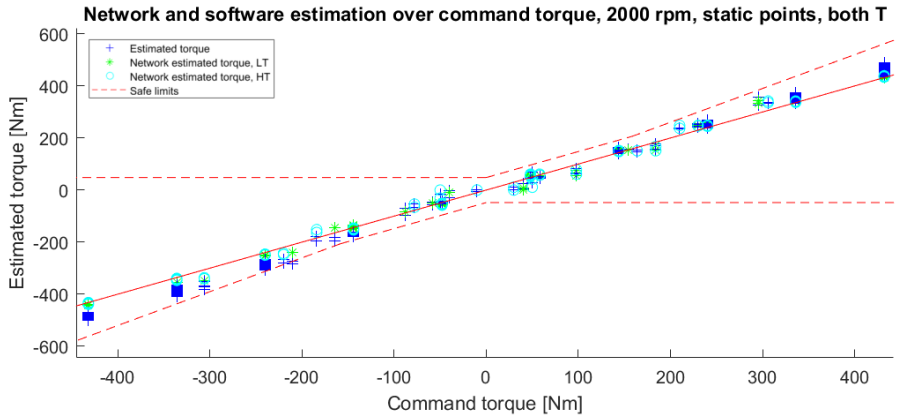

Figure 51: Comparison between estimators over command torque for static points for the two ranges of temperature at a speed of 2000 rpm.

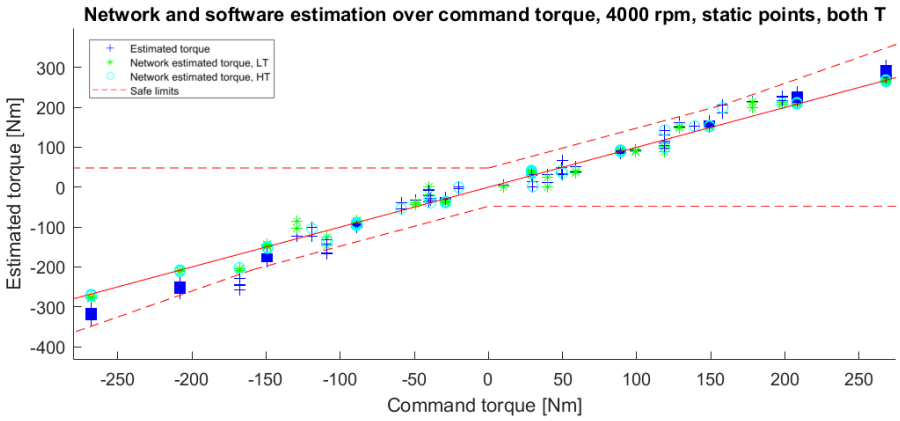

Figure 52: Comparison between estimators over command torque for static points for the two ranges of temperature at a speed of 4000 rpm.

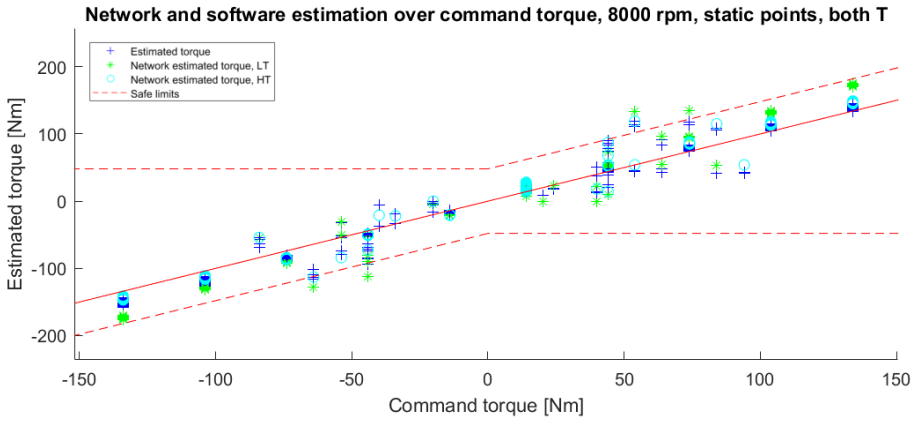

Figure 53: Comparison between estimators over command torque for static points for the two ranges of temperature at a speed of 8000 rpm.

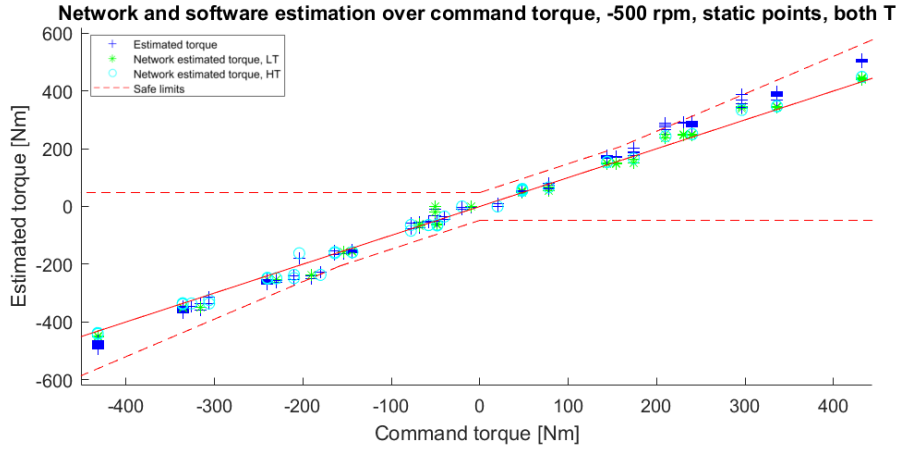

Figure 54: Comparison between estimators over command torque for static points for the two ranges of temperature at a speed of -500 rpm.

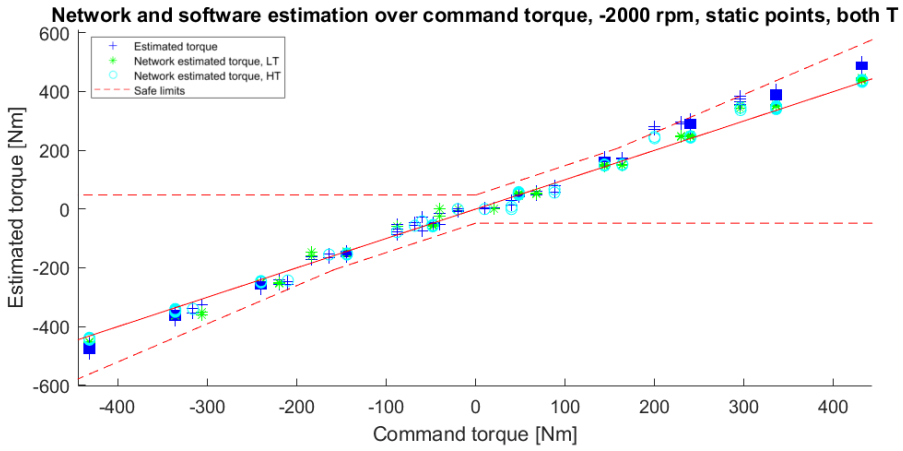

Figure 55: Comparison between estimators over command torque for static points for the two ranges of temperature at a speed of -2000 rpm.

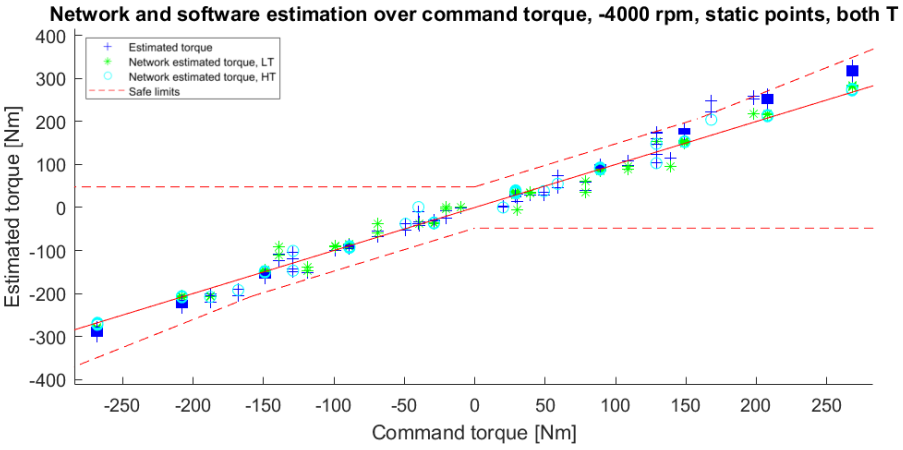

Figure 56: Comparison between estimators over command torque for static points for the two ranges of temperature at a speed of -4000 rpm.

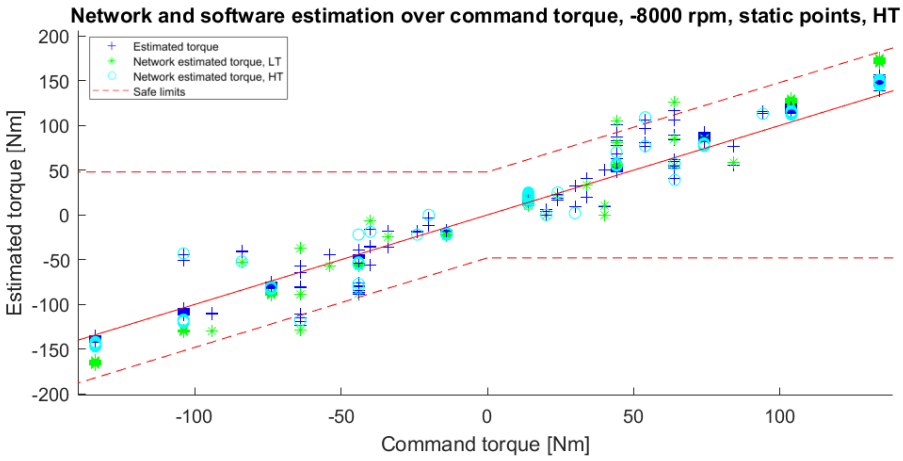

Figure 57: Comparison between estimators over command torque for static points for the two ranges of temperature at a speed of -8000 rpm.

8 Conclusions

This work is meant to present a first analysis about torque estimation with neural networks, evaluating the possibility of such option as implementation in a real system. From the results achieved and presented in the previous chapter, it is possible to draw the following conclusions:

- It is possible to use a neural network as torque estimator for this application.
- The result of the network is dependent on the temperature: for operating points at a similar temperature in respect to the one at which the training data have been collected, the network is able to give satisfactory results, fulfilling all the safety requirements. For lower temperatures, the network has problems in fulfilling safety requirements during transients. A training process which includes the stator temperature could give better results.
- The possibility of implementing the network in the software of the application so that it does not crash depends on the architecture of the network: networks with architecture up to 7 layers with 10 neurons each one and not more than 3 tansig layers (all the rest have to be pure linear layers) can be implemented without any issue. For higher number of layers, neurons or more than 3 tansig layers, the implementation is either possible but too close to the limits of load for the CPU, or impossible.
- For appropriate ranges of temperature, the network is able to carry out results comparable with the current estimator in terms of accuracy.

In addition, it is though necessary to highlights some other considerations. First, the network is loading the CPU much more than a normal estimation based on a normal equation, as specified in table 2. For systems in which the phase voltage measurement is a safe measurement according to ISO standards and the magnetic rotor flux can be calculated, it may be useless to implement such a complex estimator like a neural network. An ordinary estimator based on equation (2) should be accurate enough to prevent safety problems, despite the error introduced by the dependency on the temperature which should be corrected anyway.

Furthermore, in the case the temperature is used as input, the neural function to implement in the system would require slightly more memory because of the additional input. More tests on the board would probably be required, but the impact of the added input should not be large. Therefore, a network capable of giving satisfactory results having the temperature as input and able to stay inside the architectural boundaries previously described could probably represent a better and viable solution.

Finally, it is necessary to remind that the aim of this work is to evaluate the possibility of using a neural network as safe torque estimator in automotive applications and give some guidelines on how this can be done. In order to be able to implement this new type of application, much more testings and

assessments are needed. The network has not being tested for a larger amount of working points because of the lack of time and possibility to run the motor, due to Inmotion customer projects priority.

9 Future studies

After showing with the testings carried out that it is possible to use a neural network as safe torque estimator, it is still necessary to perform more studies to be able to have a fully reliable and proven application.

One possibility would be, as explained, to use a temperature measurement as input to the network. Stator temperature may be sufficient, but what affect the most the electromagnetic torque is actually the rotor temperature. It is unfortunately not possible to measure the rotor temperature directly at every run of the motor, but it is possible to have some measurement through contactless laser sensors. Therefore, a possibility to have a rotor temperature measurement to use as input to the torque neural network could actually be to use another neural network to estimate the rotor temperature from the stator temperature. This solution could probably be acceptable as implementation as well: if the torque neural network has to be run at 500 Hz in order to get torque points enough frequently, this problem does not occur for a temperature evaluation. The time constant of the temperature change is in fact much larger than the time constant of the torque. This means that to evaluate a change in the temperature, the system could run the rotor temperature neural network maybe every second instead of 500 times per second. Therefore, the load on the CPU would not increase dramatically.

Furthermore, the same discussion could be done for the rotor flux: measurement of this element can be taken from the control system. Thus, it could be possible to design a neural network capable of giving as result the rotor flux from safe inputs. This rotor flux could then be used in the torque estimation neural network in order to take into account the effect of the dynamic of the flux on the electromagnetic torque.

A final version of the safe torque estimation through neural network could be the implementation of three different neural networks run at different frequencies according to the constant time of each element they need to calculate. Two of them, calculating rotor flux and rotor temperature, could be used as input for the third one, which would carry out the safe torque calculation.

References

[1] Inmotion Technologies AB,
Solkraftsvägen 13, 135 70, Stockholm, Sweden.
http://en.evs-inmotion.com/

[2] Zapi S.p.A.,
Via Parma 59 - 42028 Poviglio RE, Italy
https://www.zapigroup.com/

[3] Ian Goodfellow and Yoshua Bengio and Aaron Courville, *Deep Learning*
MIT Press, 2016.
http://www.deeplearningbook.org

[4] International Organization for Standardization, *ISO 26262: 2018 Road vehicles – Functional safety*
Phoenix Arizona, 2018.
https://www.iso.org/standard/68383.htm

[5] Inmotion Technologies AB, *Inverter ACH6530*
Solkraftsvägen 13, 135 70, Stockholm, Sweden.
http://en.evs-inmotion.com/products/ach/

[6] ZF Friedrichshafen AG, *AxTrax AVE 130*
https://press.zf.com/press/en/media/media_1506.html

[7] Tom Mitchell, *Machine Learning*
McGraw-Hill, 1997.

[8] Wikipedia, *Lipschitz continuity*
https://en.wikipedia.org/wiki/Lipschitz_continuity

[9] Kurt Hornik, *Neural Networks*
Pergamon Press, 1991
https://www.sciencedirect.com/science/article/abs/pii/
089360809190009T?via\%3Dihub

[10] Jacobs, R. A., *Increased rates of convergence through learning rate adaptation, Neural Networks*
1988

[11] The MathWorks Inc, *MATLAB 9.7.0.1190202 (R2019b)*
Natick, Massachusetts, 2018

[12] The Apache Software Foundation, *Apache Subversion 1.13.0*
Retrieved 26 July 2019
https://subversion.apache.org/