

# POLITECNICO DI TORINO

Master's Degree in Computer Engineering



Master's Degree Thesis

## Calibration of atmospheric pollution sensors using quantum machine learning techniques

Supervisors

Prof. Maurizio REBAUDENGO

Prof. Bartolomeo MONTRUCCHIO

Candidate

Michele Luigi GRECO

ACADEMIC YEAR 2020-2021



## **Abstract**

The need to efficiently process large amounts of data has led information technologies to migrate to the world of Machine Learning, to be able to extract information with a variable degree of accuracy, but still quite high. The use of Quantum Computing for Machine Learning occurs in a similar, but not entirely, scenario, as the amount of data that can be processed tends not to scale easily. The purpose of this thesis is to exploit these new computing paradigms to analyze and calibrate sensors for measuring air pollution, in particular PM-2.5 and PM-10, this is one of the most interesting potential applications of Quantum Technologies: Machine Learning for the purpose of prediction. Furthermore, it is shown that some problems that are generally difficult to calculate can be easily processed by Classical Machine Learning, which is trained on one part of the initial data-set and then is tested on the rest. As a starting example, a case study on the prediction of the price of a house in the classical version is analyzed and then this example is extended to the quantum version. Using classical prediction algorithms such as Linear Regression as a basis, the next intent is to develop a methodology for evaluating the potential quantum advantage in learning and predicting correct values for air pollution detection sensors. The results are promising, compared to classical methodologies, the quantum one is able to calibrate better, with a percentage error almost always below the 10% threshold.

# Summary

The need to efficiently process large amounts of data has led information technology to migrate to the world of Machine Learning, to extract information with a variable degree of accuracy, inasmuch the Machine Learning is a compromise between the amount of data that can be processed within a reasonable time and the quality of the obtained results of the analysis process. Quantum Computing presents new techniques to approach ML problems, with promising prospects, but also with some limitations due to its early stage of maturity. The main purpose of this thesis is to exploit this new IT paradigm, namely, Quantum Machine Learning, for one of its most interesting applications: the prediction of values. These potentialities will be applied to the field of air pollution monitoring, specifically this method will be exploited for the calibration of the PM10 and PM2.5 sensors on the basis of a reference given to us by ARPA, a government agency, and taking into account the values of temperature and relative humidity. Machine learning can offer an efficient alternative to the conventional engineering scheme, i.e. the analysis of a complex problem in all its parts, which typically leads to a high development cost and to the lengthening of resolution time, precisely when the costs, time and complexity of analysis and development are the key elements. On the other side, the approach has the disadvantages of providing generally suboptimal performance, or limiting the interpretability of the solution, and can only be applied to a limited set of problems, which can be grouped into the following tasks:

- **Supervised learning:** a learning function that maps an input to an output based on example input-output pairs.
- **Unsupervised learning:** provides the information system with a series of inputs that it will reclassify and organize on the basis of common characteristics to try to make reasonings and forecasts on subsequent inputs.
- **Reinforcement learning:** aims to create autonomous agents able to choose the actions to be taken to achieve certain objectives through interaction with the environment in which they operate.

The task tackled in this thesis falls into the first category: supervised learning.

The classical reference in this specific case is the Linear Regression algorithm. Focusing on Linear Regression with Classic Machine Learning, what is covered next are various optimization algorithms, in particular the one considered the most suitable for our purpose, based on the literature and on functional tests, is Adaptive Moment Estimation (ADAM), which is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. [1] Focusing on Quantum Machine Learning, it was decided to make an overview of the characteristics of quantum systems, starting from the mathematical description of the Quantum Gates, moving on to the neurons and weights definitions, and then talking about FFNN (Feed-Forward Neural Network).

Next step is to extend the concept of Linear Regression to the quantum world, formulating the problem and adapting the cost function to the new programming environment. Before proceeding with the description of the implemented algorithm, an introduction to the existing project at the DAUIN research laboratory is performed. The Weather Station Analysis project aims to understand if it is possible with much cheaper devices than the high-end ones of governmental entities (for example, ARPA, that will be the reference in our case), to proceed with the analysis of atmospheric pollution by monitoring various climatic factors and not. The variables of interest are: Pm2.5, Pm10, Temperature, Relative Humidity.

In this context the central role is done by the calibration of the sensor, that is, to find a mathematical function that, given a value  $x$ , of a given sensor belonging to a given board, is able to return  $\hat{x}$ , i.e. the correct value that is closest to the  $y$  reference value provided by ARPA. The existing calibration methods are: Linear Regression and Random Forest in a Classical Machine Learning way.

In order to verify how much the implemented algorithms are efficient, a validation class has been used. This class is able to, given the access to the database containing the raw data, given the algorithms implemented for calibration, to return a series of metrics capable of facilitating the critical reading of the results:

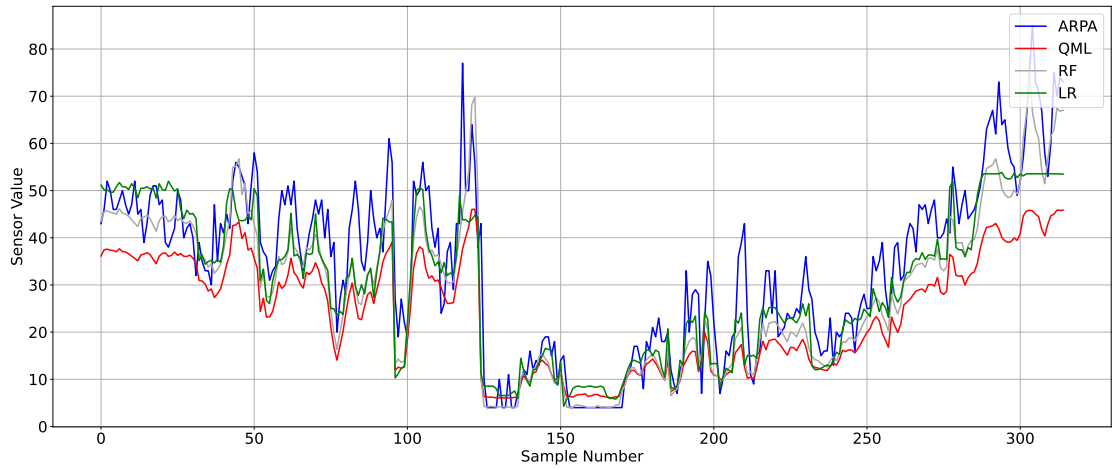
- The correlation index between the raw values and the ARPA reference.
- The correlation index between the calibrated values and the ARPA reference.
- The Gaussian error distribution of the raw and calibrated values.
- The violin plots of MAE, MSE, RMSE, CORR and R2SCORE.

The central topic of the thesis is how the classical model of Linear Regression can be related to the world of Quantum Machine Learning through a hybrid network which structure can be summarized with the following scheme: as first a classical level, then a quantum level placed in the middle, finally the other classical level. The number of qubits, or quantum bits, that is the unit of quantum information, due to the simple calculus performed, is fixed at 1.

In order to understand how the network is set up, it is useful to introduce the fundamental concept of Quantum Embedding: how data is represented as quantum states in a Hilbert space via a quantum feature map. Different configurations of the hybrid model, with different Learning Rates, Number of Epoch and Batch Size, have been analysed and tested. The tuned configuration is finally compared to the classical version of the same algorithm.

Results are quite similar to the one obtained with the classical method taken for comparison, such as Linear Regression and Random Forest. Slightly higher in terms of accuracy, the QML method has an average of 89-90% in terms of correlation index, while the counterparts LR and RF reach 89-91%. As can be seen from Fig. 1 the calibration process using QML is promising, the deviation from the reference, in most cases remains below the threshold set for 10%.

As we can see, with respect to the time required to execute the algorithm, it



**Figure 1:** Comparison between different method of calibration and ARPA reference

is not possible to conclude that the Quantum-Classical Hybrid Network brings an advantage in terms of time since, due to the strong slowness in running the simulations on the IMB's QPUs, through the Qiskit API for PennyLane, we preferred to use a local simulator. However, a simulation carried out directly on a QPU is reported at the end of the thesis for comparison.

# Acknowledgements

*Spero di non dimenticare nessuno :)*

Grazie **ai miei genitori, Rocco e Maria**, che hanno sempre creduto in me, che non mi hanno mai permesso di arrendermi ad ogni ostacolo che mi si è presentato in questo viaggio. Grazie a voi, che avete contribuito a rendere la passione di un bambino che si divertiva a smontare computer in cameretta un longimirante futuro. Grazie **a mia sorella, Benedetta**, che so dal profondo del mio cuore quanto ci tenga me. Grazie **ai miei nonni, Annunziata P., Anna G., Luigi L., Michele G., Luigi S.**, che siate ancora al mio fianco o che mi stiate guardando da lassù, vi porterò sempre nel cuore. Grazie **zio Gianni**, penso che tutta la mia passione per l'informatica sia anche merito tuo. Grazie **zia Marivelia**, ricordo ancora quel giorno in quella sala quando ti laureasti, ricordo le parole "voglio superarti con il mio voto di laurea", grazie per avermi fatto capire quel giorno che nella vita la tenacia è tutto.

Grazie **Prof. Rebaudengo** per ogni occasione che mi ha dato per dimostrarle quanto valessi, per tutti i progetti del Laboratorio di Ricerca al quale ho potuto prender parte, grazie per avermi fatto amare ancora di più l'informatica e soprattutto il politecnico, sin dal primo giorno del primo anno della Laurea Triennale, quando, ricordiamolo, il Torino perse con la Juventus per 4 a 0.

Grazie **Vivi**, per avermi sostenuto in ogni momento, per essere stata al mio fianco, per esser stata sempre disposta ad ascoltarmi ore ed ore parlare dei miei argomenti di studio, del mio lavoro, della mia routine. Ricordo quella volta di fronte al poli, quando la lezione iniziava sempre tra cinque minuti, e poi quei cinque minuti erano eterni. Grazie per aver ascoltato ogni singola lamentela, perchè si sa, a lamentarsi tutti son bravi, ma miky non lo batte nessuno. Grazie per ogni singolo gesto d'amore, per ogni cazziata, per ogni *miky devi studiare*, sei e sarai sempre nel mio cuore, e in quello di poppy.

Grazie **Giuseppe**, il mio miglior amico, senza te penso che avrei fatto molte più cazzate. Grazie per ogni telefonata alla quale eri sempre pronto a rispondere, per ogni sveglia suonata alle 7.00 di mattina ad Agosto, quando dopo una notte insonne studiare Chimica e Fisica 1 poteva solo accompagnare. Grazie per la tua fiducia, grazie per essere l'altra metà della mia coscienza. Ma soprattutto grazie di non essere andato all'estero questa estate così possiamo festeggiare la Laurea.

Grazie **Paolina**, sei stata l'amica più importante dei miei anni universitari, con te ho vissuto ogni singolo esame, ogni singola emozione che il poli e non potessero regalarci. Ricordo le notti a studiare per Sonza, quando tra un sorso di Martini Rosato e GOT c'era sempre quel tema d'esame della Baralis che mi convincevi a fare, ricordo quando il programma non compilava, perchè c'era JUMP FINE, e quello giustamente andava a FINE, ricordo i sensi di colpa, quando dovevamo studiare e invece Piazza Vittorio era sempre più magica in quelle notti d'inverno. Grazie per esser sempre stata al mio fianco, ti voglio un bene dell'anima. P.S. mena a laurearti che ho pronto l'affondo-scherzo perfetto. Heheh.

Grazie **Edoardo**, grazie collega, grazie dottorando, grazie amico, sei stata una delle persone più strane, in senso positivo eh!, che abbia conosciuto al politecnico, ne abbiamo combinate di cotte e crude, quando quelle serete dietro ai sensori RFID finivano con pizza e cazziatone alle 22.00 al poli... Grazie Edo, prometto, prima o poi mi convincerò che Java è il malessere del mondo.

Grazie a tutti i miei **compagni universitari**, ai miei amici, Klaus, Iustin, Matteo, Alberto C., Enrico, Federica, Alessandro, Alberto B., Davide, Domenico, Luca G., Rosario, Gabriel. Alcuni di voi sono stati miei colleghi fino in fondo, altri hanno preso strade diverse, per corsi di laurea diversi, grazie per aver reso ogni lezione meno noiosa, grazie soprattutto per avermi tenuto il posto quando arrivavo in ritardo.

Grazie ai miei **colleggi di Lavoro**, che mi hanno sopportato, quando invece di dedicarmi alle loro parole rispondevo "si si" e continuavo a chiedermi perché il codice python della tesi non compilasse. Un grazie particolare al mio **collega di Cuneo**, Enrico, che da 10 mesi a questa parte mi sopporta in ufficio, e che da giorni, inoltre, ha deciso che l'aria di Cuneo è bene che sia sentita per tutta la giornata lavorativa. Grazie, quindi, anche all'aria di Cuneo, che mi ha permesso di tenere ben freschi i neuroni, quando in realtà si sarebbero voluti squagliare.





*“Your work is going to fill a large part of your life,  
and the only way to be truly satisfied is to do what you believe is great work.  
And the only way to do great work is to love what you do.  
If you haven’t found it yet, keep looking. Don’t settle.  
As with all matters of the heart, you’ll know when you find it.”*  
Steve Jobs



# Table of Contents

<b>List of Tables</b>	XIII
<b>List of Figures</b>	XIV
<b>Acronyms</b>	XVII
<b>1 Introduction</b>	1
1.1 What is Machine Learning . . . . .	1
1.2 When to Use Machine Learning . . . . .	2
1.2.1 Learning Tasks . . . . .	3
1.3 Introduction to Linear Regression . . . . .	5
1.3.1 Statistical inference . . . . .	6
1.3.2 Frequentist Approach . . . . .	7
1.3.3 Taxonomy of solutions . . . . .	7
1.3.4 Linear Regression (LR) Model . . . . .	8
1.3.5 Discriminative vs. Generative Probabilistic Model . . . . .	8
1.3.6 Model Order and Model Parameters . . . . .	10
1.3.7 Overfitting and Underfitting Problem . . . . .	10
1.3.8 Influence of dataset size . . . . .	11
1.3.9 Regularization . . . . .	13
1.4 Optimizers and model accuracy . . . . .	14
1.4.1 Gradient Descent . . . . .	15
1.4.2 Gradient Descent Algorithm . . . . .	16
1.4.3 Adaptive Moment Estimation (ADAM) . . . . .	17
1.4.4 Adam Configuration Parameters . . . . .	18
1.5 Overview . . . . .	20
<b>2 Quantum Machine Learning</b>	21
2.1 What is Quantum Machine Learning . . . . .	21
2.1.1 Example of Quantum Gates . . . . .	22
2.1.2 Neurons and Weights . . . . .	23

2.1.3	Feed Forward Neural Network . . . . .	23
2.1.4	IO Structure of Layers . . . . .	24
2.1.5	Building the network . . . . .	24
2.2	Linear Regression with QML, case study . . . . .	25
2.2.1	Problem formulation . . . . .	25
2.2.2	Definition of parametrized Cost Function J . . . . .	26
2.2.3	Gradient descent algorithm in multivariable version . . . . .	27
2.2.4	Implementation in Python . . . . .	27
2.2.5	Results . . . . .	28
<b>3</b>	<b>Air pollution sensors Project</b>	<b>30</b>
3.1	Introduction . . . . .	30
3.2	Purpose of the project . . . . .	31
3.3	Role of Sensors Calibration . . . . .	31
3.3.1	Calibration Strategies . . . . .	32
3.4	Current Calibration process . . . . .	33
3.5	Validation process . . . . .	34
3.5.1	Validation metrics . . . . .	34
3.5.2	The Validation Process . . . . .	35
<b>4</b>	<b>Calibration of air pollution sensors</b>	<b>38</b>
4.1	The idea of a hybrid network . . . . .	38
4.1.1	The concept of Quantum embedding . . . . .	38
4.2	Implementation of the algorithm . . . . .	40
4.2.1	Usefully library import . . . . .	40
4.2.2	Hybrid quantum-classical layer . . . . .	40
4.2.3	Import of the dataset for training and testing calibration . . . . .	42
4.2.4	Training neural network . . . . .	42
4.2.5	Sensor calibration and Loss analysis . . . . .	43
4.2.6	Evaluation of the error percentage . . . . .	44
4.2.7	Example of <code>ws_analysis_qml.py</code> use . . . . .	44
<b>5</b>	<b>Tuning, Simulations, and Results</b>	<b>46</b>
5.1	Tuning of hyperparameter . . . . .	46
5.1.1	Tuning Number of epoches . . . . .	48
5.1.2	Tuning Learning Rate and Batch Size . . . . .	54
5.1.3	Tuning overall results . . . . .	59
5.2	Validation of calibration strategy . . . . .	60
5.2.1	PMs and Board Selection . . . . .	60
5.2.2	Percentile Optimizer range . . . . .	60
5.2.3	Time window for calibration and evaluation . . . . .	60

5.2.4	Xticks parameters . . . . .	61
5.2.5	calKinds . . . . .	61
5.2.6	Metrics and Coefficients . . . . .	62
5.3	Validation result phase . . . . .	63
5.3.1	Mean absolute error results analysis . . . . .	64
5.3.2	Mean Square Error results analysis . . . . .	65
5.3.3	Root Mean Square Error results analysis . . . . .	66
5.3.4	Correlation coefficient results analysis . . . . .	67
5.3.5	Correlation coefficient results, violin plots analysis . . . . .	68
5.3.6	Coefficient of determination results analysis . . . . .	70
5.4	Execution time analysis . . . . .	71
5.5	Results on IBM QPU . . . . .	73
<b>6</b>	<b>Conclusions and further improvements</b>	<b>75</b>
6.1	Conclusions . . . . .	75
6.2	Further improvements . . . . .	76
6.2.1	MultiThread Class Possible Implementation . . . . .	76
	<b>Bibliography</b>	<b>77</b>

# List of Tables

2.1	Case study data-set, original length was 450 lines, here an example of few lines. . . . .	29
3.1	Comparing different calibration techniques over different calibration periods (2 and 12 weeks, respectively), adopting the RMSE (Root Mean Square Error) metric. We consider all stationary sensors. . .	37
5.1	Hyperparameters tuning phase proposed value . . . . .	46
5.2	Summary of obtained results of tuning number of epoches . . . . .	53
5.3	Number of epoches selected for each configuration proposed . . . . .	53
5.4	Learning Rate and Batch Size selected for each configuration proposed with metrics indicator . . . . .	59
5.5	Final hyperparameter choiche for each configuration proposed . . .	59
5.6	Quantile range configurations . . . . .	61
5.7	Calibration method proposed for validation process . . . . .	61
5.8	Metrics for the validation of calibration method . . . . .	62
5.9	Constants for time analysis . . . . .	72
5.10	Execution time for each proposed configuration . . . . .	72

# List of Figures

1	Comparison between different method of calibration and ARPA reference . . . . .	iv
1.1	The Machine Learning approach, automatically adapting to change. [3] . . . . .	2
1.2	Example of a training set $\mathcal{D}$ with $N = 10$ points $(x_n, t_n), n = 1, \dots, N$ . [2] . . . . .	5
1.3	Illustration of underfitting and overfitting in ML learning: The dashed line is the optimal predictor (2.10), which depends on the unknown true distribution, while the other lines correspond to the predictor $t_{ML}(x) = \mu(x, w_{ML})$ learned via ML with different model orders $M$ . [2] . . . . .	11
1.4	Square root of the generalization loss $L_p(w_{ML})$ and of the training loss $L_{\mathcal{D}}(w_{ML})$ as a function of the training set size $N$ . The asymptote of the generalization and training losses is given by the minimum generalization loss $L_p(w^*)$ (cf. (2.21)) achievable for the given model order (see Fig. 2.5). [2] . . . . .	12
1.5	Comparison of Adam to Other Optimization Algorithms Training a Multilayer Perceptron [1] . . . . .	18
2.1	Representation of qubit states, unitary gates and measurements in the quantum circuit model and in the matrix formalism [17] . . . .	22
2.2	Simple structure of a hybrid quantum-classical neural network[17] .	24
3.1	Architecture of the project. The data coming from the sensors are first stored in Raspberry Pi, and then transferred to a remote server over the Wi-Fi network.[21, 3] . . . . .	31
3.2	Different kind of calibration methods used [21] . . . . .	33
3.3	Validation process $R^2$ [21] . . . . .	35



3.4	Time series comparing the reference, the raw, and the calibrated data using sensor 34, randomly selected. Calibration is performed using MLR. The different plots show MLR with different dependent variables, namely temperature (a), humidity (b), and temperature plus humidity (c).[21]	36
4.1	Example of $y$ -rotation $R_y(v)$ by an angle $v$ around the $y$ axis, that is, in the $x$ - $z$ -plane.[25]	39
5.1	Results Score graph, configuration 1: PM2.5 - offQuantile	49
5.2	Errors Score graph, configuration 1: PM2.5 - offQuantile	49
5.3	Results Score graph, configuration 2: PM10 - offQuantile	50
5.4	Errors Score graph, configuration 2: PM10 - offQuantile	50
5.5	Results Score graph, configuration 3: PM2.5 - onQuantile	51
5.6	Errors Score graph, configuration 3: PM2.5 - onQuantile	51
5.7	Results Score graph, configuration 4: PM10 - onQuantile	52
5.8	Errors Score graph, configuration 4: PM10 - onQuantile	52
5.9	Results Score heatmap, configuration 1: PM2.5 - offQuantile	55
5.10	Errors Score heatmap, configuration 1: PM2.5 - offQuantile	55
5.11	Results Score heatmap, configuration 2: PM10 - offQuantile	56
5.12	Errors Score heatmap, configuration 2: PM10 - offQuantile	56
5.13	Results Score heatmap, configuration 3: PM2.5 - onQuantile	57
5.14	Errors Score heatmap, configuration 3: PM2.5 - onQuantile	57
5.15	Results Score heatmap, configuration 4: PM10 - onQuantile	58
5.16	Errors Score heatmap, configuration 4: PM10 - onQuantile	58
5.17	Comparing MAE results on QML, LR, RF	64
5.18	Comparing MSE results on QML, LR, RF	65
5.19	Comparing RMSE results on QML, LR, RF	66
5.20	Comparing CORR results on QML, LR, RF	67
5.21	QML Correlation coefficient violin plots	68
5.22	LR Correlation coefficient violin plots	68
5.23	RF Correlation coefficient violin plots	69
5.24	Comparing $R^2$ results on QML, LR, RF	70
5.25	QML simulation results on IBM Quantum, PM 2.5	74
5.26	QML simulation results on IBM Quantum, PM 10	74



# Acronyms

**AI**

artificial intelligence

**AdaGrad**

Adaptive Gradient Algorithm

**Adam**

Adaptive Moment Estimation

**CORR**

Correlation coefficient

**CRMSE**

Corrected Root Mean Square Error

**DAG**

Directed Acyclic Graph

**FFNN**

Feed-Forward Neural Network

**GMM**

Gaussian mixture model

**LASSO**

Least Absolute Shrinkage and Selection Operator

**LR**

Linear Regression

**MAE**

Mean absolute error

**MAP**

Maximum a Posteriori

**MBE**

Mean Bias Error

**MDL**

Minimum description length

**MLR**

Multivariate Linear Regression

**ML**

Maximum Likelihood

**MSE**

Mean Square Error

**NFL**

No free lunch theorem

**R2SCORE**

Coefficient of determination

**RF**

Random Forest

**RMSE**

Root Mean Square Error

**RMSProp**

Root Mean Square Propagation

**RSS**

Residual Sum of Squares

# Chapter 1

## Introduction

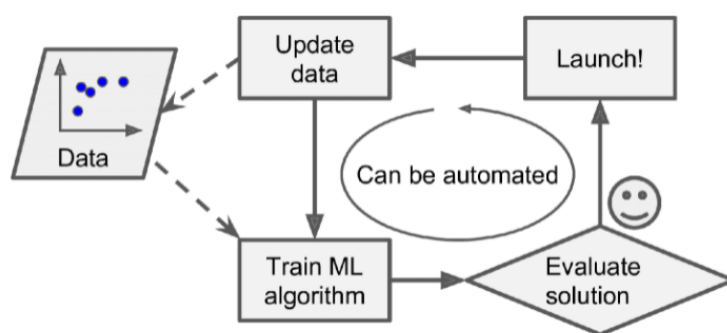
In the first part, it is introduced the main concepts of Machine Learning, and how the basic techniques that will be used in this thesis are implemented, in both an algorithmic and mathematical way.

### 1.1 What is Machine Learning

To start the treatment with machine learning methodology, we can start by exploiting the comparison with the classic scheme of conventional engineering. The process scheme starts with an in-depth analysis of the problem domain, which ends with the definition of a mathematical model. The purpose of the mathematical model is to capture the key features of the problem under study, and is typically the result of the work of a number of experts. The model just outlined is then used to obtain a solution starting from existing or from scratch solutions. As an example, consider the problem of defining a chemical process to produce a given molecule. An example is the design of speech translation or image / video compression algorithms. Both of these problems involve the definition of models and algorithms by teams of experts, such as linguists, psychologists, and signal processing practitioners, not infrequently during the course of long standardization meetings.

The engineering design model described above may be too costly and inefficient for problems in which faster or less expensive solutions are desired. The machine learning alternative solution is based on collecting large data sets, e.g., of labeled speech, images, or videos, and to use this information to train general-purpose learning machines to carry out the desired task.

While the standard engineering scheme is based on domain knowledge and on the design optimized for the problem, machine learning lets large amounts of data to set the base for algorithms and solutions. To this purpose, rather than



5. the task does not require detailed explanations for how the decision was made;
6. the task has a tolerance for error and no need for provably correct or optimal solutions;
7. the phenomenon or function being learned should not change rapidly over time; and
8. no specialized dexterity, physical skills, or mobility is required.

These criteria are useful key points for the decision of whether machine learning methods are suitable. They also offer a line of differentiation between machine learning as it is understood today, with its focus on training and computational statistics tools, and more general notions of AI (artificial intelligence) based on knowledge and common sense [6].

### 1.2.1 Learning Tasks

As learned from the literature [7, 8], we can distinguish between three different main types of machine learning problems, which are briefly introduced below.

1. **Supervised learning:** We have  $N$  labelled training examples  $\mathcal{D} = \{(x_n, t_n)\}_{n=1}^N$  where  $x_n$  represents a covariate, or explanatory variable, while  $t_n$  is the corresponding label, or response. For example, variable  $x_n$  may represent the text of an SMS, while the label  $t_n$  may be a binary variable indicating whether the SMS is spam or not. The goal of supervised learning is to predict the value of the label  $t$  for an input  $x$  that is not in the training set. However, supervised learning purpose is to generalizing the observations in the data set  $\mathcal{D}$  to new inputs. For example, an algorithm trained on a set of SMS should be able to classify a new SMS not present in the data set  $\mathcal{D}$ .

We can generally distinguish between classification problems, in which the label  $t$  is discrete, as in the example above, and regression problems, in which variable  $t$  is continuous.

An example of a regression task is the prediction of tomorrow's air pollution  $t$  based on today's air pollutions observations  $x$ . An effective way to learn a predictor is to identify from the data set  $\mathcal{D}$  a predictive distribution  $P(t|x)$  from a set of parametrized distributions. The conditional distribution  $P(t|x)$  defines a profile of beliefs over all possible of the label  $t$  given the input  $x$ . For instance, for air pollution prediction, one could learn mean and variance of a Gaussian distribution  $P(t|x)$  as a function of the input  $x$ . As a special case, the output of a supervised learning algorithm may be in the form of a deterministic predictive function  $t = \hat{t}(x)$ .

**2. Unsupervised learning:** Let consider now that we have an unlabelled set of training examples  $\mathcal{D} = \{(x_n)\}_{n=1}^N$ . Defined with a lesser degree of precision than upervised learning, unsupervised learning generally refers to the purpose of learning properties of the mechanism that generates this data set. Specific tasks and applications include clustering, which is the problem of grouping similar examples  $x_n$ ; dimensionality reduction, feature extraction, and representation learning, all related to the problem of representing the data in a smaller, better organized, or more readable space depending on the desired interpretation of the data. Finally its fundamental to notice that generative modeling is also the problem of learning a generating mechanism to produce artificial examples that are similar to available data in the data set  $\mathcal{D}$ . As a generalization of both supervised and unsupervised learning, semi-supervised learning refers to scenarios in which not all examples are labelled, with the unlabelled examples providing information about the distribution of the covariates  $x$ .

**3. Reinforcement learning:** Reinforcement learning refers to the problem of take optimal sequential decisions based on rewards or penalty received as a result of previous actions. Under supervised learning, the “label”  $t$  refers to an action to be taken when the learner is in an informational state about the environment given by a variable  $x$ . Upon taking an action  $t$  in a state  $x$ , the learner is provided with feedback on the immediate reward accrued via this decision, and the environment moves on to a different state. As an example, a marines can be trained to navigate a given environment in the presence of obstacles by penalizing decisions that result in collisions.

Reinforcement learning is hence neither supervised, since the learner is not provided with the optimal actions  $t$  to select in a given state  $x$ ; nor it's fully unsupervised, given the availability of feedback on the quality of the chosen action. Reinforcement learning is also distinguished from supervised and unsupervised learning due to the influence of previous actions on future states and rewards.

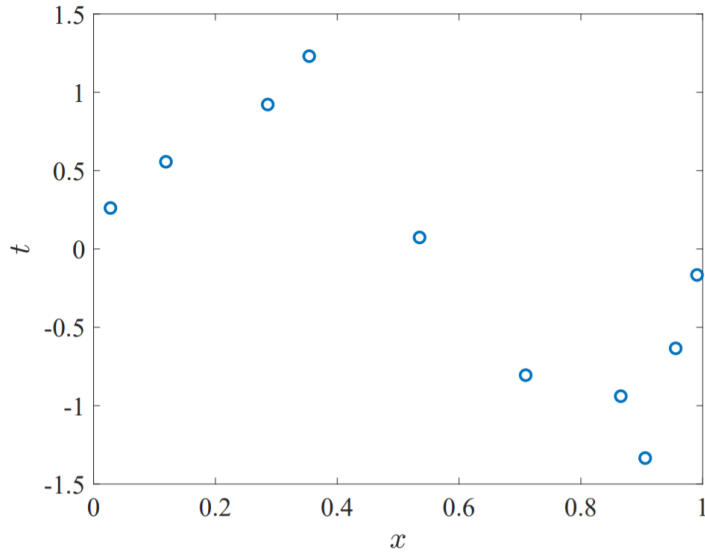
These general tasks can be further classified along the following dimensions.

- **Passive vs. active learning:** A passive learner is when is given the training examples, otherwise an active learner can affect the choice of training examples on the basis of prior observations.
- **Offline vs. online learning:** The offline learning act over a batch of training samples, otherwise the online learning processes samples in a streaming mode. It's easy to notice that the reinforcement learning operates intrinsically in an online mode, while supervised and unsupervised learning can be carried out by following either offline or online formulations.



### 1.3 Introduction to Linear Regression

In the standard formulation of a supervised learning problem, we give a training set  $\mathcal{D}$  containing  $N$  training points  $(x_n, t_n), n = 1, \dots, N$ . The observations  $x_n$  are considered free variables and known as covariates, domain points, or explanatory variables; while the target variables  $t_n$  are assumed to be dependent on  $x_n$  and are referred to as dependent variables, labels, or responses. [9] An example is illustrated in Fig. 1.2. We use the notation  $x_{\mathcal{D}} = (x_1, \dots, x_N)^T$  for



**Figure 1.2:** Example of a training set  $\mathcal{D}$  with  $N = 10$  points  $(x_n, t_n), n = 1, \dots, N$ . [2]

the covariates and  $t_{\mathcal{D}} = (t_1, \dots, t_N)^T$  for the labels in the training set  $\mathcal{D}$ . Starting from this data, the goal of supervised learning is to identify an algorithm to predict the label  $t$  for a new domain point  $x$ . The outlined learning task is clearly impossible in the absence of additional information on the mechanism relating variables  $x$  and  $t$ . With reference to Fig. 1.2, we can assume that  $x$  and  $t$  are related by a function

$$t = f(x) \tag{1.1}$$

With some characteristics, such as smoothness, we have no way of predicting the label  $t$  for an unobserved domain point  $x$ . This observation is formalized by the NFL (No free lunch theorem): one cannot learn rules that generalize to unseen examples without making assumptions about the mechanism generating the data. At this point we define the inductive bias as the set of all assumptions made by the learning algorithm .

The key point of this discussion, however, is a key difference between memorizing and learning. While the first means the mere retrieval of a value  $t_n$  corresponding to an already observed pair  $(x_n, t_n) \in \mathcal{D}$ , learning implicate the capability to predict the value  $t$  for an unseen domain point  $x$ .

Learning, otherwise, converts experience in the mathematical form of  $\mathcal{D}$  into knowledge in the form of a predictive algorithm. This is well captured by the following quote by Jorge Luis Borges: “*To think is to forget details, generalize, make abstractions.*” [10].

Another important definition for supervised learning is the loss means: loss is the error in the prediction of a new label  $t$  for an unobserved explanatory variable  $x$ . Finally, now we can say that the goal of supervised learning is that of identifying a predictive algorithm that minimizes the generalization loss. How exactly to formulate this problem, however, depends on one’s viewpoint, on the nature of the model that is being learned, and on who are the variables of interest.

### 1.3.1 Statistical inference

We specifically consider the inference problem of predicting a rv  $t$  given the observation of another rv  $x$  under the assumption that their joint distribution  $p(x, t)$  is known. As a matter of terminology, it can be noticed that here it is used as the term “inference” as it is typically intended in the literature on probabilistic graphical models[11], that diverging from its use in other branches of the machine learning literature, such as pattern recognition[12]. To define and exploit the problem of optimal inference, we can start by defining a non-negative *loss function*  $\ell(t, \hat{t})$ . This defines the loss incurred when the correct value is  $t$  while the estimate is  $\hat{t}$ .

An important example is the  $\ell_q$  loss:

$$\ell_q(t, \hat{t}) = |t - \hat{t}|^q \quad (1.2)$$

which includes as a special case the *quadratic loss*  $\ell_2(t, \hat{t}) = (t, \hat{t})^2$ , and the *0-1 loss*, or *detection error*,  $\ell_0(t, \hat{t}) = (t, \hat{t})_0$ , where  $|a|_0 = 1$  if  $a \neq 0$  and  $|a|_0 = 0$  otherwise. Once a loss function is defined, the optimal prediction  $\hat{t}(x)$  for a given value of the observation  $x$  is computed by minimizing the *generalization loss*:

$$L_p(\hat{t}) = E_{(x,t) \sim p_{xt}}[\ell(t, \hat{t}(x))] \quad (1.3)$$

The syntax  $L_p$  emphasizes the dependence of the generalization loss on the distribution  $p(x, t)$ .

The solution of this problem is given by the optimal prediction, also known as Bayes’ rule :

$$\hat{t}^*(x) = \arg \min_{\hat{t}} E_{p_{t|x}}[\ell(t, \hat{t})|x] \quad (1.4)$$

This can be noticed by using the law of iterated expectations

$$E_{(x,t) \sim p_{xt}}[\cdot] = E_{x \sim p_x}[E_{t \sim p_{t|x}}[\cdot|x]] \quad (1.5)$$

Equation (1.4) shows that the optimal prediction  $\hat{t}^*(x)$  is a function of the posterior distribution  $p(t|x)$  of the label given the domain point  $x$  and of the loss function  $\ell$ . Therefore, knowing the posterior  $p(t|x)$ , the optimal prediction Equation (1.4) can be evaluated for any desired loss function, without the necessity to know the joint distribution  $p(x, t)$ .

The goal of supervised learning methods is to obtain a predictor  $\hat{t}(x)$  that performs closer as possible to the optimal predictor  $\hat{t}^*(x)$ , based only on the training set  $\mathcal{D}$ , and without knowledge of the joint distribution  $p(x, t)$ .

The closeness in performance is measured by the difference between the generalization loss  $L_p(\hat{t})$  obtained by the trained predictor and the minimum generalization loss  $L_p(\hat{t}^*)$  of the optimal predictor, which depends on the true distribution  $p(x, t)$ .

### 1.3.2 Frequentist Approach

In this thesis it is specifically treated the Frequentist Approach, as it will be the method adopted for the implementation of the software for the calibration of pollution analysis devices. Therefore, the Bayesian approach will not be dealt with. According to the frequentist point of view, the training data points  $(x_n, t_n) \in \mathcal{D}$  are independent identically distributed (i.i.d.) rvs drawn from a true, and unknown, distribution  $p(x, t)$ :

$$(x_n, t_n) \stackrel{\text{i.i.d.}}{\sim} p(x, t) \quad i = 1, \dots, N. \quad (1.6)$$

The new observation  $(x, t)$  is also independently generated from the same true distribution  $p(x, t)$ ; the domain point  $x$  is observed and the label  $t$  must be predicted. Since the probabilistic model  $p(x, t)$  is not known, one cannot solve directly problem (1.4) to find the optimal prediction that minimizes the generalization loss  $L_p$  in (1.3). Before discussing the available solutions to this problem, it is worth observing that the definition of the “true” distribution  $p(x, t)$  depends in practice on the way data is collected. As in the example of the “beauty AI” context, if the rankings  $t_n$  assigned to pictures  $x_n$  of faces are affected by racial biases, the distribution  $p(x, t)$  will reflect these prejudices and produce skewed results [62].

### 1.3.3 Taxonomy of solutions

There are two main ways to address the problem of learning how to perform inference when not knowing the distribution  $p(x, t)$ :

- *Separate learning and (plug-in) inference*: Learn first an approximation, say  $p_{\mathcal{D}}(t|x)$ , of the conditional distribution  $p(t|x)$  based on the data  $\mathcal{D}$ , and then plug this approximation in (1.4) to obtain an approximation of the optimal decision.

$$\hat{t}_{\mathcal{D}}(x) = \arg \min_{\hat{t}} E_{t \sim p_{\mathcal{D}}(t|x)}[\ell(t, \hat{t})|x] \quad (1.7)$$

- *Direct inference via Empirical Risk Minimization (ERM)*: Learn directly an approximation  $\hat{t}_{\mathcal{D}}(\cdot)$  of the optimal decision rule by minimizing an empirical estimate of the generalization loss (1.3) obtained from the data set.

The notation  $L_{\mathcal{D}}(\hat{t})$  highlights the dependence of the empirical loss on the predictor  $\hat{t}(\cdot)$  and on the training set  $\mathcal{D}$ . In practice, as we will see, both approaches optimize a set of parameters that define the probabilistic model or the predictor.

Furthermore, the first approach is generally more flexible, since having an estimate  $p_{\mathcal{D}}(t|x)$  of the posterior distribution  $p(t|x)$  allows the prediction (1.7) to be computed for any loss function. In contrast, the ERM solution is tied to a specific choice of the loss function  $\ell$ . In the rest of this section, we will start by taking the first approach.

### 1.3.4 Linear Regression (LR) Model

In the following, we will consider the following running example inspired by [12]. In the example, data is generated according to the true distribution  $p(x, t) = p(x)p(t|x)$ , dove  $x \sim \mathcal{U}(0, 1)$  and

$$t|x = x \sim \mathcal{N}(\sin(2\pi x), 0.1) \quad (1.8)$$

The training set in Fig.1.2 was generated from this distribution. If this true distribution were known, the optimal predictor under the  $\ell_2$  loss would be equal to the conditional mean

$$\hat{t}^* = \sin(2\pi x) \quad (1.9)$$

Finally, the minimum generalization loss is  $L_p(\hat{t}^*) = 0.1$ .

### 1.3.5 Discriminative vs. Generative Probabilistic Models

In order to use an approximation  $p_{\mathcal{D}}(t|x)$  of the predictive distribution  $p(t|x)$  based on the data  $\mathcal{D}$ , we will proceed by first selecting a family of parametric probabilistic models, also known as a *hypothesis class*, and then learning the parameters of the model to fit the data  $\mathcal{D}$ .

Let consider as an example the linear regression problem introduced above, we start by modelling the label  $t$  as a *polynomial function* of the domain point  $x$  added to a Gaussian noise with variance  $\beta^{-1}$ . Parameter  $\beta$  is the precision, i.e., the

inverse variance of the additive noise. The polynomial function with degree  $M$  can be written as

$$\mu(x, w) = \sum_{j=0}^M w_j x^j = w^T \phi(x) \quad (1.10)$$

where we have defined the weight vector  $w = [w_0 \ w_1 \ \cdots \ w_M]^T$  and the feature vector  $\phi(x) = [1 \ x \ x^2 \ \cdots \ x_M]^T$ . The vector  $w$  defines the relative weight of the powers in the sum (2.11). This assumption corresponds to adopting a parametric probabilistic model  $p(t|x, \theta)$  defined as

$$t|x = x \sim \mathcal{N}(\mu(x, w), \beta^{-1}) \quad (1.11)$$

with parameters  $\theta = (w, \beta)$ . Having fixed this hypothesis class, the parameter vector  $\theta$  can be then learned from the data  $\mathcal{D}$ , as it will be discussed. In the example above, we have parametrized the posterior distribution. Alternatively, we can parametrize and learn the full joint distribution  $p(x, t)$ . These two alternatives are introduced below.

### 1. Discriminative probabilistic model

With this first class of models, the posterior, or predictive, distribution  $p(t|x)$  is assumed to belong to a hypothesis class  $p(t|x, \theta)$  defined by a parameter vector  $\theta$ . The parameter vector  $\theta$  is learned from the data set  $\mathcal{D}$ . For a given parameter vector  $\theta$ , the conditional distribution  $p(t|x, \theta)$  allows the different values of the label  $t$  to be discriminated on the basis of their posterior probability.

In particular, once the model is learned, one can directly compute the predictor (2.6) for any loss function. As an example, for the linear regression problem, once a vector of parameters  $\theta_{\mathcal{D}} = (w_{\mathcal{D}}, \beta_{\mathcal{D}})$  is identified based on the data  $\mathcal{D}$  during learning, the optimal prediction under the  $\ell_2$  loss is the conditional mean  $\hat{t}_{\mathcal{D}}(x) = E_{t \sim p(t|x, \theta_{\mathcal{D}})}[t|x]$ , that is,  $\hat{t}_{\mathcal{D}}(x) = \mu(x, w_{\mathcal{D}})$ .

### 2. Generative probabilistic model

Instead of learning directly the posterior  $p(t|x)$ , one can model the joint distribution  $p(x, t)$  as being part of a parametric family  $p(x, t|\theta)$ . Note that, as opposed to discriminative models, the joint distribution  $p(x, t|\theta)$  models also the distribution of the covariates  $x$ . Accordingly, the term “generative” reflects the capacity of this type of models to generate a realization of the covariates  $x$  by using the marginal  $p(x|\theta)$ .

Once the joint distribution  $p(x, t|\theta)$  is learned from the data, one can compute the posterior  $p(t|x, \theta)$  using Bayes’ theorem, and, from it, the optimal predictor (1.7) can be evaluated for any loss function. Generative models make stronger assumptions by modeling also the distribution of the explanatory variables. As a result, an improper selection of the model may lead to more significant

bias issues. However, there are potential advantages, such as the ability to deal with missing data or latent variables, such as in semi-supervised learning.

### 1.3.6 Model Order and Model Parameters

In the linear regression example, the selection of the hypothesis class required the definition of the polynomial degree  $M$ , while the determination of a specific model  $p(t|x, \theta)$  in the class called for the selection of the parameter vector  $\theta = (w, \beta)$ . As we will see, these two types of variables play a significantly different role during learning and should be clearly distinguished, as discussed next.

1. **Model order  $M$  (and hyperparameters):** The model order defines the “capacity” of the hypothesis class, that is, the number of the degrees of freedom in the model. The larger  $M$  is, the more capable a model is to fit the available data. For instance, in the linear regression example, the model order determines the size of the weight vector  $w$ . More generally, variables that define the class of models to be learned are known as *hyperparameters*. As we will see, determining the model order, and more broadly the hyperparameters, requires a process known as *validation*.
2. **Model parameters  $\theta$ :** Assigning specific values to the model parameters  $\theta$  identifies a hypothesis within the given hypothesis class. This can be done by using learning criteria such as ML (Maximum Likelihood) and MAP (Maximum a Posteriori).

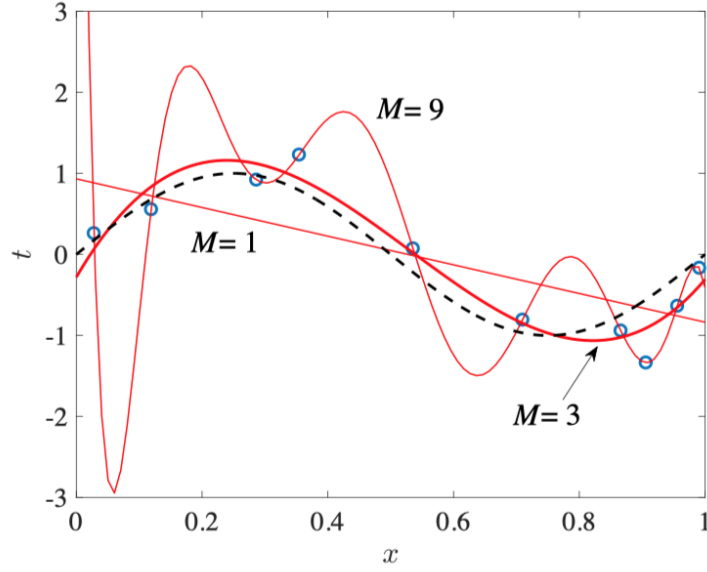
### 1.3.7 Overfitting and Underfitting Problem

Adopting the  $\ell_2$  loss, let us now compare the predictor  $t_{ML}(x) = \mu(x, w_{ML})$  learned via ML with the optimal, but unknown, predictor  $\hat{t}^*(x)$  in (2.10). To this end, Fig. 1.4 shows the optimal predictor  $\hat{t}^*$  as a dashed line and the ML-based predictor  $t_{ML}(x)$  obtained with different values of the model order  $M$  for the training set  $\mathcal{D}$  in Fig. 1.2 (also shown in Fig. 1.4 for reference).

We begin by observing that, with  $M = 1$ , the ML predictor underfits the data: the model is not rich enough to capture the variations present in the data. As a result, the training loss  $L_{\mathcal{D}}(w_{ML})$  is large. In contrast, with  $M = 9$ , the ML predictor overfits the data: the model is too rich and, to account for the observations in the training set, it yields inaccurate predictions outside it. In this case, the training loss  $L_{\mathcal{D}}(w)$  is small, but the generalization loss is large.

$$L_p(w_{ML}) = E_{(x,t) \sim p_{xt}}[\ell(t, \mu(x, w_{ML}))] \quad (1.12)$$

With overfitting, the model is memorizing the training set, rather than learning how to generalize to unseen examples.



**Figure 1.3:** Illustration of underfitting and overfitting in ML learning: The dashed line is the optimal predictor (2.10), which depends on the unknown true distribution, while the other lines correspond to the predictor  $\hat{t}_{ML}(x) = \mu(x, w_{ML})$  learned via ML with different model orders  $M$ . [2]

The choice  $M = 3$  appears to be the best in comparison with the optimal predictor. Note that this observation is in practice precluded given the impossibility to determine  $\hat{t}^*(x)$  and hence the generalization loss.[13]

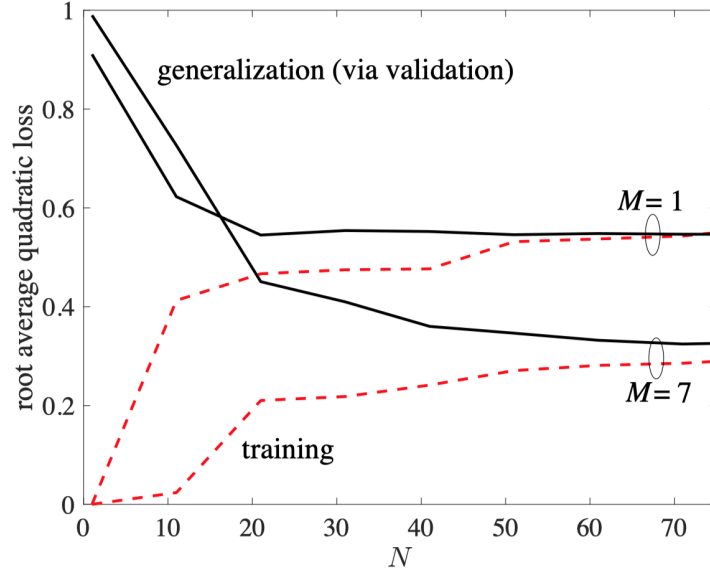
### 1.3.8 Influence of dataset size

Extrapolating from the behavior observed in Fig. 1.4, we can surmise that, as the number  $N$  of data points increases, overfitting is avoided even for large values of  $M$ . When the training set is big compared to the number of parameters in  $\theta$ , we expect the training loss  $L_{\mathcal{D}}(w)$  to provide an accurate measure of the generalization loss  $L_p(w)$  for all possible values of  $w$ .

Informally, we have the approximation  $L_{\mathcal{D}}(w) \simeq L_p(w)$  simultaneously for all values of  $w$  as long as  $N$  is large enough. Therefore, the weight vector  $w_{ML}$  that minimizes the training loss  $L_{\mathcal{D}}(w)$  also (approximately) minimizes the generalization loss  $L_p(w)$ . It follows that, for large  $N$ , the ML parameter vector  $w_{ML}$  tends to the optimal value  $w^*$  (assuming for simplicity of argument that it is unique) that minimizes the generalization loss among all predictors in the model, i.e.,

$$w^* = \arg \min_w L_p(w) \quad (1.13)$$

To better understand what we have pointed, we can give a numerical example,



**Figure 1.4:** Square root of the generalization loss  $L_p(w_{ML})$  and of the training loss  $L_{\mathcal{D}}(w_{ML})$  as a function of the training set size  $N$ . The asymptote of the generalization and training losses is given by the minimum generalization loss  $L_p(w^*)$  (cf. (2.21)) achievable for the given model order (see Fig. 2.5). [2]

Fig.1.3 plots the square root of the generalization and training losses versus  $N$ , where the training sets were generated random from the true distribution. From the figure, we can make the following important assumptions:

- Overfitting, as measured by the gap between training and generalization losses, vanishes as  $N$  increases. This is a consequence of the discussed approximate equalities  $L_{\mathcal{D}}(w) \simeq L_p(w)$  and  $w_{ML} \simeq w^*$ , which are valid as  $N$  grows large, which imply the approximate equalities  $L_{\mathcal{D}}(w_{ML}) \simeq L_p(w_{ML}) \simeq L_p(w^*)$ .
- Training loss  $L_{\mathcal{D}}(w_{ML})$  tends to the minimum generalization loss  $L_p(w^*)$  for the given  $M$  from below, while the generalization loss  $L_p(w_{ML})$  tends to it from above. This happens because, as  $N$  increases, it becomes more difficult to fit the data set  $\mathcal{D}$ , and  $L_{\mathcal{D}}(w_{ML})$  increases. Contrary as  $N$  grows large the ML estimate becomes more accurate, because of the increasingly accurate approximation  $w_{ML} \simeq w^*$ , and thus the generalization loss  $L_p(w_{ML})$  decreases.
- Selecting a smaller model order  $M$  yields an improved generalization loss when



the training set is small, while a larger value of  $M$  is desirable when the data set is bigger. When  $N$  is small, the estimation error caused by overfitting dominates the bias caused by the choice of a small hypothesis class. In contrast, for sufficiently large training sets, the estimation error vanishes and the performance is dominated by the bias induced by the selection of the model.

### 1.3.9 Regularization

As we saw before, the model will have a low accuracy if it is overfitting, so one of the major aspects of training a machine learning model is avoiding overfitting. This happens because the model is trying too hard to capture the noise on the training dataset. By noise we mean the data points that don't really represent the true properties of the data, but random chance. Learning such data points makes the model more flexible, at the risk of overfitting. *Regularization* is a form of regression that constrains/regularizes or shrinks the coefficient estimates towards zero. In other words, this technique discourages learning a more complex or flexible model to avoid the risk of overfitting [14].

We have seen above that the MAP learning criterion amounts to the addition of a regularization function  $R(w)$  to the ML or ERM learning loss. This function penalizes values of the weight vector  $w$  that are likely to occur in the presence of overfitting, or, generally, that are improbable on the basis of the available prior information. The net effect of this addition is to effectively decrease the capacity of the model, as the set of values for the parameter vector  $w$  that the learning algorithm is likely to choose from is reduced. As a result, as seen, regularization can control overfitting and its optimization requires validation. A simple relation for linear regression looks like this. Here  $Y$  represents the learned relation and  $\beta$  represents the coefficient estimates for different variables or predictors( $X$ )

$$Y \approx \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots + \beta_p x_p \quad (1.14)$$

The fitting procedure involves a loss function, known as RSS (Residual Sum of Squares). The coefficients are chosen such that they minimize this loss function.

$$RSS = \sum_{i=1}^n \left( y_i + \beta_0 - \sum_{j=1}^p \beta_j x_{ij} \right)^2 \quad (1.15)$$

The 1.15 will adjust the coefficients based on the training data. If there is noise in the training data, then the estimated coefficients won't generalize well to the future data. This is where regularization comes in and shrinks or regularizes these learned estimates towards zero.

We note that the same techniques, such as ridge regression and LASSO (Least

Absolute Shrinkage and Selection Operator), can also be introduced independently of a probabilistic framework in an ERM formulation. Furthermore, apart from the discussed addition of regularization terms to the empirical risk, there are other ways to perform regularization.

One approach is to modify the optimization scheme by using techniques such as early stopping [15]. Another is to augment the training set by generating artificial examples and hence effectively increasing the number  $N$  of training examples. Related to this idea is the technique known as bagging.

With bagging, we first create a number  $K$  of bootstrap data sets. Bootstrap data sets are obtained by selecting  $N$  data points uniformly with replacement from  $\mathcal{D}$  (so that the same data point generally appears multiple times in the bootstrap data set). Then, we train the model  $K$  times, each time over a different bootstrap set. Finally, we average the results obtained from the models using equal weights. If the errors accrued by the different models were independent, bagging would yield an estimation error that decreases with  $K$ . In practice, significantly smaller gains are obtained, particularly for large  $K$ , given that the bootstrap data sets are all drawn from  $\mathcal{D}$  and hence the estimation errors are not independent [16].

## 1.4 Optimizers and model accuracy

We have previously dealt with the loss function, which is a mathematical way of measuring how wrong your predictions are. During the training process, we tweak and change the parameters (weights) of our model to try and minimize that loss function, and make our predictions as correct and optimized as possible. However, how exactly do you do that? How do you change the parameters of the model, by how much, and when?

This is where *optimizers* come in. They tie together the loss function and model parameters by updating the model in response to the output of the loss function. In simpler terms, optimizers shape and mold your model into its most accurate possible form by futzing with the weights. The loss function is the guide to the terrain, telling the optimizer when it's moving in the right or wrong direction.

For a useful mental model, we can think of a hiker trying to get down a mountain with a blindfold on. It's impossible to know which direction to go in, but there's one thing she can know: If she's going down (making progress) or going up (losing progress). Eventually, if she keeps taking steps that lead her downwards, she'll reach the base. Similarly, it's impossible to know what your model's weight should be right from the start. However, with some trial and error based on the loss function (whether the hiker is descending), you can end up getting there eventually.

### 1.4.1 Gradient Descent

This algorithm is used across all types of machine learning (and other math problems) to optimize. Gradient descent is a first-order optimization algorithm which depends on the first-order derivative of a loss function. It calculates that which way the weights should be altered so that the function can reach a minima. Through backpropagation, the loss is transferred from one layer to another and the model's parameters, also known as weights, are modified depending on the losses so that the loss can be minimized. The algorithm of gradient descent can be implemented following the following instruction:

1. Calculate what a small change in each individual weight would do to the loss function (i.e. which direction should the hiker walk in)
2. Adjust each individual weight based on its gradient (i.e. take a small step in the determined direction)
3. Keep doing steps #1 and #2 until the loss function gets as low as possible

In mathematical form, this can be translated into

$$\Theta = \Theta - \alpha \cdot \nabla J(\Theta) \quad (1.16)$$

The tricky part of this algorithm (and optimizers in general) is understanding the gradients, which represent what a small change in a weight or parameter would do to the loss function. Gradients are partial derivatives and are a measure of change. They connect the loss function and the weights; they tell us what specific operation we should do on our weights—add 5, subtract .07, or anything else—to lower the output of the loss function and thereby make our model more accurate.

The cost function can be mathematically defined as follows:

$$J(\theta) = 1/2m \sum_{i=1}^m (h(\theta)^{(i)} - y^{(i)})^2 \quad (1.17)$$

Note that the cost function is for linear regression, for other algorithms the cost function will be different and the gradients would have to be derived from the cost function. Others important parameters of Gradient Descent are:

- *Learning Rate*: is a tuning parameter in an optimization algorithm that determines the step size at each iteration while moving toward the minimum of a loss function. Since it influences to what extent the newly acquired information overrides old information, it metaphorically represents the speed at which a machine learning model "learns". In the adaptive control literature, the learning rate is commonly referred to as gain.

- *Epochs*: one epoch is when an entire dataset is passed forward and backward through the neural network only once. Since one epoch is too big to feed to the computer at once, we divide it in several smaller batches. Passing the entire dataset through a neural network is not enough. What is typically done is to pass the full dataset multiple times to the same neural network.
- *Batch Size*: total number of training examples present in a single batch. As was previously mentioned, you can't pass the entire dataset into the neural net at once. Therefore, what is done is to divide the dataset into a number of batches or sets or parts.
- *Iterations*: iteration is the number of batches needed to complete one epoch.

### 1.4.2 Gradient Descent Algorithm

Basically the Gradient Descent can be implemented in this way.  
First of all, we define the cost function:

```
1 def cal_cost(theta,X,y):  
2     '''  
3     Calculates the cost for given X and Y. The following shows and  
4     example of a single dimensional X  
5     theta = Vector of thetas  
6     X      = Row of X's np.zeros((2,j))  
7     y      = Actual y's np.zeros((2,1))  
8     where:  
9     j is the no of features  
10    '''  
11    m = len(y)  
12    predictions = X.dot(theta)  
13    cost = (1/2*m) * np.sum(np.square(predictions-y))  
14    return cost  
15
```

Then we can define the Gradient Descent algorithm:

```
1 def gradient_descent(X,y,theta,learning_rate=0.01,iterations=100):  
2     '''  
3     X      = Matrix of X with added bias units  
4     y      = Vector of Y  
5     theta=Vector of thetas np.random.randn(j,1)  
6     learning_rate  
7     iterations = no of iterations  
8     '''
```

```

8   Returns the final theta vector and array of cost history over no
9   of iterations
10  '''
11  m = len(y)
12  cost_history = np.zeros(iterations)
13  theta_history = np.zeros((iterations, 2))
14  for it in range(iterations):
15
16      prediction = np.dot(X, theta)
17
18      theta = theta - (1/m)*learning_rate*( X.T.dot((prediction - y)))
19      theta_history[it, :] = theta.T
20      cost_history[it] = cal_cost(theta, X, y)
21
22  return theta, cost_history, theta_history

```

### 1.4.3 Adaptive Moment Estimation (ADAM)

Adam optimization is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments. Adam is different from classical stochastic gradient descent: Stochastic gradient descent maintains a single learning rate (termed alpha) for all weight updates and the learning rate does not change during training. A learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds. Adam combines the advantages of two other extensions of stochastic gradient descent. Specifically:

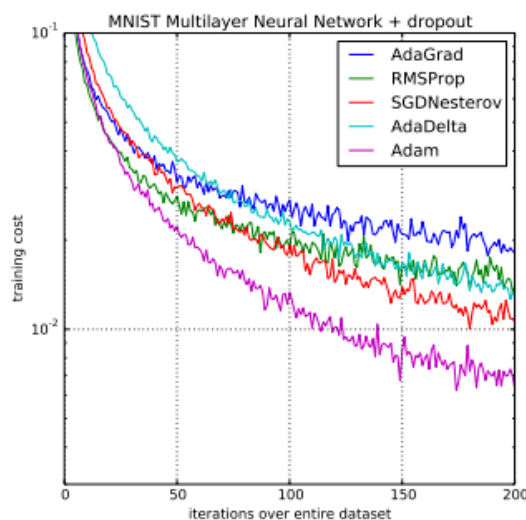
- **AdaGrad (Adaptive Gradient Algorithm):** that maintains a per-parameter learning rate that improves performance on problems with sparse gradients (e.g., natural language and computer vision problems).
- **RMSProp (Root Mean Square Propagation):** that also maintains per-parameter learning rates that are adapted based on the average of the recent magnitudes of the gradients of the weight (e.g., how quickly it is changing). This means the algorithm does well on online and nonstationary problems (e.g. noisy).

Instead of adapting the parameter learning rates based on the average first moment (mean) as in RMSProp, Adam also makes use of the average of the second moments of the gradients (uncentered variance).

Specifically, the algorithm calculates an exponential moving average of the gradient and the squared gradient, and the parameters beta1 and beta2 control the decay rates of these moving averages.

The initial value of the moving averages and beta1 and beta2 values close to 1.0

(recommended) result in a bias of moment estimates towards zero. This bias is overcome by first calculating the biased estimates before then calculating bias-corrected estimates.[1]



**Figure 1.5:** Comparison of Adam to Other Optimization Algorithms Training a Multilayer Perceptron [1]

#### 1.4.4 Adam Configuration Parameters

- **alpha:** also referred to as the learning rate or step size. The proportion that weights are updated (e.g., 0.001). Larger values (e.g., 0.3) results in faster initial learning before the rate is updated. Smaller values (e.g., 1.0E-5) slow learning right down during training
- **beta1:** the exponential decay rate for the first moment estimates (e.g., 0.9).
- **beta2:** the exponential decay rate for the second moment estimates (e.g., 0.999). This value should be set close to 1.0 for problems with a sparse gradient (e.g., NLP and computer vision problems).
- **epsilon:** is a very small number to prevent any division by zero in the implementation (e.g., 10E-8).

---

**Algorithm 1** Adam optimizer algorithm. All operations are element-wise even powers. Good values for the constants are  $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$ .  $\epsilon$  is needed to guarantee numerical stability.

---

```

1: procedure ADAM( $\alpha, \beta_1, \beta_2, f, \theta_0$ )
2:    $\triangleright \alpha$  is the stepsize
3:    $\triangleright \beta_1, \beta_2 \in [0, 1)$  are the exponential decay rates for the moment estimates
4:    $\triangleright f(\theta)$  is the objective function to optimize
5:    $\triangleright \theta_0$  is the initial vector of parameters which will be optimized
6:    $\triangleright$  Initialization
7:    $m_0 \leftarrow 0$   $\triangleright$  First moment estimate vector set to 0
8:    $v_0 \leftarrow 0$   $\triangleright$  Second moment estimate vector set to 0
9:    $t \leftarrow 0$   $\triangleright$  Timestep set to 0
10:   $\triangleright$  Execution
11:  while  $\theta_t$  not converged do
12:     $t \leftarrow t + 1$   $\triangleright$  Update timestep
13:     $\triangleright$  Gradients are computed w.r.t the parameters to optimize
14:     $\triangleright$  using the value of the objective function
15:     $\triangleright$  at the previous timestep
16:     $g_t \leftarrow \nabla_{\theta} f(\theta_{t-1})$ 
17:     $\triangleright$  Update of first-moment and second-moment estimates using
18:     $\triangleright$  previous value and new gradients, biased
19:     $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ 
20:     $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ 
21:     $\triangleright$  Bias-correction of estimates
22:     $\hat{m}_t \leftarrow \frac{m_t}{1 - \beta_1^t}$ 
23:     $\hat{v}_t \leftarrow \frac{v_t}{1 - \beta_2^t}$ 
24:     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$   $\triangleright$  Update parameters
25:  end while
26:  return  $\theta_t$   $\triangleright$  Optimized parameters are returned
27: end procedure

```

---

## 1.5 Overview

The following chapters will first introduce the methodologies used so far, then, following an introduction on the work already started in the research department of the Department of Automation and Information Technology of the Politecnico di Torino, finally it will be explained how to relate these technologies to the case study.

- Chapter 2 presents the main concepts of Quantum Machine Learning, and how to implement Linear Regression model through Quantum Neural Networks.
- Chapter 3 introduces the Air Pollution Project, illustrating from the beginning how the project has been setted up.
- Chapter 4 presents the idea of how to exploit the technologies of Quantum Machine Learning to create a hybrid network that is able to refine the calibration of sensors in a fast and punctual way.
- Chapter 5 reports the hyperparameter tuning phase, exploits the simulation plots, and evaluates the obtained results.
- Chapter 6 draws the conclusions and proposes some possible further improvements related to the overall project.



## Chapter 2

# Quantum Machine Learning

In the second part, it will be introduced the concept of Quantum Machine Learning and how this feature can be correlated with the air pollution sensors calibration. In this chapter, we explore how a classical neural network can be partially quantized to create a hybrid quantum-classical neural network.

### 2.1 What is Quantum Machine Learning

Quantum computing refers to the manipulation of quantum systems to process information. The ability of quantum states to be in superposition can thereby lead to a substantial speed-up of a computation in terms of complexity, since operations can be executed on many states at the same time. The basic unit of quantum computation is the qubit, that can be defined in a mathematical way like:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \quad (2.1)$$

with  $\alpha, \beta \in \mathbb{C}$  and  $|0\rangle, |1\rangle$  in the two-dimensional Hilbert space  $H_2$ .

The absolute squares of the amplitudes are the probability to measure the qubit in the 0 or the 1 state, and quantum dynamics always maintain the property of probability conservation by

$$|\alpha|^2 + |\beta|^2 = 1 \quad (2.2)$$






In mathematical language, this means that transformations that map quantum states to other quantum states (so-called quantum gates) have to be unitary. Through single qubit quantum gates, we are able to manipulate the basis state, amplitude or phase of a qubit (for example, through the so-called X-gate, the Z-gate, and the Y-gate, respectively), or put a qubit with  $\beta = 0$  ( $\alpha = 1$ ) into an

equal superposition (the Hadamard or H-gate) :

$$\alpha = \beta = \frac{1}{\sqrt{2}} \text{ or } \alpha = \frac{1}{\sqrt{2}} \quad \beta = -\frac{1}{\sqrt{2}} \quad (2.3)$$

Multi-qubit gates are often based on controlled operations that execute a single qubit operation only if another (ancilla or control qubit) is in a certain state. One of the most important gates is the two-qubit XOR-gate, which flips the basis state of the second qubit in case the first qubit is in state  $|1\rangle$ . A two-qubit gate that will be mentioned later is the SWAP-gate exchanging the state of two qubits with each other.

### 2.1.1 Example of Quantum Gates

qubit states	$\begin{cases}  0\rangle \leftarrow \\  1\rangle \leftarrow \end{cases}$	$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}$
X		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Hadamard		$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
XOR		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
SWAP		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Measurement		

**Figure 2.1:** Representation of qubit states, unitary gates and measurements in the quantum circuit model and in the matrix formalism [17]

Quantum gates are usually expressed as unitary matrices, this can be seen in Fig. 2.1. The matrices operate on  $2^n$ -dimensional vectors that contain the amplitudes of the  $2^n$  basis states of a  $n$ -dimensional quantum system.

For example, the XOR-gate working on the quantum state  $|\psi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$  would look like

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \cdot \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \quad (2.4)$$

and produce  $\psi' = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$ .

The art of developing algorithms for a potential quantum computer is to use such elementary gates to create a quantum state that has a relatively high amplitude

for states that represent solutions for the given problem. A measurement in the computational basis then produces such a desired result with a relatively high probability. Quantum algorithms are usually repeated a number of times since the result is always probabilistic. For a comprehensive introduction to quantum computing, we refer to the standard textbook by Nielsen and Chuang [18] as well as Plenio and Vitelli's presentation of the concept of quantum information [19]. In quantum machine learning, quantum algorithms are developed to solve typical problems of machine learning using the efficiency of quantum computing. This is usually done by adapting classical algorithms or their expensive subroutines to run on a potential quantum computer. The expectation is that in the near future, such machines will be commonly available for applications and can help to process the growing amount of global information. The emerging field also includes approaches vice versa, namely well-established methods of machine learning that can help to extend and improve quantum information theory.[20]

## 2.1.2 Neurons and Weights

A neural network is ultimately just an elaborate function that is built by composing smaller building blocks called neurons.

A *neuron* is typically a simple, easy-to-compute, and nonlinear function that maps one or more inputs to a single real number. The single output of a neuron is typically copied and fed as input into other neurons. Graphically, we represent neurons as nodes in a graph and we draw directed edges between nodes to indicate how the output of one neuron will be used as input to other neurons. It's also important to note that each edge in our graph is often associated with a scalar value called a *weight*. The idea here is that each of the inputs to a neuron will be multiplied by a different scalar before being collected and processed into a single value. The objective when training a neural network consists primarily of choosing weights such that the network behaves in a particular way.

## 2.1.3 Feed Forward Neural Network

It is also worth noting that the particular type of neural network we will concern ourselves with is called a FFNN (Feed-Forward Neural Network). This means that as data flows through our neural network, it will never return to a neuron it has already visited. Equivalently, you could say that the graph which describes our neural network is a DAG (Directed Acyclic Graph). Furthermore, we will stipulate that neurons within the same layer of our neural network will not have edges between them.

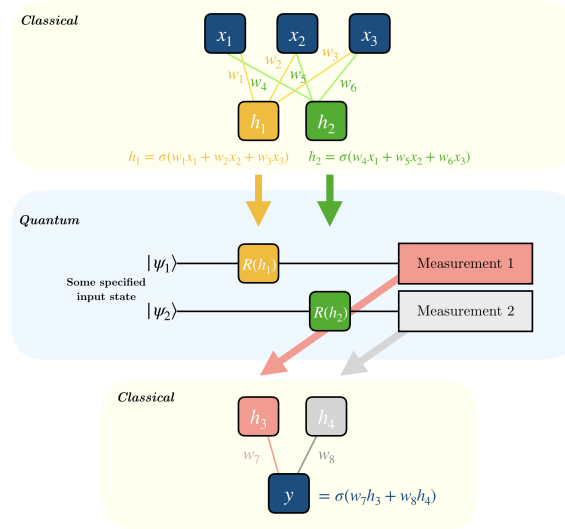
### 2.1.4 IO Structure of Layers

The input to a neural network is a classical (real-valued) vector. Each component of the input vector is multiplied by a different weight and fed into a layer of neurons according to the graph structure of the network. After each neuron in the layer has been evaluated, the results are collected into a new vector where the  $i$ 'th component records the output of the  $i$ 'th neuron. This new vector can then be treated as an input for a new layer, and so on. We will use the standard term *hidden layer* to describe all but the first and last layers of our network.

### 2.1.5 Building the Network

To create a quantum classical neural network, it can be implemented as a hidden layer of our neural network using a parameterized quantum circuit. By "parameterized quantum circuit", we mean a circuit where the rotation angles for each gate are specified by the components of a classical input vector. The outputs from our neural network's previous layer will be collected and used as the inputs for our parameterized circuit. The measurement statistics of our quantum circuit can then be collected and used as inputs for the following layer.

A simple example is depicted in Fig. 2.2:



**Figure 2.2:** Simple structure of a hybrid quantum-classical neural network[17]

Here,  $\sigma$  is a nonlinear function and  $h_i$  is the value of neuron  $i$  at each hidden layer.  $R(h_i)$  represents any rotation gate about an angle equal to  $h_i$  and  $y$  is the final prediction value generated from the hybrid network.

About backpropagation the quantum circuit can be seen as a black box and the gradient of this black box with respect to its parameters can be calculated as follows:

$$\nabla_{\theta} QC(\theta) = QC(\theta + s) - QC(\theta - s) \quad (2.5)$$

where QC = Quantum Circuit

## 2.2 Linear Regression with QML, case study

In the case study taken by the Developer guidelines on IBM Quantum Computing Developer Recipes, they show how to calculate the multivariate linear regression to find a prediction function in case the output depends on more than one  $x$  variable, such as house extension and the number of rooms.

What we will discuss in this chapter is how a low-level QPU works, the passage is fundamental, as it is useful to understand the rest of the work, where it was considered easy to use a higher level of abstraction, through the PennyLane library, a cross-platform Python library for differentiable programming of quantum computers.

### 2.2.1 Problem formulation

We are looking for a function

$$h_{\theta}(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 \quad (2.6)$$

able to mark the prediction, where

$$\theta_0, \theta_1, \theta_2 \quad (2.7)$$

are the parameters to be found.

An example dataset is used in witch, the first column is square feet, the second is room number, and third is the price.

The cost function is defined as the squared error function calculated between the function evaluation  $h_{\theta}(x)$  and the training data  $y$  and its means a calculation of the difference between evaluated and measured data. The purpose is to minimize this difference. In the multivariable case, we preliminary need to normalize data. It means that the feature  $x_i$  can be very different in scale. It means that the  $J(\theta)$  function can be very stretched in the direction of the largest scale feature. To prevent that is preferred to normalize the data, trying to maintain the feature  $x_i$  in the range  $-1 \leq x_i \leq 1$ .

To do this, we subtract to each  $x_i$  value the mean of all  $x_i$  values and divide it by the standard deviation of  $x_i$  data.

$$x_{i,n} = \frac{x_i - \mu_i}{S_i} \quad (2.8)$$

$$\mu_i = \frac{1}{m} \sum_{i=1}^m x_i \quad S_i = \sqrt{\frac{\sum_{i=1}^m x_i^2 - \mu_i^2}{m}} \quad (2.9)$$

To do this calculus fast we can use the numpy function `np.mean`, the same way for the standard deviation we can use `np.std`. After that, it is added a 'ones' column before the normalized  $x$  value and set to  $\theta$  the  $\theta$  value. This must be done to take account of the intercept in the prediction function. To add the column after having set it to 'ones' by means of the `np.ones` function, it can be used as the instruction `column stack`.

The choice of learning rate  $\alpha$  parameter is very critical :

- Choosing it too small, we can have a small convergence.
- Choosing it too big,  $J(\theta)$  may not decrease with every iteration and may not converge at all.

In the example, it is suggested to check the results varying the value of  $\alpha$  starting from 0.01 and increasing this value each time multiplying by 3. Some values can be : 0.01, 0.03, 0.1, 0.3 .

Another important warning is that in reading phase the data variable could be set to an integer data type ( instead of real ) and so when we normalize its value it can be rounded to 0. To prevent that we initialize a new vector of zeros and use it to make the calculation. In this case, this new array is defined as real.

## 2.2.2 Definition of parametrized Cost Function J

The choice of using a function is motivated by its recurrence in various parts of the algorithm.

The squared error function can be defined as follows:

$$J(\theta_0, \theta_1, \dots) = J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x_i) - y_i)^2 \quad (2.10)$$

If we consider the value of  $h_{\theta}(x)$  function the previous expression becomes:

$$J(\theta_0, \theta_1, \dots) = J(\theta) = \frac{1}{2m} \sum_{i=1}^m (X\theta - y_i)^2 \quad (2.11)$$

where  $X$  is a  $3 \times 3$  matrix and  $\theta$  is a vector  $3 \times 1$ .

By multiplying these two matrices it's obtained a vector that can be subtracted from the  $y$  vector. Finally, this is the function target to minimize.

### 2.2.3 Gradient descent algorithm in multivariable version

For each iteration cycle

$$\theta_j := \theta_j - \alpha \frac{\delta J(\theta_0, \theta_1)}{\delta \theta_j} \quad (2.12)$$

where  $j$  is the  $\theta$  index and  $\alpha$  is the so called learning rate, which defines the “length” of the single step of iteration, initially set at 0.01, but later refined through hyper-parameter tuning.

The update value of  $\theta_0, \theta_1, \theta_2$  must be simultaneous:

$$temp_j := \theta_j - \alpha \frac{\delta J(\theta)}{\delta \theta_j} \quad (2.13)$$

$$\theta_j := \theta_j - \frac{\alpha}{m} \sum_{i=1}^m (h_{\theta}(x_i) - y_i) x_i \quad (2.14)$$

then

$$\theta_j := temp_j \quad (2.15)$$

### 2.2.4 Implementation in Python

First of all, let us include the basic calculation libraries, these libraries are based on Toffoli and controlled phase gates:

- Quantum Fourier transform of  $|q\rangle$ , of length  $n$ .
- Inverse Fourier transform of  $|q\rangle$ , of length  $n$ .
- Controlled-controlled phase gate with phase  $\theta$
- Quantum Fourier transform of  $q$ , controlled by  $p$ .
- Inverse quantum Fourier transform of  $p$ , controlled by  $p$ .

Then we need to define the decimal implementation, because we work with decimal numbers, in particular we define:

- Decimal implementation of multiplication.
- Decimal implementation of subtraction. It takes into account the relative numbers, that is the case in which the first number is minor than the second one by putting a - sign.

Another useful method that has been implemented is the quantum determinant function of a 2x2 matrix.

Let us now put all together for the final algorithm for price prediction:

- Starting from  $\theta = (X^T X)^{-1} X^T y$  we calculate classically  $X^T X$ . This provides a generic 3x3 matrix.
- We use the minmax function to get the min and max values, and then we call the function `standardizeMatrix` which returns an integer scaled matrix with values from 0 to 7.
- Then we call `getMatrixInverse` to obtain the inverse of the standardized.
- Then we de-standardize the inverse obtaining an approximated inverse of the original matrix (the approximation depends on the variance of the values in the matrix).
- We complete the elaboration providing the q-calculated theta and we use it to perform the price prediction.

### 2.2.5 Results

Considering the dataset shown in Tab. 2.1, the algorithm lends itself well to predicting the price of a 1650 3-room house. What we want to point out in this section, however, is the complexity of managing the implementation of the solution. At first it was necessary to define the quantum circuit that complied with the requirements of the formal problem, then we moved on to the implementation of the basic functions, which were partly already defined by means of some libraries. Finally, we had to implement the function for the management of multiplication and subtraction with decimal digits, since, as in the case study, also in the final project for the calibration of the sensors need to work with decimal numbers is evident, as both the correlated variables and the variables subject to calibration they are expressed in decimal format. To better understand the results obtained, let us compare the result of the Qml algorithm with the standard one of the ml and with the mathematical solution, based on the use of matrices.

Keep in mind that the exact result, having a home with 1380 sq ft and 3 bedrooms, amounted to \$212,000

- Normal Solution algorithm result:  
 $\theta_0 = 89597.909$   $\theta_1 = 139210.674$   $\theta_2 = -8738.019$   
 Predicted value : \$255494.582  
 Absolute error on prediction: 43494.582  
 % error on prediction: 20.51%



Squared feet	Room number	Price
2104	3	399900 \$
1600	3	329900 \$
2400	3	369000 \$
...	...	...
1416	2	232000 \$
3000	4	539900 \$
1985	4	299900 \$

**Table 2.1:** Case study data-set, original length was 450 lines, here an example of few lines.

- Gradient Descent algorithm result  
 $\theta_0 = 340311.977$   $\theta_1 = 107880.128$   $\theta_2 = -5066.007$   
 Predicted value : \$256289.590  
 Absolute error on prediction: 44289.59  
 % error on prediction: 20.89%
- Quantum Computing algorithm result  
 $\theta_0 = -46903.716$   $\theta_1 = 98926.358$   $\theta_2 = 56727.495$   
 Predicted value : \$259797.144  
 Absolute error on prediction: 47797.144  
 % error on prediction: 22.54%

From the comparison of the results obtained, it is clear that none of the three methods can provide a reliable prediction, i.e., that it respects a percentage error below 10%.

The other observation that immediately stands out is that the QML solution has the lowest score.

Taking into account the above, it was considered easier to switch to a library that would allow all this to be managed with a greater degree of abstraction, the hybrid quantum classic machine learning choice is motivated by the ease of management and understanding of classical neural network models and by the greatest performance and reliability , through which the selective loading of data was efficiently managed according to the desired preferences.

## Chapter 3

# Air pollution sensors Project

In this chapter, what we are going to do is an introduction to the Weather Station Analysis project, of which Sensor Calibration occupies a central role. All reference tables and figures presented in this chapter are taken from a previous paper describing the overall project.[21]

### 3.1 Introduction

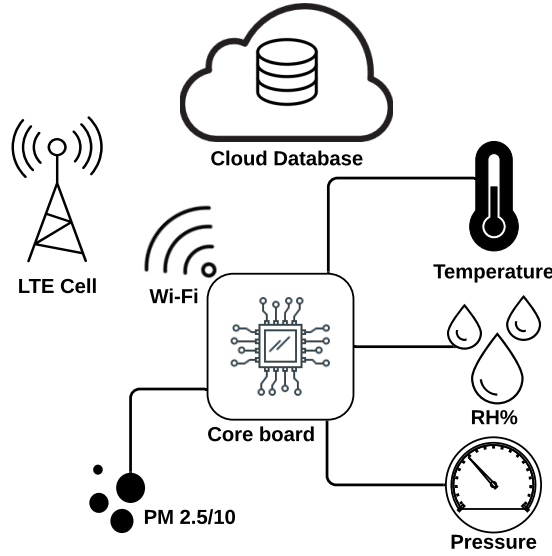
Air pollution, especially air quality and particulate matter, have recently fueled the interests of government, industry and academia to find new solutions for their monitoring, particularly what they have been trying to do is to understand if it is possible to monitor these environmental parameters through low-cost detection strategies. However, this has led to the need to analyze all the limitations of these low-cost sensors, in particular their sensitivity to aging, environmental conditions and cross-sensitivity to pollutants. The current solution to the problem, at least in part, has been to limit its use over time.

Within the project developed by the DAUIN research department, what has been developed is a low-cost particulate monitoring system, based on special acquisition cards, used to monitor air quality on sensor platforms. Different data acquisition and processing methods were analyzed in detail, in particular the aim is to, through the various calibration methods that will be proposed and analyzed, it was to understand if, given a value of a low-cost sensor, it is possible to calibrate this value in order to make it as reliable as possible with respect to the reference value given by ARPA.

The collected sensor data amount to about 50GB of data gathered in six months during winter. Tests of statically immovable stations include an analysis of accuracy

and sensors' reliability made by comparing our results with more accurate and expensive standard  $\beta$ -radiation sensors. Tests on mobile stations have been designed to analyze the reactivity of our system to unexpected and abrupt events. With respect to other approaches, the proposed methodology has been proved to be extremely easy to calibrate, to offer a very high sample rate (one sample per second), and to be based on an open-source software architecture.

### 3.2 Purpose of the project



**Figure 3.1:** Architecture of the project. The data coming from the sensors are first stored in Raspberry Pi, and then transferred to a remote server over the Wi-Fi network.[21, 3]

In this project, it was designed, builded, and verified a low-cost, open-source air-quality system which is based on special-purpose acquisition boards, it was deployed to stationary platforms, and it was devised for participatory sensing strategies. It was following article 18.5 of Italian Decree 155/2010 on the dissemination of air quality data, which absorbs EU directive 2008/50/CE.

### 3.3 Role of Sensors Calibration

Once the data has been collected and saved in a specific database, what was done was to focus on the most important part of the project, namely the calibration of the values and the validation of the calibrated data.

Regarding the data calibration process, the target was to maintain a fairly high and rigid standard, as particularly reliable calibration models are critical for the success of dense sensor networks deployed in urban areas of developed countries. In this scenario, pollutant concentrations are often found at the lower end of the spectrum of global pollutant concentrations, and poor signal-to-noise ratio and cross-sensitivity can hinder the network's ability to provide reliable results. With this in mind, the calibration phase has been designed with extreme care, using both the Multivariate Linear Regression model and the Random Forest [22], [23] machine learning algorithm.

Furthermore, the collected results were compared with different calibration methods and were validated over different periods to evaluate the quality of the result over time.

### 3.3.1 Calibration Strategies

In the original version of the paper, two calibration methods are exploited:

- *The MLR (Multivariate Linear Regression) Model:*

In Multivariate Linear Regression Models, regression analysis is used to predict the value of one or more responses from a set of predictors. Let  $(x_1, x_2, \dots, x_n)$  be a set of predictors (dependent variables) believed to be related to a response (independent) variable  $Y$ . The linear regression model for the  $j$ -th sample unit has the form

$$Y_j = \beta_0 + \beta_1 \cdot x_{j1} + \beta_2 \cdot x_{j2} + \dots + \beta_r \cdot x_{jr} + j \quad (3.1)$$

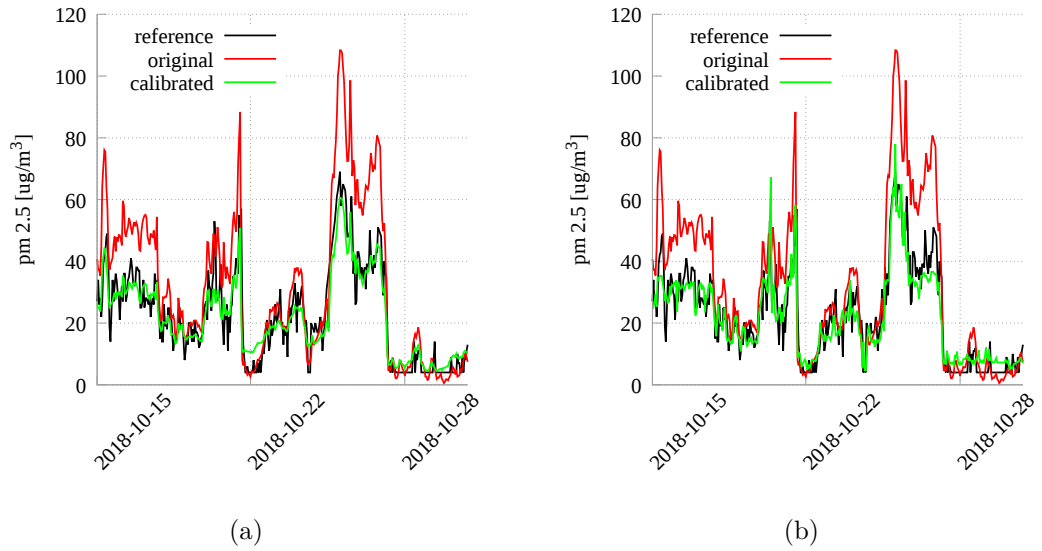
where  $j$  is a random error and the  $\beta$  are unknown (and fixed) regression coefficients. The value  $\beta_0$  is the intercept. With  $n$  independent observations, we can write one model for each sample unit or we can organize everything into vectors and matrices as:  $Y = X \cdot \beta + e$ . The training data are used to calculate the model coefficients, and the model performance is evaluated on withheld testing data. Separate MLR models are usually developed for each sensor and each measure.

- *The RF (Random Forest) Model:*

A Random Forest Model is a machine learning algorithm for solving regression or classification problems. It works by constructing an ensemble of decision trees using a training data set. The mean value from that ensemble of decision trees is then used to predict the value for the new input data. To develop a random forest model, we must specify the maximum number of trees that make up the forest, and each tree is constructed using a bootstrapped random sample from the training data set. The origin node of the decision tree is split into

subnodes by considering a random subset of possible explanatory variables. The training algorithm splits the tree based on which of the explanatory variables in each random subset is the strongest predictor of the response. This process of node splitting is repeated until a terminal node is reached. The user can specify the number of random explanatory variables considered at each node, the maximum number of subnodes, or the minimum number of data points in the node as the indication to terminate the tree.

### 3.4 Current Calibration process



**Figure 3.2:** Different kind of calibration methods used [21]

Considering that PM2.5 is strongly affected by the influence of meteorological factors, what has been done was to exploit this dependency and correlation between the sensor error and these weather factors. Particularly the Honeywell HPM115S0-XXX particulate sensor has a relatively high accuracy ( $\pm 15 \mu\text{g}/\text{m}^3$  from 0 to  $100 \mu\text{g}/\text{m}^3$ ), considering the extremely low price and the technology used. However, a filter and calibration of the collected data is necessary to eliminate possible offsets, peaks and linearity errors.

For this purpose, during the calibration phase, all the sensors were positioned near the ARPA fixed physical station in the City of Turin, which uses  $\beta$ -radiation technology to provide high-precision measurements. The ARPA station provides the hourly average data, which in this project have been used as a reference for all the connection data from the sensor cards. It tends to point out that the

hourly average data, obtained with the ARPA *beta* -radiation approach, are fully consistent with the measurements of the gravimetric sensor.

As for our boards, first of all various filters were applied to remove outliers (GaussianMix) and any unwanted data, such as peaks. In particular, what has been done is to calculate the average with a variable width window for the refinement of the analyzed samples. Subsequently, the collected values were grouped, initially on a 1 sample per second basis, by hourly average, in order to have data directly comparable with those of ARPA. The results were very satisfactory as, most of the samples are in the  $(\pm 15 \mu g/m^3)$  range, which is reasonable considering the sensitivity of the sensor.

Finally, the calibration process was performed on the hourly averages calculated. As previously introduced in this chapter, the methods we chose for calibration were: multivariate linear regression and the random forest. However, three types of MLR were applied, namely, using only temperature, humidity only, and both temperature and humidity. In all cases, RF calibration takes into account both temperature and humidity.

## 3.5 Validation process

### 3.5.1 Validation metrics

The way to quantify the accuracy of a fitting model is by minimizing some error function that measures the misfit between the output and the response function for any given value of the data set. In the following, we will use several metrics defined as in the `SciKitLearn` Python Library [24].

- The coefficient of determination is the proportion of the variance in the dependent variable that is predictable from the independent variable:

$$R^2(y, \hat{y}) = \frac{\sum_{i=0}^{n_{\text{samples}}-1} (\hat{y}_i - \bar{y})^2}{\sum_{i=0}^{n_{\text{samples}}-1} (y_i - \bar{y})^2} \quad (3.2)$$

- The mean squared error (MSE) measures the average of the squares of the errors. It is the second moment (about the origin) of the error and thus incorporates the variance of the calibration curve:

$$\text{MSE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \cdot \sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i)^2, \quad (3.3)$$

- The Mean Bias Error (MBE) is usually adopted to capture the average bias in a prediction.

$$\text{MBE}(y, \hat{y}) = \frac{1}{n_{\text{samples}}} \cdot \sum_{i=0}^{n_{\text{samples}}-1} (y_i - \hat{y}_i). \quad (3.4)$$

- The root mean squared error (RMSE) allows comparing different sizes of data sets, since it is measured on the same scale as the target value. It is obtained as the square root of the MSE, i.e.,

$$\text{RMSE}(y, \hat{y}) = \sqrt{\text{MSE}(y, \hat{y})}. \quad (3.5)$$

- The CRMSE is the Root Mean Square Error (RMSE) corrected for bias, i.e., it is defined as:

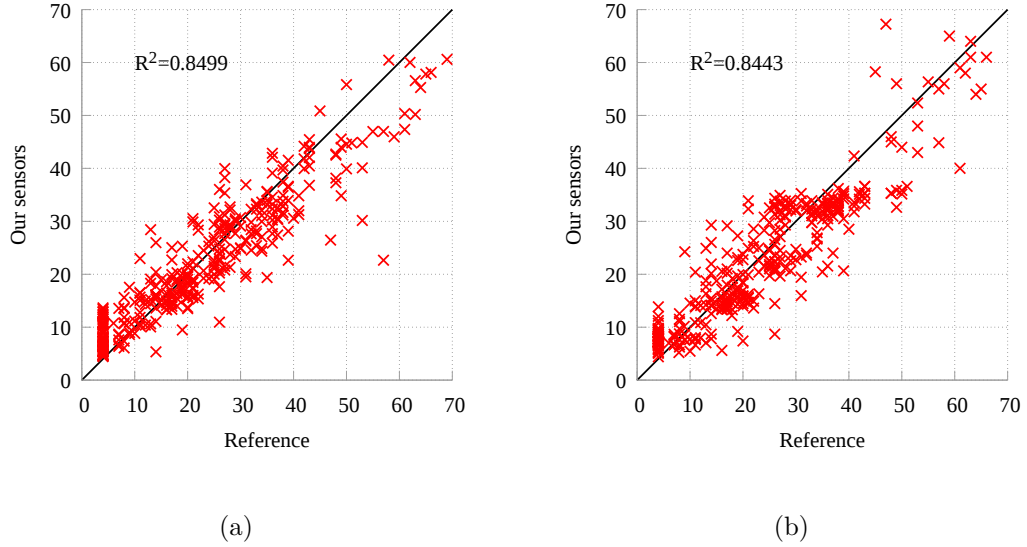
$$\text{CRMSE} = \text{RMSE} \cdot \text{sign}(\sigma_{\text{model}} - \sigma_{\text{reference}}) \quad (3.6)$$

where  $\sigma$  is the standard deviation of the measure.

- The correlation coefficient (Pearson product-moment correlation coefficient) is defined as the covariance of the variables divided by the product of their standard deviations.

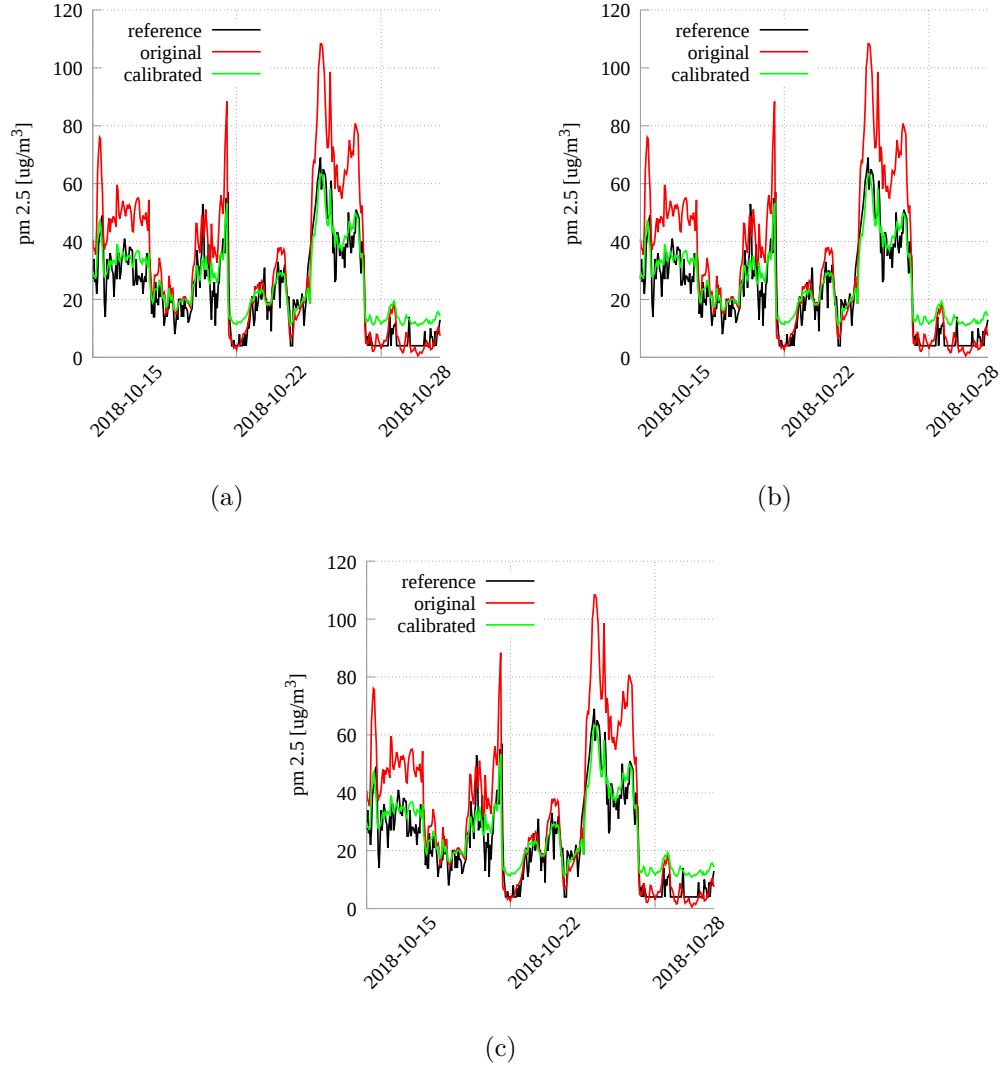
$$\rho_{y, \hat{y}} = \frac{\text{cov}(y, \hat{y})}{\sigma_y \cdot \sigma_{\hat{y}}} \quad (3.7)$$

### 3.5.2 The Validation Process



**Figure 3.3:** Validation process  $R^2[21]$

To test the performance of the two different calibration models, we first calibrate our sensors using the data collected in the first 2 weeks of October 2018. Then, we validate these calibration methods using samples collected in the last 2 weeks of the same month. In this period, we compare the concentrations obtained after



**Figure 3.4:** Time series comparing the reference, the raw, and the calibrated data using sensor 34, randomly selected. Calibration is performed using MLR. The different plots show MLR with different dependent variables, namely temperature (a), humidity (b), and temperature plus humidity (c).[21]



calibration with the measured reference concentrations.

Fig. 3.4 shows the obtained results for one single sensor (sensor 34), randomly selected using the MLR model. For this model, the three plots present the data gathered as dependent variables, only temperature, only humidity, and both variables as free variables. For all graphics, the calibrated plot is far more stable than the original one. However, it is not possible to outline a strategy that is clearly better than the other proposal.

To deepen our analysis, Figs. 3.2, 3.3 compares the MLR model with the RF one (again using sensor 34). As for Fig. 3.4, calibration is performed during the first 2 weeks of October and validation during the last 2 weeks. In this case, we use both the temperature and humidity as free variables. The charts report time series (Fig. 3.2(a) and 3.2(b)) and scatter plots (Fig. 3.3(a) and 3.3(b)). Unexpectedly (please see Zimmerman et al. [23]), our results show no advantage for the RF model with respect to the MLR one. On the contrary, the MLR model seems to outperform the RF model.

To better evaluate our results and to better assess the overall model performance, we performed calibration and validation tests for longer periods. A secondary target of this analysis is to find the best trade-off between the calibration effort and the error obtained. We consider calibration periods varying from the 2 weeks used so far up to 12 weeks, starting in October and ending in November 2018. In all cases, the validation period has been selected in December 2018.

Table 3.1 reports the RMSE (Root Mean Square Error) (i.e., the Root Mean Square Error, computed as defined by Equation 3.5) and the data correlation (computed following Equation 3.7) for a representative sensor with different calibration periods (2 and 12 weeks, respectively).

		2 weeks 323 samples			12 weeks 1851 samples		
		T	H	T+H	T	H	T+H
LR	RMSE	19.03	14.17	14.42	13.49	10.04	11.66
	Correlation	0.89	0.88	0.89	0.88	0.88	0.89
RF	RMSE	25.41	21.77	23.50	13.85	12.63	11.33
	Correlation	0.82	0.80	0.78	0.85	0.88	0.89

**Table 3.1:** Comparing different calibration techniques over different calibration periods (2 and 12 weeks, respectively), adopting the RMSE (Root Mean Square Error) metric. We consider all stationary sensors.

## Chapter 4

# Calibration of air pollution sensors

In this chapter is discussed the third calibration approach, such as using Quantum Machine Learning to calibrate the PM Sensors. The idea was born as an extension of what is illustrated in the previous chapter, that is, starting from known algorithms, such as that of Linear Regression in the classical field of machine learning, trying to extend this approach and related to the world of quantum machine learning that has recently made its way through advanced computing projects, which due to the high number of resources required become complex to manage in a traditional way.

### 4.1 The idea of a hybrid network

The main purpose of the `ws_analysis_qml.py` class is to perform a calibration of particulate sensors through the Linear Regression method using Angle Embedding, since we have a limited number of features, and the classical data are represented by float numbers, and one Single Qubit, because the required calculations on the data are quite simple.

#### 4.1.1 The concept of Quantum embedding

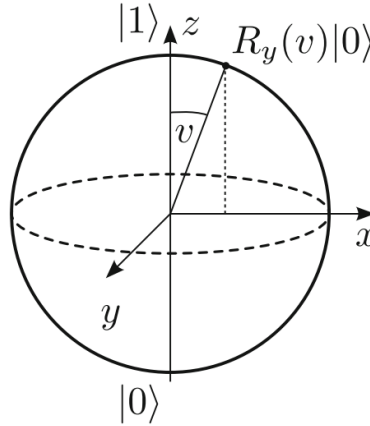
A quantum embedding represents classical data as quantum states in a Hilbert space via a quantum feature map. It takes a classical datapoint  $x$  and translates it into a set of gate parameters in a quantum circuit, creating a quantum state  $|\phi x\rangle$ . This process is a crucial part of designing quantum algorithms and affects their computational power. To embed this data into  $n$  quantum subsystems ( $n$  qubits or  $n$  qumodes for discrete and continuous variable quantum computing, respectively), we can use various embedding techniques:

- **Basis-Embedding:** each input is associated with a computational basis state of a qubit system. Therefore, classical data has to be in the form of binary strings. The embedded quantum state is the bit-wise translation of a binary string to the corresponding states of the quantum subsystems.
- **Amplitude-Embedding:** data is encoded into the amplitude of a quantum state. A normalized classical  $\mathcal{N}$ -dimensional datapoint  $x$  is represented by the amplitudes of a  $n$ -qubit quantum state  $|\phi_x\rangle$  as

$$|\phi_x\rangle = \sum_{i=0}^N x_i |i\rangle \quad (4.1)$$

where  $N = 2^n$ ,  $x_i$ , is the  $i$ -th element of  $x$ , and  $|i\rangle$  is the  $i$ -th computational basis state.

- **Angle-Embedding:** encodes  $N$  features into the rotation angles of  $n$  qubits, where  $N \leq n$ . This method is performed by applying rotations on the  $x$ -axis or  $y$ -axis using quantum gates along with the values that have to be encoded. If we want to apply angle embedding on a dataset, the number of rotations will be the same as the number of features in the dataset. The  $n$ -dimensional sample would take  $n$ -number of qubits to generate the set of quantum states. Fig. 4.1 show an example of  $y$ -rotation  $R_y(v)$  by an angle  $v$  around the  $y$  axis, that is, in the  $x$ - $z$ -plane.



**Figure 4.1:** Example of  $y$ -rotation  $R_y(v)$  by an angle  $v$  around the  $y$  axis, that is, in the  $x$ - $z$ -plane.[25]

## 4.2 Implementation of the algorithm

In this section, the key parts of the algorithm are explained.

### 4.2.1 Useful library import

In this preliminary phase, all the required libraries are imported, noticing that to perform several tests, both Tensorflow-gpu and Tensorflow-cpu libraries [26] will be used for testing and evaluating of the algorithm in the presence of Nvidia CUDA<sup>®</sup> Core GPU [27] (for reference all test with CUDA Core are performed with a Nvidia Quadro P2200 and Nvidia Tesla 4 Grapic Card). The library chosen to interface with quantum computing is PennyLane [28], that was preferred to use as, compared to the others taken into consideration, it was first of all the most complete in terms of documentation and secondly it did not require manual configuration of quantum circuits, thus managing to maintain a high level of code abstraction.

In addition to the specific libraries for classical and quantum Machine Learning, the potential of the `tt numpy` library has been used to perform calculations and prepare the data for the analysis; the `matplotlib` and `seaborn` libraries has been used for the plots of graphs and heatmaps.

### 4.2.2 Hybrid quantum-classical layer

The stages of the construction of the levels are as follows:

- We start from creating a single qubit to represent one feature and encoding it using AngleEmbedding Template.
- First, we create a device, then Qnode, that is defined as a quantum processor of at least one qubit.
- In Qnode we have a quantum function , where we accept an input on one wire, a qubit, in this case.
- StronglyEntanglingLayers [29] allows us to train Quantum Layers using parameters. This layer consists of single qubit rotations and entanglers, inspired by the circuit-centric classifier design [25]. The argument weight contains the weights for each layer.

```

1  class StronglyEntanglingLayers(weights , wires , ranges=None,
2  imprimitive=None, do_queue=True)

```

- Keras [30] will create the weight and pass it to our Qnode and in turn to Strongly-Entangling Layer Notice that a single Strongly-Entangling Layer can have multiple repeated layers, each of this layer has three trainable parameters that can be adjusted.
- The shape of the weights is hence [layer, no of qubits, 3].
- For the predicted output, we use a single neuron Dense Layer with linear activation.
- Since we have a linear regression problem, the selected loss type is the mean squared error (MSE).

The above results in the following Python implementation:

```
1 n_qubits = 1
2 layers = 3
3 data_dimension = 2
4 dev = qml.device("default.qubit", wires=n_qubits)
5
6
7 @qml.qnode(dev)
8 def qnode(inputs, weights):
9     qml.templates.AngleEmbedding(inputs, wires=range(n_qubits))
10    qml.templates.StronglyEntanglingLayers(weights, wires=range(
11        n_qubits))
12    return [qml.expval(qml.PauliZ(i)) for i in range(n_qubits)]
13
14 weight_shapes = {"weights": (layers, n_qubits, 3)}
15
16 #Creating model levels
17 qlayer= qml.qnn.KerasLayer(qnode, weight_shapes, output_dim=n_qubits)
18 clayer1= tf.keras.layers.Dense(n_qubits, activation='relu')
19 clayer2= tf.keras.layers.Dense(data_dimension, activation="linear")
20
21 # Setting model levels
22 model= tf.keras.models.Sequential([clayer1, qlayer, clayer2])
23
24 # Selection of the model Optimizer
25 opt = tf.keras.optimizers.Adam(learning_rate=0.01)
26
27 # Selection of the model Loss
28 model.compile(opt, loss='mse')
```

### 4.2.3 Import of the dataset for training and testing calibration

The data are subsequently imported and the setup of the training network is performed

```

1 X = tmp[regressor_list]
2 yarr = tmp['arpa']
3
4 X_train, X_test = split_list(X)
5 Y_train, Y_tets = split_list(yarr)
6
7 Y = np.array(Y_train, dtype=np.float32)
8 X = pd.DataFrame(X_train.values.reshape(len(X_train), 2))
9 Y = Y.reshape((len(Y_train), 1))
10
11 scale_x = MinMaxScaler()
12 x = scale_x.fit_transform(X)
13 scale_y = MinMaxScaler()
14 y = scale_y.fit_transform(Y)
15
16 # Plot of data pre-calibration
17 no_calib = plt.figure(figsize=(50, 10))
18 plt.plot(Y, label="Arpa Rif.", linewidth=2)
19 plt.plot(X[0], label="Sensor noCalib", linewidth=2)
20 plt.legend(loc=2, prop={'size': 20})
21 plt.show()

```

### 4.2.4 Training neural network

Here we train the combined Hybrid Model and plot the summarized history for loss, the N\_EPOCH and BATCH\_SIZE parameter values will be discussed in the next chapter, when we will talk about hyperparameter tuning.

```

1 # Training the Quantum Hybrid Neural Network
2 # Key parameter Number of epoch = N_EPOCH
3 # and Batch Size = BATCH_SIZE
4 # will be set ad hoc after the refinement
5 # process for each single usable configuration
6 # of the calibration method
7
8 history = model.fit(x, y, epochs=N_EPOCH, batch_size=BATCH_SIZE)

```

## 4.2.5 Sensor calibration and Loss analysis

After the train phase, we can perform a test with some new values to understand the reliability of the trained network.

```
1 model.summary()
2
3 # Plot the Loss summary
4 plt.plot(history.history['loss'])
5 plt.title('model loss')
6 plt.ylabel('loss')
7 plt.xlabel('epoch')
8 plt.legend(['train'], loc='upper left')
9 plt.show()
10
11 # Restore dataset before check
12 # calibration result
13 X = tmp[regressor_list]
14 X = pd.DataFrame(X.values.reshape(len(X), 2))
15
16 Y = np.array(yarr, dtype=np.float32)
17 Y = Y.reshape((len(yarr), 1))
18
19 scale_x = MinMaxScaler()
20 x = scale_x.fit_transform(X)
21
22 scale_y = MinMaxScaler()
23 y = scale_y.fit_transform(Y)
24
25 # Predict calibrated value
26 pred = model.predict(x)
27
28 res = scale_y.inverse_transform(pred)
29
30 # Saving result in CSV format
31 # for further analysis
32 pd.DataFrame(res).to_csv("results/result.csv")
33
34 # Plot uncalibrated vs calibrated data
35 calib = plt.figure(figsize=(50, 10))
36 plt.plot(Y, label="Arpa", linewidth=3)
37 plt.plot(X[0], label="uncal sensor", linewidth=2)
38 plt.plot(res[:, 0], label="cal sensor", linewidth=3)
39 plt.legend(loc=2, prop={'size': 20})
40 plt.gcf().autofmt_xdate()
41 plt.show()
```

### 4.2.6 Evaluation of the error percentage

Finally, in the test phase of the algorithm, before the integration with the `ws_analysis.py` class, a plot of %error was performed to have an estimation of how good is the calibration. The target error threshold has been fixed at 10%

```

1 def errorFunc(data):
2     return (data - np.min(data)) / (np.max(data) - np.min(data))
3
4 percentage_error= []
5
6 for pred,real in zip(res[:,0],Y):
7     percentage_error.append(abs(pred-real)/real)
8
9 error_threshold = [10]* len(time_series)
10
11 error = plt.figure(figsize=(50, 10))
12 plt.plot(time_series, error_threshold, "red", label="10%")
13 plt.plot(time_series, errorFunc(percentage_error)*100,
14          label="%Error",lineWidth=3)
15 plt.plot(time_train, np.zeros(len(time_train)), "red",
16          label="Train Set",lineWidth=4)
17 plt.plot(time_test, np.zeros(len(time_test)), "blue",
18          label="Test Set",lineWidth=4)
19 plt.legend(loc=2, prop={'size': 20})
20 plt.gcf().autofmt_xdate()
21 plt.show()

```

### 4.2.7 Example of ws\_analysis\_qml.py use

A complete call example of the calibration function is the following. The function requires the following parameters:

- **s**: Sensor value.
- **ws\_id**: Wheater Station ID (will be always 0, since we have only one Wheater Station).
- **start\_dt\_calibration**: begin of the time window where the calibration will be applied.
- **end\_dt\_calibration**: end of the time window where the calibration will be applied.
- **use\_temp\_bool**: say whether the temperature will be used as a correlated value.



- **use\_rh\_bool**: say whether the relative humidity will be used as a correlated value.
- **hreshold\_value**: specifies the maximum value used for peak suppression.
- **report\_mode\_bool**: enables / disables the detailed report mode, including printing of the graph for each single step of the algorithm. This parameter will subsequently be divided in the tuning phase to facilitate access to some graph and metric generating components of the algorithm.

```

1 nBoards = 12
2 boardNumbers = [10]
3
4 start_dt_calibration = '2018-10-16'
5 end_dt_calibration = '2018-10-31'
6
7 calKind = 'qml'
8 pmKinds = ['pm10']
9
10 for s in board_list[board][pmKind]:
11     strS = "s" + str(s)
12     #others cal_kind are skipped
13     if calKind == 'qml':
14         calRet =
15             x.calibrate_sensor_qml(
16                 s, 0,
17                 start_dt_calibration,
18                 end_dt_calibration,
19                 use_temp= use_temp_bool,
20                 use_rh= use_rh_bool,
21                 threshold= threshold_value,
22                 report= report_mode_bool
23             )

```

## Chapter 5

# Tuning, Simulations, and Results

In this chapter, it is reported the hyperparameter tuning phase, the simulation plots are exploited, and finally the obtained results are evaluated.

### 5.1 Tuning of hyperparameter

In this preliminary phase, the main purpose is to search for the optimal hyperparameters to have a calibration whose error percentage is as much as possible below the threshold set at 10%.

The hyperparameters taken into consideration are the following, for each of them, the range of values through which they have been tested is proposed in Tab. 5.1.

To reduce the computational time, all other available calibration methods have

Hyperparameter	Proposed values
Number of Epoch	32, 64, 128, 256, 512
Batch Size	5, 10, 15, 20, 25, 30, 50
Learning Rate	0.001, 0.005, 0.01, 0.1

**Table 5.1:** Hyperparameters tuning phase proposed value

been disabled from the `ws_analysis.py` class. In addition, a special wrapper function has been created, as shown below, which allows the export of the results in CSV.

Subsequently, these files were analyzed using a specific script that outlines the salient features. In particular, two evaluation metrics have been defined for each individual configuration.

The Result Score was defined as

$$RS = \frac{\sum_{i=0}^N |arpa_i - result_i|}{N} \quad (5.1)$$

Then the average of the Percentage Error, e.g the Error Score is:

$$ES = \frac{\sum_{i=0}^N \frac{|arpa_i - result_i|}{arpa_i}}{N} \quad (5.2)$$

It should be noted that, for the metrics just defined, the lower value represents a better data.

```

1      x.calibrate_sensor_qml(
2      s,
3      0,
4      start_dt_calibration,
5      end_dt_calibration,
6      use_temp=False,
7      use_rh=True,
8      threshold=150.0,
9      report=False,
10     plot=True,
11     plot_name=    "/perMin_" + str(<value>) +
12                "_perMax_" + str(<value>),
13     directory=directory,
14     min_percentile=min_percentile,
15     max_percentile=max_percentile,
16     use_quantile=<value>,
17     is_tuning=<value>,
18     is_heatmap=<value>
19 )

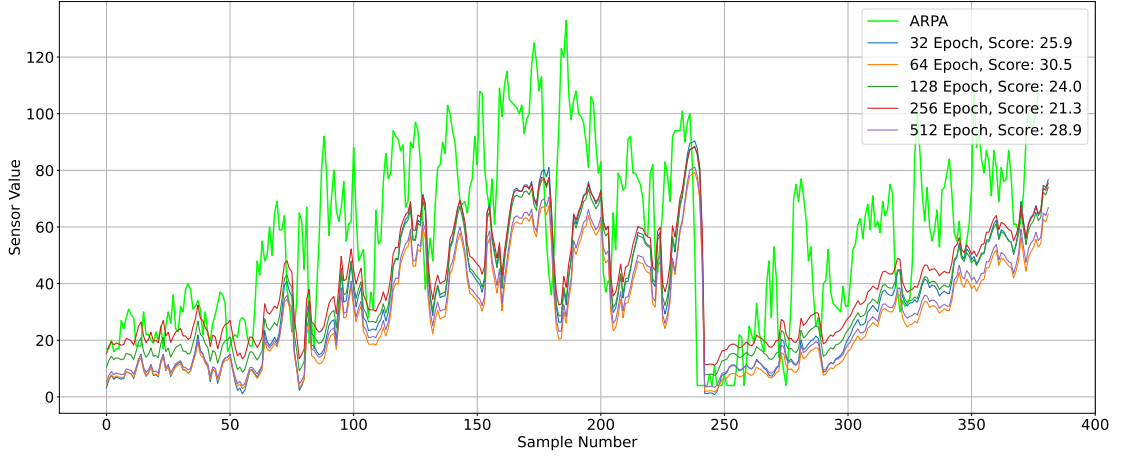
```

### 5.1.1 Tuning Number of epoches

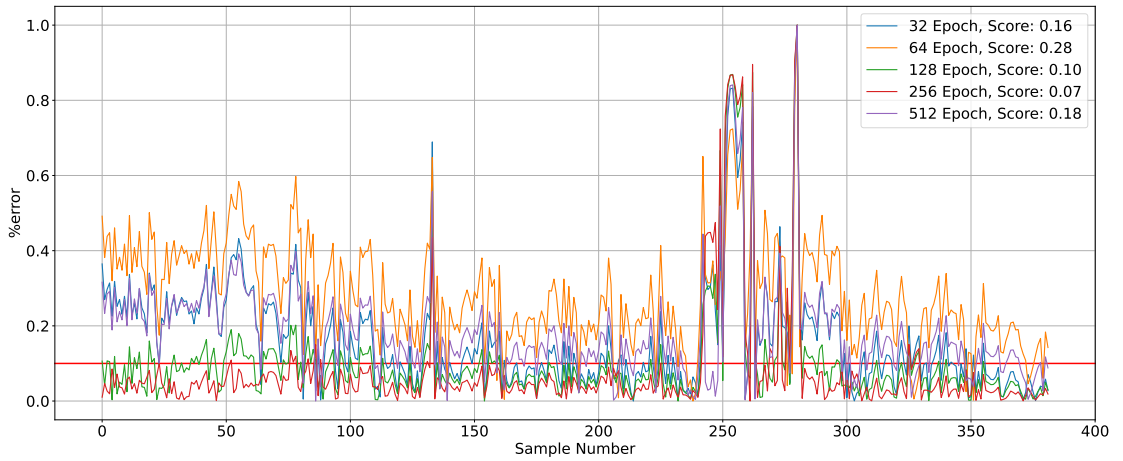
As first, we start by finding the best value of the number of epoches. This analysis is done by keeping a fixed value of Batch Size and Learning Rate to, respectively, 10 and 0.1. During the refining of the number of epoches, several configurations were tested:

- Configuration 1:  
 Batch Size = 10,  
 Learning Rate = 0.1,  
 Time Range = 2019/02/01 - 2019/02/16,  
 Sensor: *s123*,  
 Type = *PM2.5*,  
 Percentile optimization = Disabled.
- Configuration 2:  
 Batch Size = 10,  
 Learning Rate = 0.1,  
 Time Range = 2019/02/01 - 2019/02/16,  
 Sensor: *s124*,  
 Type = *PM10*,  
 Percentile optimization = Disabled.
- Configuration 3:  
 Batch Size = 10,  
 Learning Rate = 0.1,  
 Time Range = 2019/02/01 - 2019/02/16,  
 Sensor: *s123*,  
 Type = *PM2.5*,  
 Percentile optimization = Enabled,  
 Percentile range = 0.33 - 0.66.
- Configuration 4:  
 Batch Size = 10,  
 Learning Rate = 0.1,  
 Time Range = 2019/02/01 - 2019/02/16,  
 Sensor: *s124*,  
 Type = *PM10*,  
 Percentile optimization = Enabled,  
 Percentile range = 0.33 - 0.66.

- Configuration 1:  
Batch Size = 10,  
Learning Rate = 0.1,  
Time Range = 2019/02/01 - 2019/02/16,  
Sensor: *s123*,  
Type= *PM2.5*,  
Percentile optimization = Disabled.

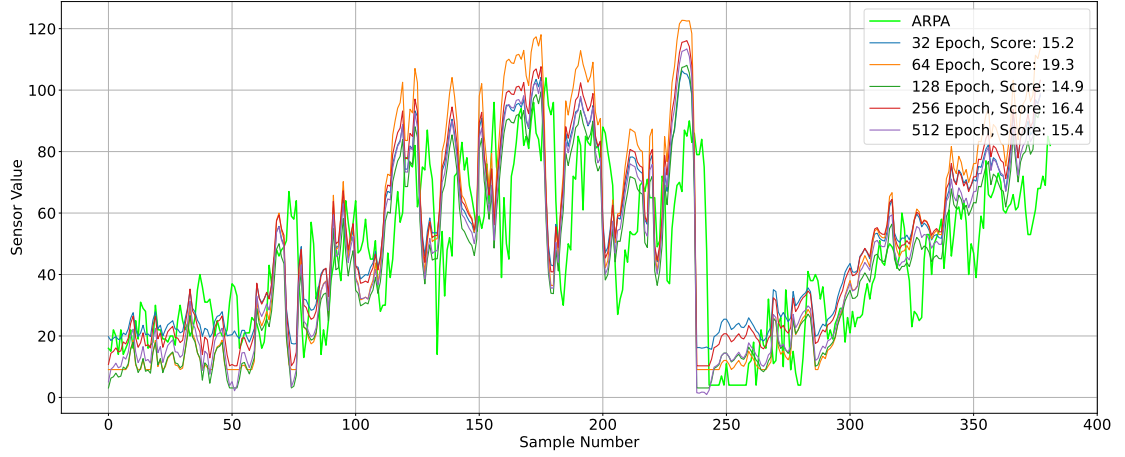


**Figure 5.1:** Results Score graph, configuration 1: PM2.5 - offQuantile

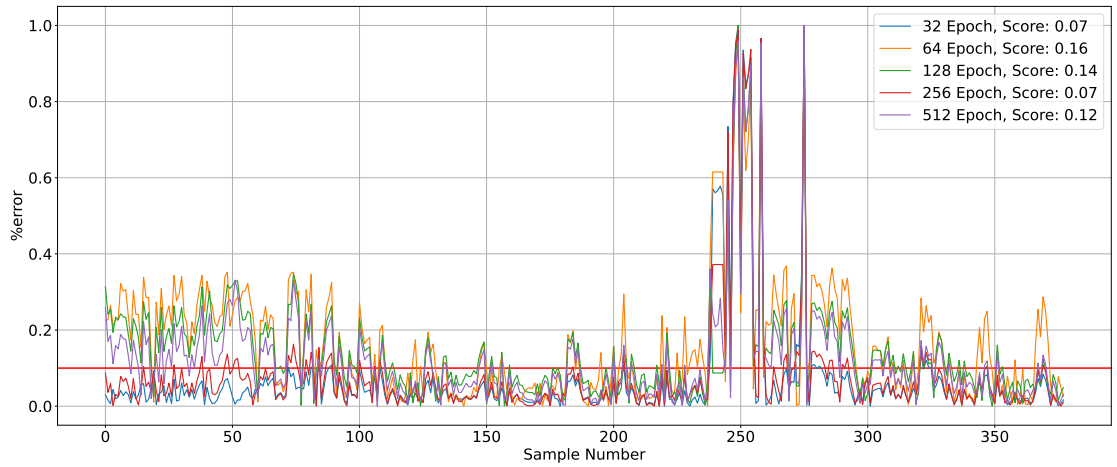


**Figure 5.2:** Errors Score graph, configuration 1: PM2.5 - offQuantile

- Configuration 2:  
Batch Size = 10,  
Learning Rate = 0.1,  
Time Range = 2019/02/01 - 2019/02/16,  
Sensor: *s124*,  
Type= *PM10*,  
Percentile optimization = Disabled.

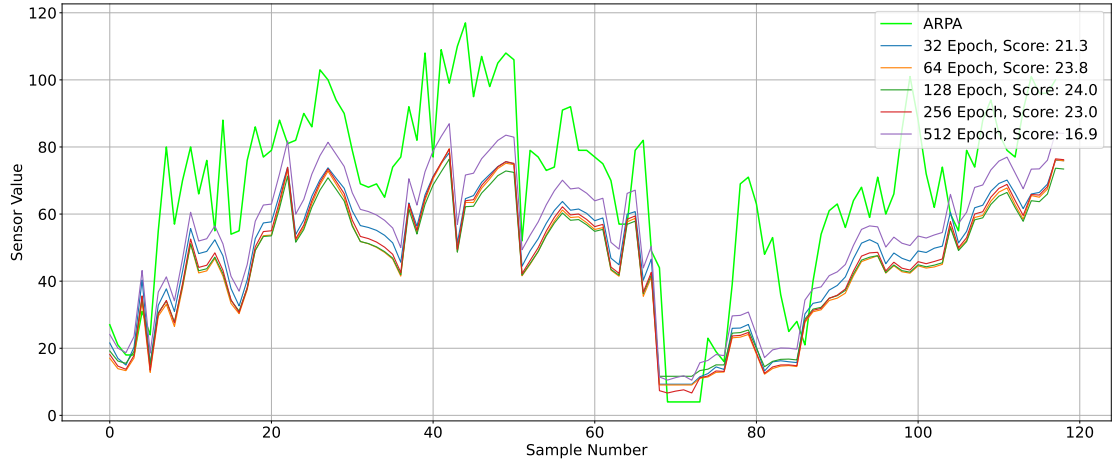


**Figure 5.3:** Results Score graph, configuration 2: PM10 - offQuantile

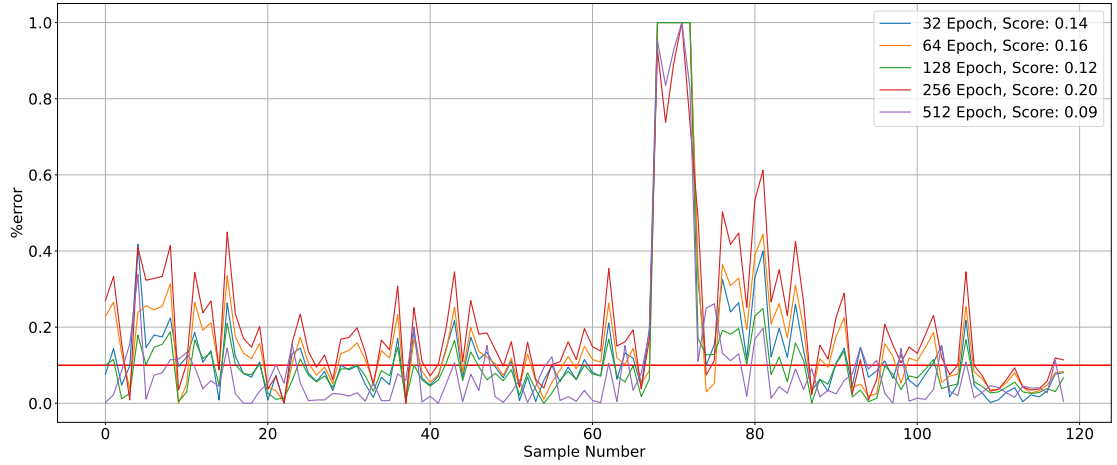


**Figure 5.4:** Errors Score graph, configuration 2: PM10 - offQuantile

- Configuration 3:  
 Batch Size = 10,  
 Learning Rate = 0.1,  
 Time Range = 2019/02/01 - 2019/02/16,  
 Sensor: *s123*,  
 Type= *PM2.5*,  
 Percentile optimization = Enabled,  
 Percentile range = 0.33 - 0.66.

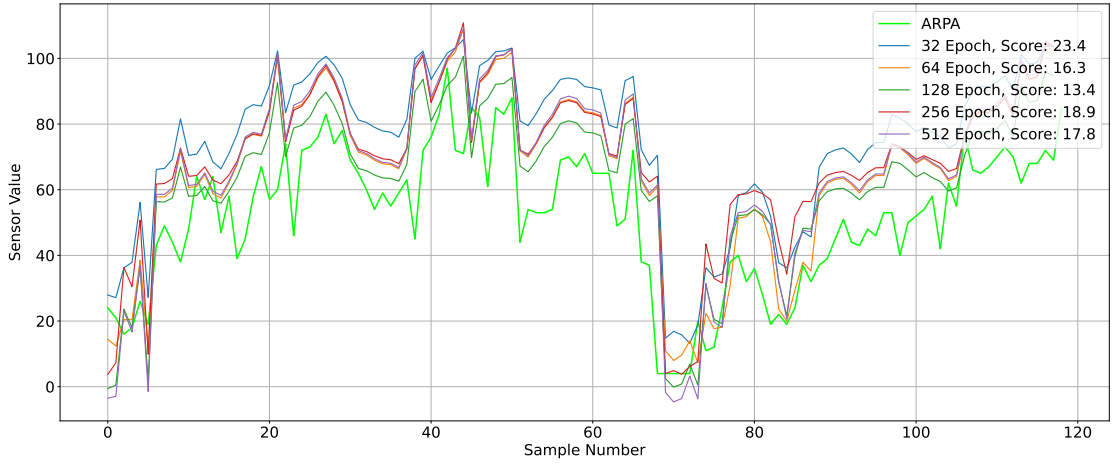


**Figure 5.5:** Results Score graph, configuration 3: PM2.5 - onQuantile

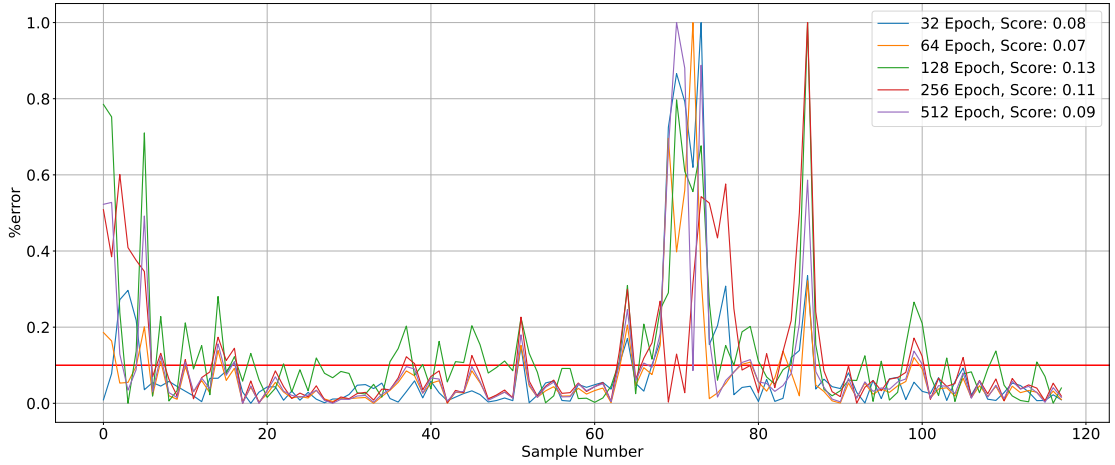


**Figure 5.6:** Errors Score graph, configuration 3: PM2.5 - onQuantile

- Configuration 4:  
 Batch Size = 10,  
 Learning Rate = 0.1,  
 Time Range = 2019/02/01 - 2019/02/16,  
 Sensor: *s124*,  
 Type= *PM10*,  
 Percentile optimization = Enabled,  
 Percentile range = 0.33 - 0.66.



**Figure 5.7:** Results Score graph, configuration 4: PM10 - onQuantile



**Figure 5.8:** Errors Score graph, configuration 4: PM10 - onQuantile



Configuration	Batch Size	SR	ES	SC
Config. 1 PM 2.5 - offQuantile	32	25.90	16 %	
	64	30.50	28 %	
	128	24.00	10 %	
	256	21.30	7 %	◀
	512	28.90	18 %	
Config. 2 PM 10 - offQuantile	32	15.20	7 %	◀
	64	19.30	16 %	
	128	14.90	14 %	
	256	16.40	7 %	
	512	15.40	12 %	
Config. 3 PM 2.5 - onQuantile	32	21.30	14 %	
	64	23.80	16 %	
	128	24.00	12 %	
	256	23.00	20 %	
	512	16.90	9 %	◀
Config. 4 PM 10 - onQuantile	32	23.40	8 %	
	64	16.30	7 %	◀
	128	13.40	13 %	
	256	18.90	11 %	
	512	17.80	9 %	

**Table 5.2:** Summary of obtained results of tuning number of epoches

Configuration	Epoch Number	SR	ES
Config. 1 PM 2.5 - offQuantile	256	21.30	7 %
Config. 2 PM 10 - offQuantile	32	15.20	7 %
Config. 3 PM 2.5 - onQuantile	512	16.90	9 %
Config. 4 PM 10 - onQuantile	64	16.30	7 %

**Table 5.3:** Number of epoches selected for each configuration proposed

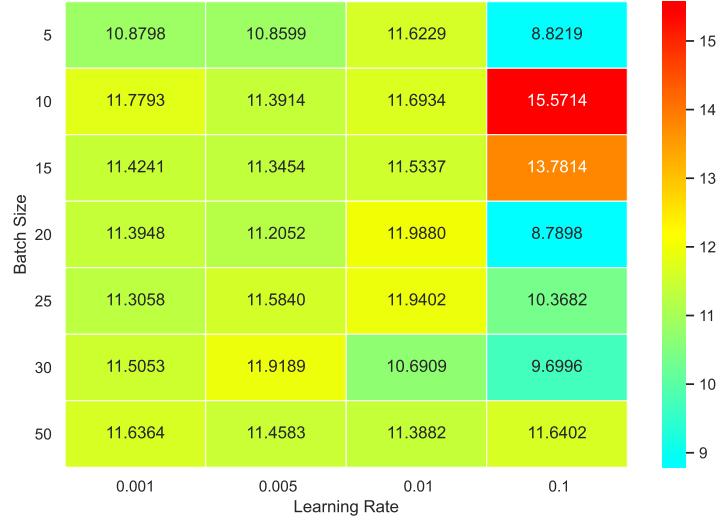
The Tab. 5.2 summarizes the results obtained, divided by configuration. On the basis of the results obtained, the optimal configuration is shown in Tab.5.3, by number of epoches for each single configuration proposed.

### 5.1.2 Tuning Learning Rate and Batch Size

Once the optimal number of epoches for each configuration was set, a gridsearch was carried out to find the best combination of Learning Rate and Batch Size for each configuration previously proposed (Config. 1-4). During gridsearch optimization, the same configurations as before were used:

- Configuration 1:  
Epoch Number = 256,  
Time Range = 2019/02/01 - 2019/02/16,  
Sensor: *s123*, Type= *PM2.5*,  
Percentile optimization = Disabled.
- Configuration 2:  
Epoch Number = 32,  
Time Range = 2019/02/01 - 2019/02/16,  
Sensor: *s124*,  
Type= *PM10*,  
Percentile optimization = Disabled.
- Configuration 3:  
Epoch Number = 512,  
Time Range = 2019/02/01 - 2019/02/16,  
Sensor: *s123*,  
Type= *PM2.5*,  
Percentile optimization = Enabled,  
Percentile range = 0.33 - 0.66.
- Configuration 4:  
Epoch Number = 64,  
Time Range = 2019/02/01 - 2019/02/16,  
Sensor: *s124*,  
Type= *PM10*,  
Percentile optimization = Enabled,  
Percentile range = 0.33 - 0.66.

- Configuration 1:  
 Epoch Number = 256,  
 Time Range = 2019/02/01 - 2019/02/16,  
 Sensor: *s123*, Type= *PM2.5*,  
 Percentile optimization = Disabled.

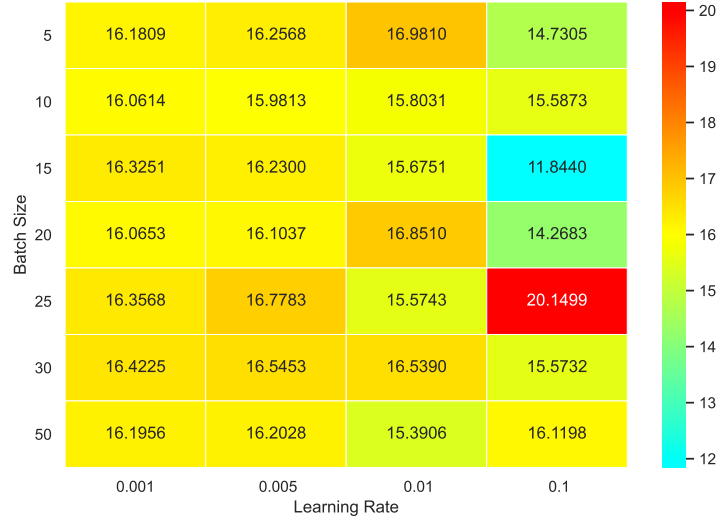


**Figure 5.9:** Results Score heatmap, configuration 1: PM2.5 - offQuantile



**Figure 5.10:** Errors Score heatmap, configuration 1: PM2.5 - offQuantile

- Configuration 2:  
 Epoch Number = 32,  
 Time Range = 2019/02/01 - 2019/02/16,  
 Sensor: *s124*,  
 Type= *PM10*,  
 Percentile optimization = Disabled.



**Figure 5.11:** Results Score heatmap, configuration 2: PM10 - offQuantile

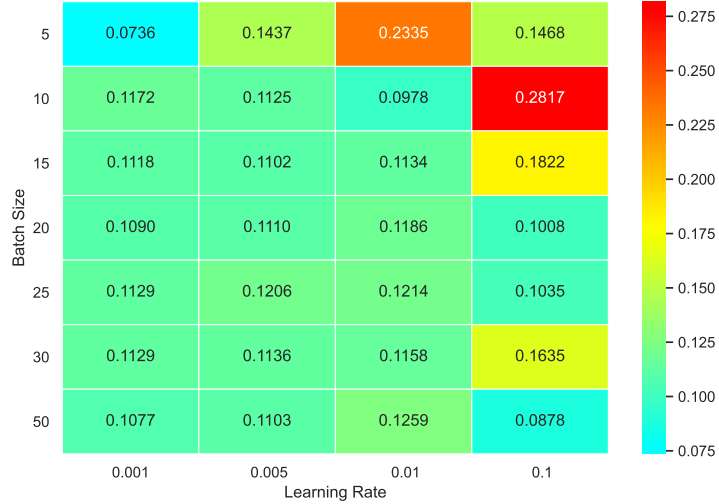


**Figure 5.12:** Errors Score heatmap, configuration 2: PM10 - offQuantile

- Configuration 3:  
 Epoch Number = 512,  
 Time Range = 2019/02/01 - 2019/02/16,  
 Sensor: *s123*,  
 Type= *PM2.5*,  
 Percentile optimization = Enabled,  
 Percentile range = 0.33 - 0.66.



**Figure 5.13:** Results Score heatmap, configuration 3: PM2.5 - onQuantile



**Figure 5.14:** Errors Score heatmap, configuration 3: PM2.5 - onQuantile

- Configuration 4:  
 Epoch Number = 64,  
 Time Range = 2019/02/01 - 2019/02/16,  
 Sensor: *s124*,  
 Type= *PM10*,  
 Percentile optimization = Enabled,  
 Percentile range = 0.33 - 0.66.



**Figure 5.15:** Results Score heatmap, configuration 4: PM10 - onQuantile



**Figure 5.16:** Errors Score heatmap, configuration 4: PM10 - onQuantile

On the basis of the results obtained, the optimal configuration is shown in Tab.5.4, by Learning Rate and Batch Size for each single configuration proposed.

Configuration	Learning Rate	Batch Size	SR	ES
Config. 1				
PM 2.5 - offQuantile	0.1	20	8.78	8.07 %
Config. 2				
PM 10 - offQuantile	0.1	15	11.84	6.94 %
Config. 3				
PM 2.5 - onQuantile	0.01	10	5.75	9.78 %
Config. 4				
PM 10 - onQuantile	0.1	10	8.44	7.32 %

**Table 5.4:** Learning Rate and Batch Size selected for each configuration proposed with metrics indicator

### 5.1.3 Tuning overall results

After tuning all hyperparameters of interest, i.e., epoch Number, Batch Size, and Learning Rate, Tab. 5.5 shows the optimal configuration for each possible configuration of the calibration method, i.e., for different PMs (2.5 and 10) and by enabling or not the quantile-based optimizer.

Configuration	Epoch Number	Learning Rate	Batch Size
Config. 1			
PM 2.5 - offQuantile	256	0.1	20
Config. 2			
PM 10 - offQuantile	32	0.1	15
Config. 3			
PM 2.5 - onQuantile	512	0.01	10
Config. 4			
PM 10 - onQuantile	64	0.1	10

**Table 5.5:** Final hyperparameter choiche for each configuration proposed

## 5.2 Validation of calibration strategy

After tuning all hyperparameters of interest, a reliability test of the calibration method was carried out based on the same metrics seen in Chapter 3. The purpose is to make a direct comparison between the calibration methods already present and the one proposed in this thesis, to highlight the potential of this alternative solution based on Quantum Machine Learning.

Before analyzing the results obtained by comparing the various calibration methods, the configuration of the `Violins.py` class is reported, i.e., the class used for comparing the calibrations carried out by the various methods.

### 5.2.1 PMs and Board Selection

Given the total of 12 boards present in the Weather Station, and given the high time required for calibration and data analysis, it was decided to continue the validation by choosing board number 10 as a reference. This choice is justified because, based on a preliminary analysis carried out in another project, it turns out to be the most reliable board.

We define first of all the number of boards as:

$$N_{boards,available} = 12 \quad (5.3)$$

Then we select only one to perform validation:

$$N_{selected\ board} = 10 \quad (5.4)$$

### 5.2.2 Percentile Optimizer range

The quantile-based optimizer arises from the idea that, at given relative humidity values, some sensors tend to deviate significantly from the real value of the PM. What this component does is an outlier detection calibration via GMM (Gaussian mixture model), which can be viewed as an extension of the ideas behind k-means. A Gaussian mixture model attempts to find a mixture of multidimensional Gaussian probability distributions that best model any input dataset. Tab. 5.6 summarizes the proposed configurations and, for each, the range used by the Gaussian mixture model.

### 5.2.3 Time window calibration and evaluation

The calibration time windows represent the set of data on which each calibration method trains. Different time windows of variable width were chosen, all in the range of dates between 1 October 2018 and 1 March 2019. In particular, the



Config.	Quantile range
Config. 1	0.00, 0.33
Config. 2	0.33, 0.66
Config. 3	0.66, 1.00

**Table 5.6:** Quantile range configurations

analysis focused on windows with a duration of 2, 4, 8, and 16 weeks. Otherwise, the evaluation time windows represent the set of data on which each calibration method performs its own test to understand if the neural network has succeeded or not in correctly fitting the data. As for the time windows of the training phase, the same methodology was also used in the test phase.

### 5.2.4 Xticks parameters

The Xticks parameters represent the points of interest used in the analysis of the metrics of the Violins class. Precisely, they are specific dates in which the proposed calibration result and the real value of the PM detected by ARPA will be observed. The criterion for choosing these parameters is in line with the previous time windows used in the training and evaluation phase. In particular, periods have been selected where the occurrence of particular atmospheric events has meant that the sensors will record singular events, such as the PM peak due to fireworks during New Year's Eve.

### 5.2.5 calKinds

The calKinds represent all calibration methods proposed and under analysis. Tab. 5.7 show all the proposed methods available and a brief description of them, with an indication of those chosen for comparison with the new method proposed.

Calibration method	Description	Selected
lr	Classic Linear Regression (LR)	YES
lrt	LR with Temperature	NO
lrh	LR with Relative Humidity	NO
lrth	LR with boot Temp. and Rel. Hum.	NO
rf	Random Forest	YES
qml	Quantum Machine Learning LR	YES

**Table 5.7:** Calibration method proposed for validation process

### 5.2.6 Metrics and Coefficients

The metrics through which the existing calibration methods and the proposed new one will be evaluated and compared are indicated in the metrics vector, there is also another coefficient vector where the coefficients returned by the calibration methods that support this type of data will be saved. This is not the case of the Random Forest and Quantum Linear Regression, as being neural networks do not return coefficients, but pretrained neural networks.

Tab. 5.7 recaps all the used metrics and their tags.

Metrics tag	Description
mae	Mean absolute error
mse	Mean Square Error
rmse	Root Mean Square Error
corr	Correlation coefficient
r2score	Coefficient of determination

**Table 5.8:** Metrics for the validation of calibration method

## 5.3 Validation result phase

In this section, the obtained results will be proposed, compared with the preexisting calibration methods, such as Linear Regression, Random Forest, and Quantum Linear Regression.

This was done through the `Violins.py` class, which gives different calibration methods is able to return different comparison metrics, in particular the ones we are going to see are:

- MAE (Mean absolute error)
- MSE (Mean Square Error)
- RMSE (Root Mean Square Error)
- CORR (Correlation coefficient)
- R2SCORE (Coefficient of determination)

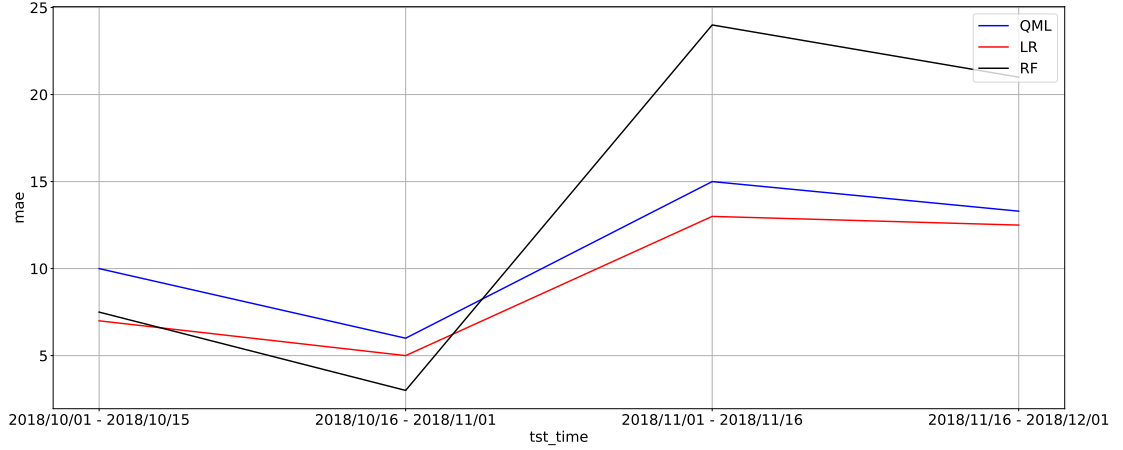
Considering the multitude of time ranges on which the new proposed calibration method has been validated, the results obtained over a single period, chosen randomly, will be shown below. In order to be complete, all comparison graphs generated will be shown for all proposed `Xtick` values.

The chosen time window is : 2018/10/01 - 2018/12/01

### 5.3.1 Mean absolute error results analysis

The Mean Absolute Error median plots are shown below. As it can be seen if Fig. 5.17, the error of the QML method is slightly higher compared to the LR, while it is much better than the RF algorithm.

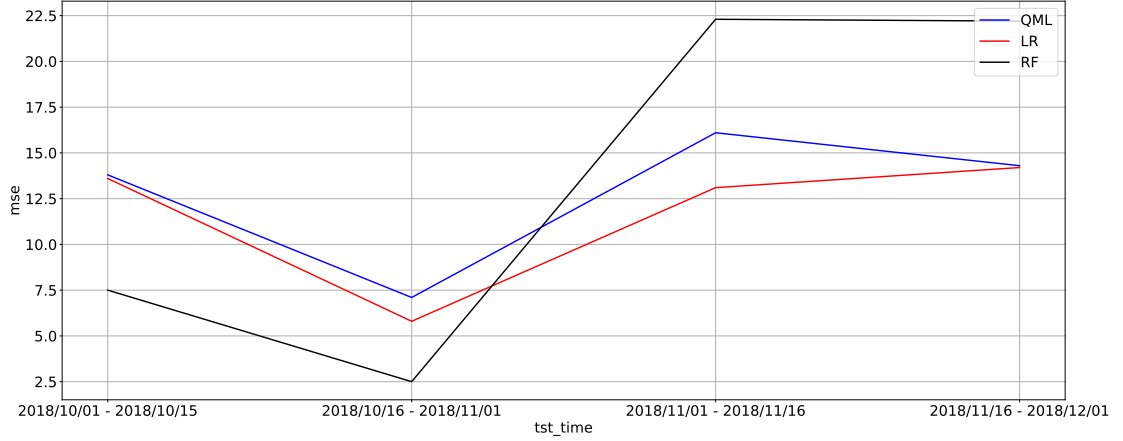
This result is motivated by the fact that the data is processed several times in more cycles and with greater refinement than RF. The error threshold is therefore in line with what was expected.



**Figure 5.17:** Comparing MAE results on QML, LR, RF

### 5.3.2 Mean Square Error results analysis

The Mean Square Error instead indicates the average quadratic discrepancy between the values of the observed data, in our case the reference ARPA and the values of the estimated data, such as the output of the different calibration methods.

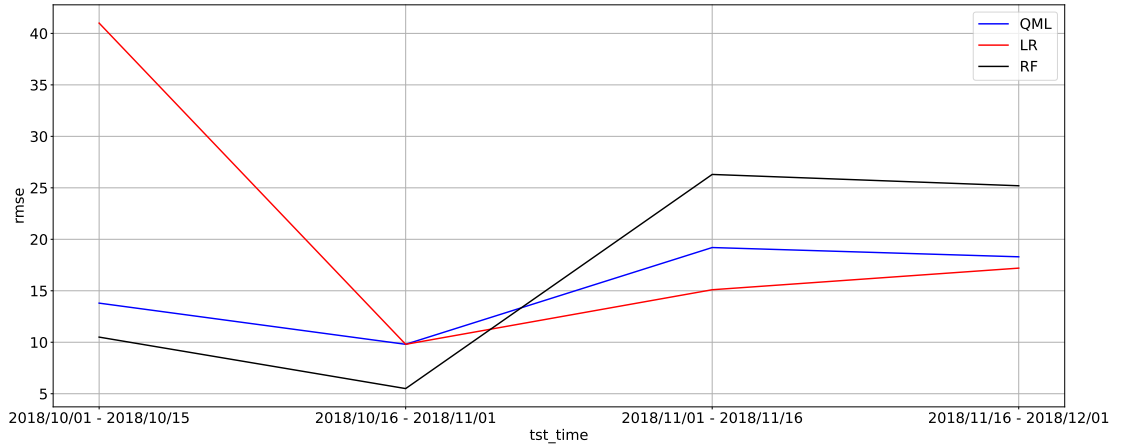


**Figure 5.18:** Comparing MSE results on QML, LR, RF

What emerges from this analysis is underlined in Fig. 5.18, even in this situation, the results perfectly follow our expectations, as seen in the previous section, and confirmed in this one, the method based on QML is always preferable to the RF, while the LR continues to have a lower score on errors.

### 5.3.3 Root Mean Square Error results analysis

The RMSE represents the square mean of these differences. These deviations are called residuals when calculations are performed on the data sample used for estimation and are called errors, or forecast errors, when they are calculated out of samples. RMSE is a measure of accuracy to compare the prediction errors of different models for a particular dataset and not between datasets, as it depends on the scale.

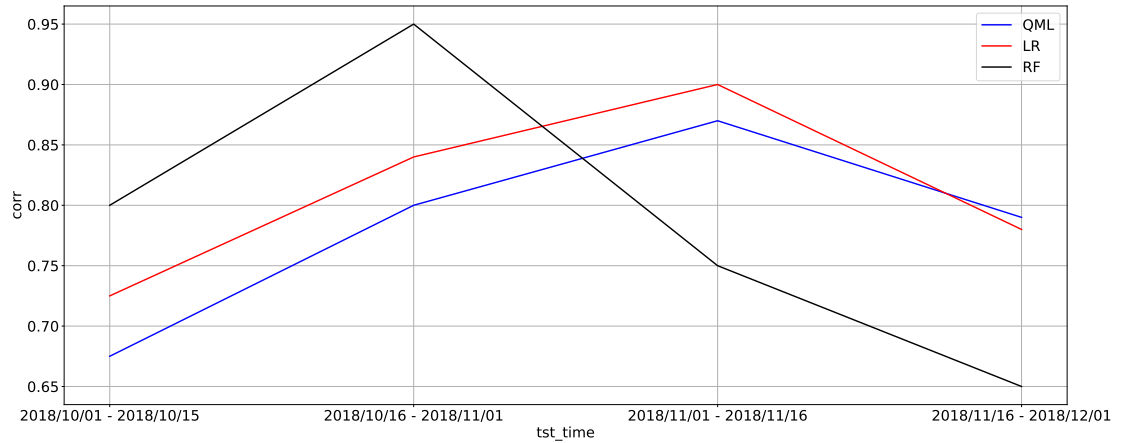


**Figure 5.19:** Comparing RMSE results on QML, LR, RF

Finally, also in this case, as showed in Fig. 5.19, the error metrics lead to the conclusion that the QML-based method has a reliability index that can be compared to that of the classical LR method. As for the comparison with the RF, it is difficult to define a winner, as the RF-based algorithm tends to have a high variance of data reliability, however, based on the tests and simulations performed, the reliability of the QML is being in most cases higher than that of the RF.

### 5.3.4 Correlation coefficient results analysis

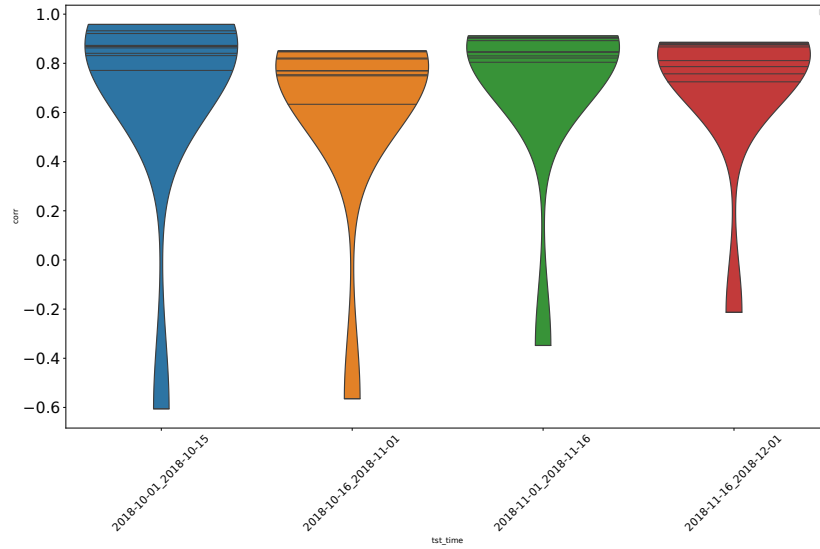
What can be seen from the Fig. 5.20 is that given the data correlation index, i.e., the linearity relationship possibly present between the calibrated data and those of the APRA reference, the RF calibration method is initially the most efficient, which is denied by subsequent tests. These tests show the optimal nature of the results obtained by the method based on QML and LR, the correlation index remains quite high, fully satisfying the preestablished requirements.



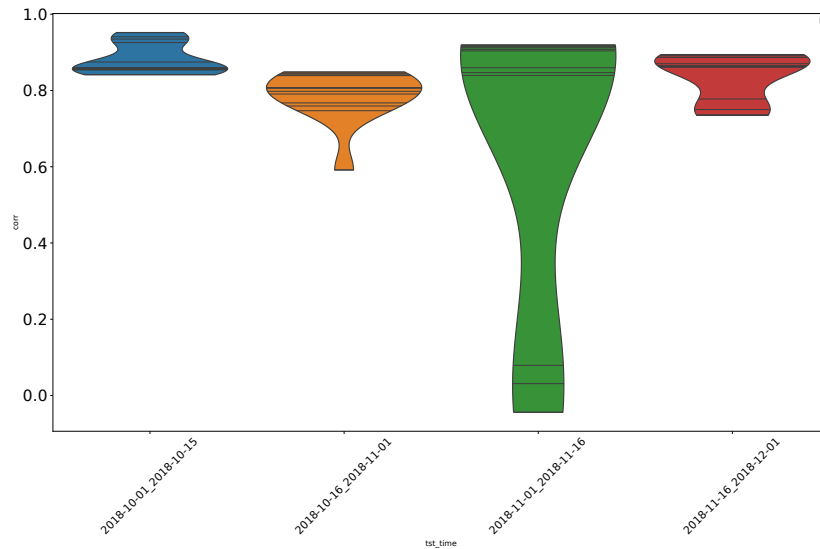
**Figure 5.20:** Comparing CORR results on QML, LR, RF

### 5.3.5 Correlation coefficient results, violin plots analysis

Violin plots are similar to box plots, except that they also show the probability density of the data at different values, usually smoothed by a kernel density estimator. As you can imagine a greater size for a given value, it indicates the presence of more data in a given range.

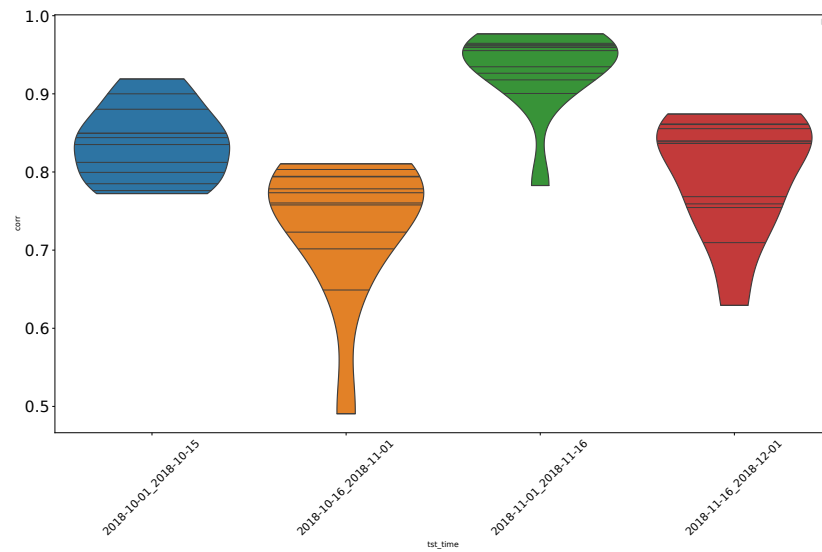


**Figure 5.21:** QML Correlation coefficient violin plots



**Figure 5.22:** LR Correlation coefficient violin plots





**Figure 5.23:** RF Correlation coefficient violin plots

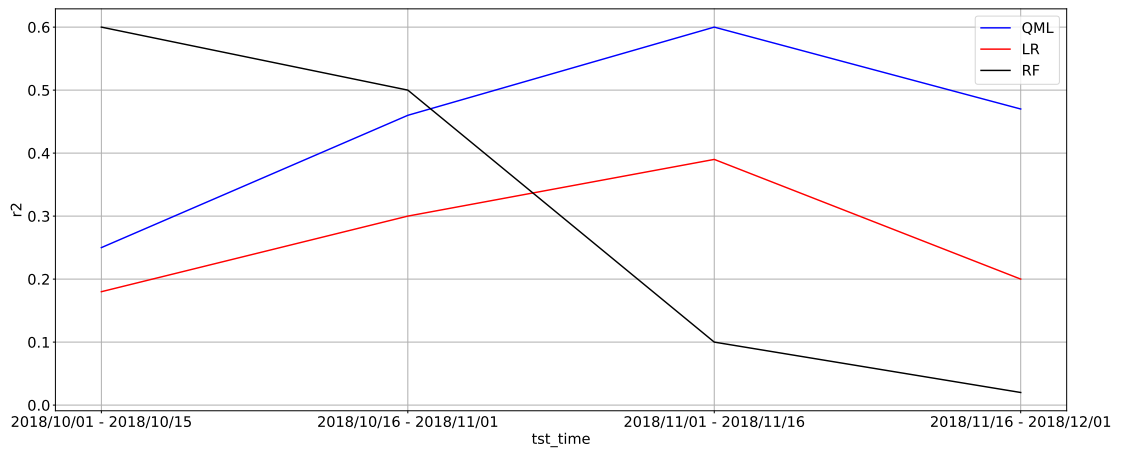
What can be seen from the graph above is first of all the presence of some outlier values, highlighted in the narrow lower part of the graph.

The other thing that stands out immediately to the eye is the concentration of the values in the high range, this is very positive, as being a correlation index, we can interpret this data as a strong correlation between the data calibrated by the QML method and those of the ARPA reference.

### 5.3.6 Coefficient of determination results analysis

Remembering the definition of correlation coefficient  $R^2$ , such as the proportion of variance in the dependent variable that is predictable from the independent variable. Fig. 5.24 show the obtained results. The metrics and test periods remain unchanged.

Analyzing the results obtained, we notice that  $R^2$  is in most cases is not very close to 1, so the calibration algorithm does not its job very well. However, the values obtained for the QML-based method are quite satisfactory and in line with what was expected, the gap compared to the previous calibration methods in terms of quality is remarkable.



**Figure 5.24:** Comparing  $R^2$  results on QML, LR, RF

## 5.4 Execution time analysis

The final step for the comparison of the calibration methods concerns the computational times involved. As previously mentioned, the method based on Quantum Machine Learning cannot be particularly fast, since, first of all, it is performed by means of a simulator of a quantum computer which the algorithm accesses through the PennyLane library.

Second, the training process requires a variable amount of execution time, based on the configuration of the hyperparameters. In particular, taking the worst case, that is, configuration number 3, it requires an Epoch Number equal to 512.

From the simulations carried out, the following data result:

- **Step execution time:**

$$T_{step} = 8s \quad (5.5)$$

- **Test execution time:**

$$T_{test} = 10s \quad (5.6)$$

These data, which correspond to a pure average of those obtained, will be used as a basis for the following calculations.

To perform the theoretical calculation of the overall execution time of the `violins.py` class, according to the parameters specified for execution, the following steps were performed:

1. First we define the time required to perform the training of the neural network as:

$$T_{train} = T_{step} \cdot N_{epoch} \quad (5.7)$$

2. Second, we define the overall time for testing as

$$T_{test,tot} = T_{test} \cdot N_{test\ windows} \quad (5.8)$$

3. Now we can define the overall time for the QLM calibration kind.

$$T_{QML} = (T_{train} + T_{test,tot}) \quad (5.9)$$

4. We define next the time required for one sensor, considering the number of quantile slots, as:

$$T_{sensor} = T_{QML} \cdot N_{quantile\ slot} \quad (5.10)$$

5. Now we have to remind that for each board there are a different number of sensors:

$$T_{board} = T_{sensor} \cdot N_{sensors} \quad (5.11)$$

6. Once we have reached the time required for execution on a single board we must take into account the type of calibration required, that is PM2.5 or PM10 or both.

$$T_{PMkinds} = T_{board} \cdot N_{selected\ PMkinds} \quad (5.12)$$

7. Finally we multiply the number just obtained by the number of training windows

$$T_{tot} = T_{PMkinds} \cdot N_{train\ windows} \quad (5.13)$$

Tab. 5.9 show the constants used for time analysis, since we use only a part of the available sensor data, since, as will be noted below, the execution times tend to grow very rapidly.

Variable	Description	Value
$N_{test\ windows}$	Number of test windows	20
$N_{quantile}$	Number of quantile range	3
$N_{calKinds}$	Number of cal kinds	1
$N_{sensors}$	Number of sensors	4
$N_{boards}$	Number of boards	1
$N_{pmKinds}$	Number of pmKinds	1
$N_{train\ windows}$	Number of train windows	20

**Table 5.9:** Constants for time analysis

Tab. 5.10 resume the execution time for each proposed configuration.

Configuration	Epoch Number	Execution time (hours)
Config. 1		
PM 2.5 - offQuantile	256	149
Config. 2		
PM 10 - offQuantile	32	30
Config. 3		
PM 2.5 - onQuantile	512	286
Config. 4		
PM 10 - onQuantile	64	47

**Table 5.10:** Execution time for each proposed configuration

## 5.5 Results on IBM QPU

To carry out this type of calibration through direct QPU, the PennyLane-Qiskit plugin was used, which allows automatic translation of information in quantum circuits.

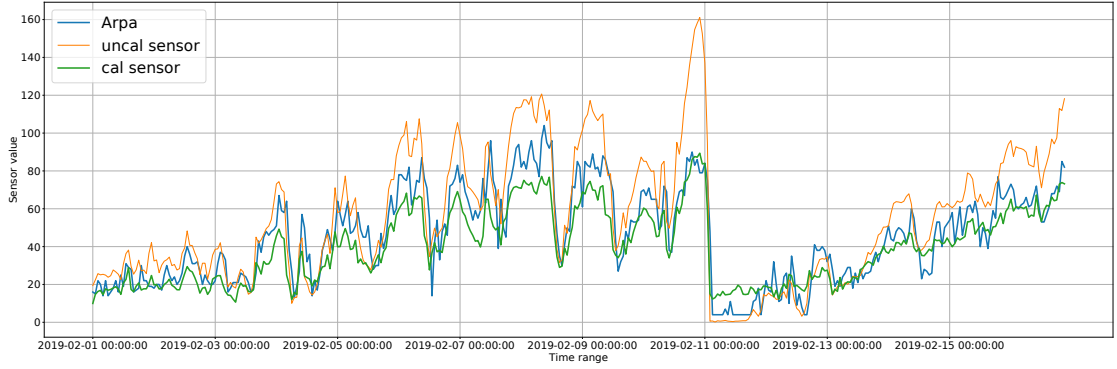
In particular, the simulator configuration is as follows:

- The qiskit.ibmq device has been chosen, which allows direct execution with auto translation of information to the remote QPU of IBM.
- The chosen backend is ibmq\_bogota, whose configuration includes a maximum total of 5 qubits, for a total quantum volume of 32 units. The quantum processor used is the Falcon r4L.

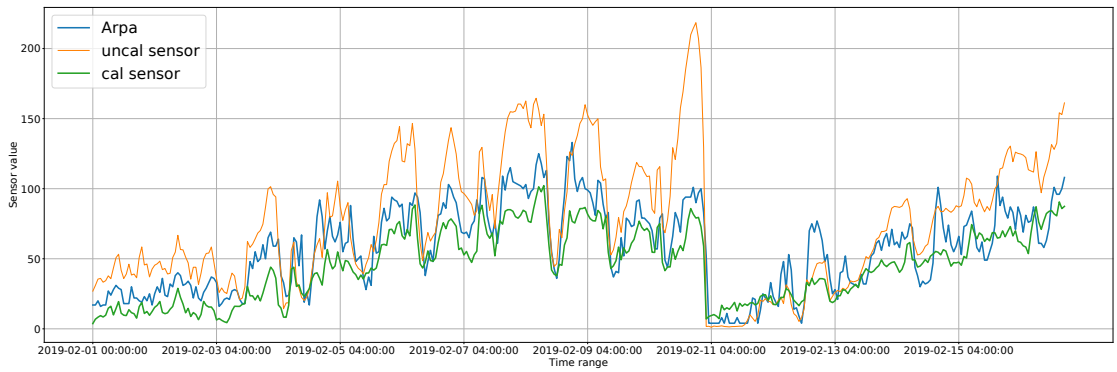
The main disadvantage, which led to the exclusion of the use of a remote QPU, was the fact that the chosen backend, like all others available, are shared among millions of users. Therefore, the single execution of a task requires not indifferent waits, and since for our calibration process the algorithm requires to iterate several times over the data, we preferred to use a local simulator, which at the expense of the quality of the data obtained, however, allowed to carry out all testing and validation processes in a reasonable time.

The configurations chosen for the simulation carried out on the quantum computer are the following:

- Configuration 1:  
Results in Fig. 5.25  
Epoch Number = 256,  
Learning Rate = 0.1  
Batch Size = 20  
Time Range = 2019/02/01 - 2019/02/16,  
Sensor: *s123*,  
Type = *PM2.5*,  
Percentile optimization = Disabled.
- Configuration 2:  
Results in Fig. 5.26  
Epoch Number = 32,  
Learning Rate = 0.1  
Batch Size = 15  
Time Range = 2019/02/01 - 2019/02/16,  
Sensor: *s124*,  
Type = *PM10*,  
Percentile optimization = Disabled.



**Figure 5.25:** QML simulation results on IBM Quantum, PM 2.5



**Figure 5.26:** QML simulation results on IBM Quantum, PM 10

What can be seen from the graphs is that, as expected, the simulation carried out on QPU is able to calibrate the sensors with a greater degree of precision than the simulator used locally.

However, as already mentioned previously, the long wait before receiving a response from the remote QPU prevents its intensive use, as the time to carry out a complete calibration would grow exponentially. This turns out to be one of the strongest limitations in the field of quantum programming today.

## Chapter 6

# Conclusions and further improvements

### 6.1 Conclusions

The `ws_analysis.py` class has been analyzed in detail, and based on the comparison parameters seen in the previous chapter, it can be said that the results are promising. Analyzing the comparison graph it is possible to note that in most cases the deviation in the absolute value between the ARPA reference and the calibrated value of a sensor remains in most cases below the threshold set at 10%.

Taking this into account, it is therefore possible to consider the method through Quantum Machine Learning a valid alternative to the calibration methods already proposed, as in terms of execution time, the results are perfectly in line with those of the other methods.

Although, as just said, QML represents a valid alternative, it is nevertheless necessary to take into consideration the biggest limiting factor found in this thesis. During the simulations carried out with the help of the IMB QPUs, we found that a significant waiting time to access the cloud resources was not among the largest QPUs, there was still a lot of pending jobs.

The conclusion we reach is therefore that the world of QML is certainly a promising world, through which very complex calculations can be carried out in a much shorter time than the classic ML, but, nowadays, this technology is not easily accessible and usable by everyone as the waiting times of the free QPUs made available by various companies cannot guarantee manageable execution times.

## 6.2 Further improvements

The next step to improve the only flaw that occurred during the implementation of the thesis is to make the calibration class multithreaded.

A multithreaded approach would significantly reduce the execution time, since, given the parallelizability of the operations, specifically the execution of the calibration method for each sensor, one could think of calibrating several sensors at the same time.

This would certainly lead to the problem of managing all synchronization mechanisms at the thread level, but on the other hand the gain in terms of execution time would be high.

What we intend to propose is a possible implementation of this class through the use of the Python 3.8 **Threading** Library, a class that acts as a wrapper to the internal part of the calibration, i.e. the passage through which the calibration algorithm is chosen and performed on the given sensor.

### 6.2.1 MultiThread Class Possible Implementation

```
1 class generateDFQuantileMultiThreadWrapper (threading.Thread):
2     def __init__(self,
3         threadID,          #Thread Identifier
4         pmKind,            #PM10 or PM2.5
5         board,             #Board ID
6         s,                 #Sensor ID
7         win_percentile,    #Percentile Ranges
8         calKind,           #Calibration Type
9         x,                 #ws_analisis self
10        df, dfCoeff,        #Coefficients
11        directory,         #Root for Plots
12        time_windows_calibration, #Windows for Calib
13        time_windows_evaluation  #Windows for Test
14    ):
15
16        threading.Thread.__init__(self)
17        #Omitted parameter self init
18
19        #Isolated Thread for train and test
20        def run(self):
21            print("Starting " + self.name)
22            #Run Calibration
23            print("Exiting " + self.name)
24            return df, dfCoeff
```



# Bibliography

- [1] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG] (cit. on pp. iii, 18).
- [2] Osvaldo Simeone. *A Brief Introduction to Machine Learning for Engineers*. 2018. arXiv: 1709.02840 [cs.LG] (cit. on pp. 2, 5, 11, 12).
- [3] Aurélien Géron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly Media, 2017. ISBN: 978-1491962299 (cit. on pp. 2, 31).
- [4] Erik Brynjolfsson and Tom Mitchell. «What can machine learning do? Workforce implications». In: *Science* 358.6370 (2017), pp. 1530–1534. ISSN: 0036-8075. DOI: 10.1126/science.aap8062. eprint: <https://science.sciencemag.org/content/358/6370/1530.full.pdf>. URL: <https://science.sciencemag.org/content/358/6370/1530> (cit. on p. 2).
- [5] Salma Jamal, Sukriti Goyal, Abhinav Grover, and Asheesh Shanker. «Machine Learning: What, Why, and How?» In: *Bioinformatics: Sequences, Structures, Phylogeny*. Ed. by Asheesh Shanker. Singapore: Springer Singapore, 2018, pp. 359–374. DOI: 10.1007/978-981-13-1562-6\_16. URL: [https://doi.org/10.1007/978-981-13-1562-6\\_16](https://doi.org/10.1007/978-981-13-1562-6_16) (cit. on p. 2).
- [6] George J. Aulisio. «Common Sense, the Turing Test, and the Quest for Real AI». In: *The European Legacy* 25.1 (2020), pp. 105–107. DOI: 10.1080/10848770.2019.1598142. eprint: <https://doi.org/10.1080/10848770.2019.1598142>. URL: <https://doi.org/10.1080/10848770.2019.1598142> (cit. on p. 3).
- [7] Ethem Alpaydin. *Introduction to Machine Learning*. 3rd ed. Adaptive Computation and Machine Learning. Cambridge, MA: MIT Press, 2014. ISBN: 978-0-262-02818-9 (cit. on p. 3).
- [8] Erik G Learned-Miller. «Introduction to supervised learning». In: *I: Department of Computer Science, University of Massachusetts* (2014) (cit. on p. 3).

- [9] Xin Yan and Xiao Gang Su. *Linear Regression Analysis*. WORLD SCIENTIFIC, 2009. DOI: 10.1142/6986. eprint: <https://www.worldscientific.com/doi/pdf/10.1142/6986>. URL: <https://www.worldscientific.com/doi/abs/10.1142/6986> (cit. on p. 5).
- [10] Stephen M. Stigler. «The Seven Pillars of Statistical Wisdom». In: (2016). DOI: 10.4159/9780674970199 (cit. on p. 6).
- [11] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. Adaptive computation and machine learning. MIT Press, 2009. ISBN: 9780262013192. URL: <https://books.google.co.in/books?id=7dzpHCHzNQ4C> (cit. on p. 6).
- [12] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738 (cit. on pp. 6, 8).
- [13] Haider Khalaf Jabbar and R. Khan. «Methods to Avoid Over-Fitting and Under-Fitting in Supervised Machine Learning (Comparative Study)». In: 2014 (cit. on p. 11).
- [14] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014. ISBN: 1461471370 (cit. on p. 13).
- [15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016 (cit. on p. 14).
- [16] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006 (cit. on p. 14).
- [17] *Documentation Reference — Qiskit 0.27.0*. URL: <https://qiskit.org/> (visited on 06/01/2021) (cit. on pp. 22, 24).
- [18] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. 10th. USA: Cambridge University Press, 2011. ISBN: 1107002176 (cit. on p. 23).
- [19] M. B. Plenio and V. Vitelli. «The physics of forgetting: Landauer’s erasure principle and information theory». In: *Contemporary Physics* 42.1 (2001), pp. 25–60. DOI: 10.1080/00107510010018916 (cit. on p. 23).
- [20] Maria Schuld, Ilya Sinayskiy, and Francesco Petruccione. «An introduction to quantum machine learning». In: *Contemporary Physics* 56.2 (2015), pp. 172–185. DOI: 10.1080/00107514.2014.964942 (cit. on p. 23).

- [21] Bartolomeo Montrucchio, Edoardo Giusto, Mohammad Ghazi Vakili, Stefano Quer, Renato Ferrero, and Claudio Fornaro. «A Densely-Deployed, High Sampling Rate, Open-Source Air Pollution Monitoring WSN». In: *IEEE Transactions on Vehicular Technology* 69.12 (2020), pp. 15786–15799. DOI: 10.1109/TVT.2020.3035554 (cit. on pp. 30, 31, 33, 35, 36).
- [22] Alessandro Bigi, Michael Mueller, Stuart K Grange, Grazia Ghermandi, and Christoph Hueglin. «Performance of NO, NO<sub>2</sub> low cost sensors and three calibration approaches within a real world application». In: *Atmospheric Measurement Techniques* 11.6 (2018), pp. 3717–3735. DOI: 10.5194/amt-11-3717-2018 (cit. on p. 32).
- [23] Naomi Zimmerman, Albert A. Presto, Srinivasa P. N. Kumar, Jason Gu, Aliaksei Hauryliuk, Ellis S. Robinson, Allen L. Robinson, and R. Subramanian. «A machine learning calibration model using random forests to improve sensor performance for lower-cost air quality monitoring». In: *Atmospheric Measurement Techniques* 11.1 (2018), pp. 291–313. DOI: 10.5194/amt-11-291-2018 (cit. on pp. 32, 37).
- [24] *API Reference — scikit-learn 0.21.3 documentation*. URL: <https://scikit-learn.org/stable/modules/classes.html%5C#module-sklearn.metrics> (visited on 06/01/2021) (cit. on p. 34).
- [25] Maria Schuld, Alex Bocharov, Krysta M. Svore, and Nathan Wiebe. «Circuit-centric quantum classifiers». In: *Physical Review A* 101.3 (Mar. 2020). ISSN: 2469-9934. DOI: 10.1103/physreva.101.032308. URL: <http://dx.doi.org/10.1103/PhysRevA.101.032308> (cit. on pp. 39, 40).
- [26] *API Reference — TensorFlow Core v2.5.0 documentation*. URL: [https://www.tensorflow.org/api\\_docs/python/tf](https://www.tensorflow.org/api_docs/python/tf) (visited on 06/01/2021) (cit. on p. 40).
- [27] *API Reference — CUDA Toolkit v11.4.0 documentation*. URL: <https://docs.nvidia.com/cuda/index.html> (visited on 06/01/2021) (cit. on p. 40).
- [28] *API Reference — PennyLane Core 0.16.0*. URL: <https://pennylane.readthedocs.io/en/stable/> (visited on 06/01/2021) (cit. on p. 40).
- [29] *API Reference — PennyLane: StronglyEntanglingLayers*. URL: <https://pennylane.readthedocs.io/en/stable/code/api/pennylane.templates.layers.StronglyEntanglingLayers.html> (visited on 06/01/2021) (cit. on p. 40).
- [30] *API Reference — Keras 2.4.3*. URL: <https://keras.io/api/> (visited on 06/01/2021) (cit. on p. 41).