POLITECNICO DI TORINO

Master's Degree in Data Science Engineering

Master's Degree Thesis

From Quantum Quirks to Clear Skies: A Not-So-Dusty Road to Sensor Calibration

Supervisors

Prof. Bartolomeo MONTRUCCHIO

Dr. Edoardo GIUSTO

Dr. Pietro CHIAVASSA

Dr. Mohamad GHAZI VAKILI

Candidate

Lorenzo BERGADANO

Academic year 2023-2024

Summary

The thesis delves into the methodology for calibrating optical fine-dust sensors using advanced techniques such as Deep Learning and Quantum Machine Learning (QML). The project's goal is to compare four advanced algorithms from the classical and quantum worlds in order to understand their differences and explore possible alternative methods to improve the accuracy and reliability of particulate matter readings in urban air quality monitoring.

By leveraging state-of-the-art technology, including feed-forward neural networks (FFNN), Long Short-Term Memory (LSTM) models, Variational Quantum Regressor (VQR), and Quantum LSTM (QLSTM), the research aims to address the limitations of current sensor calibration methods. These methods show promise in improving the calibration accuracy, thus enhancing the reliability of data used for environmental policy-making and public health campaigns.
A detailed comparison between classical and quantum machine learning models forms the core of the research. Through rigorous testing, including hyperparameter tuning and cross-validation, the thesis evaluates the performance of traditional neural networks against quantum models in sensor calibration, exploring whether for such problems a quantum advantage is foreseeable. Despite the nascent stage of quantum hardware and the current superiority of classical models in terms of accuracy, the findings highlight the promising future of quantum models in computational efficiency and potential scalability.

The thesis describes the practical implementation of the proposed calibration models, detailing the process of data preprocessing, model selection, and calibration. The methodology involves removing data anomalies, aggregating the data by making the median of all fine dust sensors in all installed boards, and comparing with reference measurements provided by ARPA, the regional environmental protection agency, in order to adjust sensor output.
A hyperparameter tuning and cross-validation pipeline is applied to individual models to better understand learning performance and possibly increase it.

The implementation of such models was done by building an open source framework in Python that allows integration with famous libraries such as Pytorch and Pennylane for building neural networks and quantum circuits respectively.
Such a framework facilitates the training and visualization phase of the results of any model, using the TensorBoard product as a graphical dashboard. At the end, the results obtained from the implementation of the classical and quantum models are illustrated and compared with each other, focusing on the advantages and disadvantages in using QML to solve the sensor calibration problem.

The comparative analysis between classical and quantum models in sensor calibration showcases a promising future for quantum models in environmental monitoring and other applications. The calibration results of the best models

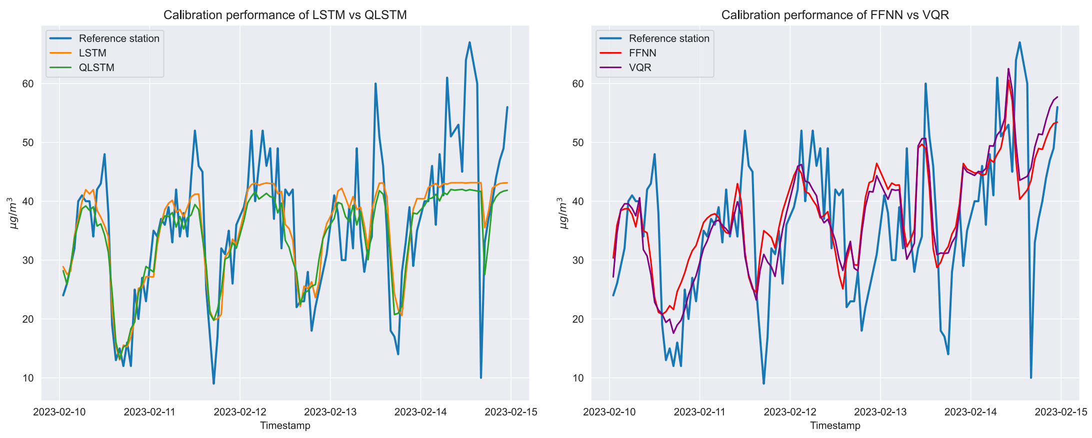


Figure 1: The calibration performance obtained from all models developed in this project in comparison with the reference station values, observing a limited time range from February 10 to 15, 2023.

Hyperparameters tuning result	**FFNN**	**VQR**	**LSTM**	**QLSTM**
Test-set size	25%	25%	30%	30%
L1 on test-set	2.9202	2.8577	2.7684	2.6978

Table 1: The minimum loss function calculated with the mean absolute error (L1) for each of the four models during the hyperparameters tuning phase.

found during hyperparameters tuning are shown in Table 1. These results were calculated on the same test-set portion and with the same loss function in order to be compared with each other.

Finally, through cross-validaton and the study of the weights used in the architecture, it was possible to demonstrate how quantum models are able to produce generalized learning on the dataset under consideration and use fewer weights to solve the problem than classical models.
It is shown that the LSTM model, which uses a total of 482 weights and no dropout techniques, tends to overfit on the dataset compared to the QLSTM model, which, in contrast, uses only 66 weights and achieves similar performance on each fold of the validation.
The Variational Quantum Regressor (VQR) surpasses classical feed-forward (FFNN) models by leveraging quantum phenomena like entanglement and superposition, enabling more efficient exploration of data's parameter space and the discovery of complex patterns missed by classical neural networks.
The public repository where the framework's source code resides at its most up-to-date version is available on the *official GitHub repository*.

In summary, the research makes a significant contribution to the comparison of Deep Learning and Quantum Machine Learning algorithms in the field of sensor calibration. By building an open source framework, it is possible to quickly and easily visualize model training results and discover how the innovative implementation of Quantum LSTM achieved a higher level of accuracy with less architectural complexity.
This suggests the potential of this research to make environmental monitoring techniques and urban management using Artificial Intelligence algorithms faster and more accessible. Future research directions include refining quantum algorithms, exploring new machine learning techniques, and expanding applications to diverse sensors and systems. The promising future of Quantum Machine Learning (QML) in environmental science and beyond, fueled by quantum computing advancements, is emphasized, along with the critical need for interdisciplinary collaboration to tackle real-world challenges.

Acknowledgements

ACKNOWLEDGMENTS

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial intelligence

ANN

Artificial Neural Network

RNN

Recurrent Neural Netwokr

LSTM

Long short-term memory

FFNN

Feed-Forward neural network

QML

Quantum machine learning

VQC

Variational quantum circuit

VQA

Variational quantum algorithm

VQR

Variational quantum regressor

PM

Particulate matter

IoT	Internet of Things
ARPA	Agenzia regionale per la protezione ambientale
SGD	Stochastic Gradient Descent
ADAM	Adaptive Moment Estimation
SSR	Sum of Squared of Residulas
OLS	The Ordinary Least Squares
QAOA	Quantum Approximate Optimisation Algorithm
NISQ	Noisy Intermediate-Scale Quantum
QLSTM	Quantum Long short-term memory
ReLU	Rectified Linear Unit

Chapter 1

Introduction

This first chapter introduces the topic behind this thesis, namely an explanation of what this project is about in detail, how air quality monitoring works, and what techniques are currently used to carry out this activity.

1.1 Overview of the project

This master's thesis delves into the emerging field of sensor calibration for fine dust detection in the air, a critical area of study for public health and environmental sustainability. In an era increasingly focused on smart, connected cities, this work fits into an ambitious project led by the Politecnico di Torino. The goal is to develop a distributed network of particulate matter sensors that are reliable, connected, and economically viable.

In the urban context of Turin, the management and monitoring of air quality have traditionally been entrusted to ARPA, the regional environmental protection agency. However, the main challenge lies in the high cost of the currently used monitoring stations, which reach thousands of euros. This thesis aims to address this issue by exploring the feasibility of low-cost monitoring stations that maintain high standards of precision and reliability.

The focus of this study is the use of advanced artificial intelligence techniques for the calibration of more economically accessible sensors. Particular attention is given to the use of neural networks and their potential in combination with quantum circuits, especially "Variational Quantum Circuits" (VQC). These represent an innovative frontier in the simulation of artificial neural networks, offering promising prospects in improving the accuracy of low-cost sensors.

Through this approach, the thesis aims not only to contribute to the field of environmental engineering and quantum computing but also to provide practical and sustainable solutions for the cities of the future, starting with the urban context of Turin. Ultimately, the work presented here aims to demonstrate how technological innovation and advanced research can converge to address environmental challenges of global significance.

The following work was conducted within the laboratory of DAUIN ("Department of Automation and Computer Science") of the Polytechnic University of Turin. Previous studies carried out by the same laboratory can be found at the following paper: [1].

1.2 Air pollution monitoring

The use of sensors for fine-dust detection is a quickly developing topic that touches on public health, technology, and environmental research. Because of the growing worries about air pollution and its effects on the environment and human health, this field has become extremely important.

Particulate matter (PM), also known as fine dust, is made up of microscopic particles that are suspended in the atmosphere. PM_{10} and $PM_{2.5}$, which stand for particles with dimensions smaller than 10 and 2.5 micrometres, respectively, are the sizes that are most often examined. These particles are particularly dangerous since they have the ability to infiltrate the circulation and delve far into the lungs, leading to a host of health issues.

Sensor technology is the foundation of fine-dust detection. These sensors are intended to identify and gauge the amount of airborne particulate matter present. A variety of sensors are available, such as optical sensors that measure the size and concentration of dust particles using light scattering techniques and electrochemical sensors that determine the chemical makeup of the particles.

Because optical sensors are effective at detecting a broad range of particle sizes, they are utilised extensively. Usually a laser or LED, they operate by generating a light source and then detecting the light that is dispersed by dust particles. The concentration and size of the particles in the air are ascertained by analysing this scattering.

These days, fine-dust sensors are frequently linked with other technological

Figure 1.1: GP2Y1010AU0F Optical Fine-Dust Sensor

systems. For example, a large number of sensors are currently connected to Internet of Things (IoT) networks, enabling data gathering and real-time monitoring. Constant monitoring is made possible by this integration, which also makes it easier to gather huge datasets for analysis and forecasting.

The applications for fine-dust sensors are numerous. They work in environmental monitoring stations to provide the government and the public with up-to-date information on air quality. The issue of health advisories and the formulation of policies pertaining to air pollution depend on this information.
To maintain a safe working environment and to adhere to environmental requirements, fine-dust sensors are employed in industrial settings to monitor and manage the air quality. These sensors may also be used in smart home systems, where they can monitor and enhance indoor air quality.

Fine-dust sensors are important, but they have problems, particularly with precision and dependability. Sensor readings can be impacted by variables such as temperature, humidity, and the presence of other substances in the air. Additionally, efforts are being made to lower the cost of these sensors and increase their accessibility for the general population.

The development of increasingly complex and precise sensors as well as the use of artificial intelligence to more accurately assess and forecast trends in air quality are potential future developments in this subject. The development of portable and personal monitoring tools that people may use to evaluate the air quality in their immediate surroundings is also gaining popularity.
In summary, the field of sensor-based fine-dust detection is essential to the effort to reduce air pollution. Sensor technology has the potential to significantly improve our knowledge of and ability to regulate air quality, which will eventually lead to improved health and a cleaner environment as it develops.

1.3 State of the art in air pollution monitoring methods

In order to produce more precise, timely, and thorough data, the state of the art in air pollution monitoring has greatly evolved in recent years by utilising new technology and approaches. Here is a summary of the approaches used today to monitor air pollution, along with a discussion of their benefits and drawbacks:

Satellite Monitoring

- **Pros**
  - Satellites can monitor large areas, including remote and inaccessible regions
  - They provide consistent data over time, allowing for long-term trend analysis
  - Advanced satellites offer near real-time data on air quality
- **Cons**
  - Satellites might not detect small-scale pollution sources or variations
  - Cloud cover and weather conditions can interfere with data collection
  - Satellite technology is expensive to develop, launch, and maintain

Ground-Based Monitoring Stations

- **Pros**
  - These stations offer precise measurements of various pollutants
  - They provide specific data for targeted locations, useful for local policy-making
  - Capable of measuring a wide range of pollutants
- **Cons**
  - Each station only covers a small area, requiring many stations for comprehensive coverage
  - Regular maintenance and calibration are needed, which can be costly
  - Stations can be subject to vandalism or theft, especially in remote areas

Mobile Monitoring (Vehicles and Drones)

- **Pros**
  - Can be moved around to gather data from different locations
  - Useful for studying specific sources of pollution or hotspots
  - Fill gaps in data where fixed stations are not present
- **Cons**
  - Mobile units might carry fewer sensors due to size and weight constraints.
  - Especially for drones, limited battery life can restrict monitoring time
  - Use of drones can be limited by local regulations

Personal Monitoring Devices

- **Pros**
  - Provide data on personal exposure levels, useful for health studies
  - Increase public awareness and engagement in air quality issues
  - Small and easy to carry around
- **Cons**
  - These devices may be less accurate than professional equipment
  - Often measure only a few common pollutants
  - Data from various devices might not be directly comparable

IoT and Sensor Networks

- **Pros**
  - Continuous monitoring provides real-time data
  - Networks can be easily expanded by adding more sensors
  - Can be integrated into smart city infrastructures for holistic monitoring
- **Cons**
  - Sensors require regular calibration and maintenance
  - Managing and analyzing the large volumes of data can be challenging
  - Sensors can be sensitive to environmental factors, affecting accuracy

Artificial Intelligence and Machine Learning

- **Pros**
  - Can predict pollution trends and patterns
  - AI can efficiently process large datasets from various sources
  - Useful in understanding complex environmental interactions
- **Cons**
  - The accuracy of AI predictions is dependent on the quality of input data
  - Developing and implementing AI systems can be complex and costly
  - AI decision-making processes can be opaque, making it difficult to understand how conclusions are reached

Every approach has its own advantages and disadvantages. The best strategy frequently combines various techniques, capitalising on their advantages to produce a thorough and precise picture of air pollution. We may anticipate more advancements in coverage, precision, and the capacity to assess and forecast changes in air quality as technology develops.

Using ARPA "Ground-Based monitoring stations" as a reference, the study of "AI and Machine Learning" techniques is the main goal of this thesis in order to enhance sensor calibration. Artificial intelligence techniques will be compared with their quantum counterparts.

Chapter 2

Technologies

For each of these technologies, theory and examples will be explained.In this second chapter we explore the technological underpinnings of our experimental work, providing an in-depth analysis of the tools, frameworks, and approaches that are essential to our work. This analysis aims to offer a thorough grasp of the theoretical and mathematical foundations guiding these technologies rather than just a descriptive one. We start by outlining the historical evolution and contemporary status of each technology, emphasising its applicability and room for innovation in the computer science engineering sector.

2.1 Machine Learning

Within artificial intelligence, machine learning is a sophisticated discipline that focuses on creating algorithms that can learn from data and make judgements. This discipline has significantly changed how problem-solving is approached across a range of fields, shifting the focus from explicit programming to data-driven approaches.

The idea of learning from experience is at the heart of machine learning. When algorithms are exposed to big data sets, they can find patterns and connections. Following then, choices or predictions regarding fresh, unobserved data are based on these discoveries. This process is comparable to human learning, in which information is acquired via experience and used to decision-making.
While there are many different approaches to machine learning, they all entail utilising a dataset to train a model. The model begins as a simple hypothesis and develops over time as it processes data, increasing its accuracy and dependability. The quality and quantity of the data supplied are critical to the success of machine learning; more complete datasets typically provide more accurate models.

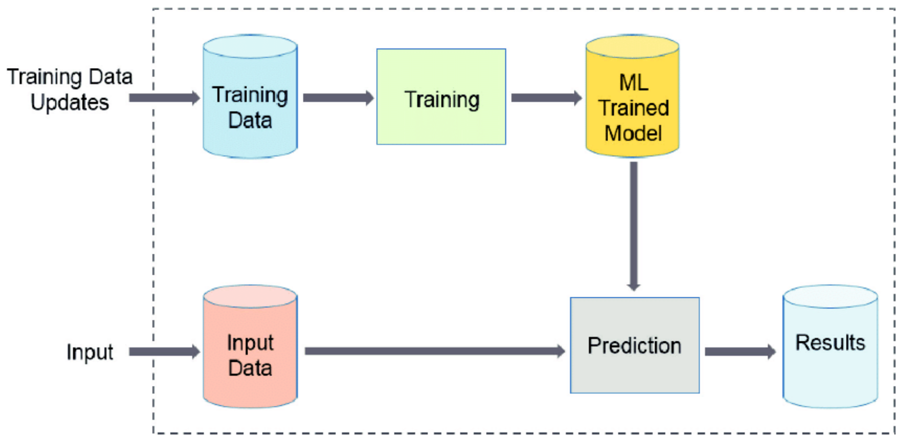


Figure 2.1: A generic Machine Learning workflow

In Fig. 2.1 a generic workflow used to build a Machine Learning model (without validation) can be observed: a portion of the dataset, called the "traning-set," is used to train the model that will perform the predictions on another portion of the dataset that has never been seen before.
Ensuring that models generalise adequately to new data—a notion known as generalization—is one of the major issues in machine learning. This implies that the model should function well on both fresh, untrained data and the data it was trained on. Overfitting is a typical mistake when a model performs poorly on fresh data because it is overly adapted to the training set.
Applications for machine learning are numerous and range from simple ones like email filtering to sophisticated ones like autonomous driving or medical diagnostics. It is a vital tool in today's technology environment because of its versatility and effectiveness in managing large, complicated datasets.

All things considered, machine learning signifies a fundamental change in computational methodologies, placing an emphasis on data-driven, adaptive, predictive models as opposed to the static, rule-based systems of classical computing. This change has created new opportunities for technology, enabling robots to become more than just tools for carrying out instructions—rather, they are now sentient beings with the ability to learn, adapt, and make judgements.

2.1.1 Supervised learning

As learned from the literature [2], we can distinguish between three different main types of machine learning problems: Supervised learning, Unsupervised learning and Reinforcement learning.

A key component of machine learning is supervised learning, which is the process of training algorithms on a labelled dataset in which each data point is associated with an accurate response or result. The'supervised' part describes the direction these labels provide to the algorithm during its training phase, which is similar to a teacher monitoring a student's progress in class.
In supervised learning, the dataset serves as an instructor by giving the algorithm instances to work with. Input data (pictures, text, or numerical characteristics) and a matching label or target (such as a category for classification tasks or a continuous value for regression tasks) make up each sample in the dataset. The algorithm's goal is to identify correlations or patterns between the inputs and the labels that correspond to them.
The algorithm is trained iteratively by first generating predictions from the input data and then modifying its internal parameters in response to how accurate these predictions turn out to be. The programme measures the discrepancy between the actual labels and its predictions using a mathematical calculation called the loss function. As training goes on, the algorithm's predictions get more accurate because the goal is to minimise this loss.
Unseen data is used to assess the algorithm's performance following training. This assessment is essential because it determines how well the algorithm can apply the patterns it has learnt to fresh, untested samples. A good supervised learning model is one that keeps a high degree of accuracy when faced with fresh data, in addition to doing well on its training set.

Here are some examples of famous machine learning algorithms of the "supervised learning" type:

- **Linear Regression:** Used for predicting a dependent variable value (y) based on a given independent variable (x). It is best suited for regression problems.

- **Logistic Regression:** Despite its name, logistic regression is used for classification problems, not regression. It predicts the probability of occurrence of an event by fitting data to a logistic curve.

- **Support Vector Machines (SVMs):** These are powerful algorithms used for both classification and regression tasks. They work by finding the best hyperplane that separates data points of different classes.

- **Decision Trees:** This algorithm models decisions and their possible consequences as a tree, which is particularly useful for classification and regression.

- **Random Forests:** An ensemble method that operates by constructing multiple decision trees during training and outputting the class that is the mode of the classes (classification) or mean prediction (regression) of the individual trees.

- **K-Nearest Neighbors (K-NN):** It's a simple algorithm that stores all available cases and classifies new cases based on a similarity measure (e.g., distance functions).
- **Neural Networks:** Although they are often associated with deep learning, simple neural networks can also be used for a variety of supervised learning tasks.

2.1.2 Unsupervised learning

The process of training an algorithm on data without predetermined labels or responses is referred to as "unsupervised learning," as the publication [3] demonstrates. Unsupervised learning algorithms have to make sense of the data without explicit instructions on what to search for, in contrast to supervised learning, where the training data contains both input and output labels. This type of learning is similar to giving raw data to a machine and expecting it to find correlations, patterns, or structures on its own.

The algorithm searches the data in unsupervised learning to look for innate structures. The two basic methods employed in this field for this kind of investigation are clustering and association. A collection of things is clustered when they are arranged so that the objects in the same group, or cluster, resemble one another more than the ones in other groups. Conversely, association involves identifying patterns that characterise significant parts of the data, including recurring events or correlations between various variables.

Unsupervised learning is powerful because it can reveal hidden patterns in data that are not immediately obvious. This capacity is especially helpful in exploratory data analysis, where the objective is to comprehend and identify the data's underlying structure. It's also very effective in more sophisticated tasks like dimensionality reduction, where the algorithm can identify the most relevant characteristics in a dataset, or in anomaly detection, where the aim is to uncover data points that differ considerably from the rest of the dataset.

Some of the most renowned unsupervised learning algorithms include:

- **K-Means Clustering:** This is a popular clustering algorithm that partitions data into K distinct, non-overlapping subsets or clusters. It assigns each data point to the nearest cluster center and iteratively refines the positions of these centers.

- **Hierarchical Clustering:** Unlike K-means, hierarchical clustering does not require the number of clusters to be specified beforehand. It builds a hierarchy of clusters either by merging smaller clusters into larger ones (agglomerative) or by breaking down a large cluster into smaller ones (divisive).
- **DBSCAN (Density-Based Spatial Clustering of Applications with Noise):** DBSCAN is a clustering algorithm that groups together points that are closely packed together while marking points that lie alone in low-density regions as outliers.
- **Principal Component Analysis (PCA):** PCA is a technique used for dimensionality reduction. It transforms the data into fewer dimensions while retaining most of the variability in the data, making it easier to analyze and visualize.

2.1.3 Reinforcement learning

Compared to supervised and unsupervised learning, reinforcement learning (RL) is a unique and dynamic subfield in machine learning. Through behaviour in an environment to accomplish a goal, an agent learns to make decisions through reinforcement learning. Rather than giving clear instructions on what to do, reinforcement learning (RL) focuses on learning how to behave depending on input from the environment, such as rewards or punishments.
The idea of the agent-environment interaction is at the core of reinforcement learning. Everything an agent interacts with is considered part of the environment, and the agent itself is the learner or decision-maker. In the environment, the agent makes decisions (actions), and the environment reacts to these decisions by presenting the agent with new scenarios. The environment also offers incentives, which are numerical values that show how well the agent is doing in relation to its objective. The agent's long-term goal is to maximise the cumulative payoff.

There are many different fields in which reinforcement learning is applied, such as robotics, gaming, autonomous cars, and personalised recommendation systems. Nevertheless, RL has special difficulties, such the necessity for a great deal of trial and error to discover efficient policies and the difficulty of deciphering environmental conditions and incentives. In general, reinforcement learning emphasises experience-based learning and the modification of tactics to accomplish certain objectives, providing a dynamic and participatory approach to machine learning.

Among the most well-known algorithms for reinforcement learning ([4]) are:

- **Q-Learning:** This is a value-based algorithm that learns the value of an action in a particular state. It uses a Q-table to store Q-values, which are estimates of the total reward an agent can expect to receive after taking a given action in a given state.

- **Deep Q Networks (DQN):** Building upon Q-learning, DQN integrates deep learning by using neural networks to approximate Q-values. It's famous for its success in playing Atari games at a level comparable to human players.

- **Policy Gradients:** This approach learns a policy function directly without having to maintain a value table or model. The algorithm adjusts the parameters of the policy in a direction that increases expected rewards.

2.2 Artificial neural network

A subfield of computer science called artificial intelligence (AI) is concerned with building machines that are able to carry out activities that would typically need human intellect. Learning, thinking, problem-solving, perception, language comprehension, and creativity are all included in these activities. Artificial Intelligence (AI) integrates concepts from several fields, such as computer science, mathematics, psychology, linguistics, and neuroscience.

AI has a rich and complicated history. Although the idea of manufactured creatures has long been present in mythology and storytelling, the foundation for this idea was set by the formal laws of logic developed by philosophers such as Aristotle. Calculators and mechanical reasoning were first conceptualised in the 17th century by philosophers such as Leibniz and Descartes.
The development of electronic computers in the 1940s marked a pivotal point, leading to the modern era of AI. A landmark moment was Alan Turing's 1950 paper proposing the Turing Test as a measure of machine intelligence.

John McCarthy first used the phrase "Artificial Intelligence" in 1956 at the Dartmouth Conference, which officially launched the subject. Expert systems and symbolic issues were the main subjects of early study.
But there have been difficulties with AI as well. Due to unfulfilled expectations, the discipline has gone through many "AI winters," or times when funding and enthusiasm have decreased. These caused changes in emphasis, particularly in the areas of machine learning and neural networks.
There have been major improvements in AI thanks to its comeback, particularly with deep learning and neural networks. With notable advancements in natural language processing, picture recognition, and autonomous systems, artificial intelligence (AI) is already influencing a wide range of industries, including healthcare, banking, transportation, and entertainment. AI applications have grown even more

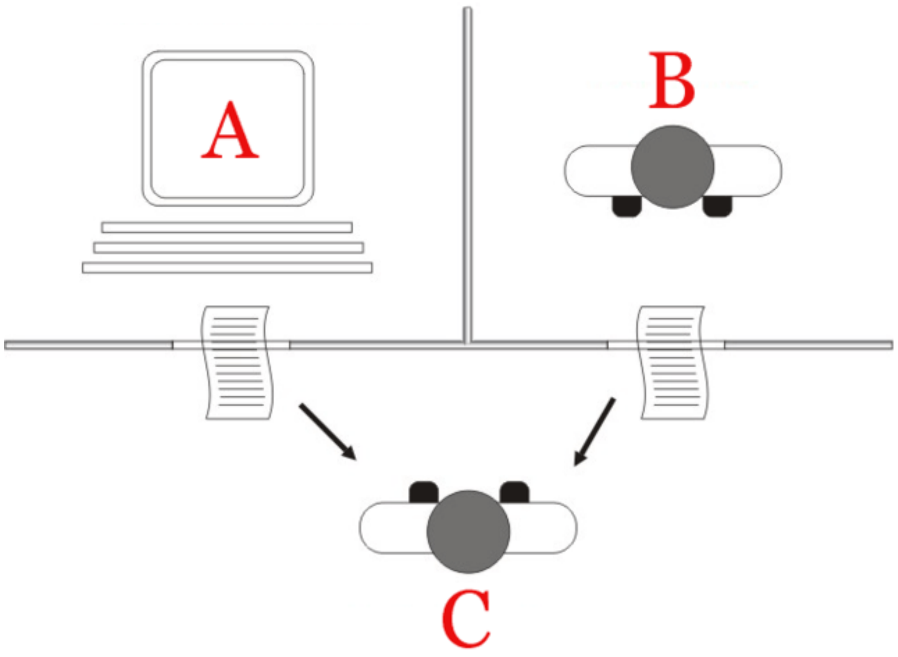


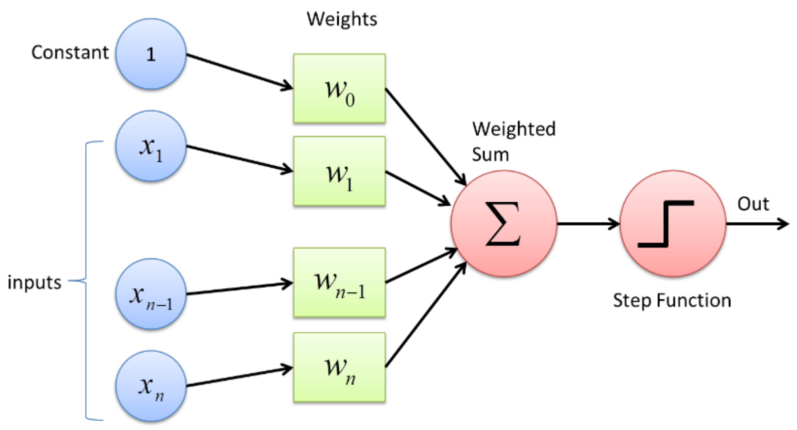
Figure 2.2: The Alan Turing's test used as a measure of machine intelligence

through interaction with other technologies like big data and the Internet of Things.

Artificial Intelligence consists of several essential components, such as machine learning, which is the foundation of AI and focuses on algorithms for prediction and decision-making; neural networks and deep learning, which are effective at recognising patterns and inspired by the human brain; natural language processing, which allows computers to comprehend and produce human language; robotics, which combines AI with mechanical engineering for autonomous tasks; and expert systems, which are made for solving complex problems using rule-based systems. Lastly, the area of AI is becoming more and more concerned with safety and ethical problems related to AI, addressing things like algorithmic bias, the effect on the workforce, and making sure AI development is both safe and helpful. The goals of AI in the future are to increase general intelligence, improve human-AI cooperation, and address ethical issues.

Modern artificial intelligence is based on Artificial Neural Networks (ANNs), which are modelled after the biological neural networks seen in animal brains. A neural network (ANN) is a computing system composed of several basic, highly linked processing components that process data based on how each element changes its dynamic state in response to outside inputs.

The McCulloch-Pitts neuron model, developed in 1943 by logician Walter Pitts and neurologist Warren McCulloch, lay the groundwork for neural networks. This model represented biological neurons in a mathematically simplified manner. In 1958, Frank Rosenblatt developed the Perceptron, a primitive neural network with basic pattern recognition capabilities. This was a big advance since it brought in a network that could learn from information.



Source

Figure 2.3: The Perceptron by Frank Rosenblatt (1958)

Here's a brief explanation of how the Perceptron works:

1. **Input Values:** The perceptron receives multiple input values. Each of these inputs corresponds to a feature in the dataset. For instance, in a dataset for image recognition, each input might be the intensity of a pixel.

2. **Weights and Bias:** Each input value is assigned a weight, which the perceptron will learn during training. There's also a bias, which can be thought of as a weight for an input that's always 1. The weights and bias are initially set to random values and are adjusted during the learning process.

3. **Summation and Activation:** The perceptron multiplies each input by its weight, sums all these products, and then adds the bias. This sum is then passed through an activation function. A common activation function used in perceptrons is the step function, which outputs either 0 or 1. This step function thresholds the sum: if the sum is above a certain threshold, the perceptron outputs 1; otherwise, it outputs 0.

4. **Output:** The output is a simple binary decision (e.g., yes/no, 1/0) that categorizes the input data into one of two classes.

5. **Learning Process:** During training, the perceptron adjusts its weights and bias based on the errors in its predictions. If the perceptron's prediction is wrong, the weights are adjusted in the direction that reduces the error. This process is repeated many times over the dataset, allowing the perceptron to learn from its mistakes.

2.2.1 Feed-Forward neural network (FFNN)

An artificial neural network in which there is no cycle formed by the connections between the units is called a feed-forward neural network.
This type of architecture serves as the basis for several deep learning models and is employed in a wide range of machine learning applications. An input layer, one or more hidden layers, and an output layer make up the layers that make up the network's structure. Neurons, or nodes, comprise each layer, and all of the neurons in one layer are connected to all of the neurons in the layer above it.

Here's a closer look at its workings, along with the mathematical foundation. The input layer receives the input features of the data: each neuron in this layer represents a feature in your dataset.
Then, the hidden layers perform computations and transformations on the inputs. The number of hidden layers and the number of neurons in each hidden layer are design choices that depend on the complexity of the task.
At the end, the output layer produces the final output of the network, which could be a class label in classification problems or a continuous value in regression.
The operation of this type of neural network is divided into two steps: forward propagation and backpropagation.

Forward propagation. In each neuron, the first step is a linear transformation of the inputs.
Given an input vector $X = [x_1, x_2, ..., x_n]$, the linear transformation in a neuron is calculated as $Z = XW + b$ where W is the weight matrix, b is the bias vector, and Z is the output vector of the linear transformation.
The linear transformation's result Z is then passed through an activation function σ, producing the output $A = \sigma(Z)$.
The activation function introduces non-linearity, allowing the network to learn complex patterns. Common activation functions include Sigmoid, Tanh, ReLU (Rectified Linear Unit), etc.
This process (linear transformation followed by activation) is repeated for each

layer. The output of one layer A becomes the input to the next layer.

Mathematical Example for a Single Layer.
Consider a layer with two inputs x_1, x_2 connected to two neurons.
The equations for the neurons' outputs could be represented as follows:

- For neuron 1: $Z_1 = w_{11}x_1 + w_{12} + x_2 + b_1$, then $A_1 = \sigma(Z_1)$
- For neuron 2: $Z_2 = w_{21}x_1 + w_{22} + x_2 + b_2$, then $A_2 = \sigma(Z_2)$

Where:

- w_{ij} represents the weight from input j to neuron i
- b_i is the bias for neuron i
- Z_i is the linear transformation's output for neuron i
- σ is the activation function
- A_i is the activation function's output for neuron i, which will be used as input for the next layer.

Backpropagation. Following forward propagation, a loss function (such as mean squared error for regression or cross-entropy for classification) is used to quantify the difference between the network's predicted and actual output.
Reducing this loss is the aim of training. Backpropagation is used to update the weights and biases in the network. It involves:

1. Computing the Gradient: The loss function's partial derivatives with respect to each network weight and bias, showing how adjustments to these variables may impact the loss.
2. Updating Parameters: The weights and biases are then updated in the direction that minimally reduces the loss, typically using gradient descent or variations thereof. The amount of adjustment is controlled by the learning rate.

To conclude, in order to generate predictions, a feed-forward neural network goes through several stages of processing. It learns by employing a technique known as backpropagation to modify its weights and biases in response to the prediction error.
Activation functions and the depth of the network introduce non-linearity, which allows the architecture to learn intricate patterns.

2.2.2 Gradient Descent

Gradient Descent is a fundamental optimisation method that is extensively used in machine learning and other fields where function reduction or maximisation is required [5]. This method is widely used in artificial neural networks due to its efficiency in optimizing complex models with vast numbers of parameters.
Fundamentally, the goal of gradient descent is to minimise the loss function, which is a mathematical depiction of the discrepancy between the model's predictions and the actual data.
The fundamental idea behind this approach is to utilise the gradient, or slope, of the loss function in relation to the model parameters. By using this information, the parameters are iteratively adjusted in order to minimise the loss.
This optimisation technique makes use of a simple yet effective idea: one may travel in the other direction to decrease the loss by knowing the direction in which the loss function grows. The gradient indicates the direction of the sharpest climb. It is a vector made up of the partial derivatives of the loss function with respect to each parameter. The local minimum of the function, or the set of parameters that produce the least error, may be found by travelling in the other direction.
Mathematically, Gradient Descent is formulated as follows: given a loss function $J(\theta)$, where θ represents the parameters of the model, the goal is to find the set of parameters θ^* that minimizes $J(\theta)$. The gradient of J at θ, denoted as $\nabla J(\theta)$, gives the direction of the steepest ascent. To minimize J, one updates θ by moving a small step in the opposite direction of $\nabla J(\theta)$:

$$\theta_{new} = \theta_{old} - \alpha \nabla J(\theta_{old}) \tag{2.1}$$

In this case, the scalar α stands for learning rate, which establishes the magnitude of the step made in the opposite direction of the gradient. Selecting the right number for α is essential, as a too small value will result in extremely sluggish convergence, while a very large value could cause the algorithm to overshoot the minimum and perhaps diverge.
Gradient Descent is a procedure that gradually approaches the minimum of J by iteratively adjusting the parameters θ through the use of this update algorithm. This iterative process goes on until convergence, which is usually understood to be the completion of a predefined number of iterations or the moment at which the change in the loss function between iterations is less than a predefined threshold.
In order to quantitatively illustrate this, let's have a look at a basic linear regression model, in which the objective is to fit a line to a set of points in such a way that the total of the squared differences between the actual and predicted values is as little as possible. In this instance, the loss function may be shown as follows:

$$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \tag{2.2}$$

where m is the number of training examples, $h_\theta(x^{(i)}) = \theta_0 + \theta_1 x^{(i)}$ is the hypothesis function predicting the output given input $x^{(i)}$, and $y^{(i)}$ is the actual output.
The partial derivatives of $J(\theta_0, \theta_1)$ with respect to θ_0 and θ_1 are:

$$\frac{\partial J}{\partial \theta_0} = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) \tag{2.3}$$

$$\frac{\partial J}{\partial \theta_1} = \frac{1}{m} \sum_{i=1}^{m} \left((h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}\right) \tag{2.4}$$

These gradients are then used to update θ_0 and θ_1 respectively, as per the Gradient Descent update rule.

Gradient Descent minimises the loss function by performing these updates again and again until the linear regression model's optimum parameters are reached. This is a fundamental approach in statistical and machine learning optimisation, as demonstrated by this procedure, which also shows how effective it is at optimising model parameters.

2.2.3 Gradient Descent Algorithm

We show the operation of the Gradient Descent method from the algorithmic point of view use the Python programming language. To better understand how it works, we will simplify the algorithm use a quadratic function as the function to be minimized.

The only library needed for the purpose of implementation is numpy: a library that allows us to manipulate complex data structures for mathematical purposes.

```python
import numpy as np
```

First, we define our objective function to be minimized.

```python
# Define the function to be minimized
def f(x, y):
    return x**2 + y**2
```

Given the simplicity of the chosen objective function, we can define the derivative of that function with respect to its parameters (x and y).

```
# Define the partial derivatives of the function with respect to x and y
def df_dx(x, y):
    return 2 * x
def df_dy(x, y):
    return 2 * y
```

Below is the algorithm describing the operation of Gradient Descent.

```
# Define the gradient descent algorithm
def gradient_descent(start_x, start_y, learning_rate, num_iterations):
    # Initialize the parameters
    x = start_x
    y = start_y

    # Perform the gradient descent iterations
    for i in range(num_iterations):
        # Calculate the gradients
        grad_x = df_dx(x, y)
        grad_y = df_dy(x, y)

        # Update the parameters
        x = x - learning_rate * grad_x
        y = y - learning_rate * grad_y

    return x, y, f(x, y)
```

Finally, you can run the algorithm by calling the function above. The constants in the problem were chosen for demonstration purposes of this execution.

```
start_x, start_y = 8, 8
learning_rate = 0.1
num_iterations = 20
x_opt, y_opt, f_opt = gradient_descent(start_x, start_y, learning_rate, num_iterations)
```

A 3-d graph of the execution of the algorithm on the same data is shown.

2.2.4 Recurrent neural network (RNN)

A kind of artificial neural networks known as recurrent neural networks (RNNs) have nodes connected in a directed graph along a temporal sequence. It may now display temporal dynamic behaviour for a time series as a result. RNNs have the

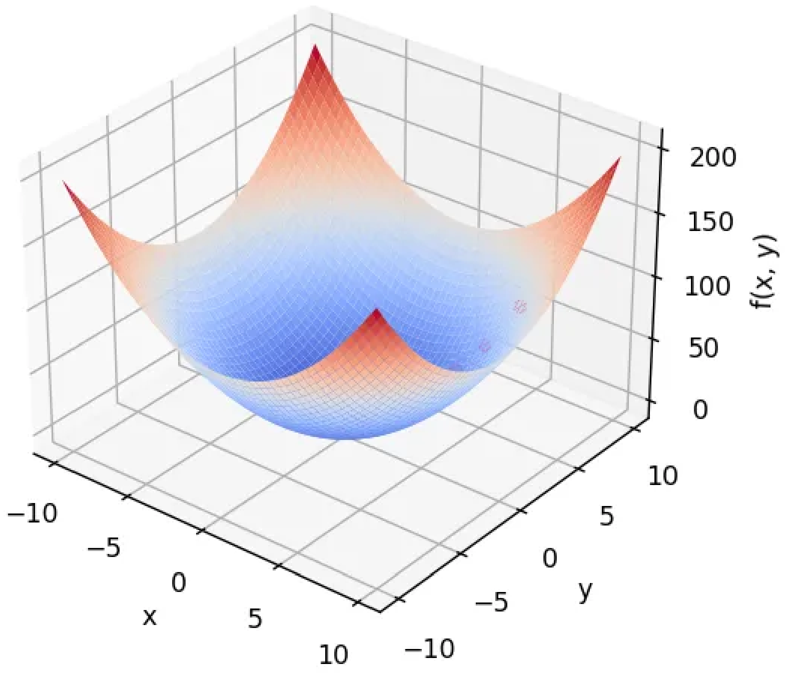


Figure 2.4: Gradient descent algorithm results

ability to process input sequences using their internal state, or memory, in contrast to feed-forward neural networks (FFNNs). They can therefore be used for tasks like speech recognition or unsegmented, linked handwriting recognition.

An in-depth look at this type of technology is desired because some of the laboratory work in this thesis was carried out using this type of architecture in order to create a model capable of learning how to calibrate fine dust sensors based on their performance over time.
In more detail, a specific recurrent neural network model called "Long short-term memory" (LSTM) was used.
The origins of RNNs may be discovered in the 1980s, when David Rumelhart and John Hopfield conducted seminal research [6]. A method to comprehend how networks may function as associative memory was provided by John Hopfield's 1982 introduction of Hopfield Networks, which can be seen as a forerunner to RNNs. Backpropagation over time is a technique for training RNNs that was presented by David Rumelhart, Geoffrey Hinton, and Ronald Williams in 1986. During this time, it became clearer how neural networks might represent temporal dynamics, a property that is essential for processing data sequences like time series, text, or voice.

The main factor that makes RNNs so successful with sequential data is their looping nature, which makes information persistent. Every node in a layer of an RNN carries out the same function, but with various inputs—current input as well as information that has already been received. In essence, it possesses a "memory" that records data on computations made thus far. RNNs have the capacity to store information in their internal state for extended periods of time. The vanishing gradient problem, which causes gradients to trend to zero or explode throughout the backpropagation process, makes learning long-term relationships difficult to capture in reality.

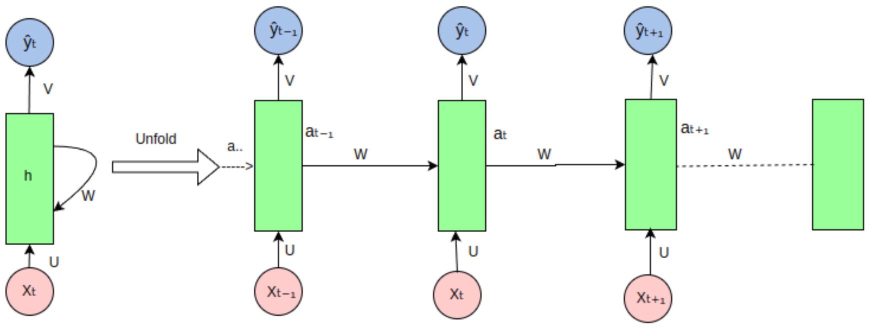


Figure 2.5: Recurrent Neural Network architecture explained [7].

RNNs offer a number of benefits. They are perfect for applications in language modelling, machine translation, speech recognition, and time-series prediction because they are naturally adapted to processing data sequences. They are able to comprehend context and provide predictions depending on the order of data inputs because of their capacity to model temporal dynamics and recall prior inputs.
RNNs are not without limitations, though. Because of the disappearing and ballooning gradient issues, they are renowned for being challenging to train. Furthermore, compared to models like Convolutional Neural Networks (CNNs) that can process inputs in parallel, their sequential structure makes them less effective for parallel processing. When working with lengthy sequences, this constraint becomes substantial and results in lengthier training durations.

2.2.5 Long short-term memory (LSTM)

Recurrent neural networks (RNNs) have limits. Long Short-Term Memory (LSTM) networks are a kind of RNN that was created to overcome these constraints, especially with regard to learning long-term dependencies. LSTMs were first presented by Hochreiter and Schmidhuber in 1997, and they have since grown to be a mainstay in the deep learning space for applications that call for modelling sequential data over extended periods of time [6].
The intricate cell structure of LSTM networks, which consists of many gates—the

input, forget, and output gates—is its primary innovation. By deciding whether information should be retained or lost as data progresses through the sequence, these gates enable LSTMs to keep a more constant gradient throughout training. Due to the vanishing gradient problem, standard RNNs frequently struggle to capture long-term dependencies in sequential data; this design helps LSTMs overcome this obstacle.

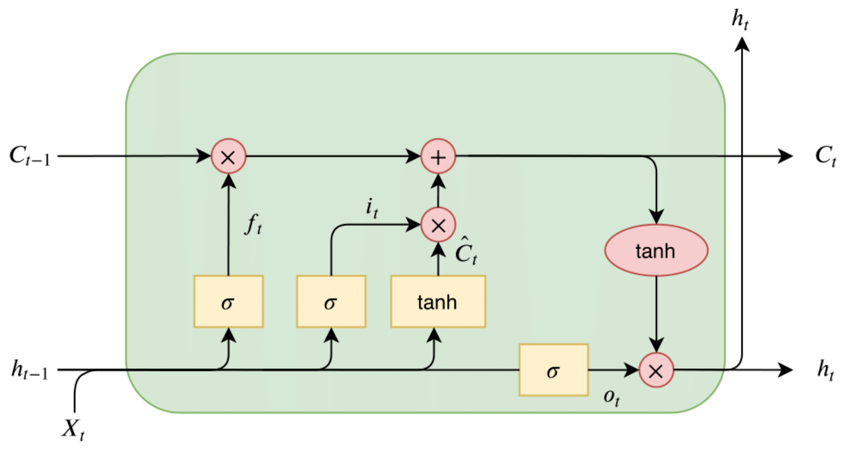


Figure 2.6: Long short-term memory architecture explained.

Based on Fig.2.6, an LSTM cell consists of the following components:

- **Cell state** (C_t): This acts as the "memory" of the LSTM, carrying relevant information throughout the processing of the sequence.
- **Hidden state** (h_t): This is the output of the LSTM cell at step t, which can be used for predictions or passed to other layers in the network.
- **Input gate** (i_t): Determines how much of the new information should be added to the cell state.
- **Forget gate** (f_t): Decides how much of the existing information in the cell state should be retained or discarded.
- **Output gate** (o_t): Controls how much of the cell state is passed to the hidden state.

From [6] it is possibile to understan the operation of an LSTM cell, in fact: given an input sequence $x_1, x_2, ..., x_t$, the LSTM updates its cell state C_t and hidden state h_t at each time step t using the following equations.

1. **Forget Gate:**

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.5}$$

Here σ is the sigmoid function, W_f is the weight matrix for the forget gate, b_f is the bias term, and $[h_{t-1}, x_t]$ represents the concatenation of the previous hidden state and the current input.

2. **Input Gate:**
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.6}$$
Where W_i is the weight matrix for the input gate, and b_i is the bias term.

3. **Candidate Cell State:**
$$\hat{C}_t = tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2.7}$$
It is a candidate value for updating the cell state, where W_C is the weight matrix, and b_C is the bias term.

4. **Cell State Update:**
$$C_t = f_t \cdot C_{t-1} + i_t \cdot \hat{C}_t \tag{2.8}$$
The old cell state C_{t-1} is updated to the new cell state C_t by forgetting certain information (as determined by f_t) and adding new candidate information (as determined by i_t and $\hat{C}_t$).

5. **Output Gate:**
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.9}$$
As always, W_o is the weight matrix for the output gate, and b_o is the bias term.

6. **Hidden State Update:**
$$h_t = o_t \cdot tanh(C_t) \tag{2.10}$$
The hidden state h_t is updated based on the output gate and the current cell state, allowing the LSTM to output information based on the cell's memory.

Through its gating mechanisms, LSTM is able to maintain a balance between remembering important information and discarding irrelevant material, which is fundamental to its success. Processing and predicting sequences with long-term relationships require this equilibrium.
An important development in neural networks' processing of sequential input is represented by LSTMs. LSTMs have expanded the usefulness and efficacy of neural networks in a range of difficult tasks using sequential data by overcoming the shortcomings of classic RNNs, notably in learning long-term dependencies.
They can selectively recall and forget information thanks to their special design, which is made up of several gating mechanisms. This makes them an effective tool for modelling temporal dynamics.

2.3 The linear regression problem

As mentioned in the first chapters, the objective of this thesis is to study the calibration of optical sensors for fine dust in order to increase their accuracy by bringing them to simulate the same behavior as those installed by ARPA.
To do this, the linear regression problem must be addressed because the value of particulate matter in the air (PM2.5) is a unit in $\mu g/m^3$ that varies over time.

Modelling the connection between a dependent variable and one or more independent variables is done using the core statistical and machine learning technique of linear regression. Finding the linear equation that most accurately predicts the dependent variable from the independent variables is the aim. One of the most basic and traditional types of regression analysis, it dates back to the 19th century and was made famous by Francis Galton's work on regression towards mediocrity in hereditary height [8].
In the late 18th and early 19th centuries, astronomy and geodesy provided the framework for the development of the notion of regression and the least squares approach, which is the foundation of linear regression. The least squares approach was separately invented in the early 19th century by Carl Friedrich Gauss and Adrien-Marie Legendre. The ideas of correlation and regression were first established by Francis Galton's work in the late 19th century on the link between parents' and children's heights. Galton also developed the term "regression" and showed a regression towards the mean.

From a mathematical perspective, by fitting a linear equation to observed data, linear regression describes the connection between the dependent variable Y and one or more independent variables X_i. Simple linear regression is the most basic type of linear regression. It involves two variables and is represented by the following equation:

$$Y = \beta_0 + \beta_1 X + \epsilon \tag{2.11}$$

where Y is the dependent variable, X is the independent variable, β_0 is the y-intercept of the regression line, β_1 is the slope of the regression line and ϵ represents the error term, accounting for the difference between observed and predicted values.
In multiple linear regression, where there are two or more independent variables:

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_n + \epsilon \tag{2.12}$$

Solving this type of problem requires working on the coefficients (β) in the equation. The goal is to identify the line—or hyperplane, in multiple regression terms—that minimises the total squared differences between the values that the linear model predicts and the values that are observed. This is called the criteria

of least squares.
Mathematically, this involves minimizing the sum of squares of residuals (SSR):

$$SSR = \sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{2.13}$$

where Y_i is the observed value (coming from your dataset) and $\hat{Y}_i$ is the predicted value based on the regression model.
The Ordinary Least Squares (OLS) estimations are obtained by minimising the SSR with respect to the coefficients. The following formula may be used to get the OLS estimates of β_0 and β_1 for simple linear regression based on sample data:

$$\beta_1 = \frac{\sum_{i=1}^{n}(X_i - \overline{X})(Y_i - \overline{Y})}{\sum_{i=1}^{n}(X_i - \overline{X})^2} \tag{2.14}$$

$$\beta_0 = \overline{Y} - \beta_1\overline{X} \tag{2.15}$$

where $\overline{X}$ and $\overline{Y}$ are the sample means of X and Y, respectively.

Under specific circumstances (known as the Gauss-Markov theorem), the OLS estimator possesses favourable statistical properties: it is unbiased, consistent, and has the lowest variance among linear estimators.
Along with these other presumptions, linear regression additionally requires that the data be linear, independent, homoscedastic (i.e., have constant variance of error terms), and normal for certain statistical tests. If these presumptions are broken, the model may need to be adjusted or alternative estimating methods may need to be used.

2.4 Optimization and accuracy

The use of artificial neural networks as a tool to deal with linear regression to study the behavior of fine dust sensors during a predetermined time frame was mentioned. Despite its considerable power and precision, this type of architecture also turns out to be "black boxed" today, meaning that it is not easy to explain all the process that went into arriving at the result.
For this reason, it is necessary to study and test different types of optimizers and accuracy metrics, evaluating which one best fits the dataset and the type of problem.

Optimizers in deep learning are algorithms or techniques that modify the neural network's parameters, including weights and learning rate, in order to lower losses. Optimizers assist in minimising (or maximising) an objective function with respect to its parameters, known as the loss function.

An example of an optimizer is the Gradient Descent method mentioned earlier. We now show the behavior of other optimizers that were actually used in the laboratory work done in this thesis.

2.4.1 Stochastic Gradient Descent (SGD)

A key optimisation technique for deep learning model training is stochastic gradient descent (SGD). It is a variant on the gradient descent technique (saw before), which is used to determine a function's minimum.
The term "stochastic" in SGD refers to the fact that, unlike in conventional gradient descent, only randomly chosen subsets of the dataset (mini-batches) are used to carry out the optimisation. This method can improve the models' generalisation features while also accelerating computation.

The core idea behind this method is still the same as gradient descent: adjusting the model's parameters (weights) iteratively to minimize a loss function. The loss function measures the difference between the model's predictions and the actual target values for the training data.
The main innovations introduced by this method relate to optimizations in the amount of computation performed as individual steps of the parent algorithm.

Let's say we have 10 characteristics and 10,000 data points.
As many terms as there are data points in the sum of squared residuals—10,000 terms in our example—are included. It will really take $10000 \cdot 10 = 100{,}000$ calculations to compute the derivative of this function for each feature, as we must do for each iteration. It typically takes 1000 iterations to finish the process, which means that $100{,}000 \cdot 1000 = 100000000$ calculations are involved. Gradient descent is sluggish on large datasets since it is essentially an overhead.

SGD randomly select a small subset of the data (a mini-batch) instead of the entire dataset and then it calculates the gradient of the loss function with respect to the parameters, but only for the selected mini-batch. This gives an approximation of the gradient over the whole dataset.
Once the gradient is obtained, the parameters are updated in the same way that you already know.

$$w = w - \eta \cdot \nabla_w L \tag{2.16}$$

where w is the parameter to update, ∇ is the learning rate (which decide the size of the step) and $\nabla_w L$ is the gradient of the loss function with respect to w.
Then, this entire process is repeated for a number of iterations or until the loss function converges to a minimum.
Another advantage over Gradient Descent is related to the convergence properties:

for certain types of non-convex loss functions common in deep learning, SGD can help escape local minima and find better generalizing solutions.

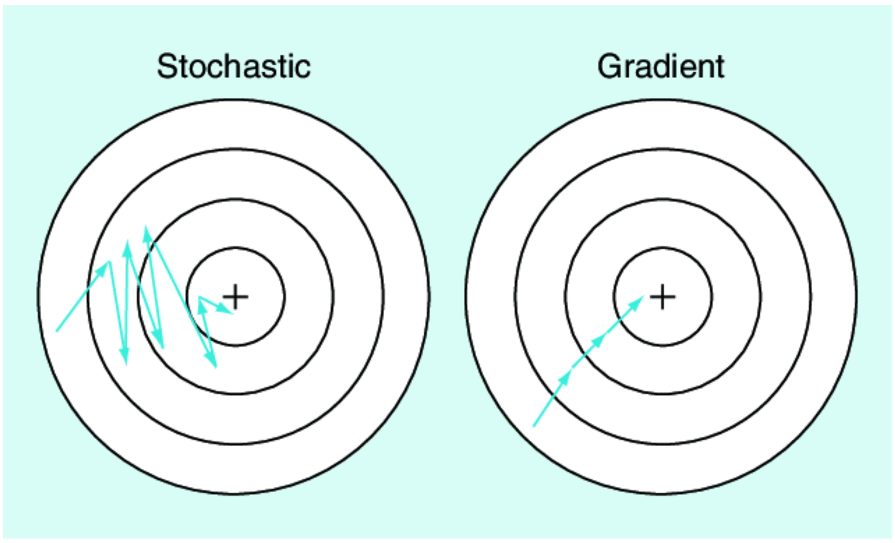


Figure 2.7: Stochastic gradient descent compared with gradient descent [9].

Several advances in deep learning have been built upon the foundation of stochastic gradient descent because of its efficiency and ease of use. In complicated deep learning environments, however, variations of SGD with momentum or adjustable learning rates (like Adam) are frequently chosen due to their enhanced convergence qualities.

2.4.2 Adaptive Moment Estimation (ADAM)

Adaptive Moment Estimation, commonly known as Adam, is an optimization algorithm designed to adjust the learning rates of parameters by estimating the first and second moments of gradients.
It's designed to replace the traditional stochastic gradient descent (SGD) procedure by computing adaptive learning rates for each parameter. Adam combines the best properties of two other extensions of stochastic gradient descent: Adaptive Gradient Algorithm (**AdaGrad**) and Root Mean Square Propagation (**RMSProp**).

Here's a thorough explanation of Adam's workflow from the standpoint of deep learning.
Two vectors, m and v, are initialised to zeros by Adam. The gradient square and running averages of the gradients will be stored in these vectors, respectively. Additionally, two hyperparameters that regulate the exponential decay rates of these moving averages are β_1 and β_2, which are usually near to 1.

Adam calculates the gradient of the loss function with respect to the parameters ($\nabla\theta$) for the current batch for every parameter change. Next, it applies the following formulae to update the running averages of the gradients (m) and their squares (v):

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot \nabla\theta \tag{2.17}$$

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot (\nabla\theta)^2 \tag{2.18}$$

where t indicates the current time step.
Because of their initialization, m and v are biassed towards zero in the first time steps. In order to account for this, Adam applies bias correction:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.19}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.20}$$

This correction helps in stabilizing the updates in the initial training phase.
At the end, Adam updates the parameters using the bias-corrected estimates:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{2.21}$$

where η is the learning rate, and ϵ is a small scalar (e.g., 10^{-8}) added to prevent division by zero.

This is the algorithm expressed in pseudo code of how Adam works.

2.4.3 Loss functions

Since the main problem of this study is a regression, there are several alternative methods to evaluate the accuracy of the developed model.

They are referred to as "alternative" methods because accuracy is a concept that is most frequently coupled with classification problems. In classification, accuracy metrics perform a spot check on what the model predicted and the expected output; if these two coincide, then the correct prediction counter is increased by 1. Building on this basic concept, several metrics have been developed to help evaluate the accuracy of a classifier, such as: confusion matrix, accuracy score, precision, recall, F1-score, etc.

With a regression problem, these metrics can no longer be used since the predicted value is not a membership class but a continuous numerical value.
This is why they are referred to as "**Loss functions**".

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

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

These are functions that make it possible to calculate how "close" or "similar" the prediction is to the expected value: the closer the prediction, the less the loss. In order to obtain a good predictive model, the goal is to minimize the loss function.

The following loss functions were used in the laboratory work of this thesis:

Loss function	**Formula**
Mean Square Error (MSE)	$\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2$
Root MSE (RMSE)	$\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(Y_i - \hat{Y}_i\right)^2}$
Mean Absolute Error (MAE) / L1 loss	$\frac{1}{n}\sum_{i=1}^{n}\left\lvert Y_i - \hat{Y}_i\right\rvert$

Table 2.1: Loss functions used in carrying out the work of this thesis

We can assume there is a dataset of size n and, after using the model to generate n-predictions, the cost function is calculated where Y_i is the i-th real value and $\hat{Y}_i$ is the i-th predicted value.
All three types of loss functions have one feature in common: the lower their values, the better the model (unlike other famous metrics used in regression such as R^2 and Adjusted-R^2).

The Mean Squared Error (MSE) is a quadratic scoring rule and it calculates the average squared difference between the estimated and actual values. The mean square of all the errors is used to compute it. The discrepancy between the expected and actual numbers is referred to as the error in this context. Squaring is required to eliminate any unfavourable indicators. Additionally, it has the effect of emphasising the greater discrepancies more. MSE's sensitivity to outliers is one of its main features. Outliers have a disproportionately significant impact on the MSE since the errors are squared. This implies that the presence of outliers in the data may cause the MSE to be very high and distort the model's interpretation of the data.

Next, we have the Root Mean Squared Error (RMSE), which is the square root of the MSE. Although they may appear unnecessary, square roots and squaring have a function. We can simplify understanding by returning the units to the original units of the output variable by calculating the square root of the mean square error (MSE). The square root helps to slightly offset the impact of outliers, but RMSE retains all of the characteristics of MSE, including its vulnerability to them. When a Gaussian distribution of error terms is anticipated, RMSE is frequently chosen because it can give a more understandable sense of the average error size.

Mean Absolute Error (MAE) is the last loss function. It calculates the average size of mistakes in a series of predictions without taking direction into account. It is computed as the mean of the absolute deviations between the observed and anticipated values. The MAE is not affected by outliers like the MSE and RMSE are. Because large deviations are linearly weighted, MAE offers a more reliable metric when outliers are present. Because of this, MAE is especially helpful in situations where you wish to prevent outliers from having a disproportionately negative impact on the model's performance.

Depending on the particulars of the given regression situation, each of these loss functions has distinct qualities and applicability. MSE and RMSE are appropriate for models where higher mistakes are undesirable and should be minimised since they are more harsh when it comes to them. However, MAE is better suited for scenarios in which outliers are predicted or uncontrollable since it offers a more comprehensible estimate of average error and is more tolerant of them.

2.5 Quantum computing

Using the ideas of quantum physics, information may be processed in ways that are essentially distinct from those of conventional computing, marking a significant breakthrough in the science of computation. Quantum computing is centred on the goal of solving complicated problems faster than conventional computers, which might transform whole industries by solving issues that were previously unsolvable.

[10] The advancement of quantum mechanics itself in the early 20th century laid the conceptual foundation for quantum computing. But serious progress towards the realisation of quantum computers did not occur until the 1980s. David Deutsch in 1985 and Richard Feynman in 1982 were influential individuals who explained the possibilities of quantum computing. While Deutsch outlined the quantum Turing machine and established the theoretical framework for quantum computing, Feynman suggested that a quantum system may be effectively emulated using another quantum system.
Significant progress has been made in quantum theory, algorithms, and technology throughout the years. Significant turning points include the factoring of huge numbers technique developed by Peter Shor in 1994 [11], which would have taken classical computers an unreasonably long time to complete, and the database searching algorithm developed by Lov Grover in 1996 [12], which showed a quantum speedup over classical algorithms.

In terms of information representation, quantum computing differs from classical computing from the very beginning. Quantum computers employ quantum bits, or **qubits**, as the lowest unit of information, whereas classical computers use bits. Because of the quantum mechanical concept of **superposition**, a qubit, unlike a bit, which can only be either 0 or 1, can exist in a state that is a simultaneous superposition of both 0 and 1.
This superposition allows quantum computers to process a vast amount of possibilities concurrently. Another key principle is entanglement, a phenomenon where qubits become interconnected and the state of one (no matter how far apart they are) can instantaneously affect the state of another. This capability is crucial for quantum computing's potential speedup.

The mathematical description of a qubit state is typically represented using Dirac notation, or "bra-ket" notation, where a qubit $|\psi\rangle$ can be written as a linear combination of the basis states $|0\rangle$ and $|1\rangle$:

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle \tag{2.22}$$

In this equation, α and β are complex numbers that describe the probability amplitudes of the qubit being in state $|0\rangle$ or $|1\rangle$. The probabilities of measuring the qubit in either state are given by $|\alpha|^2$ and $|\beta|^2$ respectively, with the condition that: $|\alpha|^2 + |\beta|^2 = 1$.

Beyond the capabilities of traditional computers, quantum computing holds great promise for addressing difficult issues in material science, medication development, and optimisation. Examples of these applications include quantum cryptography and quantum communication. Because they can solve issues that are now intractable owing to computing limits, quantum computers have the potential to completely transform a variety of disciplines.
Still, there are a number of important issues to be resolved, such as error rates, qubit coherence durations, and quantum system scalability. Realising the full potential of quantum computing requires developments in fault-tolerant quantum computing, quantum error correction, and innovative quantum algorithms.

The next several decades should see the release of useful quantum computing applications, which makes this a fascinating area for experimentation and study as research and development picks up speed. The direction of quantum computing in the future will surely be shaped by the junction of theoretical understandings, technical developments, and real-world applications.

2.5.1 Quantum gates

Similar to logic gates in conventional circuits, quantum gates modify qubit states and serve as the fundamental components of quantum circuits. They are the basic building blocks for quantum circuits.
Quantum gates use the concepts of superposition and entanglement to manage the quantum states of qubits, as opposed to conventional logic gates, which apply binary operations to bits (0 or 1). This allows for a wider range of operations that support the potential capability of quantum computing.

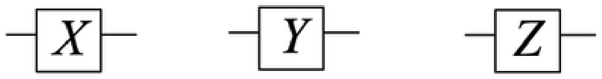
Quantum gates operate on qubits using unitary transformations, which are mathematical operations characterized by certain properties that ensure the total probability of all possible outcomes remains 1. A unitary transformation U applied to a quantum state $|\psi\rangle$ results in a new quantum state $|\psi'\rangle$, which can be represented mathematically as:

$$|\psi'\rangle = U\,|\psi\rangle \tag{2.23}$$

A 2x2 unitary matrix may be used to represent a quantum gate for a single qubit, and a 2-dimensional vector can be used to represent the qubit's quantum state. The qubit's vector and the gate's matrix are then multiplied to get the gate's action on the qubit. For example, the quantum state $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ can be represented as a vector $\binom{\alpha}{\beta}$, and a quantum gate U acting on $|\psi\rangle$ transforms it according to the matrix multiplication $U \cdot \binom{\alpha}{\beta}$.

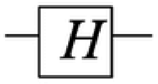
Quantum computing relies on many quantum gates, which are comparable to the conventional AND, OR, and NOT gates. Among the most well-known are:

- **Pauli Gates (X, Y, Z):** Pauli gates perform phase-shifting and quantum bit-flipping. The Z gate modifies a qubit's phase, the Y gate performs a bit and phase flip, and the X gate (which is comparable to a classical NOT gate) flips a qubit's state.
- **Hadamard Gate (H):** By generating superpositions, the Hadamard gate transforms a defined state—either $|0\rangle$ or $|1\rangle$—into a collection of states. It is essential for generating superpositions of states to initialise quantum algorithms.
- **Controlled NOT Gate (CNOT):** The CNOT gate is a two-qubit gate that flips the second qubit (target) if the first qubit (control) is in state $|1\rangle$. It's essential for creating entangled states (Fig.2.10).
- **Toffoli Gate (CCNOT):** If the first two qubits are in the $|1\rangle$ state, the Toffoli gate acts on three qubits, flipping the third qubit. Since it can implement any



$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

Figure 2.8: Pauli gates (X, Y, Z)



$$\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

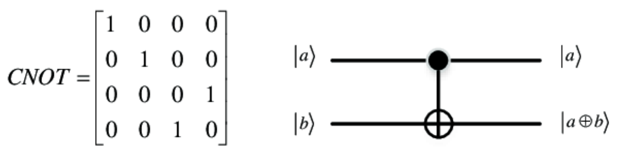
Figure 2.9: Hadamard gate (H)

quantum computation, it is a universal gate for quantum computing (Fig.2.11).

In summary, quantum gates bring up computational possibilities that are not possible with traditional technology by enabling the manipulation of information in fundamentally new ways. In order to fully utilise quantum gates despite their present limitations, quantum algorithms and error correction strategies are still being developed.

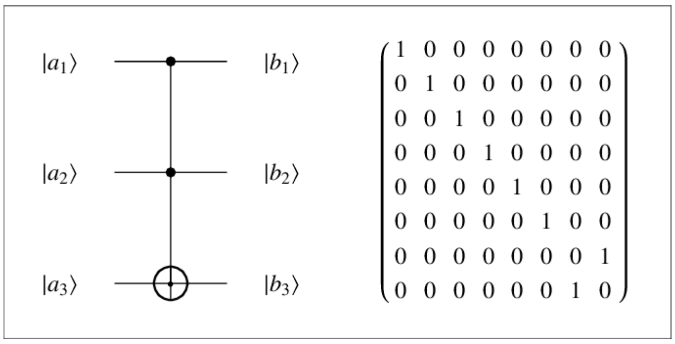
2.6 Quantum Machine Learning

With the goal of using the concepts of quantum physics to improve or speed up machine learning algorithms, quantum machine learning, or QML, is a promising fusion of quantum computing with machine learning. The area arose from the realisation that quantum computing has novel computational capabilities that may be able to tackle some problems—including machine learning problems—more



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

Figure 2.10: Controlled NOT Gate (CNOT)



$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure 2.11: Toffoli gate (CCNOT)

quickly than traditional computers.

Since academics started looking into the possibilities of quantum algorithms in the late 20th century, QML has its roots in the early advances in quantum computing itself [13]. But the phrase "quantum machine learning" didn't catch on until the 2010s, when theoretical developments and advances in quantum hardware made the real-world applications of quantum computing more plausible. Because quantum algorithms are essential to many machine learning techniques, early research concentrated on finding ways to speed up linear algebra work using them. During this time, quantum versions of support vector machines, principal component analysis, and other techniques were developed.
The need to address the rising computing needs and complexity of some machine learning tasks—tasks that are difficult for traditional computers to complete in a reasonable amount of time—led to the development of QML. These include, but are not limited to, simulating complicated systems, large data analysis, and difficult optimisation challenges. These kinds of problems are frequently encountered in domains such as artificial intelligence research, financial modelling, and drug development.

When quantum computing's built-in benefits, such parallelism and entanglement, can be used to handle data in ways that classical computing cannot, quantum machine learning can be very useful. In particular, QML has potential in:

- **Increasing the speed of linear algebra computations**: A lot of machine learning techniques rely on calculations that, on classical computers, scale poorly with data size, but may run exponentially faster on quantum computers.

- **Managing high-dimensional data:** High-dimensional vectors and matrices may be compactly represented and manipulated by quantum systems, which may provide more effective methods of handling complicated, high-dimensional information.

- **Optimisation issues:** In certain optimisation tasks, which are essential for training machine learning models, quantum algorithms may be able to identify optimum solutions more quickly than conventional algorithms.

Fundamentally, QML uses quantum algorithms to outperform classical algorithms in tasks like classification, pattern recognition, and optimisation. One popular method is to use quantum embedding, which is the act of encoding data into quantum states so that the quantum system may process and analyse data in its high-dimensional quantum space.
The quantum rendition of traditional machine learning methods is a fundamental component of QML. To identify the ideal parameters for a particular model, for instance, the Quantum Approximate Optimisation Algorithm (QAOA) is made to address optimisation issues, a common task in machine learning. Similar to this, quantum algorithms for linear systems of equations seek to solve for x in $Ax = b$ under specific conditions exponentially quicker than conventional methods; this has immediate implications in the resolution of linear regression issues.

In terms of mathematics, a quantum machine learning algorithm could construct a quantum state that reflects the data ($|\psi_{data}\rangle$) by executing a sequence of quantum gates, or unitary operations. Further quantum operations are then applied to this state in order to accomplish the appropriate machine learning objective. When the result is measured, the quantum state collapses to classical data that may be understood as the computation's consequence.

In conclusion, Quantum Machine Learning offers a look into a future where processing power and the laws of quantum mechanics might open new capabilities in data analysis, pattern recognition, and beyond. It sits at the nexus of quantum computing and artificial intelligence. Even though it is still in its infancy, QML research and development are poised to address some of the most difficult and urgent problems in computational science and machine learning.

2.6.1 Linear Regression with QML

To illustrate the power of Quantum Machine Learning (QML) through a simple example, let's consider the problem of linear regression.
The goal is to find the line (in two dimensions) that best fits the data. The classical equation for a line is $y = mx + b$, where m is the slope and b is the y-intercept.

As it is possible to observe in Section 2.3, given a dataset of n points $\{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, we aim to minimize the cost function which is often the sum of squared residuals (i.e: the differences between observed and predicted values).
By encoding the linear regression issue into a quantum system and applying quantum methods to determine the ideal parameters that minimise the cost function, we may solve the problem in a quantum context. Since linear regression may be thought of as the solution to a system of linear equations, the quantum algorithm for linear systems of equations, or HHL (Harrow, Hassidim, and Lloyd) algorithm, can be modified for this use.

Encoding Data into Quantum States.
The first step involves encoding the data $\{(x_i, y_i)\}$ into a quantum state. This is achieved through quantum embedding, which maps classical data into the amplitudes of a quantum state. For simplicity, let's consider each x_i and y_i as components of vectors x and y respectively. The goal is to encode these vectors into quantum states $|x\rangle$ and $|y\rangle$.

Formulating the Problem.
Finding a vector θ (containing m and b) that minimises the norm $||\mathbf{A}\theta - y||^2$ may be used to express the linear regression issue. Here, $\mathbf{A}$ is a matrix with x values and an extra column of ones for the intercept term b. We seek to solve the system of linear equations $\mathbf{A}\theta = y$ that result from this formulation.

Quantum Linear System Algorithm (HHL).
The HHL algorithm is used to solve the linear system $\mathbf{A}\theta = y$ quantumly. The algorithm proceeds in several steps:

- Prepare the quantum state $|y\rangle$ corresponding to the vector y.
- Apply quantum phase estimation to estimate the eigenvalues of $\mathbf{A}$.
- Use the eigenvalues to perform controlled rotations on an ancilla qubit, encoding the solution into the amplitudes of a quantum state.
- Apply the inverse of the quantum phase estimation step to retrieve the state encoding the solution.

- Measure the ancilla qubit. If it is in the desired state, the system collapses to a state proportional to the solution vector $|\theta\rangle$, from which we can extract the regression coefficients.

The sparsity and condition number of **A** have a big impact on how well these stages can be completed, which is necessary for the HHL algorithm to work. The main conclusion is that the HHL algorithm may be able to solve the linear system more quickly than conventional approaches, particularly for large and sparse systems, even if a thorough mathematical analysis of each step would need a thorough study of quantum algorithms and linear algebra.

2.6.2 Variational Quantum Circuit

Parameterized quantum circuits, or variational quantum circuits, or VQCs, are at the forefront of quantum computing research at the moment, particularly in the field of quantum machine learning (QML). These circuits belong to a class of quantum algorithms that use the adaptability of quantum systems to optimise a given objective function in order to solve issues that are difficult for conventional computers to solve. The use of VQCs in near-term quantum devices—also known as Noisy Intermediate-Scale Quantum (NISQ) technology—makes them especially noteworthy.

A Variational Quantum Circuit is a hybrid quantum-classical algorithm framework that uses a quantum circuit with tunable parameters, optimized through a classical feedback loop [14]. The fundamental feature of a VQC is its capacity to modify quantum gate settings in order to minimise (or maximise) a given objective function. For this reason, it is an ideal tool for machine learning applications, quantum chemistry simulations, and optimisation challenges.

A new era in the practical applications of quantum computing is being heralded by VQCs, which provide a flexible architecture that has demonstrated tremendous promise in a range of computational fields. Because of their special combination of quantum and conventional processing techniques, they are highly adaptive and can solve issues that classical algorithms cannot now solve. Quantum optimisation is one of the key areas where VQCs have had a significant influence as they provide innovative methods for more quickly identifying the best solutions for particular problem classes. This capacity is especially important for operations research, logistics, and complex systems analysis, where optimisation is a key component.
Furthermore, the creation of VQCs is extremely beneficial to the field of quantum machine learning as it lays the groundwork for improving traditional machine learning models as well as creating brand-new quantum-native models. By taking use of the inherent parallelism and entanglement of quantum mechanics, VQCs provide

routes to possibly better performance in tasks like classification, grouping, and regression. This benefit might result in quicker training durations or more accurate models, both of which are essential for data-intensive applications including natural language processing, healthcare, and finance.
Apart from these domains, another domain that is very amenable to modification by VQCs is quantum chemistry. These circuits' capacity to mimic chemical and molecular processes presents previously unheard-of possibilities for materials research and medicine development. VQCs have the potential of providing more accurate simulations of complicated quantum events within molecules, something that traditional computer approaches frequently struggle to do. By offering previously unreachable insights into molecular interactions and reactions at the quantum level, this capacity might expedite the creation of novel medications and materials.

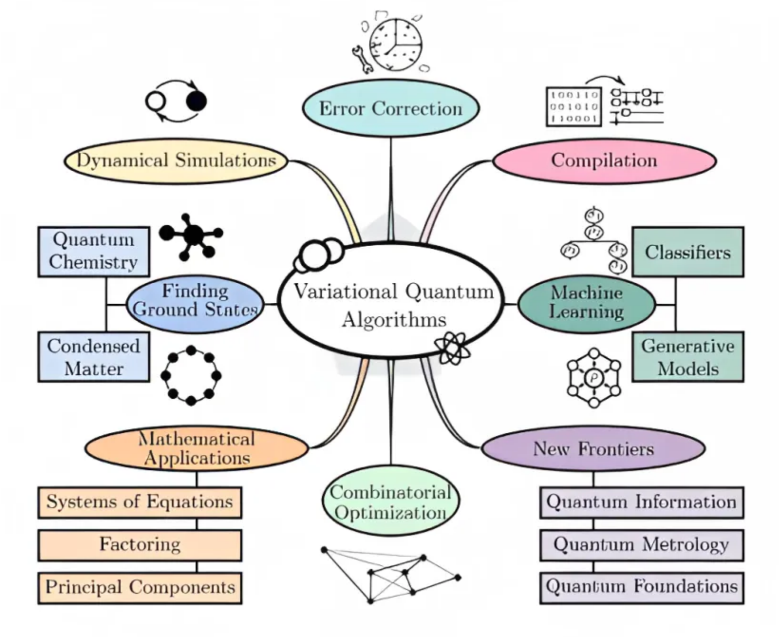


Figure 2.12: Applications of variational quantum algorithms - source: [14]

The Variational Principle in quantum mechanics serves as the basis for quantum variational algorithms. These algorithms compute approximations of solutions to a given issue, are independent of the problem description, and comprise modular circuit components. The flexibility of Variational quantum algorithms (VQAs) results in a range of algorithmic structures with differing degrees of complexity; yet, all variational algorithms share a fundamental foundation (Fig.2.13):

- **Problem initialization:** The quantum computer is started in the default

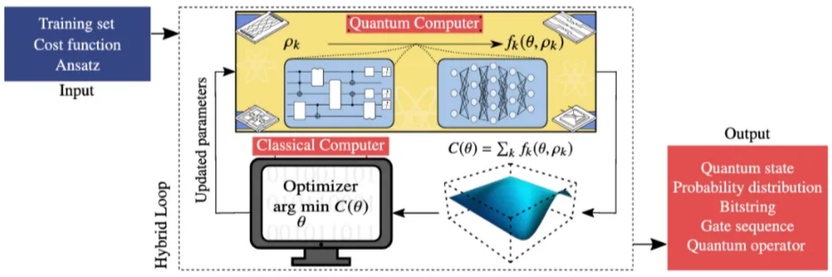


Figure 2.13: Core framework by VQAs - source: [14]

state $|00..0\rangle$. It is then changed into the reference state $|R\rangle$ by use of a non-parameterized unitary reference operator Ur.

- **Circuit Ansatz:** (the circuit's design), which might be user-dependent or fixed for an algorithm. assists us in reaching the intended state $|Y(x)\rangle$ from the default state $|00..0\rangle$. A variational operator of the type $Uv(x)$, which is a group of explorable parameterized states, aids in the transformation. Ansatz operators can be seen as a group of:

$$U_A(x) = U_V(x)U_R \tag{2.24}$$

- **Cost function:** To systematise the specific aim that we ultimately want to maximise or minimise, an objective function unique to the situation is needed. We represent it as a linear combination of Pauli operators operating on a quantum system, or $C(t)$.

- **Training process:** This subroutine effectively computes a function from the quantum circuit's output measurements and iteratively adjusts the circuit parameters until it finds the best answer. In the event that a less-than-ideal solution is already identified, the data may be utilised by starting the optimisation process from this point on.

Within the limitations of existing technology, Variational Quantum Circuits provide a potential way to harness the power of quantum computing. Even before completely fault-tolerant quantum computers are accessible, VQCs provide a method to investigate the advantages of quantum computing in a variety of applications, from machine learning to quantum chemistry, by fusing the best aspects of quantum and conventional computing. They are an important topic of study and experimentation in the effort to realise useful quantum computing applications because of their versatility and effectiveness in using NISQ devices.

Chapter 3

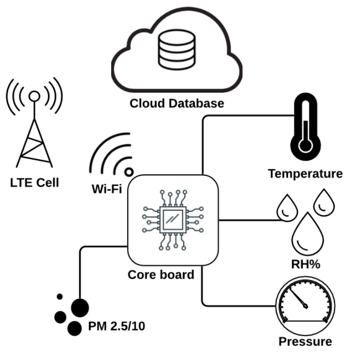
Solution and Methodology

This chapter outlines the technique and suggested solution for the calibration of optical fine-dust sensors, which addresses the issues that have been found. The technique offers a new solution by utilising state-of-the-art technology and processes, all based on a thorough examination of the issue space. We start by describing the theoretical foundations of our methodology, including the fundamental ideas and structures that guide our resolution. Subsequently, we provide our suggested system design, including its constituent parts, their interplay, and their combined roles in accomplishing the intended results. Special attention is given to the innovative techniques and tools employed, as well as the rationale behind their selection.

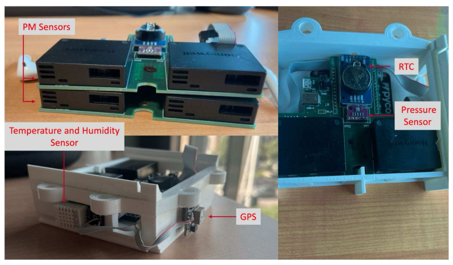
3.1 Proposed solution

The main objective of this thesis is to study how Deep Learning and Quantum Machine Learning (QML)-based approaches can impact the calibration of sensors for fine dust detection.
Enhancing the precision and dependability of particle matter (PM) readings in urban air quality monitoring requires better calibration of the sensors. Although inexpensive and small enough for extensive monitoring networks, low-cost light-scattering PM sensors have drawbacks in terms of precision and accuracy because of things like miniaturisation, environmental variables (such as relative humidity), and the intrinsic unpredictability in particle composition. By modifying sensor output to match reference measurements, calibration helps to mitigate these problems and enhances data quality by lowering errors. Producing accurate data on air quality is essential for environmental policy-making, public health campaigns, and the creation of more advanced, self-governing monitoring systems. By calibrating the data, one may be sure that it is reliable and significant information for evaluating air quality and making judgements.

As already mentioned, this project is a follow-up to a series of projects conducted by the Department of Control and Computer Engineering at the Polytechnic University of Turin that began in 2015.
The first study [1] recounts the design, development and installation phase of low-cost area quality monitoring boards compared to those provided by ARPA.
The architecture designed for the implementation of these stations is expressed in the figure 3.1a.



(a) Architecture of the proposed system from [1].



(b) Air monitoring station hardware from [15].

Figure 3.1: Air monitoring station architecture and hardware, deployed over ARPA reference station.

The primary computer in these stations is a Raspberry Pi Zero Wireless, which runs the ARM-based Arch Linux operating system. Data is gathered by the system from six distinct sensors. Using light scattering, four Honeywell HPMA115S0-XXX sensors determine the levels of $PM_{2.5}$ and PM_{10} particles. In this work, we only take into account $PM_{2.5}$ because the sensor calculates PM_{10} from $PM_{2.5}$ rather than measuring it directly.
The BME280 sensor records air pressure, while the DHT22 sensor detects temperature and relative humidity (Fig.3.1b).
Every two seconds, temperature and relative humidity are measured at the air monitoring stations, and every five seconds, atmospheric pressure is measured. PM2.5 is measured at these intervals.
We set up fourteen monitoring stations on the roof of the Rubino ARPA station in order to carry out the investigation. They were placed on the property $1.5 \sim 2m$ from the reference grade devices' air inlets.

In subsequent studies by the same laboratory, the focus has been on developing a new framework that can make these stations more reliable and automatic (i.e., without manual analysis done by an operator).

Starting with the raw data, collected from all 6 sensors installed at each station, we address a pipeline that has been shown to statistically improve the accuracy with which the sensors do their work and also their resistance to failure.
Such a pipeline is depicted in Figure 3.2.

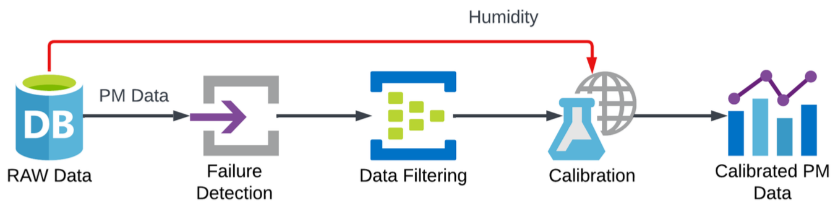


Figure 3.2: Actual framework overview.

Failure detection stage: it focuses on examining raw sensor data to find instances of sensor failure, paying special attention to the difficulty provided by embedded photosensor defects. These mistakes are troublesome because they result in continuous data production without any signs of failure, which makes it challenging to identify defects, particularly when the errors exhibit random or non-deterministic patterns.
Data Filtering stage: it is responsible for locating and removing point abnormalities that compromise particulate matter (PM) measurement accuracy. These abnormalities are frequently caused by noise or outside influences, resulting in abrupt, high-frequency shifts that raise errors and produce outliers. with order to improve data quality with respect to ground truth values, a variety of filters were tested in an effort to lessen noise and outliers.
This procedure improves the quality of the data for subsequent analysis and calibration processes by precisely eliminating outliers while maintaining the integrity of the data set.
Calibration Model stage: it focuses on improving sensor accuracy through a "push-button" calibration phase that doesn't need human participation. This is especially crucial for large-scale installations, because the volume of data generated makes human data analysis impossible. By taking this step, low-cost light-scattering PM sensors' total data quality is greatly improved, guaranteeing accurate and dependable data for environmental monitoring and analysis.

The purpose of this thesis focuses on the design and implementation of future work left undone in the "Calibration" phase of the framework in Figure 3.2.
More in detail, the use of Deep Learning and quantum approaches to Machine Learning in place of the basic methods used in the work done so far is studied. The comparison of the results obtained is the real value brought by this thesis.

Thus, calibration serves to increase the precision and accuracy with which these (low-cost) sensors can make measurements. To evaluate the goodness of sensor calibration, the measured value from a reference station belonging to ARPA was taken in order to understand how "distant" the sensors' value is from that station. The aim, therefore, is to get as close as possible to the same level of accuracy as the sensors at ARPA stations but with low-cost sensors.

The proposed solution involves:

- **Dataset exploration:** Study the available dataset and extract some statistical highlights.
- **Dataset building:** Build a dataset suitable for the type of experiments that will be performed in order to produce results comparable with the current models used for calibration.
- **Model selection** and **hyperparameters tuning:** Build the artificial models and quantum circuits to perform the calibration; hyperparameters tuning will be performed to explore the space of possible configurations of each model and find out which are the best.
- **Cross-validation:** Perform cross-validation with the configured models to see if overfitting occurs on the dataset.
- **Get the results:** Produce a table with the final results obtained from each model in its best configuration (using different metrics).

The entire project was carried out by relying on a server owned by the Polytechnic University of Turin. The server has an Linux (Ubuntu 20.04.2 LTS) operating system with 245 gigabytes of RAM (Random Access Memory), an Intel Xeon Processor (Skylake, IBRS) with an x86 architecture and 28 cores available.

3.2 Data analysis and preparation

There are two main types of datasets involved in this project: the one containing data from our installed boards and the one containing data measured by ARPA. Both datasets contain the star feature of our work, which is the level of particulate matter in the air ($PM_{2.5}$ and PM_{10}). The other data collected from both our and ARPA's sensors are relative humidity level, temperature, and atmospheric pressure.

Some boards installed in Rubino (TO) did not always stay on: this was because several tests were in progress or because some sensors had broken. This generates a nonlinearity of data acquisition by the same sensors along time: we will see later how to make up for this lack of data.

In addition, it is important to note that not all of the installed boards were used for the purpose of this thesis. Some boards date back to the experiments conducted in paper [1] of which it was realized that one of the reasons why some poor quality output was being produced was due to the hardware (i.e., the type of sensors used). Therefore, in 2022, another data collection session was conducted (again at the same geographic location) but this time with updated boards: the hardware inside consists of the sensors discussed in section 3.1 (using light-scattering sensors for dust detection).

For this reason, all the data from the boards installed during the period from November 3, 2022 to June 6, 2023 was used as dataset in this thesis. The boards considered are those with the following identifiers (for internal use by the Department): 20, 21, 22, 25, 29 and 31.
In the following sections, a univariate and multivariate analysis of the data from our sensors was done in order to better understand their nature, correlations, and to identify what will be the best dataset to provide to the predictive models that will be explained later.

3.2.1 Univariate analysis

The purpose of this type of analysis is to understand the statistical nature of the features measured by our sensors. Distribution, outliers, missing or duplicate values are all that we will bring out of this analysis.
The board with identifier 20 was used to conduct the analysis. The reason for this choice is because this board remained operational throughout the time of the experiment (November 2022 to June 2023).
As we already know, all boards have six sensors inside: four sensors for fine dust detection (Honeywel), one sensor for temperature and humidity, and one last sensor for atmospheric pressure. Every 2 to 4 seconds, the sensors take the measurement and save it to the dataset, so the number of records we are dealing with travels in the millions (about 9.7 million of data collected from board 20).
We therefore show a univariate analysis by considering one feature at a time.

Particle matter feature:
As can be seen from Figure 3.3, the distribution of particulate matter observations collected by the sensor follows a Gauss bell, that is, a distribution according to the random variable "Normal". This type of distribution is optimal because it has been shown that leading machine learning algorithms learn better if a feature is distributed in the dataset in this way, i.e., they are able to produce higher quality predictions due to the naturalness of the reality represented in the observed data. The box plot (Fig.3.3), on the other hand, is useful for understanding whether

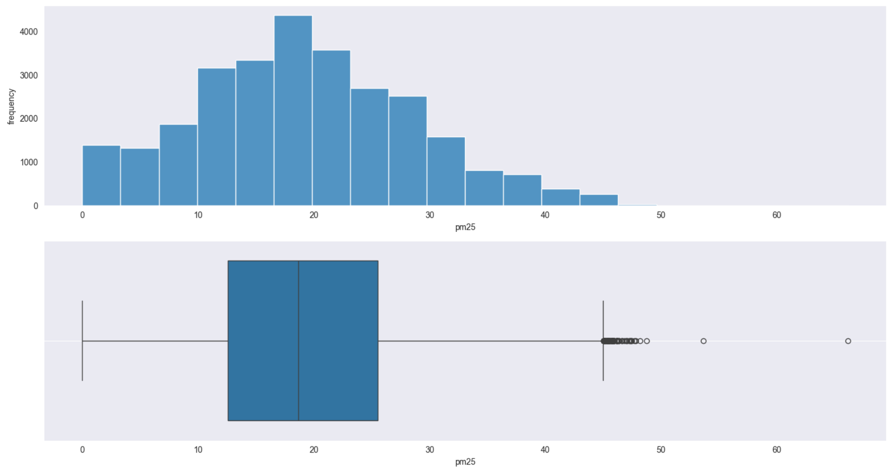


Figure 3.3: The histogram (on top) and the box-plot (on bottom) of the $PM_{2.5}$ sensor.

there are and where the outliers of a feature are. In this case, there are outliers on the right side of the blue box (where most of the data points are concentrated): this means that some points in the dataset have a value greater than the maximum statistically relevant value in this distribution.
Confirming this, also in the histogram we find an "empty" value space on the right of the Gauss bell.

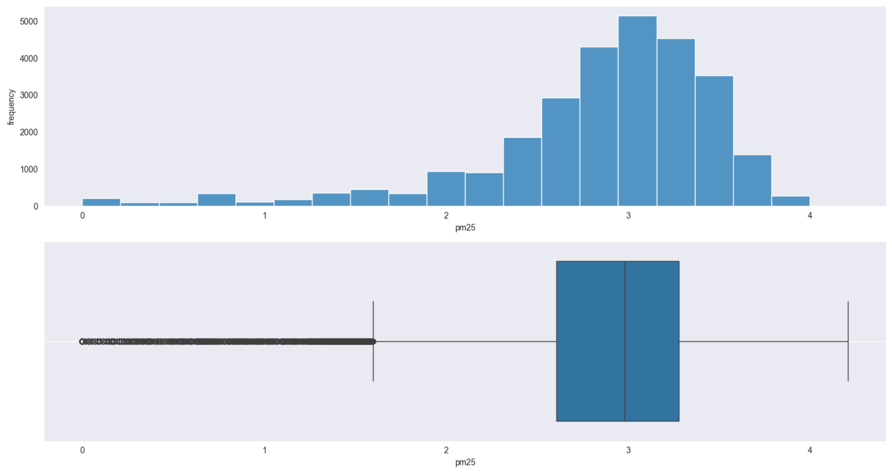


Figure 3.4: The histogram (on top) and the box-plot (on bottom) of the logarithm function of $PM_{2.5}$ sensor.

Although not entirely necessary, the logarithm function was applied to $PM_{2.5}$ features, and the results are shown in the Figure 3.4. Such a transformation, in

the world of statistical analysis, serves mainly to reduce the range of values the variable can take, reduce the number of outliers and allow the variable to distribute itself according to a Normal distribution. We say that this transformation was not necessary because, looking at the original feature in Figure X, one can see by eye how it already distributes according to a Normal distribution and the outliers do not seem to be too many.
We can see that the range of feature values varies between 0 and 4 (in the logarithmic case), whereas previously it varied between 0 and about 70. The average has moved around the numerical value 3, and the Gauss bell is even more visible.
Some outliers have accumulated below the minimum statistical threshold of the new distribution with the logarithm: however, since the range of values is so small, these outliers are less significant than those in the original distribution of the variable.

Let us now look at the scatter plot showing the distribution of $PM_{2.5}$ points along the time axis (Fig.3.5). On the right side you can find the feature's data without any kind of aggregation (i.e., near every second), while on the left side an aggregation by minutes was performed.

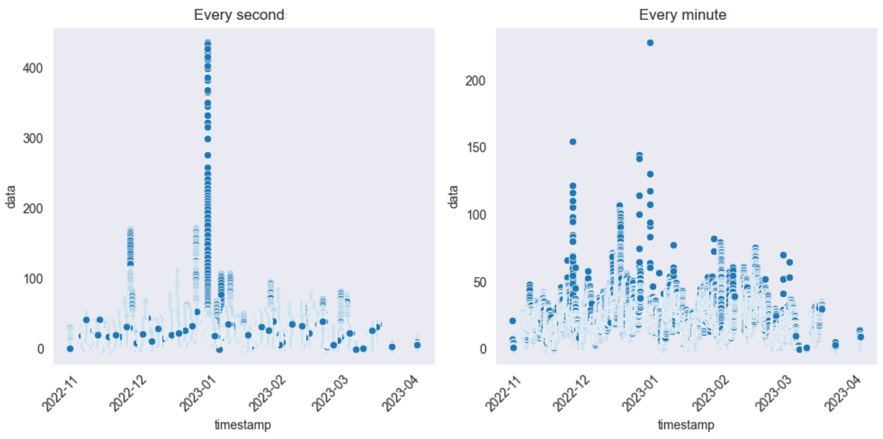


Figure 3.5: The scatter-plot of the $PM_{2.5}$ feature along the time and divided by different types of aggregation.

The concentration of blue color at certain points on the graph shows the density of the feature: the more colorful the area, the denser it is.
It is possible to see that the detailed distribution per second is more faithful to reality but consequently shows more outliers and there are many areas of dense (and therefore insignificant) points. Aggregating the dataset points to the minute, on the other hand, results in a point cloud that does not reflect the shape of the original dataset (due to the loss of information caused by aggregation) but on the other hand, the aggregated data are more concentrated in specific areas and some

outliers are gone.
The surge seen in both spike clouds indicates New Year's Eve: during this night in Turin it is customary to make use of fireworks and firecrackers, which release an incredible amount of particulate matter into the air.

Temperature feature:

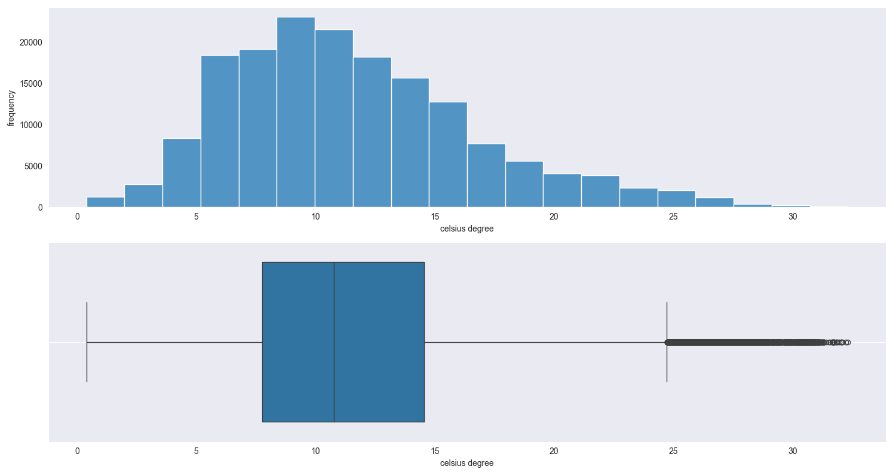


Figure 3.6: The histogram (on top) and the box-plot (on bottom) of the temperature sensor.

Figure 3.6 shows the distribution of the variable representing the temperature (in degrees celsius) measured by the appropriate sensor in board 20.
The histogram again shows a distribution similar to the Normal distribution. The only outliers encountered are above 30°C and represent the first days of June 2023. The seasonal nature of atmospheric temperatures explains the presence of some outliers in any dataset that collects this information. In fact, depending on the length of the observation period and where on Earth we are, there may be more observations with high degrees Celsius than with low degrees.
Recall that, in the case of this thesis, the data collection period is from November 2022 to June 2023. This implies a presence of cold rather than warm temperatures because the fall, winter, and spring months are present in that period.

The distribution along the time axis (Fig.3.7) shows alternating high and low values: this behavior can be explained by the alternation of night and day when temperatures are colder or warmer, respectively.
In aggregating the data by minutes, it is easier to see the temperature trend during the months of observation: from November 2022 to January 2023, there is a drop in average temperature, which then tends to rise as the warmer months of the year 2023 arrive.

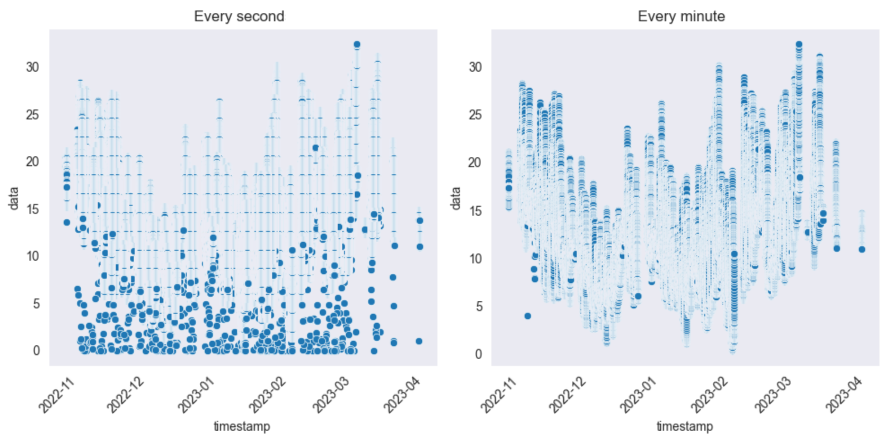


Figure 3.7: The scatter-plot of the temperature feature along the time and divided by different types of aggregation.

Humidity feature:

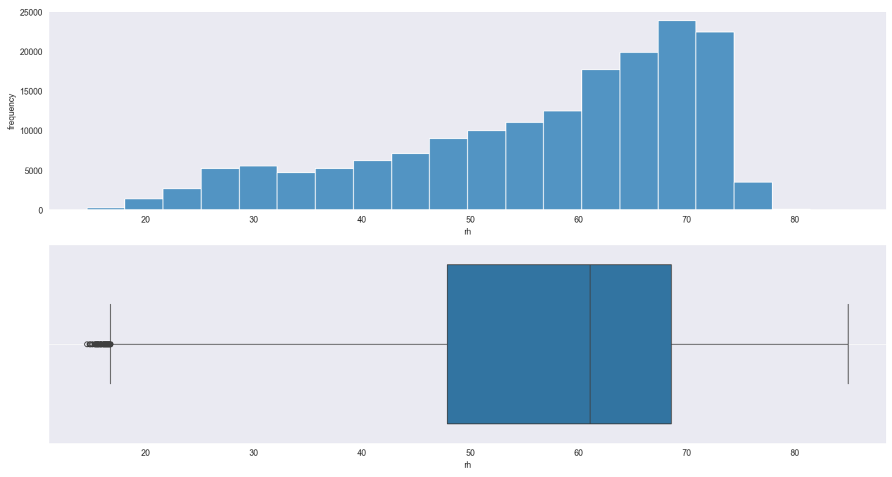


Figure 3.8: The histogram (on top) and the box-plot (on bottom) of the humidity sensor.

The moisture values collected by the sensor in board 20 (expressed in percentage of humidity in the air) do not follow a Normal distribution and have a non-uniform frequency. Outliers are present on the left side of the box-plot, that is, below the statistically relevant minimum threshold.
By applying a logarithmic transformation of the values, we are able to distribute the values slightly better but generate many outliers.

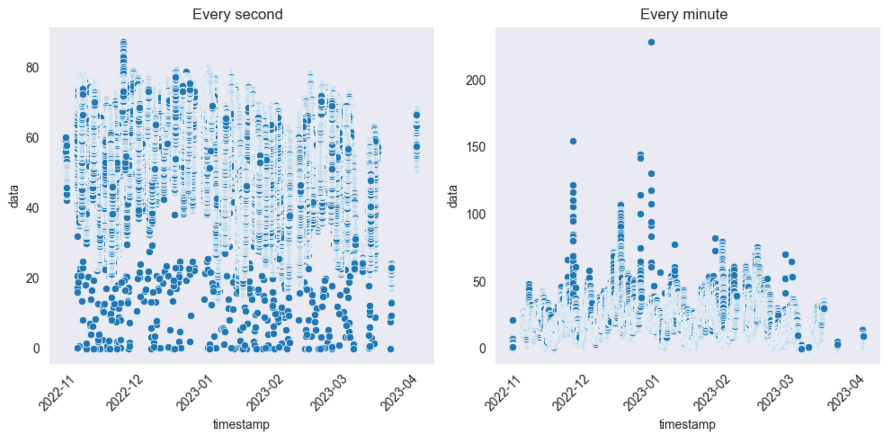


Figure 3.9: The scatter-plot of the humidity feature along the time and divided by different types of aggregation.

As evidence for the presence of outliers and a distribution of values with non-uniform frequencies, it can be seen that the scatter plot in Figure 3.9 shows a scattered cloud of points (without following a specific pattern) and some points are independent of the main mass of that cloud.
To use a feature such as this, it is highly recommended that data preprocessing techniques be used in order to improve the quality of the predictions produced by any statistical model (which will not be done and will be explained in later sections).

Pressure feature:

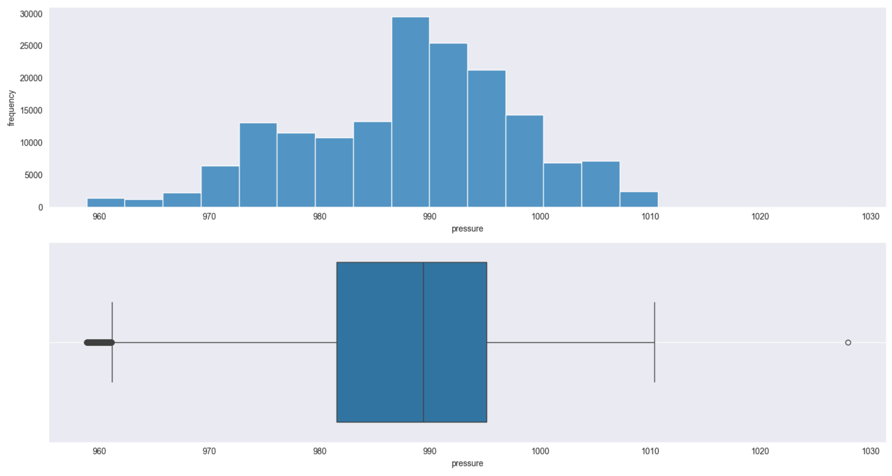


Figure 3.10: The histogram (on top) and the box-plot (on bottom) of the pressure sensor.

Our sensor pressure is measured on the order of Hectopascal, a unit of pressure equal to a millibar ($1hPa = 1mb$).
The Figure 3.10 shows a distribution similar to the Normal distribution, with several outliers outside the box-plot at both extremes. The average value recorded throughout the observation period is $990hPa$.
The recording of outliers with values that are too high can only be explained by a failure on the part of the sensor to perform the measurement. Such measurement failures can also be observed in the scatter plot at Figure 3.11.

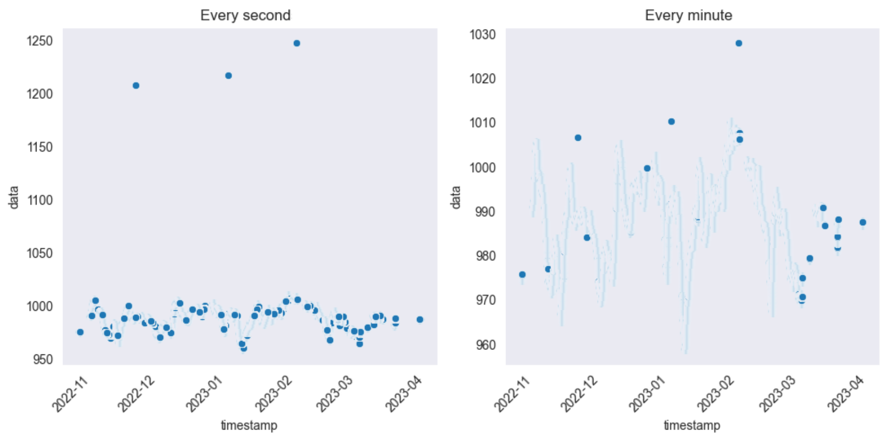


Figure 3.11: The scatter-plot of the pressure feature along the time and divided by different types of aggregation.

It is important to note how the aggregation of these values on the order of minutes (Fig.3.11) brings with it various benefits such as: the reduction of outliers and therefore the concentration of the data in a shorter range and the greater acquisition of importance and expressiveness that the data hire.

3.2.2 Multivariate analysis

Unlike univariate analysis, the one we propose in this section involves analyzing the behavior of multiple variables simultaneously. The study of natural correlations between variables is what is posed to address this type of analysis.
The variables in question are the same four seen in section 3.2.1: air particulate matter, temperature, humidity and atmospheric pressure.

As a matter of simplicity in handling a less heavy dataset (with reduced observations) and greater expressiveness and significance of the data, it was decided to approach this analysis by aggregating all variables with granularity to the minute instead of every second.

The data observation period is from November 2022 to early April 2023. The choice not to use the entire time range (November 2022-June 2023) is due to the fact that the sensors in board 20 correctly recorded measurements until early April 2024.

As a first analysis, we show the correlation among the four features by Pearson's correlation coefficient compared with Spearman's correlation coefficient [16]. Both of them are indicators of the direction and intensity of the link between two variables, but their methods of calculation and what they measure are different:

- Pearson's Correlation Coefficient (r):
  - Measures the linear relationship between two continuous variables.
  - Assumes a linear connection and a normally distributed set of data.
  - It is sensitive to outliers.
  - Computed by dividing the variables' covariance by the sum of their standard deviations.
- Spearman's Correlation Coefficient (ρ):
  - Measures the monotonic relationship between two variables, whether linear or not.
  - Does not need a normal distribution of the data; it may be used to non-linear relationships or ordinal variables.
  - Computed using the data's rankings as opposed to the data's raw form.

The only thing these coefficients have in common is the range of values they can take, in fact the value varies between -1 and 1, where 1 indicates a perfect positive linear relationship in Pearson's coefficient (or a perfect monotonic increasing relationship in Spearman's coefficient); the -1 value indicates a perfect negative linear/monotonic relationship, and 0 indicates no relationship.
Below are the two correlation matrices calculated with both coefficients mentioned (Fig.3.12). As is easily noticed, on the main diagonal of this matrix is the constant value 1: the values of a feature with itself are always highly correlated.
We also note how the correlation coefficient calculated with Pearson's formula produces results of greater significance for the purposes of analysis (i.e., negatively or positively distant from zero). For this reason, we will only consider Pearson's correlation.
Certainly, the first correlation that stands out is the one present at +59% (positive) between humidity *rh* and temperature (*temp*). The relationship between these two dimensions is explained by their natural behavior: when atmospheric temperature rises, moisture is likely to form in the air, especially in cities like Turin.

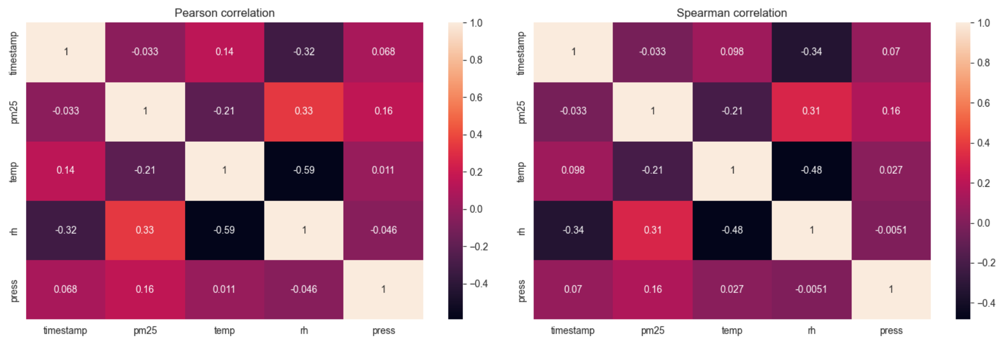


Figure 3.12: The correlation analysis: on the left the Pearson's correlation, on the right the Spearman's one.

Pearson correlation:

	timestamp	pm25	temp	rh	press
timestamp	1	-0.033	0.14	-0.32	0.068
pm25	-0.033	1	-0.21	0.33	0.16
temp	0.14	-0.21	1	-0.59	0.011
rh	-0.32	0.33	-0.59	1	-0.046
press	0.068	0.16	0.011	-0.046	1

Spearman correlation:

	timestamp	pm25	temp	rh	press
timestamp	1	-0.033	0.098	-0.34	0.07
pm25	-0.033	1	-0.21	0.31	0.16
temp	0.098	-0.21	1	-0.48	0.027
rh	-0.34	0.31	-0.48	1	-0.0051
press	0.07	0.16	0.027	-0.0051	1

If we look at the horizontal row of the $PM_{2.5}$ variable, we will observe the correlation it has with the other variables in the dataset (Fig.3.13). Aside from

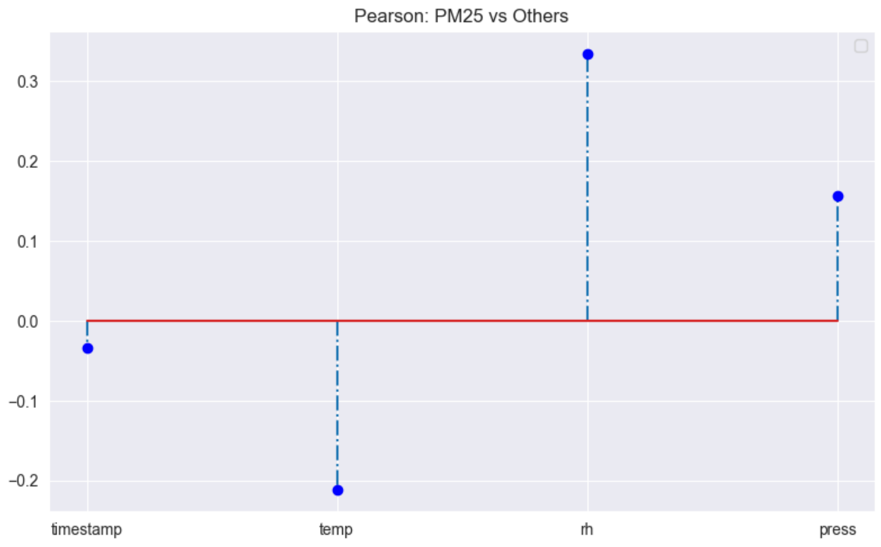


Figure 3.13: The Pearson's correlation analysis focused on the $PM_{2.5}$ feature against all the other features.

the timestamp (i.e., the day and month we are in), humidity rh proves to be the variable most correlated (positively, 33%) with particulate matter in the air. Right after we find temperature, with a negative correlation of -21%: when the temperature rises, particulate matter values tend to decrease. This phenomenon is scientifically explainable and often happens in large crowded cities, including Turin. During arid summer days, when the temperature reaches high peaks, the fine particles in the air do not have time to condense at a specific point in space.

To conclude the correlation study with the main variable of this thesis, air particulate matter correlates +16% with atmospheric pressure *press* and -0.033% with *timestamp*. The dimension representing the timestamp may be negligible at this point in the study since we are studying the correlations present between measurements taken directly from the sensors on the boards.

Given the evidence of some interesting correlations (some due to natural phenomena), we draw the scatter plot showing two variables in comparison.
The point cloud that forms represents the correlation between the 2 variables: if this is scattered it means that there is a low correlation (the correlation coefficient is close to zero), otherwise there is a visible correlation that can be positive (follows the direction of the diagonal) or negative (follows the direction of the anti-diagonal). In the Figure 3.14 we show the correlation between the most correlated variables.

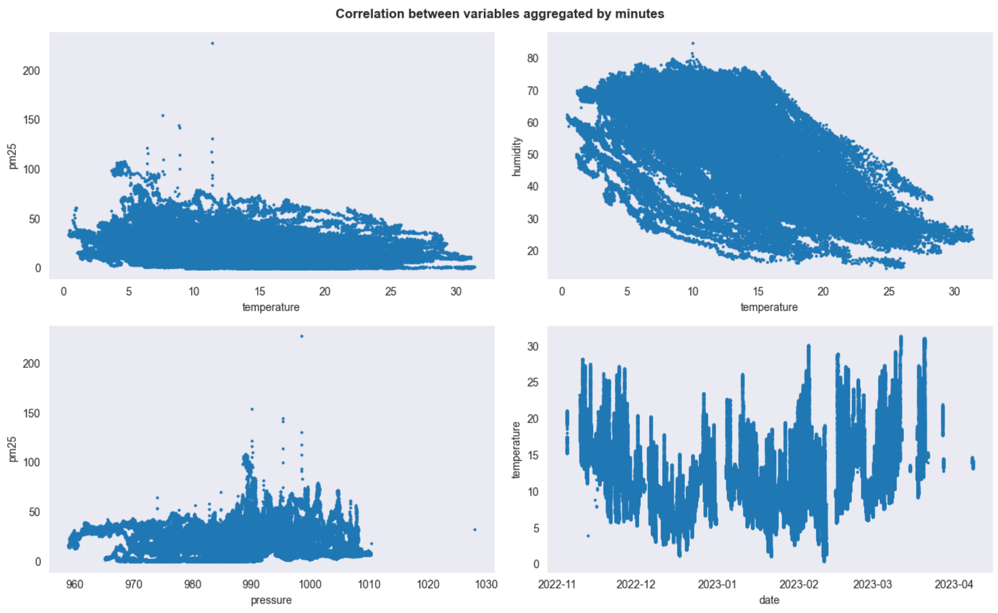


Figure 3.14: The scatter-plot between two variables with data aggregated by minutes

All variables show in Figure 3.14 are correlated with each other with a high Pearson's coefficient value compared to the matrix in Figure 3.12 (we are talking about ±30% of correlation). It can be seen in the upper right graph that there is a negative correlation between humidity and temperature; the correlation between particulate matter and pressure (lower left) or temperature (upper left) show a slight direction on the diagonal and anti-diagonal respectively.
Finally, atmospheric temperature is also correlated with the date in the most natural way there can be, which is through seasonality.

Interesting is to observe what happens if we aggregate the data at the hourly level. In this way we are definitely losing information, but we are able to better highlight the correlation between the same variables shown in Figure 3.14 but aggregated to the minute.

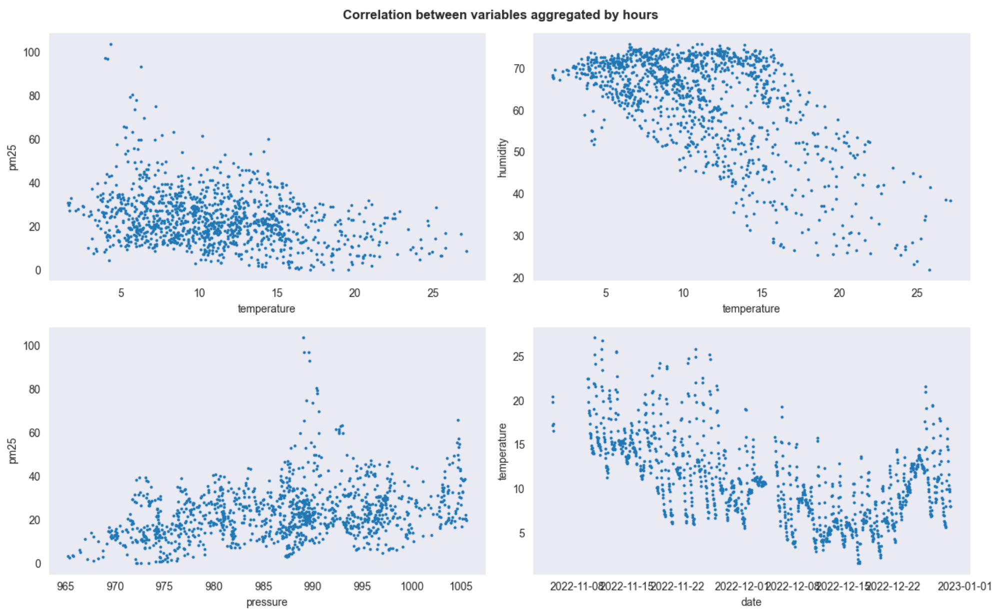


Figure 3.15: The scatter-plot between two variables with data aggregated by hours.

The low expressiveness of the data in Figure 3.15 allows us to image the diagonal on which the point cloud represented between the particulate and pressure variables is distributed; or again, the anti-diagonal on which particulate and temperature coexist together.

3.2.3 Comparison with ARPA dataset

The ARPA dataset is the one used as a reference to perform the calibration of our sensors. In this section we would like to find out how the sensors calibrated up to before the study conducted in this thesis perform compared to the data measured by ARPA.
We use the one with identifier 20 and 21 as the reference board: it is preferred to add the sensors from different boards to compare the results against the reference station.
The only preprocessing steps we perform are inherent in removing duplicates within the measurements committed by our sensors and grouping them into hours since

the data from ARPA are hourly.

The ARPA data used for this analysis were downloaded from their official web portal and consisted of: temperature data from the Royal Gardens station in Turin in 2022 and 2023, air particulate data for all of 2022 and 2023 from the reference station installed in Turin (in Via Rubino).
We show the trend of particulate air levels measured by ARPA and the sensors installed in board 20 and 21, during a specific observation period.

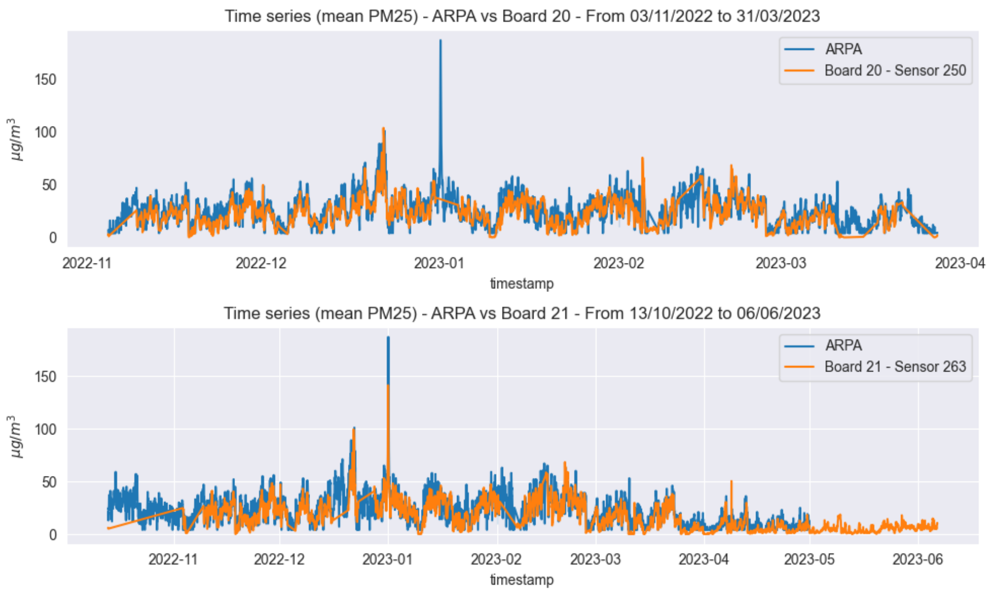


Figure 3.16: The $PM_{2.5}$ values collected by ARPA and one sensor in board 20 (on the top) and board 21 (on the bottom).

The measurements produced by the sensors in the respective boards are similar to those measured by the ARPA station. Their correlation (Pearson) is around 79% for board 20 and 84% for board 21.
One of the reasons why this high level of correlation might be even higher is certainly the lack of some sensor values in board 20 during the period in question. Indeed, some "holes" can be seen in the timeseries above (Fig.3.16). In the case of the sensor in board 21, there are some missing values at the beginning of the observation period.
These missing data from our boards are due to a momentary shutdown of these boards for testing issues or component changes. Nevertheless, the calibration work done so far [1] shows promise given the high correlation between the measurements.

Temperature, of which no calibration has ever been done, also shows a 92 percent

Pearson correlation with that measured by ARPA in the same observation period (Fig.3.17).

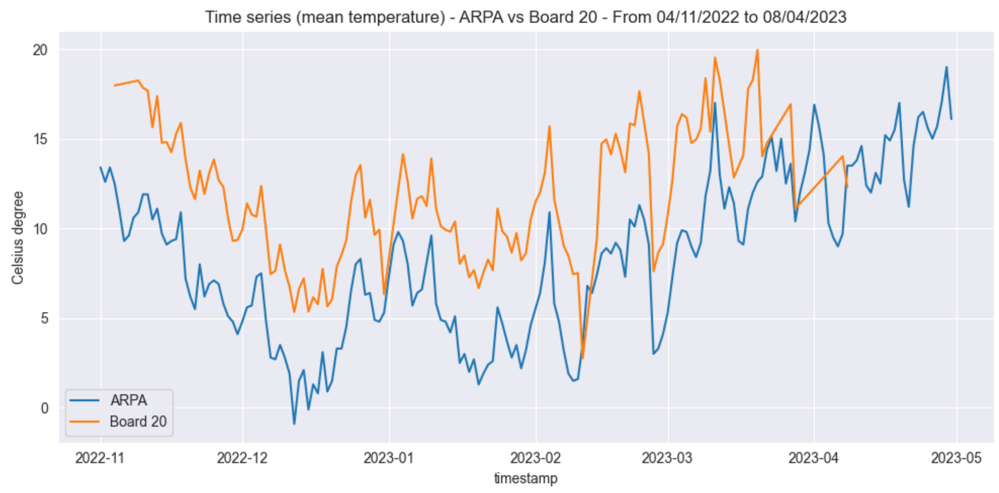


Figure 3.17: The temperature values collected by ARPA and the sensor in board 20.

Having thus looked at the types of graphs created and analyzed in this phase of the project, the highlights that emerged from doing this work on all the sensors on boards 20, 21, 22, 25, 29 and 31 are listed. Recall that there are four fine dust detection sensors for each board.

- **Board 20:**
  - The dataset is incomplete since data points are only gathered through the end of March; the dataset expires at the beginning of April.
  - The sensor with ID 258 performed terribly; it appears that there was a hardware malfunction (i.e., incorrect power levels).
  - The trend compared to ARPA data is quite similar (mean of 74% Pearson correlation); the sensor 258 is not considered in this comparison (indeed it has a -0.09% of correlation); the mid-high correlation level can be explained by the scarcity of the data points in this board.
- **Board 21:**
  - A complete dataset (until the beginning of June) is available on this board.
  - The sensor with ID 267 contains an anomaly at the end of the period: at the mid of May it seems to be switched off, indeed the values collected by this sensor are always equal to 0 from 14/05/2023 to 25/05/2023.

  Furthermore, it experienced completely incorrect PM25 levels around the end of May.
  - The trend compared to ARPA data is quite similar (mean of 80% Pearson correlation) except for the sensor 267 in which it is 62%.

- **Board 22:**
  - The board appears to have been powered off for the bulk of the experiment, hence the dataset is incomplete.
  - It's not possible to use this board for further analysis.

- **Board 25:**
  - The board has a complete dataset (until the begging of June).
  - Throughout the trial, no abnormalities are detected by any of the sensors when collecting data.
  - The trend compared to ARPA data is very similar (mean of 82% Pearson correlation).

- **Board 29:**
  - It has a complete dataset (until the begging of June).
  - During the mid of November 2022, the entire board seems to have been switched off (there are no data points in that period).
  - In the beginning of May, Sensor 367 encountered completely incorrect readings (an excessively high data point).
  - The trend compared to ARPA data is very similar (mean of 81% Pearson correlation).

- **Board 31:**
  - It has a complete dataset (until the begging of June).
  - The entire board appeared to have been turned off in the middle of November 2022 (much as the board with identification number 29); same phenomenon also happened at the start of January 2023.
  - The trend compared to ARPA data is very similar (mean of 81% Pearson correlation).

3.2.4 Building the final dataset

After the previous analyses, we came to be able to draw conclusions and construct what will be the final dataset to be used in the remaining work.
Given the goodness of the distributions and data collected from the sensors in the time range from November 2022 to June, we chose not to totally disrupt the dataset by applying data preprocessing and features engineering techniques; this also leads us to observe, in the next chapter, the results obtained from complex artificial intelligence models such as Feed-Forward Neural Network, LSTM and QML models.
This choice may lead to a loss of accuracy in the final calibration which is justified by giving emphasis to the real essence of this thesis that is the comparative study between the current ML models used and the more powerful and complex ones we will present later.

We therefore show the only data preprocessing steps that were applied to the dataset:

1. **Removal of duplicate values:** some sensors within the boards have duplicate values due to delays in transmitting them to the station's own on-board computer. Considering that the sensors measure and transmit data almost every second, some duplicates are permissible and therefore we remove them, also going to streamline the dataset.

2. **Filter data for a specific time range:** we want to work on the exact period of time that the sensors were on (almost all of them), so we cut out all observations that were measured outside this time frame.

3. **Grouping by minutes/hours:** since the dataset of measurements collected and distributed by ARPA has hourly granularity, we prefer to increase granularity to our sensor data as well. We therefore change from a measurement every second to a measurement every minute/hour, applying an arithmetic mean to reduce the dataset.

4. **Removal of null values:** since, for the entire time range used, some boards were not always on, aggregating the data by minutes or hours may have empty (or null) values. We therefore decide to remove them since they do not represent additional information for the purpose of our analysis.

These four preliminary operations listed above were reproduced serially for each sensor on each of the six boards used.

We therefore end up with several pre-processed datasets, one for each sensor on each board. What we are looking for, however, is a single dataset actualized to

best accomplish the purpose of this thesis.
To do this therefore, we have chosen to take the median of all the sensors from all the boards (divided by sensor type) and create a single dataset with minute granularity. The same thing was done to produce a single dataset but with hourly granularity. This creates a single time series that has no null values because, thanks to the median, the missed measurements of some of the off sensors were covered by those made by the active sensors.
With the models selected and discussed in Section 3.3, this thesis focuses on optimizing them within this single dataset with the purpose of calibrating the sensors. The calibration will be generic i.e., it is based on the median of all sensor measurements. In a future study, the quality of the calibration can be investigated through the same models discussed in this thesis but applied to each individual sensor in order to have an ad-hoc calibration on it.

For illustration purposes, the snipped of Python code used to create the final dataset is shown.

```python
import pandas as pd
import numpy as np
```

There are two main libraries for performing this task: numpy (for applying the median on matrices) and pandas (for reading and manipulating data from external sources such as csv files).

The *prepare_generic_dataframe* function is used to apply the preprocessing steps to each sensor on each board (regardless of sensor type).

```python
def prepare_generic_dataframe(file_name: str) -> pd.DataFrame:
    df              = pd.read_csv(file_name)
    df.timestamp    = pd.to_datetime(df.timestamp)
    df.drop_duplicates(inplace=True)
    df.sort_values(by='timestamp', inplace=True)
    df = df.loc[(df['timestamp'] >= START_DATE_BOARD) & (df['
timestamp'] <= END_DATE_BOARD)]
    df = df.groupby(pd.Grouper(key='timestamp', freq='min')).mean
().reset_index()
    df.dropna(inplace=True)
    return df
```

The sensor's observations are aggregated by minutes and all the null values are removed.
The following code's snippet shows the uploading and preprocessing phase of all

the sensors, divided by typology (i.e.: particulate matter, temperature, humidity and pressure).

```python
dataframes_pm25 = []
dataframes_temp = []
dataframes_pres = []
dataframes_rh   = []
for folder_name in tqdm(os.listdir(PM25_DIRECTORY), desc='Analyzing
    folders'):
     folder = os.path.join(PM25_DIRECTORY, folder_name)
     if os.path.isdir(folder) and len(folder.split('/')) > 3 and
    folder.split('/')[3] in PM2_MAP:
          files = PM2_MAP[folder.split('/')[3]]
          for file_name in files:
               file = os.path.join(folder, file_name)
               if os.path.isfile(file) and file.endswith(".csv"):
                    df = prepare_generic_dataframe(file)
                    if folder_name.endswith("_temp"):
                         dataframes_temp.append(df)
                    elif folder_name.endswith("_pres"):
                         dataframes_pres.append(df)
                    elif folder_name.endswith("_rh"):
                         dataframes_rh.append(df)
                    else:
                         dataframes_pm25.append(df)
```

After that, we are ready to build the final dataset by merging all the cleaned sensor's data through median. The structure that this dataset is to have is declared, and the variables that will serve as unique datasets for each sensor type are instantiated.

```python
df_final                = pd.DataFrame(columns=['timestamp', 'pm25',
    'temp', 'pres', 'rh'])
df_final['timestamp']   = pd.date_range(start=START_DATE_BOARD, end=
    END_DATE_BOARD, freq='min')
pm25_series             = []
temp_series             = []
pres_series             = []
rh_series               = []
```

Now we go on to fill these series with the median of all sensors (divided by type). This is done by running the entire time series minute by minute and collecting data from each sensor (if any).

```python
for pit in tqdm(df_final['timestamp'], desc='Building unique dataset'):
    # PM2.5
    pm25_values = []
    for df in dataframes_pm25:
        value = df[df.timestamp == pit]['data'].values
        if len(value) > 0:
            pm25_values.append(value[0])
    pm25_series.append(np.median(pm25_values) if len(pm25_values) > 0 else None)
    # Temperature
    temp_values = []
    for df in dataframes_temp:
        value = df[df.timestamp == pit]['data'].values
        if len(value) > 0:
            temp_values.append(value[0])
    temp_series.append(np.median(temp_values) if len(temp_values) > 0 else None)
    # Pressure
    pres_values = []
    for df in dataframes_pres:
        value = df[df.timestamp == pit]['data'].values
        if len(value) > 0:
            pres_values.append(value[0])
    pres_series.append(np.median(pres_values) if len(pres_values) > 0 else None)
    # Humidity
    rh_values = []
    for df in dataframes_rh:
        value = df[df.timestamp == pit]['data'].values
        if len(value) > 0:
            rh_values.append(value[0])
    rh_series.append(np.median(rh_values) if len(rh_values) > 0 else None)
```

The time complexity of this algorithm is $\mathcal{O}(n \cdot d)$ where n are all measurements present in the specified date range and d are the number of all sensors used to conduct this study (about 36 sensors involved). It is aware to admit that this complexity is high and that it is possible to reduce it, but for the purpose of this thesis it is not necessary.

3.3 The selected models

After explaining in detail the purpose of this project and the steps in constructing the dataset that will be used, this section explains the selected artificial intelligence models. All the reasoning behind this choice is well explained in the following paragraphs in order to provide a complete picture of the work carried out.

The purpose of this experimental work is to use complex models for fine dust sensor calibration. The term "complex" is meant in relation to the work done in previous studies completed by the same research laboratory ([1] [17] [15]). In fact, in these previous works, the statistical models used are a Random Regressor Forest and a Multivariate Linear Regressor. Both models that, given their architecture, we can fall within the set of Machine Learning models.
Moreover, in addition to the use of more complex models, we want to make a comparison with the equivalent of the quantum world to observe how a decisive paradigm shift (related to quantum computing) can impact in a real-word problem.

Given the above considerations, the models that were chosen for this thesis are as follows:

1. **Feed-Forward Neural Network** (FFNN): We discussed about this kind of model in depth in sec.2.2.1.
2. **Long short-term memory** (LSTM): We discussed about this kind of model in depth in sec.2.2.5.
3. **Variational Quantum Regressor** (VQR): This model is based on the same principle as the variational quantum circuits discussed in sec.2.6.2 but with the major difference that the variational params are used to learn a regression and not a classification.
4. **Quantum LSTM** (QLSTM): This model is based on the study conducted in this research [18] and involves creating a variational circuit that simulates the behavior of an LSTM cell but using qubits.

For each of the models mentioned, it is explained in detail why it was chosen and how it was designed in order to perform a calibration.

3.3.1 FFNN model

The mathematical operation of this type of model has been extensively explained in section 2.2.1.
The reason this type of model was chosen is because it represents the classical concept of an "Artificial Neural Network" found in Deep Learning. The research lab's interest in using this type of network was the main reason for this thesis to study its adoption. In addition to the possibility of being able to choose from different formulas for calculating the loss function, the type of optimizer to be applied on the weights of individual neurons at the end of backpropagation, and

other configuration parameters, the advantage of using this model is the freedom in the choice of architecture.
It is possible to compose different neural network architectures by changing the numbers of neurons and hidden layers involved. Figure 3.18 shows the standard

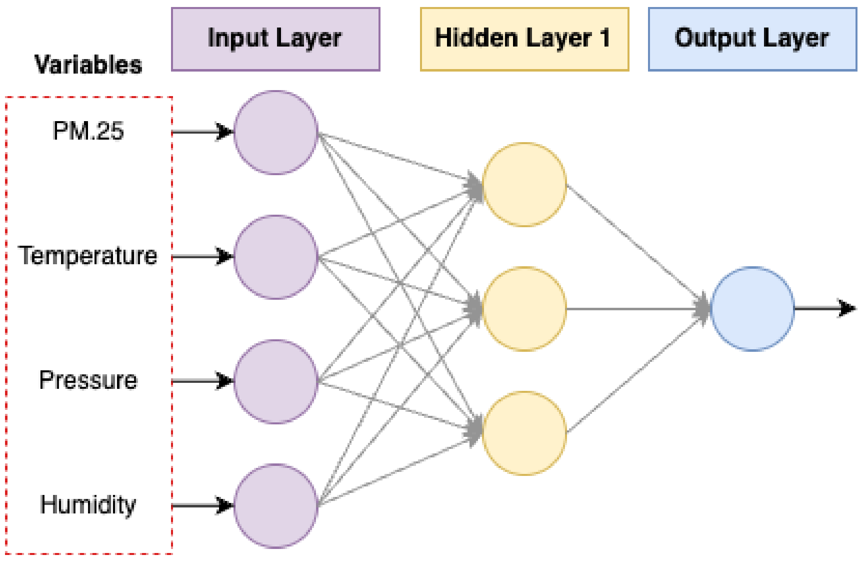


Figure 3.18: The standard architecture of a FFNN.

architecture of an FFNN. The number of input and output neurons are predetermined a priori according to the nature of the problem to be addressed.
Example: in output we will always have only one neuron since our problem involves predicting the value of particulate matter in the air as close as possible to that of ARPA, using also the values produced by the other sensors, all at a specific date in time.
As for the input layer, we will see next the strategies chosen in order to model the data in the dataset at hand.
In the central part of this architecture reside the hidden layers: they can be from one to as many as desired, and each of them can hold an arbitrary number of neurons. There is a need to keep in mind that, depending on the nature of the problem being addressed, a corpus architecture could lead to well-known problems such as gradient vanishing or overfitting. The study of the impact caused by architecture variation on the final calibration will be addressed through hyperparameter tuning, which will be discussed in Chapter 4.1.

We show the implementation of the Python class representing an FFNN used in the thesis project.

```python
import torch.nn as nn
class MyNeuralNetwork(nn.Module):
    def __init__(self, input_size: int, output_size: int, hidden_size: int, hidden_size_2: int = 90, hidden_size_3: int = 30):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, output_size),
        )
        if hidden_size_2 is not None and hidden_size_3 is not None:
            self.net = nn.Sequential(
                nn.Linear(input_size, hidden_size),
                nn.ReLU(),
                nn.Linear(hidden_size, hidden_size_2),
                nn.ReLU(),
                nn.Linear(hidden_size_2, hidden_size_3),
                nn.ReLU(),
                nn.Linear(hidden_size_3, output_size),
            )
        elif hidden_size_2 is not None:
            self.net = nn.Sequential(
                nn.Linear(input_size, hidden_size),
                nn.ReLU(),
                nn.Linear(hidden_size, hidden_size_2),
                nn.ReLU(),
                nn.Linear(hidden_size_2, output_size),
            )
    def forward(self, x):
        out = self.net(x)
        return out
```

As can be seen from the code snippet above, the neural network requires the size of the input, output and at least one hidden layer as mandatory parameters. It remains optional to declare the presence of a second and third hidden layer.
Through this implementation, it is possible to easily manipulate the final architecture that the neural network will have, simply by choosing which parameters in the class constructor to pass and which not. Forward propagation is implemented in the `forward` method of the snippet.
This method is responsible for taking the input tensor and passing it through the network that is implemented by the `torch.nnn.Sequential` class. This class takes care of executing, in a sequential manner, the sequence of operations to which the input tensor must undergo to get to the output layer.

Within each neuron there is a function that transforms the linear summation between weights, input and bias: this function is called the "activation" function.

Neural networks rely heavily on the activation function of individual neurons, which adds non-linearity to the model. The network needs this non-linearity in order to learn and simulate intricate correlations and patterns in the data that linear models are unable to capture.
The chosen activation function is one of the most popular for manipulating nonlinear problems of all types: the Rectified Linear Unit (ReLU).

$$f(x) = max(0, x) \tag{3.1}$$

Positive inputs are sent through directly by the ReLU function, whereas negative inputs are clipped to zero. This indicates that the output is equal to the input for any positive input and equal to zero for any negative input. The model gains non-linearity from this straightforward process without changing the magnitude of positive input values.
ReLU's first benefit is its computational efficiency, which comes from its straightforward thresholding at zero. Because of its efficiency, it may be computed more quickly than other activation functions like sigmoid or tanh, particularly when deep networks are being trained.
The second is that ReLU does not saturate positively, in contrast to sigmoid or tanh functions. This property mitigates the vanishing gradient issue, which occurs when gradients are too tiny for the network to process information efficiently.

To conclude the design choices made on this model, it is important to specify that two types of feed-forward neural networks were created:

- **ANN#1**: This type of neural network works with the aggregated dataset to the minute.
- **ANN#2**: This type of neural network works with the aggregated dataset per hour. Its structure is identical to that in Figure 2 net of successive hidden layers that may vary.

ANN#2 network was chosen in order to make a direct comparison with the quantum equivalent (Variational Quantum Regressor) that we will see in the next sub-sections.

The dataset that presents sensor data aggregated per minute can bring the benefit of finer granularity and therefore more detailed learning by the ANN#1 network. But how can sensors be calibrated with this dataset if ARPA data is only available at an hourly granularity? We need a strategy to take advantage of the aggregated data per minute and compare it with that of ARPA.
Figure 3.19 schematizes the operation of this type of network.
From Figure 3.19 it can immediately be seen that the output layer contains a single neuron, and therefore the value produced by the network will be a single

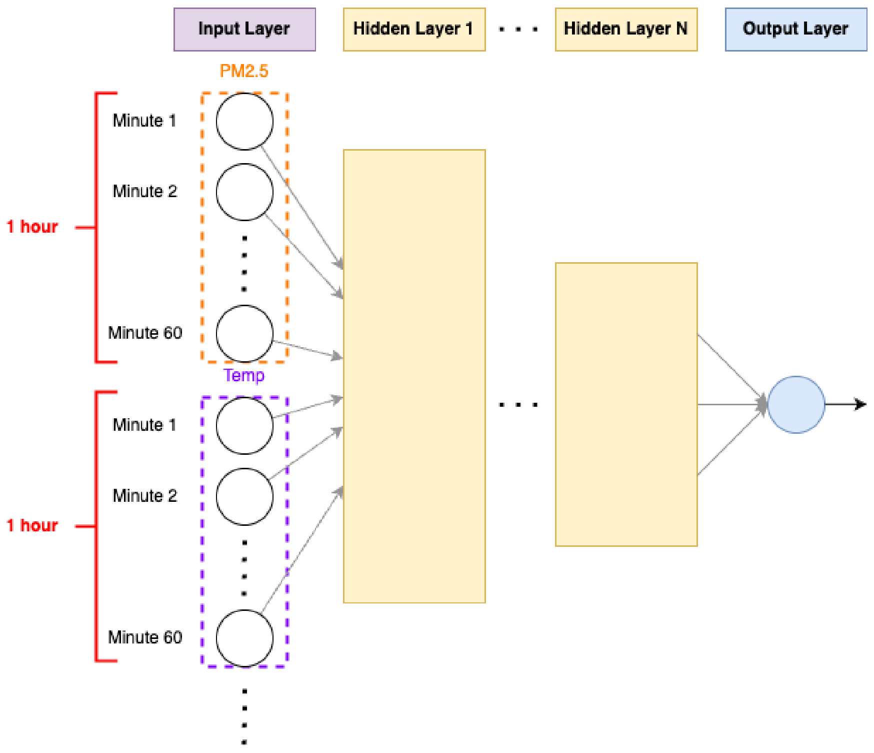


Figure 3.19: The strategy used to use data aggregated per minute to calibrate sensors based on ARPA hourly data through ANN#1 network.

number. This number represents the value of particulate matter in the air present at a given moment in time, also taking into account other variables such as temperature, pressure and humidity.
This value will then be compared with the value measured by ARPA, and the whole backpropagation part is based on the error made by the network in predicting this value. As we already know, the ARPA dataset contains only measurements made with an hourly frequency, so the prediction of the ANN#1 network, to be compared, must also refer in hourly terms.

The chosen strategy, therefore, involves using 60 variable neurons: one for each minute that makes up 1 hour.
To simplify the explanation, taking only the $PM_{2.5}$ variable as an example. The sixty measurements taken in one hour are sent as input to the network, which will then process them in order to produce a single numerical value. The output of the network is then compared, through a specific loss function, with the output of the reference hour measured by ARPA.
The next chapters of this paper will show the calibration result obtained with this

strategy and diversified into two variants: the first takes as input only the $PM_{2.5}$ variable (so the architecture has only sixty neurons in the input layer), while the second uses the other variables in this problem as well.

3.3.2 LSTM model

Again, the technical explanation on how this type of model works is given in Section 2.2.5.
Given the strong presence of the temporal dimension within this specific problem, we chose to use a model that can be a state-of-the-art in learning along a time axis. From this point of reflection, it is impossible not to think of a model such as Long short-term memory (LSTM).
Due to its recurrent nature (it belongs to the category of neural networks referred to as RNN - "Recurrent neural network"), this model is capable of maintaining relationships between data and extracting patterns by paying attention to everything that happened in the "previous steps."
LSTM is thus able to, by taking into account the history of a variable's trend over time, predict what value it may take on in the near future.

The first step to be implemented to bring a model like this to calibration concerns the dataset. To solve a generic calibration problem, our model should not predict what will happen in the next hour, but what happens in the present hour, in order to then be able to compare it with a reference measurement (called ground truth). So the dataset on which this model was trained involves making a merge between data from the reference station (ARPA) and our board (the median of all sensors from all boards, as explained in Section 3.2.4). Then, only the dimension representing the air particulate measured by ARPA was slid forward by 1 hour.
This choice allows LSTM to take into account a certain number of values from previous hours in order to predict the next one: but, having shifted the ARPA values forward by 1 hour, the model's prediction will be compared with the current reference hour and not the next one.
In summary: you are predicting the present and not the future. This is why we no longer speak of prediction but of calibration.

Another important aspect when working with this type of model is the order of the data within the final dataset. To use LSTM, one must present the data in ascending order according to a time axis-this way the model can continue to learn by referring to what it has "seen" previously.
This brings with it a limitation, namely that of not being able to shuffle the data during the training phase, and thus increasing the likelihood of creating a model that possesses a bias on the historical period on which it was trained. To overcome

this, one would need to train such a model on a time-series long enough to include all the types of samples present within our observed problem.

In addition to the number of features that will make up the hidden state of LSTM, another parameter that can be chosen a priori (called hyperparameter) is the one identified with the letter `T`.
The parameter `T` represents the number of previous time instances to be taken into account to carry out the prediction in the next time instance. Therefore, if I want to predict the value of $PM_{2.5}$ at time t, the network will use as input the values of $PM_{2.5}$ ranging from time $(t-1-T)$ to $(t-1)$.
Let's show an example. Let us consider the value of the variable t equal to `'01-01-2023 12:00:00'` and the parameter `T` equal to 5 (means I take the previous 5 time units as input). The time range that LSTM will use to make its prediction is:

$$from = (t-1-T) = \text{'01-01-2023 06:00:00'} \tag{3.2}$$

$$to = (t-1) = \text{'01-01-2023 11:00:00'} \tag{3.3}$$

Now that the reason for choosing LSTM and how it was used within the project has been explained, let us move on to see its implementation in Python code.

```python
import torch.nn as nn
class MyLSTM(nn.Module):
    def __init__(self, input_dim, hidden_dim, layer_dim, output_dim):
        super(MyLSTM, self).__init__()
        self._device=torch.device("cuda" if torch.cuda.is_available()
          else "cpu")
        self.M = hidden_dim
        self.L = layer_dim
        self.rnn = nn.LSTM(
            input_size=input_dim,
            hidden_size=hidden_dim,
            num_layers=layer_dim,
            batch_first=True)
        # batch_first to have (batch_dim, seq_dim, feature_dim)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, X):
        # initial hidden state and cell state
        h0 = torch.zeros(self.L, X.size(0), self.M).to(self._device)
        c0 = torch.zeros(self.L, X.size(0), self.M).to(self._device)
        # compute the forward propagation
        out, (hn, cn) = self.rnn(X, (h0.detach(), c0.detach()))
        # h(T) at the final time step
        out = self.fc(out[:, -1, :])
        return out
```

As input parameters, the class representing the LSTM model receives the size of the input, the number of features in the hidden state, and the number of layers. The actual implementation of the LSTM cell operation is provided by the `torch.nn.LSTM` class provided by Pytorch.
Pytorch allows the implementation of a multilayer LSTM: in this type of model, the input $x_t^{(l)}$ of the l-th layer ($l \geq 2$) is the hidden state $h_t^{(l-1)}$ of the previous layer muliplied by dropout $\delta_t^{(l-1)}$ where each $\delta_t^{(l-1)}$ is a Bernoulli random variable which is 0 with probability equals to a specified parameter which is called dropout. By default, this probability is zero, which means that no dropout is done in any of the layers that make up the multilayer LSTM.
The `num_layers` parameter is the one that decides whether to constitute a multilayer LSTM or not: by default it is set to one, but in our project we left it fixed at two. In this way, we are stacking two LSTMs together to form a stacked LSTM, with the second LSTM taking in outputs of the first LSTM and computing the final results.

LSTM layers generate outputs with a specific hidden size, determined by the number of hidden units in the LSTM. However, the desired output size for our task (e.g., the value of $PM_{2.5}$) often does not match the LSTM's hidden size. A fully connected layer is used to transform the dimensionality of the LSTM output to match the desired output size.
The fully connected layer, which offers dimensionality matching, learning capability, feature integration, and customisation flexibility, is an essential part that enables the LSTM model to interface its intricate, high-dimensional hidden representations with the particular requirements of different tasks in an efficient manner.

To conclude, forward propagation within the network involves initializing two tensors that will represent the hidden state and the cell state at time $t = 0$.
Then these two tensors are passed together with the input into the network and, at the end of the computation, they will be extracted: the output features from the last LSTM layer for each t (`out` variable), the last hidden state and cell state calculated (`hn`, `cn` variables).

3.3.3 VQR model

The variational quantum regression (VQR) model is a VQC-based model with the purpose of learning to predict a number and not a class of membership (its detailed explanation can be found in Section 2.6.2).
This model can be compared to a simple neural network, in fact it is also referred to

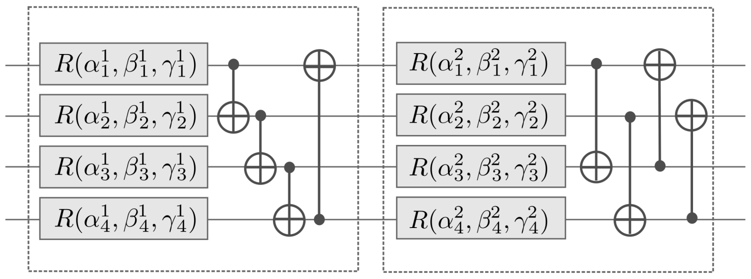
as a Quantum Neural Network. This model was chosen to propose the first comparison between the classical and quantum worlds; the model defined as "ANN#1" in Section 3.3.1 and this model were designed to achieve the same result, the difference being that one is based on the bits of classical computing while the other harnesses the power of qubits through special circuits.

We are talking about the first artificial neural network model created simply to fit the type of dataset used: in "ANN#1" the dataset has an hourly granularity, and the entire network (or circuit in this case) assumes an architecture suitable for working with this data.
The picture of the architecture is similar to the one in Figure 3.19: the major difference is that there are no neurons but embedding and ansatz layers, repeated in different ways and at will.

Before further detailing the configuration used for this model, it is important to explain how such quantum circuits were simulated in order to produce the results of this thesis. Not having a high need for qubits to be used and to stay with a focus on software rather than hardware, no emulation of that model was done. This implies that a classical computer (manipulating bits) has been used to simulate the behavior of a quantum machine, with all the disadvantages that it brings with it (i.e. the slowness or failure to submit to computational errors that can occur on an authentic machine).

Taking awareness of this, the model consists of two main parts: the data embedding layer and the quantum circuit layer (called ansatz). The former is used to transform the data from the dataset into a corresponding quantum state (this is called data embedding). The second, on the other hand, is the circuit that will perform operations on the data represented by qubits.
The quantum operations that are performed depend on the type of circuit that makes up the ansatz. Two main ansatz were used in this work: the Strongly Entangling Layer and the Basic Entangler Layer.
The Strongly Entangling Layers have the potential to provide better expressivity and problem-solving skills due to their complicated, parameterized quantum gates that may produce highly entangled states. Because of this, they are very helpful in solving challenging quantum puzzles where substantial entanglement is essential. However, because of their complexity and wider parameter space, multiple layers may make optimisation more difficult and result in longer training times and a higher vulnerability to local minima.
Conversely, Basic Entangler Layers provide a more direct method with less complicated entanglement processes. When variational quantum circuits are trained, their simplicity can lead to simpler optimisation and quicker convergence, increasing

their applicability and using less processing power. Nevertheless, as they might not produce highly entangled states as quickly as Strongly Entangling Layers, their expressivity and capacity to handle more challenging quantum issues may be limited by their lesser complexity.
For the constructed VQR model, it was chosen to use the Strongly Entangling Layers as ansatz. As you can see from Fig. 3.20, the repeated units or "layers"



$R(\alpha_1^1, \beta_1^1, \gamma_1^1)$
$R(\alpha_2^1, \beta_2^1, \gamma_2^1)$
$R(\alpha_3^1, \beta_3^1, \gamma_3^1)$
$R(\alpha_4^1, \beta_4^1, \gamma_4^1)$
$R(\alpha_1^2, \beta_1^2, \gamma_1^2)$
$R(\alpha_2^2, \beta_2^2, \gamma_2^2)$
$R(\alpha_3^2, \beta_3^2, \gamma_3^2)$
$R(\alpha_4^2, \beta_4^2, \gamma_4^2)$

Figure 3.20: The Strongly Entangling Layers with 2 layers, from Pennylane website.

that are applied to the qubits one after the other form the basis of the Strongly Entangling Layers ansatz. The purpose of each layer is to encourage significant entanglement between qubits.
Applying parameterized single-qubit rotation gates (such as RX, RY, and RZ) to each qubit is the initial step in each layer. Angles that will be optimised to minimise a selected cost function throughout the variational algorithm's execution parameterize these rotations. A sequence of entangling gates (like CNOT or CZ gates) are applied across the qubits after the single-qubit rotations. Strong entanglement is ensured by the pattern of these entangling processes, which usually covers all pairs of qubits. While the exact pattern may differ, it often combines long-range interactions with nearest-neighbor entanglements to create a dense network of entanglements across the circuit.
To conclude, a traditional optimisation procedure is used to optimise the parameters—that is, the angles in the rotation gates—in order to minimise a cost function. This function's purpose is to gauge how well the quantum circuit performs in resolving the particular issue at hand.

Concerning the emedding layer, one of the most famous was used, namely Angle Embedding. Angle Embedding is a technique that prepares a quantum state that reflects the classical data by encoding it into the parameters (angles) of quantum gates in variational quantum circuits. The fundamental concept is directly mapping

the angles of rotation gates applied to the qubits in a quantum circuit to the classical data points. For quantum machine learning and other quantum algorithms, this method offers a simple and effective way to integrate classical information into a quantum state.
The rotation performed by this type of embedding works like this: the rotation angle θ represents the classical information that is to be transformed, while the rotation is performed by one of the Pauli matrices (X, Y, Z) on one of the three possible spatial axes. The starting quantum state that is used is $|0\rangle$. In doing so, the classical information will be transformed into one of three possible rotations with degree θ, creating a new starting quantum state on which the ansatz will operate.
For instance, applying a rotation $RY(\theta)$ gate to a qubit initially in the state $|0\rangle$ transforms it into a superposition state:

$$RY(\theta)\,|0\rangle = cos(\frac{\theta}{2})\,|0\rangle + sin(\frac{\theta}{2})\,|1\rangle \tag{3.4}$$

This transformation encodes the classical information (represented by θ) into the amplitude and phase of the quantum state, enabling the quantum circuit to process classical data in a quantum framework.

Two types of circuits representing a VQR model were proposed in this thesis work:

- **Linear:** This architecture is as seen in the preceding paragraphs that is, a single AngleEmbedding layer followed by several Strongly Entangling layers representing the ansatz.
- **Non linear:** From [19] it is possible to experiment with different strategies by which data are encoded within the circuit and how these different embeddings may impact in the expressive power of the data within the final model. The nonlinearity is given by alternating different simple embedding gates with the corresponding ansatz.

From Figures 3.21 and 3.22 each vertical blue block representing an ansatz, enclosed within it is the Strongly Entangling Layer circuit shown in Figure 3.20. In the case of the nonlinear circuit, there is only one layer within the ansatz, while in the case of the linear circuit, the number of layers used becomes a parameter on which to fine tune to find the one suitable for the problem to be solved.
In Figure 3.22, something similar to the Angle Embedding seen earlier is used as the data encoding layer. It is referred to as a rotation on the spatial X-axis in which, however, the angle of rotation is the arcotangent function of the input data, multiplied by two ($RX(\theta)$ with $\theta = 2\arctan(x)$, where x is the input data).

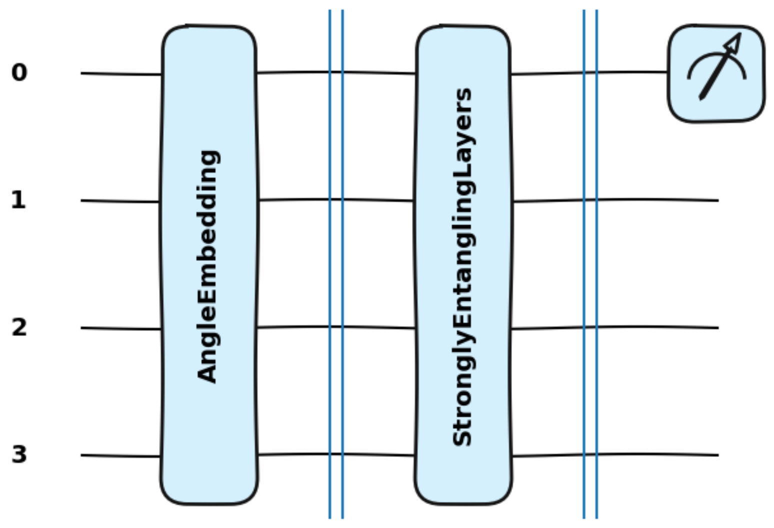


Figure 3.21: The Linear VQR circuit with 4 qubits

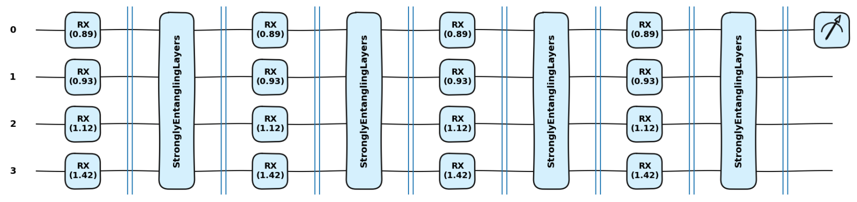


Figure 3.22: The Non Linear VQR circuit with 4 qubits, based on [19]

To conclude, in order to use this type of circuit (especially the nonlinear one), it is necessary to apply preprocessing on the input data. In fact, using the arcotangent function, it is mandatory to have the data in a range between $[-1, +1]$ otherwise the function belongs to a nonexistent domain.
Furthermore, we can explain the fact that, using a dataset with only four features ($PM_{2.5}$, temperature, humidity and pressure), it was decided to use 4 qubits within the model to represent the input data; in this way it is possible to make a fair comparison with the equivalent classical neural network which also uses four input features.

We show the Python class which, together with the open source Pennylane framework, creates the class that represents our linear Variational Quantum Regressor:

```python
import torch
import pennylane as qml
from pennylane import numpy as np
from torch import nn

class VQRLinearModel(nn.Module):
    def __init__(self, n_qubit: int, layers: int = 1, duplicate_qubits: bool = False) -> None:
        super().__init__()
        self.n_qubit = n_qubit * 2 if duplicate_qubits else n_qubit
        # initialize thetas (or weights) of NN
        shape = qml.StronglyEntanglingLayers.shape(n_layers=layers, n_wires=self.n_qubit)
        initial_weights = np.pi * np.random.random(shape, requires_grad=True)
        self.weights = nn.Parameter(torch.from_numpy(initial_weights), requires_grad=True)
        # initialize bias of NN
        self.bias = nn.Parameter(torch.from_numpy(np.zeros(1)), requires_grad=True)
    def encoder(self, x):
        qml.AngleEmbedding(x, wires=range(self.n_qubit))
    def layer(self, weights):
        qml.StronglyEntanglingLayers(weights, wires=range(self.n_qubit))
    def circuit(self, x):
        self.encoder(x)
        qml.Barrier(wires=range(self.n_qubit), only_visual=True)
        self.layer(self.weights)
        qml.Barrier(wires=range(self.n_qubit), only_visual=True)
        return qml.expval(qml.PauliZ(wires=0))
    def forward(self, X):
        # define the characteristics of the device
        dev = qml.device("default.qubit", wires=self.n_qubit)
        vqc = qml.QNode(self.circuit, dev, interface="torch")
        res = []
        for x in X:
            res.append(vqc(x) + self.bias)
        res = torch.stack(res)
        return res
```

This class shows the classic methods also seen for previous models: there is a constructor method that takes care of initializing the model by taking as input the number of qubits needed and how many layers to use (by default only one); then

there is a `forward` method that tells the forward propagation inside the quantum circuit. Finally we have the `encode` and `circuit` methods which respectively tell how the embedding layer and the ansatz are made.
As previously explained, the quantum architecture used with Pennylane is defined as `default.qubit`: this backend represents the use of a classical computer to simulate the quantum behavior of the circuit.

3.3.4 QLSTM model

This last type of model was implemented with the aim of having a comparison with the equivalent model in the classical world (i.e. LSTM). Let's start by stating that, in order to make the comparison fair, the type of model training and the dataset used are the same for both models.
In this way we can compare, in the next chapter, the results obtained from both models and have the opportunity to discuss them.

An inventive method for implementing an LSTM cell (sec. 2.2.5) inside a quantum framework is the Quantum Long Short-Term Memory (QLSTM).
QLSTMs mimic the internal architecture and computational mechanisms of an LSTM cell by means of quantum circuits, in contrast to conventional LSTMs, which function within classical computing paradigms. This quantum-based approach is intended to take advantage of the potential speed and efficiency benefits that come with quantum computing, especially when working on large-scale data processing tasks.

A QLSTM cell is made up of a few essential parts that are modified for quantum computing while still reflecting the architecture of a traditional LSTM cell:

- **Quantum Gates:** These are used to perform operations analogous to those in classical LSTM cells, including the handling of input, forget, and output gates.
- **Quantum Circuits:** Specifically created to carry out an LSTM cell's activities within a quantum computing framework, quantum circuits are used to recreate the whole LSTM cell.
- **Quantum Encoding:** A suitable quantum encoding scheme is proposed for input data, ensuring that the data can be effectively processed within the quantum circuit.
- **Activation Functions:** The QLSTM incorporates a quantized version of the sigmoid and hyperbolic tangent activation functions, which are crucial to the functioning of standard LSTM cells.

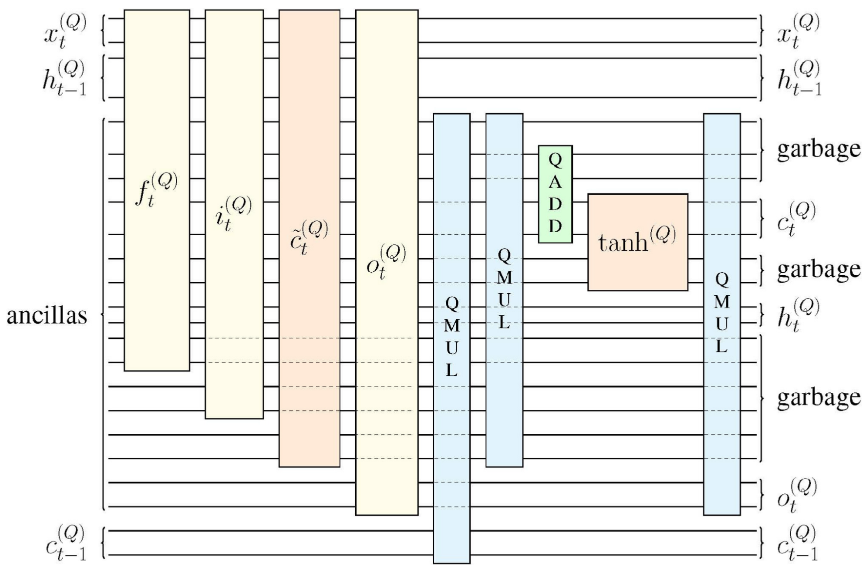
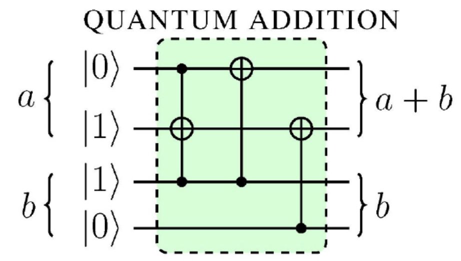


Figure 3.23: Circuit of a QLSTM cell [18]. The QMUL and QADD transformations correspond to quantum multiplication and quantum addition, respectively.

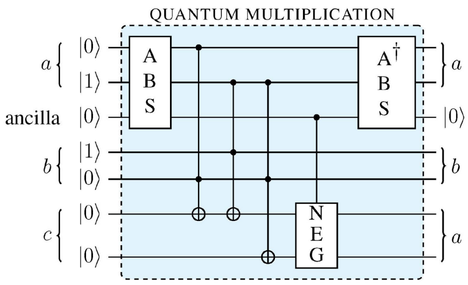
The crucial information encoding stage, in which data is converted into qubits using certain encoding strategies like basis encoding, is where a QLSTM operates. For the quantum circuit to process data effectively, this preparation is essential.
The QLSTM cell then uses specialised quantum circuits to carry out quantum arithmetic operations, such as quantum versions of addition and multiplication. These cleverly constructed procedures are intended to mimic their classical equivalents, yet they may be more efficient due to the special characteristics of quantum physics. Quantum circuits in the QLSTM perform arithmetic operations as well as quantized versions of activation functions like the sigmoid and hyperbolic tangent functions. Deterministic mapping techniques, such as the Product-Of-Sum (POS) approach, are used to do this. These techniques guarantee an exact input-output relationship that is essential to the cell's functioning.

The Quantum Addition (QADD) gate is the bitwise direct addition of one integer onto another; it is the quantum counterpart of conventional addition. The matching qubits of the two integers involved are a prerequisite for this operation to occur. In essence, quantum addition uses the entanglement and superposition concepts from quantum physics to carry out addition operations in a naturally parallel fashion, possibly providing considerable processing gains over traditional addition techniques.
Because managing the multiplication of binary integers in a quantum setting

is necessary, quantum multiplication is a more sophisticated operation than its addition equivalent, and this is what the Quantum Multiplication (QMUL) gate does. To calculate the product of two integers, this gate combines controlled logic gates with quantum arithmetic processes. The procedure ensures that the right sign is applied to the outcome by accounting for the possibility of negative integers and using an ancilla qubit to store the sign of the product. A sequence of Toffoli gates may be used in the multiplication process to simulate the binary multiplication process by building up partial products. Furthermore, auxiliary qubits engaged in the process are disentangled and returned to their starting states via uncomputing procedures.



(a) Quantum addition between two numbers a and b. The highest qubit is the least significant [18].



(b) The design of a quantum multiplication circuit [18].

Figure 3.24: The design of QMUL and QADD circuits, needed inside the QLSTM cell.

The encoding of the data in a starting quantum state is done via a simple Angle Embedding seen previously in Section 3.3.3.
The quantum layers representing the input, forgot, update and output states, typical of an LSTM cell, were implemented through an ansatz circuit whose free parameters are those that the model learns during training. In particular, two types of ansatz were used for each of these four states mentioned: the Strongly Entangler Layers and the Basic Entangler Layers (sec. 3.3.3).
The circuit of a Basic Entangler Layers ansatz is done as follows: The layers in this circuit consisting of one-parameter single-qubit rotations on each qubit, followed by a closed chain or ring of CNOT gates. Every qubit is connected to its neighbour by a ring of CNOT gates, with the last qubit being regarded as a neighbour of the first qubit.

The number of qubits used in this model is equivalent to the number of input features in the dataset. Furthermore, as in the LSTM model, here too there is the concept of hidden layers, used to determine the cell's ability to represent and

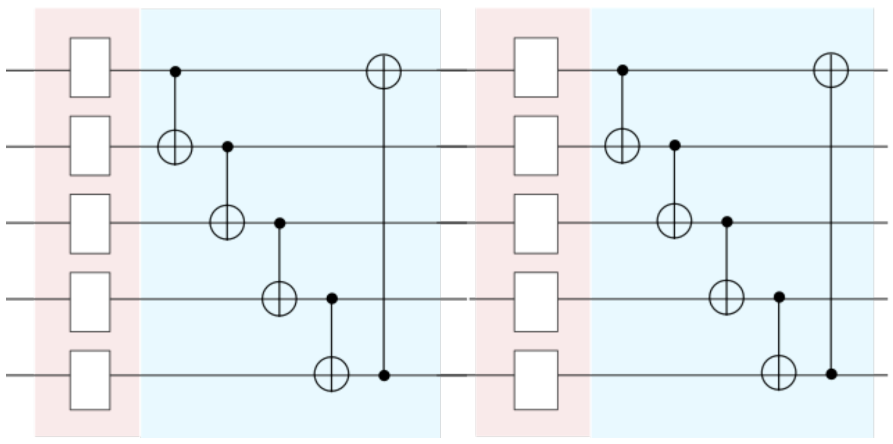


Figure 3.25: The Basic Entangler Layers with 2 layers, from Pennylane website.

maintain information through its hidden states.

The Python class representing this model was implemented using the Pytorch framework and Pennylane. Given the length of the written code, we prefer to refer the reader to the official GitHub repository in order to observe the implementation of the QLSTM class.

3.4 The framework

Having seen what this thesis project consists of, the study of the dataset at hand, the artificial and quantum models involved, it is important to make a final point about the working methodology adopted, dwelling on the framework created to fulfill the objective of this thesis.

The entire code is written in the Python programming language at version 3.9. In fact, the entire project was also tested with Python version 3.8, making the latter also compatible for the purpose of reproducibility of results.
The most important libraries used are the following:

- **Pandas:** To manipulate data structures and perform analysis on the dataset.
- **Pytorch:** To work with tensors, build neural networks and train them.
- **Pennylane**: To build the quantum circuits and carry out the simulations.
- **Tensorboard:** It is the front-end that allows all the results obtained from each training of a model to be displayed via a web page and compared through a simple user interface.

The framework built for carrying out this concept allows for building, training and comparing different models in a simple and standard way. It can be reused for generic problems including training of neural networks and visualization of the results obtained.
The public repository where the source code resides at its most up-to-date version is available on GitHub at the following web address: `https://github.com/lolloberga/quantum_weather_station`.

3.4.1 How it works

To better understand how the built framework works, we show the diagram of the Python classes that compose it. The legend for interpreting this UML diagram is

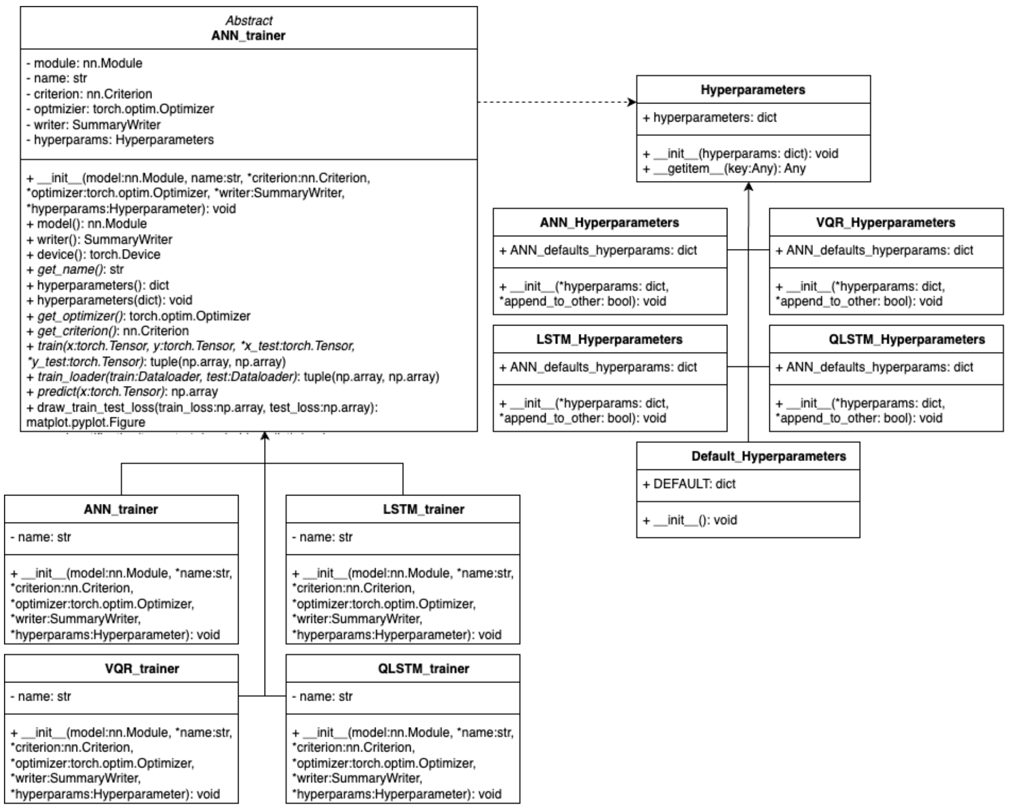

Abstract
ANN_trainer
- module: nn.Module
- name: str
- criterion: nn.Criterion
- optmizier: torch.optim.Optimizer
- writer: SummaryWriter
- hyperparams: Hyperparameters
+ __init__(model:nn.Module, name:str, *criterion:nn.Criterion, *optimizer:torch.optim.Optimizer, *writer:SummaryWriter, *hyperparams:Hyperparameter): void
+ model(): nn.Module
+ writer(): SummaryWriter
+ device(): torch.Device
+ get_name(): str
+ hyperparameters(): dict
+ hyperparameters(dict): void
+ get_optimizer(): torch.optim.Optimizer
+ get_criterion(): nn.Criterion
+ train(x:torch.Tensor, y:torch.Tensor, *x_test:torch.Tensor, *y_test:torch.Tensor): tuple(np.array, np.array)
+ train_loader(train:Dataloader, test:Dataloader): tuple(np.array, np.array)
+ predict(x:torch.Tensor): np.array
+ draw_train_test_loss(train_loss:np.array, test_loss:np.array): matplot.pyplot.Figure
Hyperparameters
+ hyperparameters: dict
+ __init__(hyperparams: dict): void
+ __getitem__(key:Any): Any
ANN_Hyperparameters
+ ANN_defaults_hyperparams: dict
+ __init__(*hyperparams: dict, *append_to_other: bool): void
VQR_Hyperparameters
+ ANN_defaults_hyperparams: dict
+ __init__(*hyperparams: dict, *append_to_other: bool): void
LSTM_Hyperparameters
+ ANN_defaults_hyperparams: dict
+ __init__(*hyperparams: dict, *append_to_other: bool): void
QLSTM_Hyperparameters
+ ANN_defaults_hyperparams: dict
+ __init__(*hyperparams: dict, *append_to_other: bool): void
Default_Hyperparameters
+ DEFAULT: dict
+ __init__(): void
ANN_trainer
- name: str
+ __init__(model:nn.Module, *name:str, *criterion:nn.Criterion, *optimizer:torch.optim.Optimizer, *writer:SummaryWriter, *hyperparams:Hyperparameter): void
LSTM_trainer
- name: str
+ __init__(model:nn.Module, *name:str, *criterion:nn.Criterion, *optimizer:torch.optim.Optimizer, *writer:SummaryWriter, *hyperparams:Hyperparameter): void
VQR_trainer
- name: str
+ __init__(model:nn.Module, *name:str, *criterion:nn.Criterion, *optimizer:torch.optim.Optimizer, *writer:SummaryWriter, *hyperparams:Hyperparameter): void
QLSTM_trainer
- name: str
+ __init__(model:nn.Module, *name:str, *criterion:nn.Criterion, *optimizer:torch.optim.Optimizer, *writer:SummaryWriter, *hyperparams:Hyperparameter): void

Figure 3.26: The UML diagram of the main part of the framework.

the standard one net of: names beginning with an asterisk "*" means that they are optional, names in italics represent an abstract method or parameter, and the solid arrow represents an extension of a class while the dashed arrow means that an instance of the target class was used as a parameter.

As can be seen from Figure 3.26, there is an abstract class called "Trainer" that is responsible for representing a generic trainer of models built with Pytorch.
It requires as input the model it is to train and a symbolic name representing it. The remaining variables such as the optimizer, the criterion by which to compute the loss function, and hyperparameters are left abstract so that the concrete class that will extend this will be given the opportunity to provide an implementation suitable for the type of trainer being created.
Therefore, to implement a trainer of a generic model, it is mandatory to extend the main abstract class. Consequently, one is obliged to provide the implementation of all the abstract methods present namely: `get_name()`, `get_optimizer()`, `get_criterion()`, `train()`, `train_loader()` and `predict()`.
Let us individually discuss the operation of these methods that serve as the backbone of the framework with regard to training and analyzing the results of a generic model:

- `get_name()`: It is mandatory to provide a symbolic name for each model you wish to train; this name will be used as an identifier for the folders that enclose the graphs and model weights produced at the end of training.
- `get_optimizer()`: Since each model may have its own weight optimization algorithm, it is important to specify which one you want to use to perform the training.
- `get_criterion()`: For the same reason as the optimizer, the loss function must also be defined before each workout in order to calculate the distance between the predicted value and the ground truth.
- `train()`: This is the main method that deals with training the model: the logic of how to carry out such training is defined within it. As input it receives a dataset divided into its train and test parts: this division is also repeated in order to work on the features to be used for training (named with the symbol X) and the one to be used for doing prediction (named y).
- `train_loader()`: This method is identical to `train()` where the major difference lies in the input received; in fact, you can use Pytorch's class called Dataloader to define the train and test set. This class allows for a number of advantages in managing the dataset before training the model, such as handling shuffle, partitioning, and data sampling.
- `predict()`: How a prediction is made, while most of the time it does not change, must also be made explicit in this method.

An observation to be made concerns the two methods for carrying out the training: although the structure of the classes forces both to be implemented, it is possible to develop the logic of only one, i.e., the one you consider most appropriate. From the abstract structure of the "Trainer" class, specific classes were developed for each model used in this project. The choice of dividing each model into a specific Python class was made to take several advantages such as: remaining compliant with the developed framework, ensuring greater readability of the code, and being able to make changes effectively without having to impact other aspects of the code. The trainers of each model have no parameters of their own, but simply implement the constructor and all inherited abstract methods.

On the right side of Figure 3.26 we notice the computer representation of hyperparameters. The "Hyperparameters" class relies on the native Python dictionary structure and is responsible for recording, updating, and providing hyperparameters at all times.
This is not an abstract class in that it implements the logic of read and write access (via the magic method `__getitem__`) to the dictionary; it also provides a default version of hyperparameters with the class "Default_Hyperparameters". The choice of this type of code structure was made because each model has its own hyperparameters that it must use to train: thus, one understands the reason for creating its own specific class for each model.

The Python classes representing the individual artificial models are represented in this way: All models extend Pytorch's torch.nn.Module class: this class represents a generic neural network and boasts all the methods needed to manage it, such as the `forward` method that implements forward propagation logic. Although two of these models represent quantum circuits, it was chosen to use Pytorch as the executive engine; this operational methodology is natively realizable by the Pennylane library that supports Pytorch. By acting in this way, another important advantage was achieved: the entire framework used is able to work regardless of the type of model, classical or quantum.
Each model declares its own parameters that are used to define it and realize the structure of the network or circuit.
Other Python classes were made to take advantage of some common utilities such as: project configuration via an external file (in yaml format), event notification system via email and SMS, batch mode reading of results published on Tensorboard, and methods to draw the trend of the loss function during epochs and model performance during its training.

3.4.2 Visualize the experiments

To conclude this chapter, we discuss the use of the Tensorboard library. All of the libraries described so far are used at the operational level to run the project and

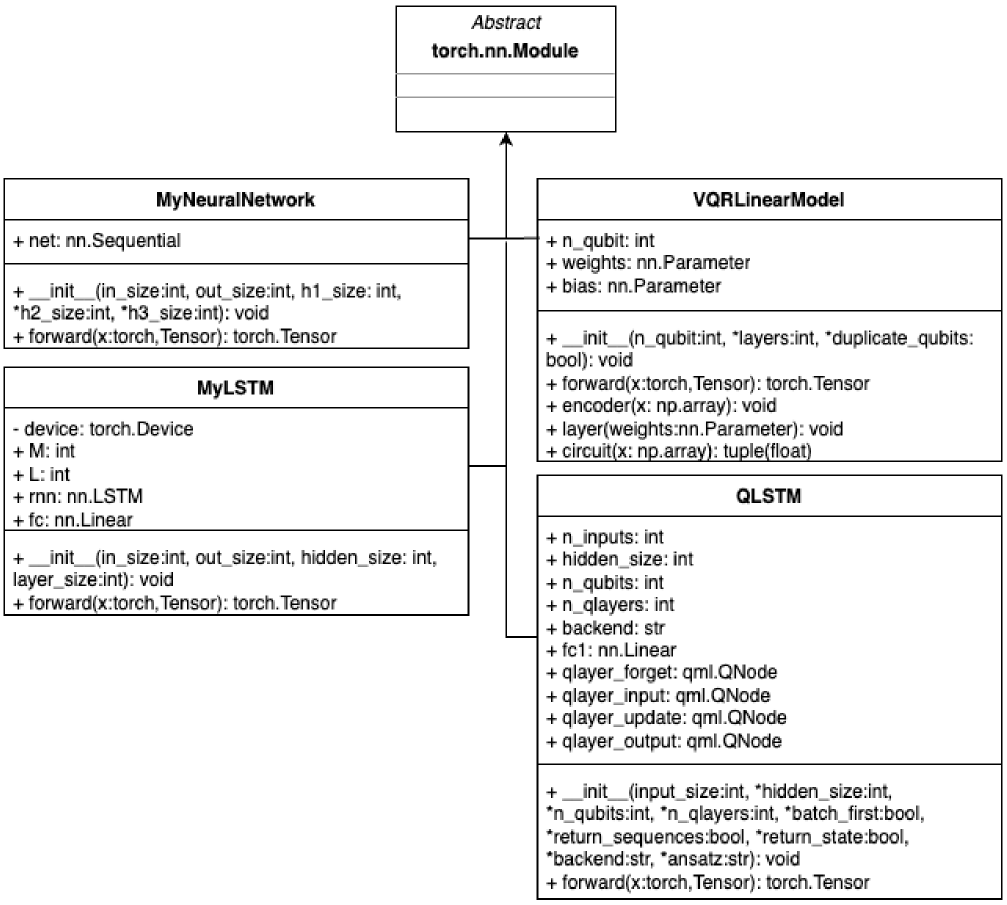

Abstract
torch.nn.Module
MyNeuralNetwork
+ net: nn.Sequential
+ __init__(in_size:int, out_size:int, h1_size: int, *h2_size:int, *h3_size:int): void
+ forward(x:torch,Tensor): torch.Tensor
MyLSTM
- device: torch.Device
+ M: int
+ L: int
+ rnn: nn.LSTM
+ fc: nn.Linear
+ __init__(in_size:int, out_size:int, hidden_size: int, layer_size:int): void
+ forward(x:torch,Tensor): torch.Tensor
VQRLinearModel
+ n_qubit: int
+ weights: nn.Parameter
+ bias: nn.Parameter
+ __init__(n_qubit:int, *layers:int, *duplicate_qubits: bool): void
+ forward(x:torch,Tensor): torch.Tensor
+ encoder(x: np.array): void
+ layer(weights:nn.Parameter): void
+ circuit(x: np.array): tuple(float)
QLSTM
+ n_inputs: int
+ hidden_size: int
+ n_qubits: int
+ n_qlayers: int
+ backend: str
+ fc1: nn.Linear
+ qlayer_forget: qml.QNode
+ qlayer_input: qml.QNode
+ qlayer_update: qml.QNode
+ qlayer_output: qml.QNode
+ __init__(input_size:int, *hidden_size:int, *n_qubits:int, *n_qlayers:int, *batch_first:bool, *return_sequences:bool, *return_state:bool, *backend:str, *ansatz:str): void
+ forward(x:torch,Tensor): torch.Tensor

Figure 3.27: The UML diagram of the classes representing the individual artificial models.

to take advantage of essential features such as creating the structure of a neural network or circuit.
Tensorboard, on the other hand, was selected as the results visualization tool. This library is a product of TensorFlow: Pytorch's competitor library for working in the world of deep learning. It was created to provide the visualization and tooling needed for machine and deep learning experimentation such as:

- Tracking and visualizing metrics such as loss and accuracy
- Visualizing the model graph
- Observing the time-varying histograms of weights, biases, or other tensors
- Projecting embeddings to a lower dimensional space

- And much more

The Tensorboard graphical representations used by our framework are different, such as: the performance of the loss function during training, the prediction performance achieved by the model, the internal structure of a model and the hyperparameters used to produce certain results.
The framework was built to save all the data previously listed on Tensorboard and partially also locally, without incurring saturation of the computer's memory. Tensorboard uses a folder on the file system to store data and therefore work: this folder is managed entirely by the framework, making this entire part invisible to the programmer's eyes. When you then want to access the visualization tool, just open a terminal on your computer in the main project folder and type the following command:

```
tensorboard --logdir=runs
```

where `--logdir=runs` indicates that the folder named "runs" is the one to use to view the results (and is also the default folder where Tensorboard saves the data). At the end of executing this command, the terminal will remain operational as a web server is running on your computer. By connecting to the address `http://localhost:6006` via any browser, you will find the Tensorboard home page where you can start viewing results.

We show some representations extracted from Tensorboard during the intermediate phase of the thesis development:

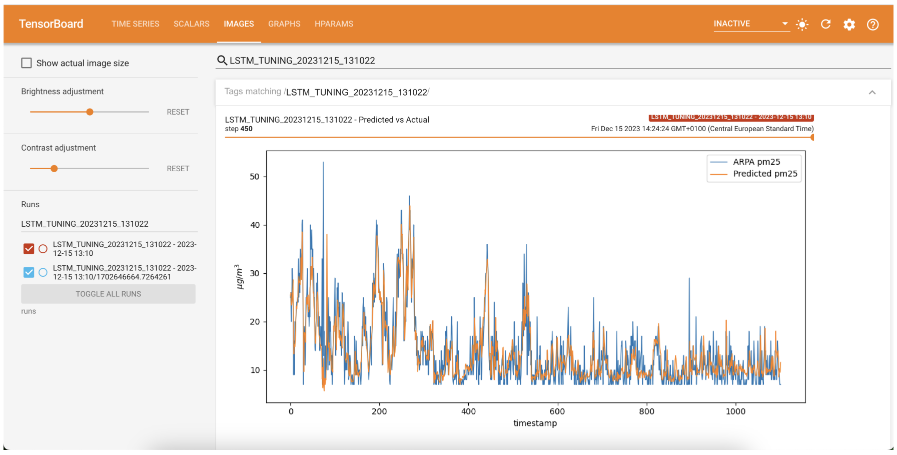
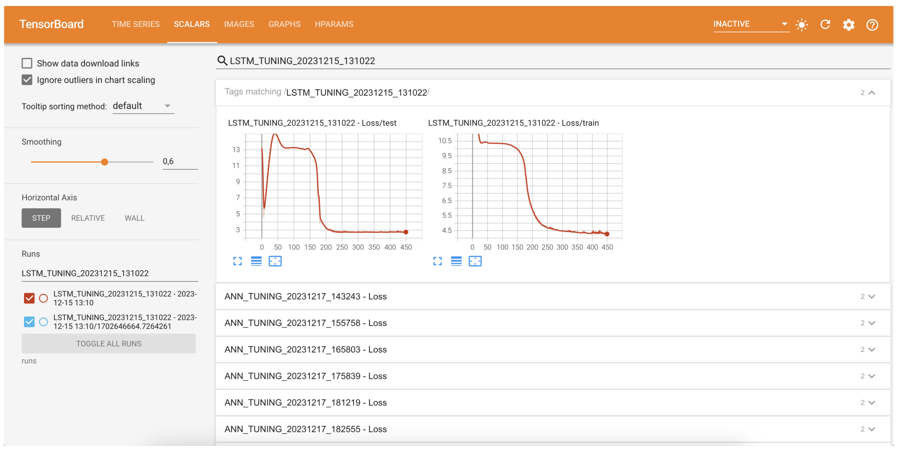

TensorBoard
TIME SERIES
SCALARS
IMAGES
GRAPHS
HPARAMS
INACTIVE
Show actual image size
Brightness adjustment
RESET
Contrast adjustment
RESET
Runs
LSTM_TUNING_20231215_131022
LSTM_TUNING_20231215_131022 - 2023-12-15 13:10
LSTM_TUNING_20231215_131022 - 2023-12-15 13:10/1702646664.7264261
TOGGLE ALL RUNS
runs
LSTM_TUNING_20231215_131022
Tags matching /LSTM_TUNING_20231215_131022/
LSTM_TUNING_20231215_131022 - Predicted vs Actual
step 450
LSTM_TUNING_20231215_131022 - 2023-12-15 13:10
Fri Dec 15 2023 14:24:24 GMT+0100 (Central European Standard Time)
ARPA pm25
Predicted pm25
50
40
30
20
10
µg/m³
0
200
400
600
800
1000
timestamp

Figure 3.28: The performance visualizations of a an LSTM model in Tensorboard.



LSTM_TUNING_20231215_131022

☐ Show data download links
☑ Ignore outliers in chart scaling

Tooltip sorting method: default

Smoothing 0,6

Horizontal Axis: STEP

Runs

LSTM_TUNING_20231215_131022

☑ LSTM_TUNING_20231215_131022 - 2023-12-15 13:10

☑ LSTM_TUNING_20231215_131022 - 2023-12-15 13:10/1702646664.7264261

runs

Tags matching /LSTM_TUNING_20231215_131022/ 2

LSTM_TUNING_20231215_131022 - Loss/test

LSTM_TUNING_20231215_131022 - Loss/train

ANN_TUNING_20231217_143243 - Loss 2

ANN_TUNING_20231217_155758 - Loss 2

ANN_TUNING_20231217_165803 - Loss 2

ANN_TUNING_20231217_175839 - Loss 2

ANN_TUNING_20231217_181219 - Loss 2

ANN_TUNING_20231217_182555 - Loss 2

Figure 3.29: The loss function evolution of a an LSTM model in Tensorboard divided by train-set and test-set.

Chapter 4

Results

In this penultimate chapter the results obtained will be shown in order to compare classical models with quantum ones in solving the calibration problem of low-cost fine dust sensors. The chapter will explain the individual steps to train and validate artificial models, showing some intermediate results. In conclusion, the comparison of each model in its best hyperparameter configuration is shown, followed by a personal comment.

4.1 Hyperparameter tuning

The first step of a Deep Learning pipeline is to configure a model suitable for the dataset on which you will have to train and try to do some training, noting the results obtained. Once you understand which hyperparameters can most influence the functioning of the model, it is time to use hyperparameter tuning.

Hyperparameter tuning is a technique that involves training a new model for every possible combination of hyperparameters. It aims to adjusting the configuration settings of the algorithms that govern the model's learning process by using the brute force, i.e. the repetition of a set of operations until the search space is exhausted. Unlike model parameters, which are learned from the data during training, hyperparameters must be set by the practitioner.
Typically, the procedure is assessing the model's performance for every combination and choosing the set that produces the best outcomes based on a predetermined criterion, such accuracy, precision, recall, or a metric unique to a certain domain.

When a model performs less than optimally with its default parameters or when the practitioner wants to extract every ounce of performance from the model, hyperparameter tweaking may be quite helpful. It becomes crucial in situations

where there is competition, like machine learning contests, or in applications where variations in performance measurements lead to notable qualitative variations in results, such high-stakes trading, financial forecasting, or medical diagnosis.
Hyperparameter tuning is not without its difficulties and disadvantages, though. In particular, for models with a high number of hyperparameters or when the search space is extensive, it can be computationally expensive and time-consuming. If the tuning is overly precise, there's also a chance of overfitting the model to the validation set. Furthermore, locating the ideal hyperparameters can occasionally be likened to trying to find a needle in a haystack, necessitating a calculated technique to thoroughly scan the search space. Among the methods employed to conduct this search are grid search, random search, Bayesian optimisation, and evolutionary algorithms. Every one of these approaches provides a unique compromise between searching the whole field and taking advantage of well-established optimal arrangements.

Table 4.1 shows the data in which we show, for each model, the size of the search space used in order to perform complete hyperparameter tuning and find the best configurations. The presence of substantial differences in search space size on some

Model	**Search space dimension**
ANN	5832 combinations
LSTM	648 combinations
VQR	162 combinations
QLSTM	8748 combinations

Table 4.1: The search space of the hyperparameters tuning divided by each model

models is justified by its complexity and freedom of choice in the architecture of the underlying network or circuit.
The "ANN" model achieves a size of 5832 combinations as the network architecture involves using one to three hidden layers with the number of neurons per layer ranging from a maximum of 128 to a minimum of 10. While for the "QLSTM" quantum model, the same space as its classical equivalent LSTM was explored but with the addition of the number of qubit and quantum layers with which to repeat the ansatz.

It is important to note that at this point in the thesis, the artificial neural network model called "ANN" has not yet been divided into its two variants mentioned in Chapter 3.3.1 (i.e., "ANN#1" and "ANN#2).
The model used for this tuning phase is similar to the one defined as "ANN#1" in which there are 60 neurons, one for each minute that makes up an hour, that will go to predict the value coming from the reference station; the only difference

is that only the value of particulate matter in the air ($PM_{2.5}$) was used as input features.

The table with the best configurations obtained at the end of tuning for each model is now shown. When a cell in this table is empty, it means that that hyperparameter was not used within the selected model. Table 4.2 shows at the bottom the

Hyperparameter	**ANN**	**LSTM**	**VQR**	**QLSTM**
N° epochs	200	450	200	400
Learning rate	0.0001	0.001	0.01	0.01
Optimizer	SGD	Adam	Adam	Adam
Criterion	L1	L1	MSE	L1
Batch size	2	-	10	-
Hidden size l1	60	600	-	15
Hidden size l2	40	-	-	-
Hidden size l3	10	-	-	-
T (hours used t o calibrate)	-	3	-	5
N° layers	-	2	-	-
N° qubits	-	-	4	7
N° quantum layers	-	-	4	5
Ansatz	-	-	linear	strongly
Loss on train-set	10.465	4.2697	0.0098668	5.2021
Loss on test-set	5.3008	2.7362	0.0026194	2.6978

Table 4.2: The results of the hyperparameters tuning divided by each model

lowest values of the loss function calculated on the training-set and test-set: these indicate the goodness of learning achieved by the model during training. Looking at these values, the small difference present between the training-set and test-set leads to the assumption that overfitting is not occurring, i.e., that the models have memorized the dataset and therefore the generalized learning of sensor calibration is not true.

The loss function value that immediately catches the eye due to its lowness compared to the others is produced by the VQR model. The reason for this different scale of values compared to the other models is because the input features passed as input to the variational circuit are scaled by $[-1, +1]$ as explained in Chapter 3.3.3. By clearly reducing the range of possible values that the characteristics of particulate matter in the air can take on, the precision of the calibration also

increases and consequently the values produced by the model will be much "closer" to those produced by the ARPA reference station.
The presence of this feature scaling operation does not allow a fair comparison with the other models in Table 4.2. Despite this, a comparison can be made via the performance graph that we will show later.
With Tensorboard's parallel coordinate visualization, it is possible to understand the trend of hyperparameters that lead to a lower loss value, and thus generate a better model.
In Figure 4.1, this visualization specific to the LSTM model is shown.

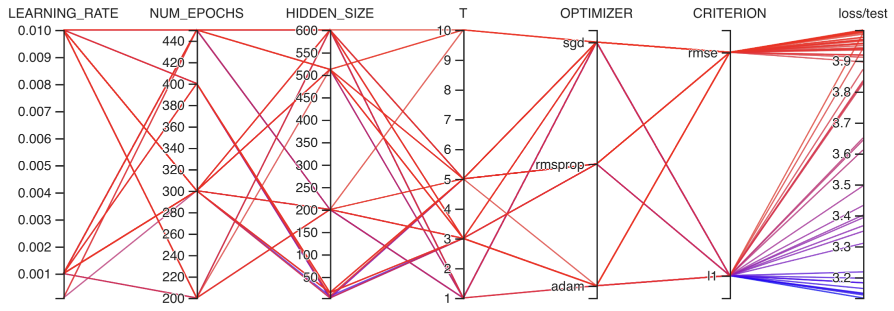


Figure 4.1: The parallel coordinates view related to LSTM model. It shows the trend of hyperparameters leading to different levels of loss.

For each type of model and hyperparameter, this type of graph supports in understanding which combination leads to a may low loss. In fact, in Figure 4.1, we can see how the L1 loss function leads the model to perform at a lower loss than the RMSE function. As for the optimization algorithms, the best loss result in the test-set was achieved with Adam, although the others available (SGD and RMSprop) produce models that maintain excellent performance for calibration purposes.
To conclude, it is easy to show that values of the hyperparameter T, which represents the number of previous hours used to make the next prediction, tend to produce better models when it is low ($T \leq 5$).

A final aspect to take into consideration to conclude the hyperparameters tuning phase is learning saturation. Initially, the number of epochs to be used in the tuning search space is given randomly. Then looking at the plot of the trend of the loss function in the training-set, it can be seen that at some point, starting from a certain epoch, all the models stop having notable improvements in learning.
The training of the QLSTM quantum network is shown as an example, in which it is evident that starting from the 270th epoch, the model shows no signs of improvement in learning, i.e. the loss function stops decreasing. Its saturation

point is found to be around epoch 270.

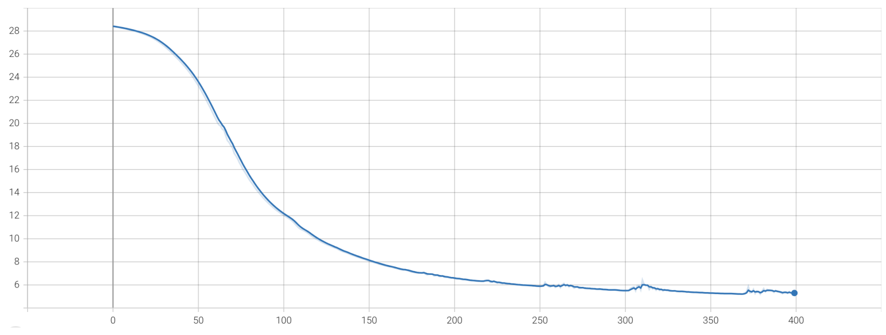


Figure 4.2: The performance of the loss function in the training set of the QLSTM model.

4.2 Improvements of the models

The value that guides the construction of an appropriate model for solving the basic problem is the loss function applied in the test-set. Indeed, recall that the test-set is that portion of the dataset that remains hidden during training and is then shown to the trained model to make predictions (or calibrations) on data it has never seen. Certainly, the fact that the loss value in the test-set is lower than that realized in the training-set is shown by the fact that the former is significantly smaller in size than the latter (we are talking about 30 percent versus 70 percent). This leads the model to make fewer errors due to the scarcity of data in the test-set than in the training-set.

It is easy to see that the "ANN" model has a higher loss value in the test-set than the others. Despite the huge size of the search space, the adopted configurations did not lead to adequate performance. This pointed out that the structure of the neural network or the data on which it trained are not adequate and can be changed radically.
To confirm this, we show in Figure 4.3 the calibration performance obtained with this model using the best configuration that emerged from tuning.

For this reason, it was chosen to create the ANN#1 and ANN#2 variants shown in Chapter 3.3.1. The former model keeps the same network architecture and changes the input features used, while the latter makes a change to both the architecture and the data.
We show the tuning results obtained from these two variants of the classic "ANN" model. The hyperparameters found at the end of tuning in Table 4.3 are very

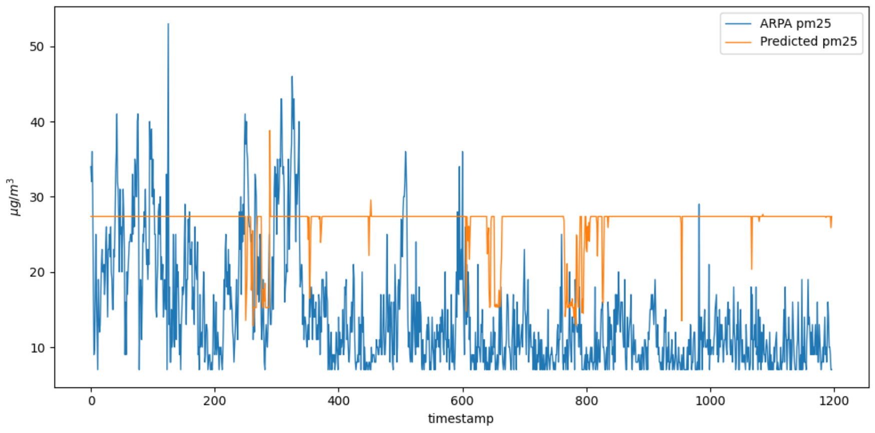


Figure 4.3: The calibration performance obtained with "ANN" model.

Hyperparameter	**ANN#1**	**ANN#2**
N° epochs	200	200
Learning rate	0.0001	0.0001
Optimizer	SGD	SGD
Criterion	L1	L1
Batch size	10	10
Hidden size l1	90	30
Hidden size l2	20	15
Hidden size l3	-	5
Loss on train-set	10.435	6.7904
Loss on test-set	4.4004	2.9202

Table 4.3: The results of the hyperparameters tuning for the ANN#1 and ANN#2 models.

similar, net of one less hidden layer present in the "ANN#1" model. The values of the loss function in the test-set of the two variants declare the ANN#2 model as the "winner": the use of a shallower (and therefore simpler) architecture combined with a dataset with hourly granularity and with all the input features available, prove to be the best compromise for creating a model suitable for the problem.
We show below the calibration performance obtained by model "ANN#2" after discovering its best configuration of hyperparameters. Compared with the performance obtained with the previous model (Fig.4.3), significant improvements in sensor calibration can be observed.

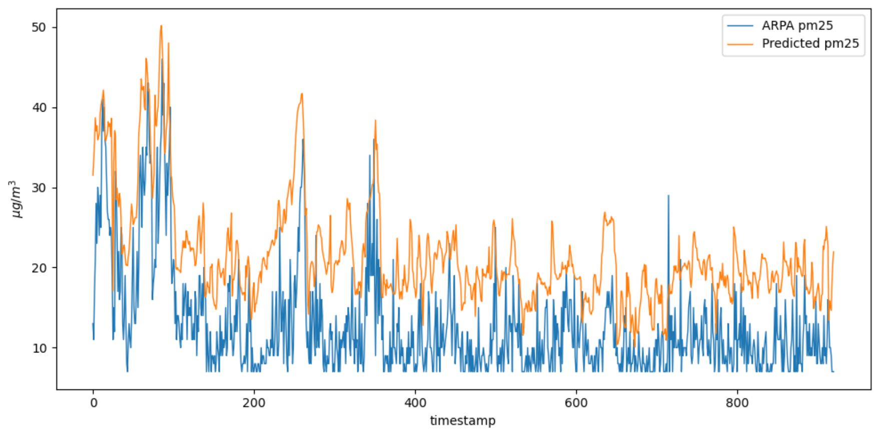


Figure 4.4: The calibration performance obtained with "ANN#2" model.

The recurrent LSTM model is the second to produce a very good result among all the comparable models in Table 4.2.The dataset it trained on is divided into 70% training-set and the remaining 30% test-set. This means that, a large sequence of data in ascending order according to time was used to teach LSTM how to calibrate the sensor.
Recall in fact, that with LSTM it is not possible to perform a shuffle of data because the data must be given in chronological order in order to exploit the true potential of this model: learning by remembering what it "has seen" before. So, using the 70 percent train-set means having a portion of the data that encompasses approximately at least 3 to 4 months of history.
It is fair to ask, for the purpose of making the project more sustainable, what would happen if I used the LSTM model to calibrate the fine dust sensors, having only 30 days of data collection? In other words, the size of the training-set would become only 15% while the remaining 85% becomes the new test-set.
We show the performance of the LSTM model with the same hyperparameters but trained on 15% of the dataset (i.e., about 30 days). As can be seen in Figure 4.5, the model did not have a chance to learn the specific characteristics of the dataset because it trained on a small portion. For example: the spike in fine particulate matter present every year on New Year's Eve (Dec. 31) due to fireworks, using only thirty days of data collected for training, this event did not come under its notice and therefore the model had no way to learn.

In addition, it can be observed that the calibration performed by this model remains balanced with respect to the high and low peaks present in the ARPA measurements; this leads one to think (and observe) that the newly created LSTM model tends to generalize the calibration results along the entire time axis.

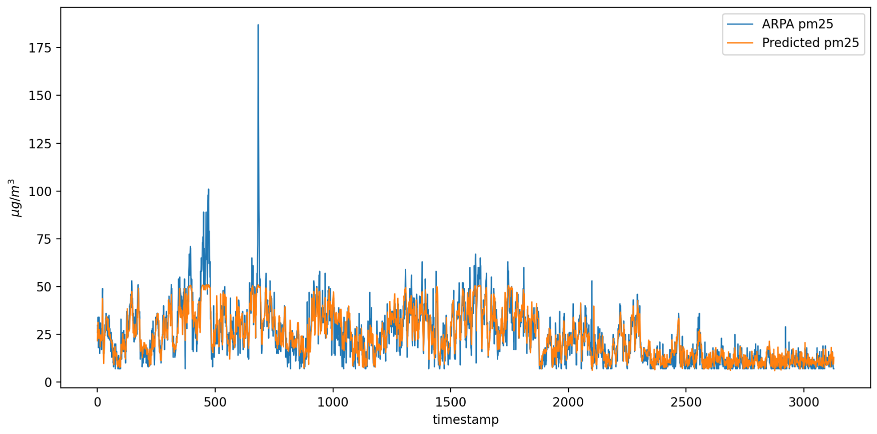


Figure 4.5: The performance of the LSTM model trained with only 15% of the dataset.

The advantages that this variant of the LSTM model brings are certainly greater generalization, that is, the possibility of having a fairly even calibration throughout the period of sensor operation, using only 30 days of data collection in order to train the model. The disadvantage is that, having a generalist model, the average accuracy calculated as the value of the loss function on the test-set will definitely be lower than in the original model.

To conclude with this type of model, one last variation was created. The purpose, this time, is to understand what is the impact on accuracy if we introduced the other input features (temperature, pressure, and humidity) during model training. In fact, the first version of LSTM presented in the previous chapter used only the feature of $PM_{2.5}$ detected by our sensors (remember: a median of our sensors). Again, the portion of the dataset on which the model trains has been kept on the 30 days: in this way, the real goal is to understand whether using the other dimensions of the dataset can make the model less generalist and more specific in performing calibration.
The test ended in failure: this latest variant of LSTM produces a model that has lower calibration accuracy than the original model (as expected) and also lower than the model that uses the same number of days to train but takes into account only one input feature.
It is thus shown that the use of the other variables collected from the sensors do not bring added value in improving the calibration, at least as far as recurrent neural models are concerned.
We show on Table 4.4 the results obtained from all variants of the LSTM model.

On the VQR and QLSTM quantum circuits, no type of enhancement was

Hyperparameter	**LSTM**	**LSTM-30days**	**LSTM-all-features**
Training-size	70%	15%	15%
N° epochs	450	450	450
Learning rate	0.001	0.001	0.001
Optimizer	Adam	Adam	Adam
Criterion	L1	L1	L1
T (hours used to calibrate)	3	3	3
Hidden size l1	600	600	600
N° layers	2	2	2
Loss on train-set	4.2697	3.4401	3.8126
Loss on test-set	2.7362	4.3276	4.4333

Table 4.4: The comparison of the results obtained among all variants of LSTM.

experimented with: the high results obtained after hyperparameter tuning and the lack of time are the two factors that contributed most to arriving at this work.

4.3 The benchmark

After understanding the ideal combination of hyperparameters through the tuning session explained in section 4.1 and after further testing and optimization of some models in section 4.2, it is time to ask whether the work done up to this point is bringing improvements or not.
The proof of the presence of these improvements must be irrefutable and grounded. For this reason, the benchmark is nothing more than a baseline measure based on scientific or statistical evidence from which it is possible to tell, based on the distance from it, whether we are in a scenario of improvement or deterioration.

In the thesis project, the value of possible loss functions applied to the original dataset was chosen as the benchmark. To be then compared with the accuracy obtained by each model, the portion of the dataset on which the loss function was calculated is only the test-set, i.e., that part of the data that is not shown to the models during their training. In this way, the volume of data used to calculate a model's goodness-of-fit will be identical to that used to calculate its benchmark. The calculation involves using the test-set in several of its dimensions: 30%, 25% and 85% (only in the case of LSTM model); the data used are represented by the value of particulate matter in the air given by the ARPA reference station and the same value measured by our sensors. The loss functions will calculate the loss present within the dataset without any preprocessing steps and without a trained model that can learn how to minimize these functions.

So to summarize, the benchmark varies according to the loss function and the portion of the test-set on which each model is trained. The last row of Table 4.5 represents the benchmark calculated on the same dataset with which the VQR model was trained; the major difference is to have all values scaled on a scale of $[-1, +1]$. If the results obtained in the previous chapters turn out to be better than the relevant benchmark, then this becomes clear evidence of the effectiveness of the artificial model in learning how to solve the problem, that is, how to calibrate the sensors in question.
The following table shows the benchmark results calculated as stated.

Benchmarks	**L1**	**MSE**	**RMSE**
Test-set 30%	5.0296	42.5333	6.5218
Test-set 25%	4.7014	33.9137	5.8235
Test-set 85%	6.6392	84.9754	9.2182
Test-set 25% rescaled	0.1013	0.0195	0.1395

Table 4.5: The benchmark results calculated for each type of loss function and portion of the test-set.

We illustrate a brief comment on the benchmark results shown in Table 4.5 compared with those obtained by the models after tuning and improvements.
The LSTM model is significantly lower than its benchmark: its best result, shown in Table 4.4, is 2.7362 with 30% of the test-set and L1 as the loss function, or it is 4.3276 with 85% of the test-set and the same loss function. We are talking about results that are equivalent to about half of the benchmark or 40% less.

The model defined as "ANN," on the other hand, is superior to the benchmark and therefore is not bringing added value in solving the problem compared to what was already there to begin with. In fact, the best result was obtained using L1 as the loss function and is 5.3008. This is further evidence that the architecture of this model is not the best for the purpose of solving the problem.
The ANN#1 model, which uses 25% test-set, on the other hand, shows marked improvements: the highest result obtained using L1 as the loss function is 4.4004 compared with its benchmark of 4.7014; this means that by also introducing the other 3 features (temperature, pressure, humidity) the model is able to better generalize the value of $PM_{2.5}$ in the air.
Despite this, ANN#2 is shown to be by far the best network among others of its type: its result using 25 percent of the test-set and L1 as the loss function and is

2.9202 compared with 4.7014 in benchmark.

As for the QLSTM model, the same reasoning for comparison with the benchmark used for the LSTM model can be applied. The current situation predicts a marked improvement over the imposed benchmark.

The VQR quantum model, comparable with the classical ANN model, has results below the benchmark, showing that it is making improvements on calibration. The VQR quantum model, comparable with the classical ANN model, has lower results than the benchmark, showing that it is making improvements on calibration. The value of the best model, trained with MSE loss function and with 25% of the test-set, is 0.0026 which compared to the benchmark of 0.0195 we are talking about 86.67% less.
By rescaling the values after training this specific model, the value of the MSE loss function on the measurements made by ARPA and the model predictions can be calculated again. The value that is obtained represents the distance between the two types of information, that is, the accuracy that the model obtained after training. This value turns out to be 33.6627 which, compared to its benchmark which is equivalent to 33.9137, (visible in Table 4.5), successfully shows that a slight improvement has been achieved (0.74% less than the benchmark).

To conclude, we show the distribution of the random variable governing the operation of the benchmarks shown, that is, the calculation of the individual loss functions on the different portions of the data.
To do this, in much the same way as the Monte Carlo method for simulating results [20], a specific percentage of the dataset (equivalent to the desired test-set) is randomly taken on which all loss functions are calculated and the result is stored. These procedures are repeated for a large number N of times: in our case $N = 10.000$.
We show the approximate probability mass functions (PMFs) for each benchmark.
Figure 4.6 and 4.7 show the PMF of the benchmark calculated on the original test-set, while Figure 4.8 uses the test-set scaled in the range in which the VQR model operates.

4.4 Cross-validation results

Through hyperparameters tuning addressed in the previous section, we were able to identify the best set of hyperparameters that produce the best pattern for the purposes of solving the basic problem. Subsequently, an improved version of some models has been tested in order to achieve better results and explore other variants.

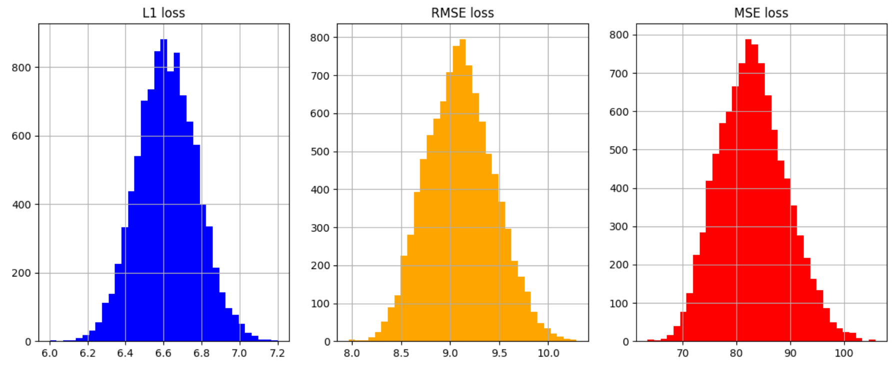


Figure 4.6: The distribution of the benchmark using the 30% of the test-set.

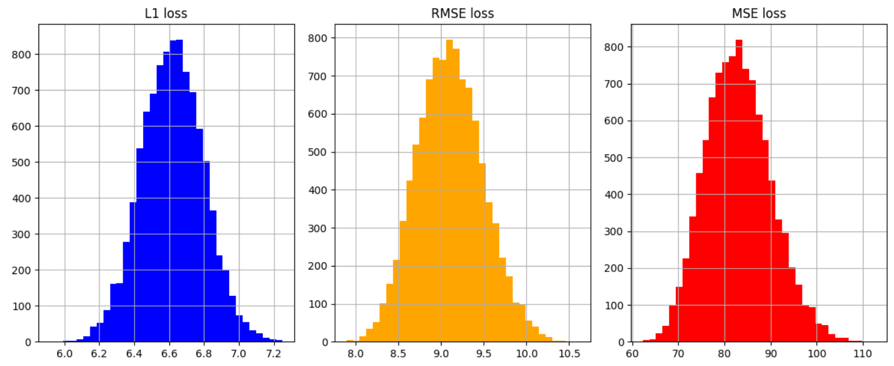


Figure 4.7: The distribution of the benchmark using the 25% of the test-set.

Now, however, it is important to ask whether different portions of the dataset may have an influence on the identified model.
To evaluate how well the findings of a statistical study will transfer to a different part of the dataset, cross-validation is a reliable machine learning approach. It is especially helpful in situations when the objective is to forecast how a model will perform with fresh, untested data. The key to cross-validation is its capacity to reuse data by partitioning the dataset into portions that are utilised for model testing and training, respectively.

K-fold cross-validation is a popular kind of cross-validation. The data is separated into 'k' subgroups using this procedure. After that, the model is evaluated on the remaining subset after being trained on 'k-1' of these subsets. Every one of the 'k' subsets is utilized as the test set precisely once during the 'k' repetitions of this operation. Then, an estimation is generated by averaging the outcomes of all 'k' trials. This method is particularly effective since it guarantees that each data point is used for testing as well as training, resulting in a thorough use of the data.

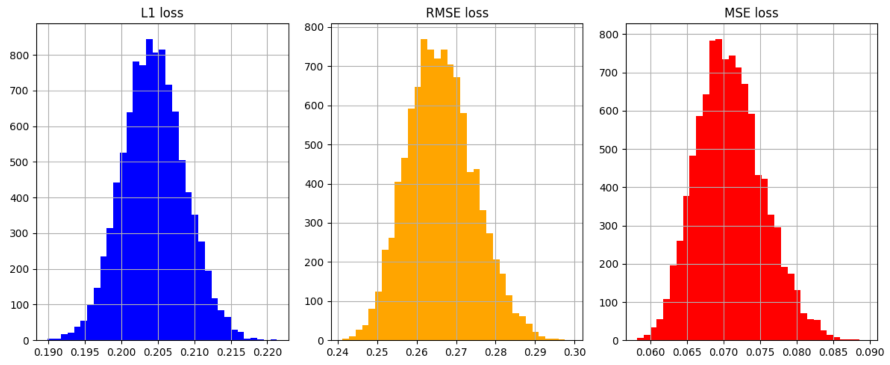


Figure 4.8: The distribution of the benchmark using the 25% of the scaled values test-set.

Other techniques are available to do cross-validation, each of them specific to each type of validation that is to be addressed; nevertheless, for this project K-fold was chosen.
Cross-validation should be used when seeking an unbiased estimate of the model's performance, as it mitigates the risk of overfitting by repeatedly evaluating the model on different subsets of data.

Nevertheless, there are several drawbacks to cross-validation. Its computational expense is among the main disadvantages. This can be a laborious procedure, especially for big datasets or sophisticated models, as the model needs to be trained and assessed 'k' times. Additionally, the validation's performance and dependability may be greatly impacted by the selection of 'k'.
Overly large 'k' values can result in models that are computationally expensive to train and may still have high test error variance because of the greater similarity between the training and test sets. Conversely, excessively small 'k' values could result in larger variation in the training data.
To use K-fold, it is necessary to understand what percentage to divide the dataset between train and test set in order to understand the value that the K-number should have. This calculation must be done for each model you want to validate.

With the LSTM model, it is necessary to take about 15 percent of the dataset as test-set (i.e., 30 days) and the remainder as train-set. To do this, the reasoning applied in order to understand the value corresponding to the K number of folds is as follows: the entire dataset contains 3683 samples, or 3683 hours which is equivalent to 153.46 days. Dividing the dataset into 5 folds would result in test sets of $\frac{3683}{5} = 736.6$ hours which equals 30.69 days. Therefore, we conclude by saying that $K = 5$ for the LSTM and QLSTM model.

For the ANN model, implementation "ANN#2" will be used as it shows a better result in the loss (Table 4.3). To take 25% of the test-set just use K-fold with $K = 4$, i.e. $\frac{1}{4}$ of the dataset will be used for testing and the remaining $\frac{3}{4}$ for training the network. The VQR model also uses the exact same dataset as ANN#2, except that the values within it have been scaled. So the number of folds used for this model are the same ($K = 4$).
In order to best highlight the results of cross-validation and improvement in model structure, we chose to draw the benchmark distribution in the same graph. In this way it will be easy to notice by how much the models are better or worse than their relative benchmark. The result of training and testing a model in each specific fold will be drawn with a dot throughout the graphs that follow.
Starting from the neural network models, we show the difference between ANN#1 and ANN#2 in the cross-validation process, compared to their benchmark. From

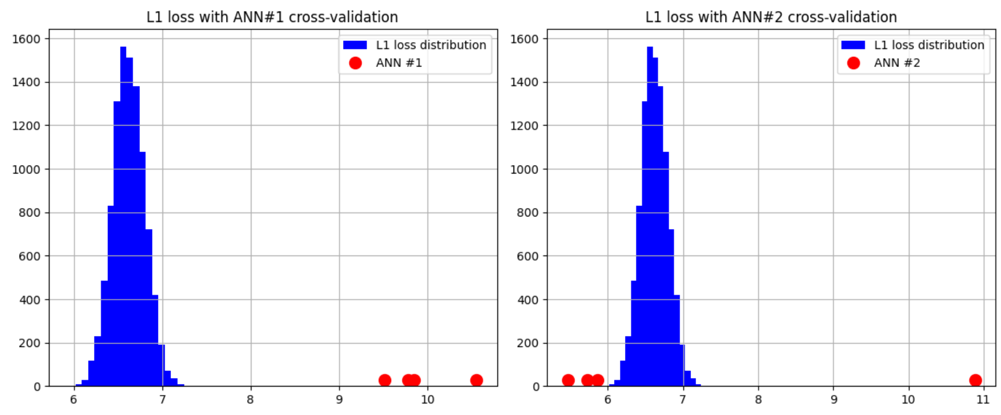


Figure 4.9: The cross-validation result of the ANN#1 (on the left) and ANN#2 (on the right) models compared to their benchmark.

Figure 4.9, it is easy to see that the improvement in this network type made in sec.4.2 brought clear improvements over the benchmark and thus over the average expected outcome.
The result of cross-validation of ANN#1 and ANN#2 models is shown in numerical terms through the following table.

Cross-validation	**Fold-1**	**Fold-2**	**Fold-3**	**Fold-4**
ANN#1	10.556	9.7848	9.8453	9.5108
ANN#2	5.8584	5.7268	5.4658	10.888

Table 4.6: The cross-validation result of the ANN#1 and ANN#2 models.

As for the quantum equivalent, validation on the VQR model was carried out according to the type of anastz used: linear or nonlinear (details can be found in Chapter 3.3.3). We can show how both models achieve excellent performance in terms of comparison with the benchmark, although the model that uses a linear ansatz, that is, a single embedding layer followed by several successive computation layers, turns out to be more accurate. We show the same results as in Figure 4.10

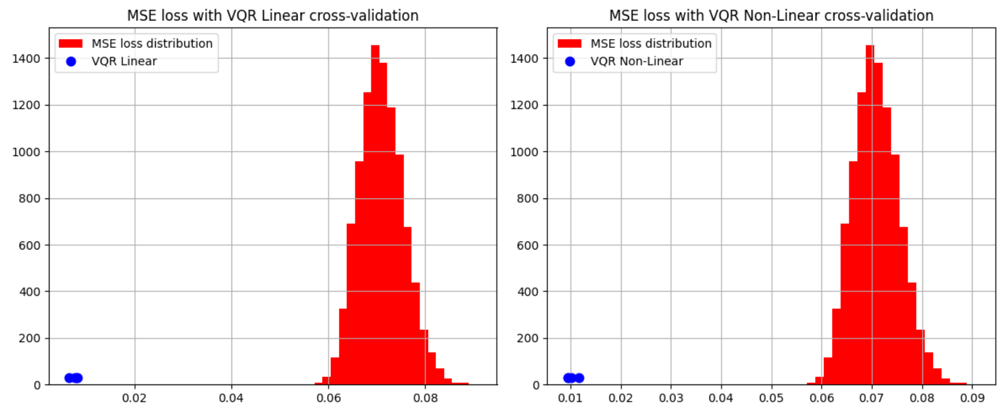


Figure 4.10: The cross-validation result of the VQR linear (on the left) and VQR non-linear (on the right) models compared to their benchmark.

but in tabular form.

Cross-validation	**Fold-1**	**Fold-2**	**Fold-3**	**Fold-4**
VQR Linear	0.0082112	0.0077559	0.0065363	0.0080037
VQR Non-linear	0.010147	0.0095828	0.0094464	0.011666

Table 4.7: The cross-validation result of the VQR model.

It can be seen that even the VQR model, trained and evaluated on different folds, i.e., random portions of the dataset, produces results far removed from the distribution of its benchmark. This implies a marked improvement over the benchmark as an attempt is made to minimize the loss function. Training on a dataset with such a narrow range of values, it is normal to observe a low error distinction for each fold.

The LSTM model, together with its quantum equivalent, turns out to be the model with the best results in order to solve the calibration problem. Indeed, we can see how, during its cross-validation, the colored dots obtained are distributed at heterogeneous distances and lower than the benchmark. This phenomenon demonstrates the lack of overfitting within the models and at the same time shows

signs of improvement over the benchmark.
We show in Figure 4.11 the results of cross-validation against the benchmark inherent in the LSMT and QLSTM models; next, we show the table with the numbers demonstrating the validation performed.

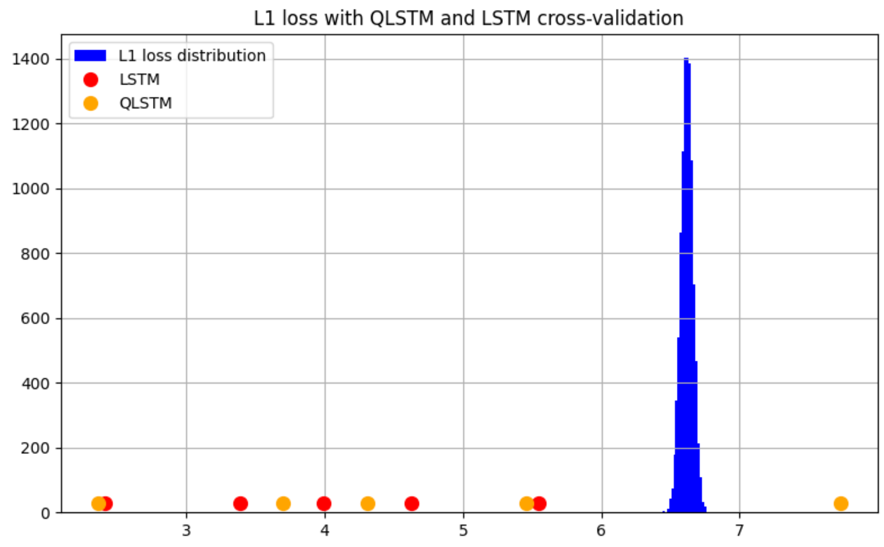


Figure 4.11: The cross-validation result of the LSTM and QLSTM models compared to their benchmark.

Cross-validation	**Fold-1**	**Fold-2**	**Fold-3**	**Fold-4**	**Fold-5**
LSTM	3.9884	5.5502	4.6265	3.3922	2.4122
QLSTM	4.3070	7.7299	5.4600	3.7013	2.3611

Table 4.8: The cross-validation result of the LSTM and QLSTM models.

The cross-validation performance of the QLSTM model seems to be worse than that of its classical equivalent, although, by averaging the loss values obtained for each fold, it can be seen that there is not that much difference, in fact a value of 4.7 is obtained for QLSTM compared to 4.0 for LSTM.

In conclusion, cross-validation is an effective method for assessing and choosing models that have the ability to function effectively with unknown data. It provides a fair method of evaluating models and sheds light on how well the model generalises outside of the particular set of training data.

4.5 Comparison between models

In this section of the chapter, performance results on the calibration of the sensors covered in this thesis for each of the four considered models will be shown.
Having explored the space of hyperparameter configurations, as well as further optimizations of some models, and validated the learning through cross-validation, it is now possible to compare these models in order to understand which one is the most promising for addressing the specific problem at hand.

Before showing the true comparison, it is important to look at the performance of each individual model in the best possible configuration for the dataset.
When we talk about performance on calibration, we mean using the best found configuration of hyperparameters of a model to make inference on the entire original dataset. In this way it will be possible to compare, throughout the time duration of data collection, the air particulate matter ($PM_{2.5}$) values measured by the ARPA reference station against those learned from the model.

For the ANN model, in its second variant "ANN#2," the performance on calibration was shown previously in Figure 4.4. We are talking about an improvement, compared with its benchmark, of 37.89% (calculated on the L1 loss with 25% of the dataset used to test the model).
While for the equivalent quantum circuit provided by the VQR model, the improvement of the MSE type loss over the benchmark, using 25% of the dataset as the test-set, is 86.67% with the scaled dataset, or 0.74% with the original dataset.
The performance of the best trained VQR model is shown graphically.

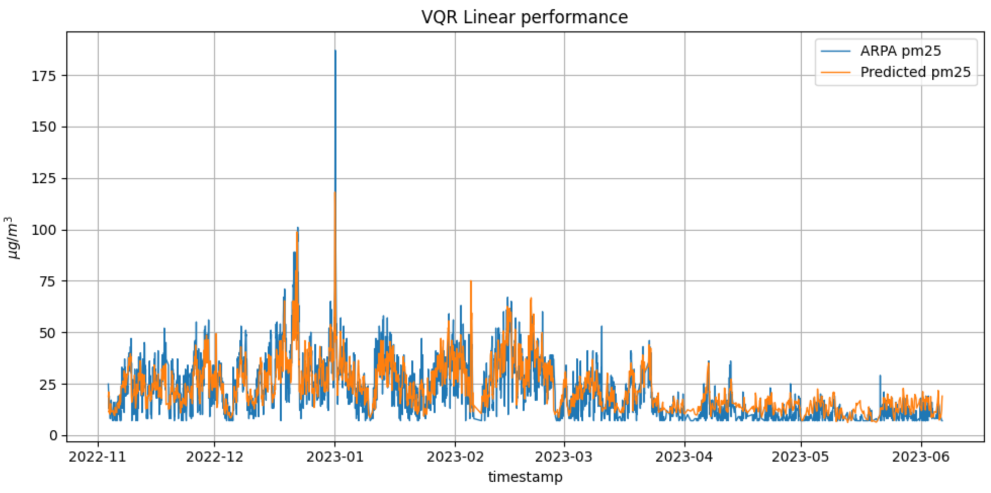


Figure 4.12: The calibration performance obtained with VQR Linear model.

As for the LSTM model, the value of the L1 loss function calculated using 30 percent of the dataset as the test-set turns out to be 45.60% lower than the value of its benchmark. We are already aware that this type of neural network, together with its equivalent quantum model, manages to produce a better result in order to solve the underlying problem. The result of the performance of this model can be seen in the follow Figure. The LSTM model shows that it generalizes very well to

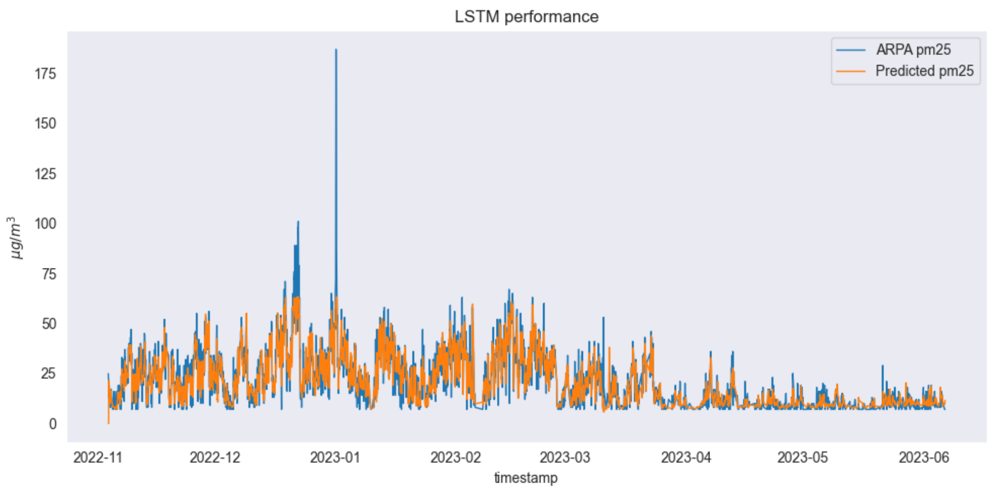


Figure 4.13: The calibration performance obtained with LSTM model.

the calibration aspects of these sensors along the time axis. In Figure 4.13, it can be seen that the trend of ARPA measurements and that learned from the model overlap throughout the timeseries, net of some outliers such as New Year's Day.

The QLSTM model, on the other hand, with the same loss function and portion of the dataset in which it tested its performance, achieves a 46.36% improvement over its benchmark. This model, compared with its equivalent in the world of classical computing, brings a slight improvement.
We show in Figure 4.14 the performance of the QLSTM model in calibrating the median of fine dust sensors against what ARPA measured.

After showing the calibration performance of each model in its best configuration and optimization, we go on to analyze the differences and behaviors present among these models compared. All four models were used to make inference on the original dataset by loading the weights and parameters found during the training phase.
Of course, it would not be correct for comparison purposes to make inference on the entire original dataset since only two parts of the dataset were used to train the models: the training-set and the test-set. This design choice implies that all models, having been trained on the training-set that corresponds to most of the original

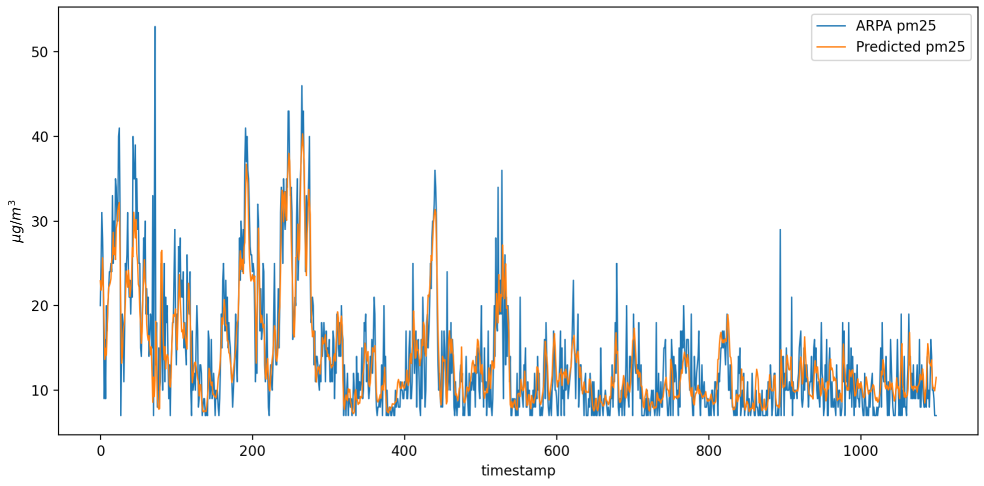


Figure 4.14: The calibration performance obtained with QLSTM model.

dataset, learned well the calibration on this part, especially all those models that tend to overfitting.

As a matter of visualizing the performance of the models compared with each other, the following graph shows their comparison behavior, taking into consideration the $PM_{2.5}$ values measured by the ARPA station sensors.

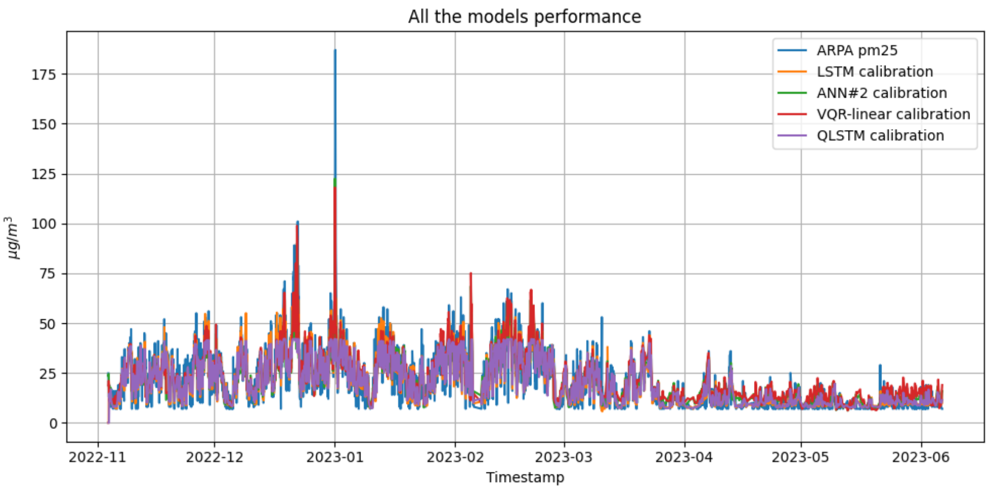


Figure 4.15: The calibration performance obtained from all models developed in this project in comparison with values from ARPA.

To better visually understand the difference between these patterns, a variant of Figure 4.15 is shown in which a shorter time frame of fifteen days is observed: from February 1 to February 15, 2023.

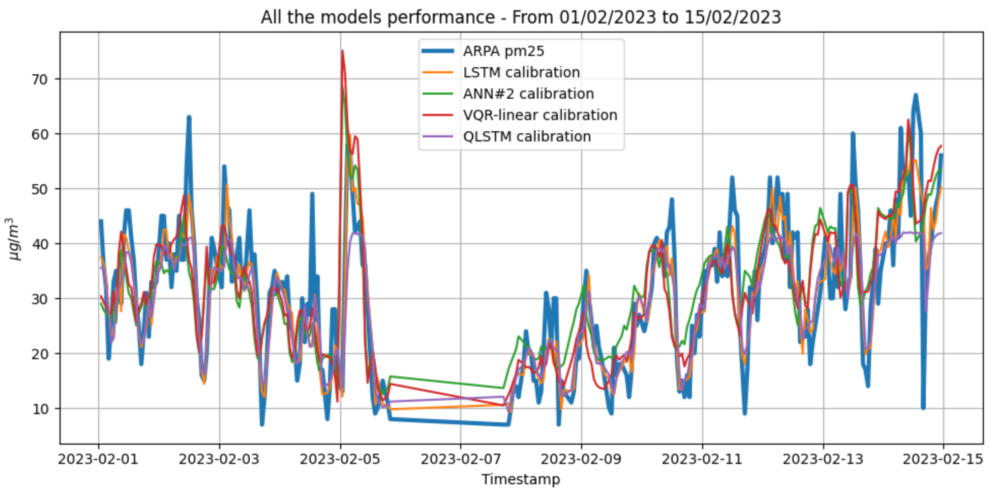


Figure 4.16: The calibration performance obtained from all models developed in this project in comparison with ARPA values, observing a time range from February 1 to 15, 2023.

It can be seen from Figure 4.16 how well the models, as a whole, understood the method of calibrating fine dust sensor values to resemble those measured by ARPA as closely as possible.
In order to represent in numbers the comparison between these models and understand which one of them succeeds in performing better, it was decided to calculate the RMSE loss function between the measurement values predicted by the model in the same unknown portion of the dataset (the test-set) and those measured by the reference station. The result is similar to that shown in the various model hyperparameter tuning tables but with the difference that a single loss function was set for all models; we report the results in the following table.

Performance metric	**ANN#2**	**VQR Linear**	**LSTM**	**QLSTM**
Test-set size	30%	30%	30%	30%
RMSE	8.0800	7.5364	6.1409	6.6605

Table 4.9: The RMSE loss function calculated for all model predictions compared with ARPA measurements.

It can easily be seen from Table 4.9 that although the portion of the test-set used is not exactly what each model trained with, the behavior maintained by the models in performing the calibration remains constant to what was noted in the previous sections. In other words: the quantum circuit manages to perform better than its equivalent in the classical world ANN#2, while between LSTM and QLSTM the winner still remains the classical model.

Comparing the first two models, i.e., the artificial feed-forward neural network and its equivalent made with variational quantum circuits, the model that produces better results is the second one. Although it does not differ much, the RMSE value of the VQR model turns out to be lower than that of ANN#2. This result implies that the lower architectural complexity of the quantum circuit tends to generalize better on the input data and thus produce a model that, over the entire dataset, manages to come closer to the ARPA measurements.
In more detail, the reasons for this quantum advantage in achieving a higher level of accuracy than its classical equivalent may be different. Quantum variational circuitry exploits phenomena such as entanglement and superposition, which can offer new ways to explore parameter space and to find nonlinear correlations and complex patterns in data that might be difficult to model with classical neural networks.
In addition, VQCs, due to their quantum nature, are capable of manipulating information exponentially more densely than classical systems. This means that they can theoretically represent much larger solution spaces with fewer resources, thus being able to better capture the complexity of given datasets.

We move on to analyze the two recurrent networks: LSTM and QLSTM. The lowest loss value, and thus the best result in terms of accuracy, is still achieved by the LSTM model. This behavior introduced curiosity and therefore we went deeper to find out the reasons.
It was realized that the comparison between the two models, despite the fact that the portions of the dataset on which the training was conducted were identical, was not entirely fair. In fact, the value of some hyperparameters in common of LSTM are completely different from those of QLSTM. This leads to noncomparability between them, as some hyperparameters of LSTM require a great computational effort because the recurrent network that is created is large and can afford excellent performance results. On the other hand, the fact that quantum hardware technology is still in the experimental stage and the simulation of these algorithms can take place on machines that do not support an arbitrary number of hyperparameters (e.g., an arbitrary number of qubits), does not allow the creation of quantum circuits that can be "as complex as desired."

It was therefore decided to train a new LSTM model again, this time with the same hyperparameters as its quantum equivalent. We will call this model

"depowered LSTM" (LSTM-d) for the purpose of recognizing it in the following paragraphs.
A new session of hyperparameters tuning was carried out for the purpose of comparing again the two models in question.

Hyperparameter	**LSTM-d**	**QLSTM**
Training-size	70%	70%
N° epochs	300	400
Learning rate	0.001	0.01
Optimizer	RMSprop	Adam
Criterion	L1	L1
T (hours used to calibrate)	3	5
Hidden size l1	15	15
N° layers	2	7
N° qubits	-	5
Ansatz	-	strongly
Loss on train-set	5.0287	5.2021
Loss on test-set	2.7684	2.6978

Table 4.10: The comparison of the results obtained between the depowered LSTM model in terms of hyperparameters values (LSTM-d) and the QLSTM model.

The benchmark of the L1 loss function using 30% of the dataset as test-set is 5.0295. Both models in Table 4.10, looking at their loss value in the test-set, turn out to be almost half of the benchmark.
It is observed that, for the first time, the accuracy of the QLSTM model, in terms of loss function minimization, is higher than its classical equivalent. This means that for the first time, using two models with the same configuration space of hyperparameters and with the only difference in the type of network architecture, on a portion of the dataset hidden during training, the QLSTM quantum model performed better than LSTM-d.

We conclude this analysis by visualizing the performance results between the LSTM-d and QLSTM models, recalculating the final accuracy of both.

As a matter of simplicity in visualization, the same time range used in Figure 4.17 was chosen as the one used in Figure 4.16. This shows that the two models maintain almost the same behavior toward the values measured by the reference station. We show the accuracy values numerically, calculating the RMSE loss on the calibration output of the two models compared with ARPA.
In order to show the accuracy levels numerically, it was chosen to carry out an

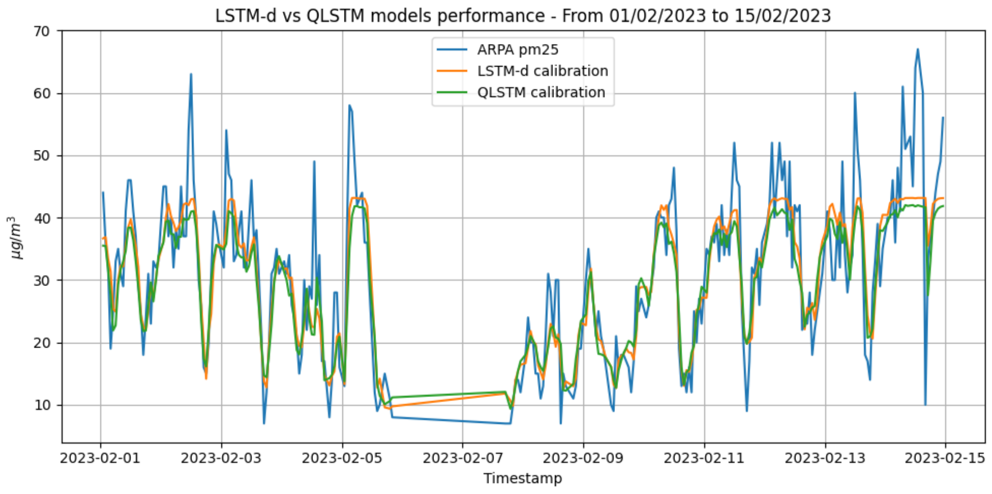


Figure 4.17: The calibration performance obtained from the LSTM-d and QLSTM models compared with ARPA values, observing a time interval between February 1 and 15, 2023.

additional cross-validation between QLSTM and LSTM-d and to extract the average of the loss function values on the different test-sets used in each fold. The number of folds used for each model remains constant at $K = 5$.

Cross-validation	**Fold-1**	**Fold-2**	**Fold-3**	**Fold-4**	**Fold-5**
LSTM-d	4.4372	7.2593	5.5691	3.5955	2.5004
QLSTM	4.3070	7.7299	5.4600	3.7013	2.3611

Table 4.11: The cross-validation result of the LSTM-d and QLSTM models.

We draw the cross-validation results between LSTM-d and QLSTM in order to understand which model turns out to be the best for calibration purposes.

The results obtained on the accuracy of the LSTM-d and QLSTM models, declare the latter as the best. Despite this, it is easy to see that the difference in accuracy between the two models is very low.
Let us therefore perform a further analysis of these models, no longer from a point of view of accuracy levels, but from a point of view of generalization on the available dataset. Understanding whether a model has learned to generalize over the dataset rather than learned by rote can be an additional discriminator that helps to understand which of the two can be considered the "best" in terms of solving the regression problem that this thesis aims to address.

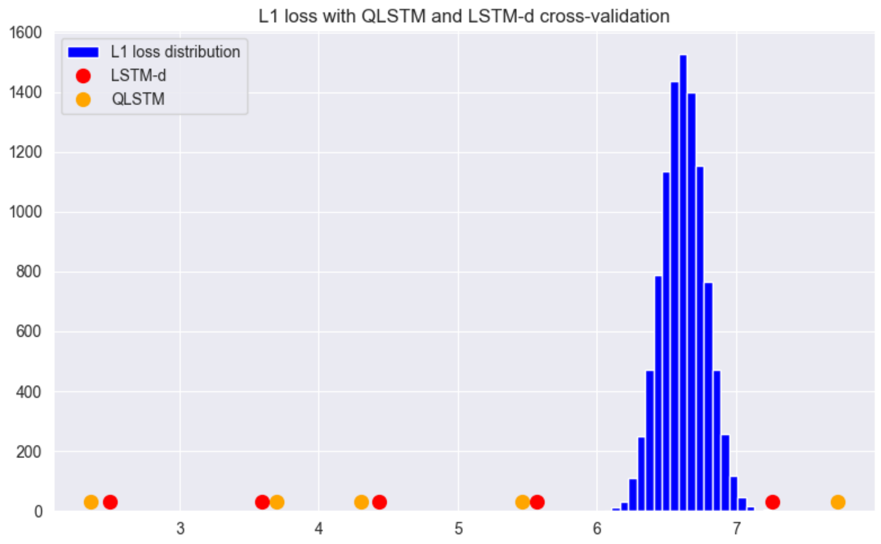


Figure 4.18: The cross-validation result of the LSTM-d and QLSTM models compared to their benchmark.

Let us start with the weights used within the LSTM architecture. The weights of a neural network are the parameters that the network learns during its training in order to minimize the loss function and therefore increase the accuracy of the model. The LSTM-d model uses a total of 482 weights, while the QLSTM model uses only 66: we are talking about a number of weights approximately seven times greater. The parts of the LSTM cell that contribute to the total calculation of the weights involved are: the bias and weights of the fully connected layer, the bias and weights of the input gate, the bias and weights of the hidden state and the number of recurrent layers.

Having models that achieve similar levels of accuracy but with a clear difference in the number of weights used leads to drawing some conclusions, such as:

- **Problem Complexity and Overfitting:** It's possible that the intricacy offered by the 482 weight model is not necessary for the problem we are seeking to solve. This might mean that overfitting will not occur and the smaller model (QLSTM) can adequately capture the problem's structure. Because of its propensity to overfit the training set, the bigger model may not yield appreciable improvements in accuracy even with a high number of weights.

- **Weight Efficiency:** Given the low number of weights but similar accuracy, the QLSTM model is able to assign each weight a more significant "role" in determining the calibration output, while in the LSTM-d model, there may be redundant or less influential weights.

- **Generalization ability:** It's possible that the model with the fewest weights can generalise to fresh data more effectively. It's possible that less complex models with fewer parameters have a better generalisation ability than more complicated ones.

This relationship between accuracy and weights used by the model highlights the importance of considering not only the performance of a model but also its efficiency and ability to generalize.

For this reason we conclude this analysis by stating that, with the same loss function, the QLSTM model achieves lower results, therefore greater calibration precision on the test-set data. The fact that this model achieves the aforementioned results using fewer weights than LSTM-d means that it has a more adequate capacity for the specific problem, indicating that the additional complexity of the LSTM-d model, which uses more weights within it, does not lead to improvements in learning the underlying structure of data.

Chapter 5

Conclusions and future works

5.1 Conclusions

The innovative journey of this thesis through the realms of Quantum Machine Learning (QML) and Deep Learning (DL) algorithms for the calibration of low-cost optical fine-dust sensors has illuminated the intricate tapestry of computational and environmental science. Through rigorous theoretical examination and empirical evaluation, this work has not only benchmarked the capabilities of these cutting-edge technologies but also spotlighted their potential and limitations in environmental monitoring, particularly in the calibration of fine-dust sensors.

The comparative analysis revealed that QML, despite its nascent stage and hardware limitations, demonstrates a promising efficiency and potential for future scalability. It suggests a paradigm shift in computational approaches to environmental challenges, particularly in sensor calibration, where QML models could offer novel insights and methodologies. Conversely, the robustness and current technological support for DL algorithms underscore their immediate applicability and effectiveness in addressing the calibration problem, showcasing a practical path towards improving air quality monitoring.

This thesis underscores the significance of embracing both traditional and quantum computational paradigms to advance our capabilities in environmental monitoring. It highlights the importance of interdisciplinary research and collaboration in tackling complex global challenges, paving the way for innovative solutions that leverage the strengths of both quantum and classical computing.
Looking forward, the convergence of QML and DL in environmental science promises

not only enhanced accuracy and efficiency in sensor calibration but also a broader implication for smart city initiatives and sustainable urban management. This work lays a foundational stone for future research avenues, inviting exploration into more complex models, larger datasets, and integration into real-world applications.

In essence, the journey from quantum quirks to clear skies is mapped out through the lens of this thesis, charting a course towards a future where technology and environmental stewardship converge for the greater good. As we stand at the crossroads of computational innovation and environmental sustainability, the insights garnered from this comparative study of QML and DL algorithms offer a beacon of hope and a guide for future endeavors in making our world a better place.

5.2 Further improvements

The exploration of quantum machine learning (QML) and deep learning (DL) algorithms in the calibration of low-cost optical fine-dust sensors presents a fertile ground for future research. The promising results obtained so far encourage the pursuit of several potential avenues to further enhance model performance, extend applicability, and refine calibration processes. This chapter outlines proposed directions for future work, emphasizing both the exploration of novel models and the application of advanced techniques to address the calibration challenge more effectively.

Application of the Long Short-Term Attention (LSTA) Model.
A significant limitation of many current models, including Long Short-Term Memory (LSTM) networks, is their dependence on the chronological order of data samples within the dataset. This requirement restricts the usability of datasets that contain samples fragmented over various points in time without maintaining linearity. To overcome this limitation, the adoption of the Long Short-Term Attention (LSTA) model [21] is proposed. Leveraging an attention matrix, the LSTA model is designed to achieve comparable results to LSTM with non-contiguous data. This approach allows for the flexible use of datasets that encompass samples collected in a non-linear fashion, potentially enhancing the model's ability to learn from a wider array of data points and improving calibration accuracy.

Model Optimization for Individual Sensors.
The proposed methodologies focus on calibrating sensors based on aggregate metrics or the median performance across all sensors. A more tailored approach is recommended for future research, wherein the best configurations of the studied

models are applied to each sensor individually. By optimizing models for the specific characteristics of each sensor, it is possible to achieve more accurate and customized calibrations. This sensor-specific strategy could significantly improve the precision and reliability of calibrations, ensuring that each sensor operates optimally within its unique environmental conditions.

Use of Neural Architecture Search (NAS) Algorithms.
Finally, the exploration of Neural Architecture Search (NAS) algorithms [22] presents a promising direction for future work. NAS algorithms automate the process of identifying the most effective neural network architecture for a given problem. Applying NAS in the context of calibrating low-cost optical fine-dust sensors could uncover novel and more efficient network architectures that are better suited to the task. This approach not only has the potential to enhance model performance but also to significantly accelerate the development cycle by automating the trial-and-error process traditionally associated with model architecture selection.

These prospective research approaches provide a path forward for using deep learning and quantum machine learning to sensor calibration. Through the exploration of these domains, scholars might expand upon the framework established by this dissertation to enhance calibration techniques, elevate model efficacy, and augment the durability and precision of inexpensive optical fine-dust sensors.

Bibliography

[1] 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. 2, 42, 45, 56, 63).

[2] Pádraig Cunningham, Matthieu Cord, and Sarah Jane Delany. «Supervised Learning». In: *Machine Learning Techniques for Multimedia: Case Studies on Organization and Retrieval.* Ed. by Matthieu Cord and Pádraig Cunningham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 21–49. ISBN: 978-3-540-75171-7. DOI: `10.1007/978-3-540-75171-7_2`. URL: `https://doi.org/10.1007/978-3-540-75171-7_2` (cit. on p. 8).

[3] H.B. Barlow. «Unsupervised Learning». In: *Neural Computation* 1.3 (Sept. 1989), pp. 295–311. ISSN: 0899-7667. DOI: `10.1162/neco.1989.1.3.295`. eprint: `https://direct.mit.edu/neco/article-pdf/1/3/295/811863/neco.1989.1.3.295.pdf`. URL: `https://doi.org/10.1162/neco.1989.1.3.295` (cit. on p. 10).

[4] Wang Qiang and Zhan Zhongli. «Reinforcement learning model, algorithms and its application». In: *2011 International Conference on Mechatronic Science, Electric Engineering and Computer (MEC).* 2011, pp. 1143–1146. DOI: `10.1109/MEC.2011.6025669` (cit. on p. 11).

[5] Sebastian Ruder. *An overview of gradient descent optimization algorithms.* 2017. arXiv: `1609.04747 [cs.LG]` (cit. on p. 17).

[6] Alex Sherstinsky. «Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network». In: *Physica D: Nonlinear Phenomena* 404 (2020), p. 132306. ISSN: 0167-2789. DOI: `https://doi.org/10.1016/j.physd.2019.132306`. URL: `https://www.sciencedirect.com/science/article/pii/S0167278919305974` (cit. on pp. 20–22).

[7] Sushmita Poudel. *Recurrent Neural Network (RNN) Architecture Explained.* `https://medium.com/@poudelsushmita878/recurrent-neural-network-rnn-architecture-explained-1d69560541ef`. 2023 (cit. on p. 21).

[8] M. H. Diskin. «Definition and Uses of the Linear Regression Model». In: *Water Resources Research* 6.6 (1970), pp. 1668–1673. DOI: `https://doi.org/10.1029/WR006i006p01668`. eprint: `https://agupubs.onlinelibrary.wiley.com/doi/pdf/10.1029/WR006i006p01668`. URL: `https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/WR006i006p01668` (cit. on p. 24).

[9] Kristy Carpenter, David Cohen, Juliet Jarrell, and Xudong Huang. «Deep learning and virtual drug screening». In: *Future Medicinal Chemistry* 10 (Oct. 2018). DOI: `10.4155/fmc-2018-0314` (cit. on p. 27).

[10] Eleanor Rieffel and Wolfgang Polak. «An introduction to quantum computing for non-physicists». In: *ACM Comput. Surv.* 32.3 (Sept. 2000), pp. 300–335. ISSN: 0360-0300. DOI: `10.1145/367701.367709`. URL: `https://doi.org/10.1145/367701.367709` (cit. on p. 31).

[11] Peter W Shor. «Introduction to quantum algorithms». In: *Proceedings of Symposia in Applied Mathematics*. Vol. 58. 2002, pp. 143–160 (cit. on p. 31).

[12] G. L. Long. «Grover algorithm with zero theoretical failure rate». In: *Phys. Rev. A* 64 (2 July 2001), p. 022307. DOI: `10.1103/PhysRevA.64.022307`. URL: `https://link.aps.org/doi/10.1103/PhysRevA.64.022307` (cit. on p. 31).

[13] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. «Quantum machine learning». In: *Nature* 549.7671 (Sept. 2017), pp. 195–202. ISSN: 1476-4687. DOI: `10.1038/nature23474`. URL: `https://doi.org/10.1038/nature23474` (cit. on p. 35).

[14] M. Cerezo et al. «Variational quantum algorithms». In: *Nature Reviews Physics* 3.9 (Sept. 2021), pp. 625–644. ISSN: 2522-5820. DOI: `10.1038/s42254-021-00348-9`. URL: `https://doi.org/10.1038/s42254-021-00348-9` (cit. on pp. 38–40).

[15] GUSTAVO ADOLFO RAMIREZ ESPINOSA. «IoT for Urban Sustainability in Smart Cities». In: (2023). DOI: `https://hdl.handle.net/11583/2983715` (cit. on pp. 42, 63).

[16] R. Artusi, P. Verderio, and E. Marubini. «Bravais-Pearson and Spearman Correlation Coefficients: Meaning, Test of Hypothesis and Confidence Interval». In: *The International Journal of Biological Markers* 17.2 (2002). PMID: 12113584, pp. 148–151. DOI: `10.1177/172460080201700213`. eprint: `https://doi.org/10.1177/172460080201700213`. URL: `https://doi.org/10.1177/172460080201700213` (cit. on p. 52).

[17] «Manuscript in preparation». In: () (cit. on p. 63).

[18] Andrea Ceschini, Antonello Rosato, and Massimo Panella. «Design of an LSTM Cell on a Quantum Hardware». In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 69.3 (2022), pp. 1822–1826. DOI: `10.1109/TCSII.2021.3126204` (cit. on pp. 63, 77, 78).

[19] Maria Schuld, Ryan Sweke, and Johannes Jakob Meyer. «Effect of data encoding on the expressive power of variational quantum-machine-learning models». In: *Phys. Rev. A* 103 (3 Mar. 2021), p. 032430. DOI: `10.1103/PhysRevA.103.032430`. URL: `https://link.aps.org/doi/10.1103/PhysRevA.103.032430` (cit. on pp. 73, 74).

[20] Robert L Harrison. «Introduction to monte carlo simulation». In: *AIP conference proceedings*. Vol. 1204. NIH Public Access. 2010, p. 17 (cit. on p. 96).

[21] Swathikiran Sudhakaran, Sergio Escalera, and Oswald Lanz. «LSTA: Long Short-Term Attention for Egocentric Action Recognition». In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2019 (cit. on p. 112).

[22] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. «A Comprehensive Survey of Neural Architecture Search: Challenges and Solutions». In: *ACM Comput. Surv.* 54.4 (May 2021). ISSN: 0360-0300. DOI: `10.1145/3447582`. URL: `https://doi.org/10.1145/3447582` (cit. on p. 113).