MASTER IN HIGH PERFORMANCE COMPUTING

Design and Implementation of a Prediction-Serving System for Runtime and Parallel Performance in Quantum ESPRESSO

Supervisor(s):
FABIO AFFINITO,
STEFANO BARONI,
STEFANO DE GIRONCOLI,
PIETRO DELUGAS

Candidate:
Mandana SAFARI

9th EDITION
2022–2023

SUMMARY

Ab initio simulations, such as those performed with QUANTUM ESPRESSO (QE), play a central role in materials science but are often limited by their high computational cost. Predicting the execution time of self-consistent field (SCF) iterations is particularly challenging, as performance depends on both the physical characteristics of the simulated system and the parallelization parameters of the underlying hardware.

This thesis investigates the use of machine learning (ML) techniques to predict the time required per SCF iteration directly from QE inputs, pseudopotentials, and computational settings. A complete workflow was designed to process raw benchmarking data into structured datasets, evaluate multiple regression approaches, and integrate the trained models into a web-based prediction-serving system.

Among the tested models, Random Forests achieved the highest overall predictive accuracy and interpretability, revealing that the number of Kohn–Sham states, total cores, and electrons are the most influential factors affecting runtime. Fully Connected Neural Networks showed comparable performance and offered smooth, consistent predictions across a wide range of execution times. Simpler models, such as Kernel Ridge Regression and Linear Regression, provided useful baselines for comparison but were less effective in capturing nonlinear dependencies.

Beyond model evaluation, a practical web interface was developed to make runtime prediction accessible to users in real time. By uploading QE input files and specifying hardware configurations, users can obtain immediate predictions of computational cost, supporting more informed resource allocation and efficient planning of large-scale simulations.

Overall, this work demonstrates how data-driven approaches can complement traditional performance modeling in high-performance computing. By combining interpretability, predictive accuracy, and real-world deployment, the developed system represents a step toward intelligent, ML-assisted simulation workflows.

ACKNOWLEDGEMENTS

Foremost, I would like to express my heartfelt gratitude to my advisors, Dr. Fabio Affinito, Prof. Stefano Baroni, Prof. Stefano de Gironcoli, and Dr. Pietro Delugas, for their continuous support, guidance, and valuable insights throughout this journey, which have been crucial to the completion of this thesis. Their encouragement, patience, and enthusiasm have been instrumental in shaping my research and academic growth.

I am also sincerely thankful to Ivan, the co-director, and Irina, the coordinator of the MHPC program, for their efforts in organizing the training courses.

I would also like to thank my friends in the MHPC program: Alfredo, Andrea, Daniel, Iara, Fernando, Keshvi, Massimo, and Neeraj, for their support and for the many enlightening scientific discussions we shared.

I remain deeply grateful to my friends in Iran, and in particular to Rayehe, for the encouraging and supportive moments of our 18-year friendship.

Above all, I wish to thank my family: my parents, my sister, and my brother, for their constant love and spiritual support throughout my life.

Last but certainly not least, I would like to thank Mohammad for his kindness, love, and support, and for being by my side through difficult times.

To my beloved spouse, Mohammad, who supports me unconditionally.

CONTENTS

1 Introduction: a general overview 1
  1.1 Motivation 1
  1.2 Problem 1
  1.3 Opportunity 2
  1.4 Contributions 2
2 Scientific Background 4
  2.1 Quantum ESPRESSO and the Kohn–Sham Cycle 4
  2.2 Machine Learning for Regression 6
    2.2.1 Linear Regression 6
    2.2.2 Kernel Ridge Regression 7
    2.2.3 Random Forest Regression 8
    2.2.4 Fully Connected Neural Network 9
  2.3 Performance Modeling in HPC 10
    2.3.1 Analytical modeling (traditional) 10
    2.3.2 Data—driven modeling (modern) 11
  2.4 Prediction-Serving Systems 12
3 Data Collection and Preprocessing 16
  3.1 The Role of Data in Machine Learning 16
  3.2 Data Collection Framework 17
    3.2.1 Automated Workflows with AiiDA 17
    3.2.2 Performance Benchmarking with JUBE 19
  3.3 Data Preprocessing 22
4 Machine Learning Models 24
  4.1 Performance Comparison 24
5 Prediction-Serving System 30
  5.1 Definition and Context 30
  5.2 Architecture of the System 30
    5.2.1 Input and Feature Extraction 30
    5.2.2 Machine Learning Model 31
    5.2.3 Prediction Output 32
    5.2.4 Web Interface 32
  5.3 Deployment 34
  5.4 Future Perspectives for the Prediction-Serving System 35
6 Conclusions and Future Work 36
A Machine Learning Model 39
  A.1 Utility Functions 39
  A.2 Model Definition 40
  A.3 Training and Prediction 41
B Prediction Serving System 43

- B.1 Flask Application Setup 43
- B.2 Prediction Endpoint 43
- B.3 HTML Template (Simplified) 50

C HPC and Cloud Integration 55
- C.1 Training on HPC with SLURM 55
- C.2 Deployment on ADAcloud 56
  - C.2.1 Systemd Service 56
  - C.2.2 NGINX Reverse Proxy 56
- C.3 Integration Workflow 57

LIST OF FIGURES

Figure 2.1 Schematic SCF cycle workflow in Quantum ESPRESSO. 5
Figure 2.2 Architecture of a prediction-serving system (based on ref. [26]). 13
Figure 3.1 Evolution of allocated, idle, and total nodes over the measurement period. 18
Figure 3.2 Distribution of Quantum ESPRESSO job outcomes. "too much RAM communication" refers to jobs terminated due to *insufficient memory in data exchange*. 19
Figure 3.3 Visualisation of benchmark structures used in JUBE, including water, nickel, perovskite, quartz, and gold systems, each tested at multiple scales and parallel setups. 21
Figure 4.1 Comparison of prediction performance across models using RMSE (left axis, blue bars) and R^2 (right axis, orange bars). 25
Figure 4.2 Comparison of real and predicted normalized time per call across models. 26
Figure 4.3 Feature importance for the Random Forest regression model. 28
Figure 5.1 Prediction-serving system web interface for end-users. 35

1 INTRODUCTION: A GENERAL OVERVIEW

Prediction is very difficult, especially if it's about the future.

—-Niels Bohr

1.1 MOTIVATION

First-principles simulations, such as those based on density functional theory (DFT), have become essential tools in materials science, chemistry, and condensed matter physics. Among the widely used software packages in this domain, QUANTUM ESPRESSO (QE) is a reference code for plane-wave-based electronic structure calculations. Despite its widespread adoption, the computational cost of such simulations is often high, especially for large systems or when high accuracy is required. Running a self-consistent field (SCF) cycle can take from minutes to several hours, depending on the input parameters and the computational resources allocated.

For researchers and HPC users, this variability poses a challenge: estimating the time required for each iteration or full calculation is difficult, yet such estimates are crucial for efficient scheduling, job submission, and resource allocation on supercomputers. Poor predictions often lead to wasted computational time, longer queue waiting times, and inefficient usage of HPC resources and energy.

1.2 PROBLEM

Traditionally, performance modeling in HPC applications has relied on analytical or semi-analytical methods. In the case of electronic-structure codes, the runtime of a single iteration in the Kohn–Sham (KS) procedure can be approximated by modeling the dominant computational kernels (e.g., diagonalization, FFTs, linear algebra routines) and their scaling with hardware characteristics such as FLOP/s, memory bandwidth, and interconnect latency. While such approaches can provide valuable insights, they often require

deep code-specific knowledge, complex performance modeling, and significant calibration effort for each new architecture.

From a user's perspective, however, a more direct approach is desirable: given an input file and a target HPC system, how long will each SCF iteration take, and how should parallelization parameters (e.g., number of MPI tasks, threads per node, pools) be chosen for optimal performance? This is a non-trivial problem, as runtime depends on a combination of physical system properties (e.g., number of electrons, basis set size) and hardware-specific configurations.

1.3 OPPORTUNITY

Machine learning (ML) offers a promising alternative to analytical performance modeling. Instead of deriving kernel-level cost functions manually, one can train ML models directly on benchmarking data to learn the mapping between input features (physical, numerical, and parallelization parameters) and performance metrics (time per SCF iteration). Such an approach enables data-driven prediction, potentially reducing the need for expert knowledge of code internals.

Moreover, once trained, these models can be deployed in a prediction-serving system, where users provide input parameters and receive real-time predictions of runtime and efficiency. This capability can accelerate scientific discovery by helping users select optimal simulation setups, minimize trial-and-error in job submissions, and improve HPC resource utilization.

1.4 CONTRIBUTIONS

In this thesis, we propose and implement a prediction-serving system for QE, with a focus on runtime prediction at the level of individual SCF iterations and on the analysis of parallelization parameters. The main contributions of this work are:

1. Data preprocessing and feature engineering: We implemented a reproducible pipeline that converts raw QE outputs and hardware information into structured datasets. This pipeline extracts both physical descriptors (e.g., number of electrons, number of bands, number of k-points) and parallelization parameters (e.g., number of cores, nodes, pools), and normalizes them into a consistent input format for ML models.

2. ML models for time prediction: We evaluated multiple regression approaches to predict the time per SCF iteration. These include linear regression (LR), kernel ridge regression

(KRR), random forest regression (RF), and a fully connected neural network (FCNN) implemented in TensorFlow/Keras. We analyzed the accuracy, generalization, and robustness of their models, and assessed the trade-offs between simple models (i.e., LR, KRR) and more complex ones (i.e., RF, FCNN).

3 Prediction-serving system for HPC users: We designed and deployed a prediction-serving system that allows users to request models for runtime predictions in real time. The system takes user-provided QE input, pseudopotentials, and parallelization parameters, runs inference through the trained ML models, and serves the predictions via a lightweight web interface. This bridges the gap between offline ML model training and practical use in production HPC environments.

4 Feature importance and interpretability: We investigated feature relevance using model-specific techniques (e.g., regression coefficients for LR/KRR, feature importance for RF). This analysis provides physical insights into how problem size and parallelization choices influence runtime, offering guidance for resource allocation and scheduling on HPC systems.

Structure of the Thesis

Chapter 2 reviews the scientific background of QE, ML methods for performance prediction, along with training details, HPC performance modeling, and prediction serving system introduction (2).

Chapter 3 describes the dataset collection, preprocessing, and feature engineering pipeline (3).

Chapter 4 Compares the ML models used in this study, evaluation metrics, and analyzes feature importance (4).

Chapter 5 explains the prediction-serving system implementation and deployment (5).

Chapter 6 concludes the thesis and outlines directions for future research (6).

2 SCIENTIFIC BACKGROUND

If I have seen further, it is by standing on the shoulders of giants.

—-Isaac Newton

2.1 QUANTUM ESPRESSO AND THE KOHN–SHAM CYCLE

Quantum ESPRESSO is a widely used open-source suite for electronic-structure simulations based on density functional theory (DFT) and plane-wave pseudopotentials [14]. It is designed for massively parallel high-performance computing (HPC) environments and has become a cornerstone of computational materials science [13].

At the heart of QE lies the self-consistent field (SCF) cycle, where the Kohn–Sham equations are solved iteratively until convergence is reached. Each iteration involves several computationally demanding steps:

1. **Hamiltonian construction**: building the effective one-electron Hamiltonian using the current electron density.

2. **Matrix diagonalization**: solving large-scale eigenvalue problems to obtain Kohn–Sham eigenstates.

3. **Fast Fourier Transforms (FFTs)**: switching between reciprocal and real space to evaluate potentials and densities.

4. **Density update and convergence check**: recomputing the charge density and monitoring convergence criteria.

The schematic algorithm (workflow) of SCF iteration is shown in Figure 2.1.

The runtime of a single SCF iteration depends on a combination of physical descriptors (such as the number of atoms, valence electrons, k-points, and plane-wave cutoff) and parallelization parameters (number of nodes, MPI processes, OpenMP threads, pools of k-points, etc.). These dependencies make runtime prediction non-trivial.

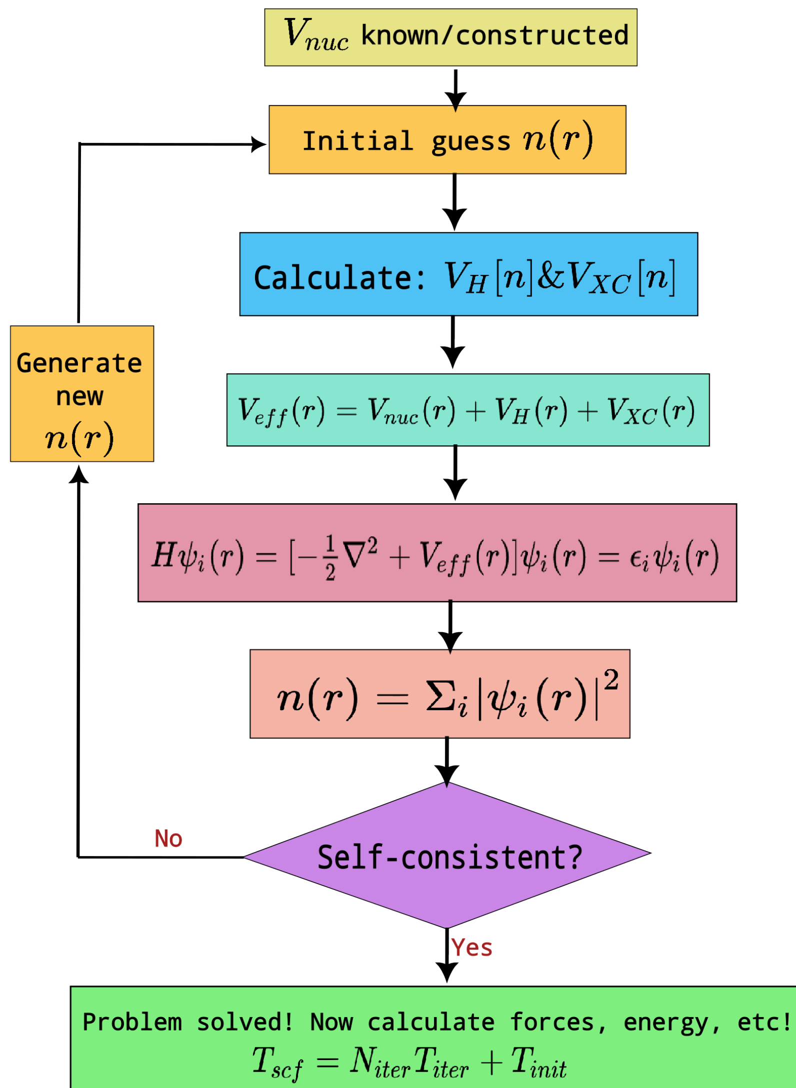

Figure 2.1: Schematic SCF cycle workflow in QUANTUM ESPRESSO.

Some of the key factors that can affect runtime are tabulated in Table 1.

For HPC practitioners, runtime estimation is particularly relevant: the efficiency of QE can vary significantly across systems and input configurations. Incorrect choices of parallel parameters may lead to poor scaling or wasted computational resources. Thus, accurate prediction of QE runtime is valuable both for users (to optimize job submission) and for schedulers (to balance workloads).

2.2 MACHINE LEARNING FOR REGRESSION

Machine learning provides a framework for approximating complex mappings between input features and continuous targets, making it suitable for runtime prediction problems. In our case, the input features correspond to both physical descriptors (e.g., electrons, k-points, plane-waves) and parallelization parameters (e.g., nodes, MPI, openMP), while the target is the runtime per SCF iteration.

Several regression algorithms have been widely applied in performance modeling:

2.2.1 Linear Regression

Linear Regression assumes a linear relationship between input features and the target variable [3, 16, 25].

$$\hat{y} = X\beta + \epsilon, \tag{2.2.1}$$

where X is the feature matrix, β is the vector of coefficients, and ϵ represents the residual error. The coefficients are estimated by minimizing the residual sum of squares:

$$\min_{\beta} \|X\beta - y\|_2^2. \tag{2.2.2}$$

This corresponds to an Ordinary Least Squares (OLS) estimator, implemented through scikit-learn's `LinearRegression` class [28]. Although simple and highly interpretable, linear regression (LR) tends to underfit the inherently nonlinear dependencies that characterize runtime behavior. In principle, the limitations of LR do not necessarily prevent its use. A classical strategy is to apply feature transformations that map the original variables into a new space where the underlying relationship with the target becomes approximately linear. Such transformations may include polynomial expansions, interaction terms, logarithmic or reciprocal mappings, or other physics-informed basis functions. Identifying an appropriate transformation, however, requires a detailed analysis of how each

feature contributes to the observed runtime, as well as a physical or empirical justification for the chosen mapping.

In this work, we did not pursue such feature-engineering efforts. Instead, we relied on more flexible models capable of capturing nonlinear relationships directly, without substantial manual intervention in the input feature space.

HYPERPARAMETERS.

- `fit_intercept`: True
- `copy_X`: True (by default)
- `n_jobs`: None (single-threaded by default)

TRAINING METHOD. The model parameters are estimated analytically using the normal equation $(X^{\top}X)^{-1}X^{\top}y$. The implementation was trained on the dataset using an 80/20 train–test split, and evaluated using cross-validation.

RESULTS. Linear Regression captures global performance trends but fails to model nonlinear interactions among computational and hardware features. It serves as a simple baseline for comparison with more expressive models.

2.2.2 Kernel Ridge Regression

Kernel Ridge Regression (KRR) extends Ridge Regression by incorporating kernel methods to model nonlinear relationships between input features and the target variable [16, 32, 33, 35]. Instead of learning coefficients in the original feature space, KRR implicitly maps the data into a high-dimensional feature space defined by a kernel function $K(\cdot, \cdot)$, allowing nonlinear dependencies to be captured without explicit feature expansion.

The model prediction is given by:

$$\hat{y}(x) = \sum_{i=1}^{N} \alpha_i \, K(x_i, x), \tag{2.2.3}$$

where x_i are training samples, α_i are dual coefficients, and $K(x_i, x)$ measures similarity between inputs. The coefficients $\boldsymbol{\alpha}$ are estimated by minimizing the regularized least-squares objective:

$$\min_{\boldsymbol{\alpha}} \|\mathbf{K}\boldsymbol{\alpha} - \mathbf{y}\|_2^2 + \lambda \, \boldsymbol{\alpha}^{\top}\mathbf{K}\boldsymbol{\alpha}, \tag{2.2.4}$$

where $\mathbf{K}$ is the kernel matrix with elements $K(x_i, x_j)$, and λ controls the regularization strength.

HYPERPARAMETERS.

- **Kernel:** Radial Basis Function (RBF)
- **Regularization parameter (α):** 1.0
- **Kernel width (γ):** Default value determined automatically by scikit-learn

TRAINING METHOD. The model was implemented using scikit-learn's `KernelRidge` class [28] with an RBF kernel and regularization parameter $\alpha = 1.0$. Model parameters were estimated analytically by solving a system of linear equations involving the kernel matrix. Training and evaluation were performed using an 80/20 train–test split.

RESULTS. KRR provides a flexible nonlinear regression approach while retaining good generalization through regularization. Compared to Linear Regression, it better captures nonlinear dependencies between computational and hardware features, yielding improved performance predictions.

2.2.3 Random Forest Regression

Random Forest is an ensemble learning method that combines multiple decision trees to improve predictive accuracy and robustness [4]. Each tree is trained on a bootstrapped subset of the data, and at each node split, a random subset of features is considered. The final prediction is obtained by averaging the outputs of all individual trees, which reduces variance and mitigates overfitting. This approach effectively partitions the feature space into regions with similar runtime behavior, making it suitable for heterogeneous and noisy datasets.

HYPERPARAMETERS.

- **`n_estimators`:** 100 (number of trees in the forest)
- **`max_depth`:** None (trees expand until leaves are pure)
- **`random_state`:** 42 (ensures reproducibility)
- **`n_jobs`:** -1 (utilizes all CPU cores for parallel training)

TRAINING METHOD. The model was implemented using scikit-learn's `RandomForestRegressor` class [28]. Each tree was trained independently on a bootstrap sample of the dataset, and the final prediction was computed as the mean of all tree outputs. An 80/20 train–test split was applied for model evaluation.

RESULTS. Random Forest Regression provides strong predictive performance and robustness to overfitting compared to single decision trees. It effectively captures nonlinear dependencies between computational and hardware parameters and performs well on small to medium-sized datasets. In performance modeling, RF offers a powerful yet interpretable ensemble baseline for nonlinear regression tasks.

2.2.4 Fully Connected Neural Network

Fully Connected Neural Networks (FCNNs) are deep feedforward architectures capable of learning complex nonlinear mappings between input features and target values [3, 15]. Unlike classical regression methods, FCNNs can approximate arbitrary nonlinear functions given sufficient depth, width, and data. In this work, a custom FCNN model was used and adapted for predicting the time-per-call of QUANTUM ESPRESSO runs.

ARCHITECTURE/ALGORITHM. The implemented FCNN was designed using TensorFlow/Keras and explicitly constructed layer by layer, rather than relying on pre-built architectures. The model consists of:

- **Input layer:** $n_{vars} = 15$ input features describing QE runs and hardware parameters.
- **Hidden layers:** Four fully connected (Dense) layers with 30, 20, 15, and 10 neurons, respectively, each using the **Swish** activation function [30].
- **Regularization:** Combined ℓ_1 and ℓ_2 weight regularization ($\lambda_1 = 10^{-5}$, $\lambda_2 = 10^{-4}$) and a Dropout layer ($p = 0.2$) to reduce overfitting [36].
- **Output layer:** A single linear neuron producing the predicted runtime per SCF iteration.

HYPERPARAMETERS.

- Activation function: Swish
- Optimizer: Adam [19] with learning rate 5×10^{-5}
- Loss function: Mean Absolute Error (MAE)
- Regularization: ℓ_1–ℓ_2 penalties and Dropout
- Batch size: 20

- Epochs: 50 (default)
- Validation split: 0.2

TRAINING METHOD. The FCNN was trained using the TensorFlow/Keras API [1]. Training employed the `fit()` function with an 80/20 train–validation split, using all available samples. The model was compiled with the Adam optimizer and MAE as both loss and evaluation metric.

NORMALIZATION AND DENORMALIZATION. To improve numerical stability and ensure balanced gradient updates, both inputs and targets were normalized internally within the model. Normalization was performed in log-space: each feature or target value was transformed as $\log(x + \epsilon)$, followed by standardization to zero mean and unit variance. This procedure stabilized training by mitigating large dynamic ranges commonly observed in raw runtime and system-size data. During inference, the model automatically performs the inverse operation, denormalizing the predicted value back to physical scale. This step ensures that the output corresponds to the real time-per-SCF iteration and is directly comparable to benchmarking data (see appendix A).

RESULTS. The FCNN model is capable of capturing highly nonlinear dependencies between computational parameters and runtime. However, it required careful regularization and hyperparameter tuning to prevent overfitting, especially on smaller datasets. The inclusion of dropout and mixed ℓ_1–ℓ_2 penalties improved generalization and stability during training.

2.3 PERFORMANCE MODELING IN HPC

Performance prediction in HPC has evolved along two distinct paradigms.

2.3.1 Analytical modeling (traditional)

Traditional approaches derive performance from first principles and simplified machine models. Representative methods include:

- Roofline analysis: bounds attainable performance by the minimum of peak FLOP/s and the product of memory bandwidth and operational intensity [43].

- Kernel-/component-based models: decompose the application into dominant kernels (e.g., BLAS xGEMM [11], FFTs [12], sparse ops, communication phases) and sum per-kernel costs using hardware parameters (FLOP/s, bandwidth, latency) and algorithmic complexity [2, 5].
- Communication models (α–β, LogP): approximate message latency and bandwidth costs for collectives and point-to-point operations [10, 40].

These methods are famous because of their lightweight, interpretability, and portability. They are considered helpful for bottleneck identification and high-level tuning.
On the dark side, they are sensitive to idealized assumptions. Also, they often miss system-specific effects (NUMA, cache behavior, contention, software stack), and require substantial code and architecture expertise to calibrate and maintain across platforms and versions.

- ***Usage in this thesis***: *Analytical models are discussed as background to situate the problem; we do not build or calibrate roofline or kernel-sum models for QE in this work.*

2.3.2 Data—driven modeling (modern)

Modern approaches learn performance directly from measurements rather than specifying it analytically. The two main modern approaches are:

- **Statistical regression**: includes linear regression, ridge regression, and kernel ridge regression (KRR). As we mentioned before, these methods provide fast, interpretable baselines when relationships are approximately linear or can be linearized with kernels [3, 15, 16].
- **Machine learning models**: such as Random Forests (ensemble trees) and FCNNs, which capture complex nonlinear interactions among physical descriptors (e.g., electrons, bands, k-points) and parallelization parameters (e.g., ranks, threads, pools). When trained on diverse data, they can generalize across hardware/software configurations [23, 37].

Generally, they automatically model complex, coupled effects without kernel-level derivations. Also, they can adapt as hardware and software evolve by retraining on new data. On the other hand, its dependency on data coverage and quality is considered a drawback. Also, extrapolation outside the training distribution is

risky. Models in this approach require validation, monitoring, and periodic retraining to remain accurate on new architectures or QE versions.
A growing body of research has investigated machine learning for performance prediction in HPC. Unlike analytical performance models (e.g., roofline or kernel-based approaches), these studies focus on learning directly from empirical measurements to capture system-specific effects such as interconnect bottlenecks, load imbalance, or software stack dependencies.

- ***Usage in this thesis***: *We adopt the data-driven paradigm: we train and evaluate LR, KRR, RF, and FCNN models on QE benchmarking datasets to predict time per SCF iteration from combined physical and parallelization features.*

2.4 PREDICTION-SERVING SYSTEMS

Machine learning in scientific computing has traditionally followed an offline workflow: data are collected from experiments or simulations, models are trained, and predictions are reported in static form (tables, plots, benchmark comparisons). While this approach has value for demonstrating feasibility, it does not provide an immediate mechanism for end users (such as scientists, engineers, or HPC practitioners) to benefit from predictions in their day-to-day workflows. As a solution, the final phase (rendering those models in live systems) is now recognized as the prediction-serving stage, suggested to developers [8].

These prediction-serving systems expose models via APIs (e.g., REST, gRPC, TensorFlow Serving, Clipper), enabling real-time inference with features such as batching, caching, low-latency execution, and model versioning [9, 21, 27].

Moreover, introducing the prediction-serving systems helps us to introduce the machine learning application workflow as a practical tool. Basically, any ML application has three phases: development, training, and inference, shown in Figure 2.2

A prediction-serving system provides the infrastructure that allows trained machine learning models to be accessed and executed in real time. Such systems expose the model through a serving layer—typically a REST API implemented with frameworks such as Flask, FastAPI, TensorFlow Serving, or TorchServe—so that users can query it dynamically with new inputs. Incoming requests are automatically preprocessed, evaluated by the model, and returned as predictions, often within milliseconds. To ensure

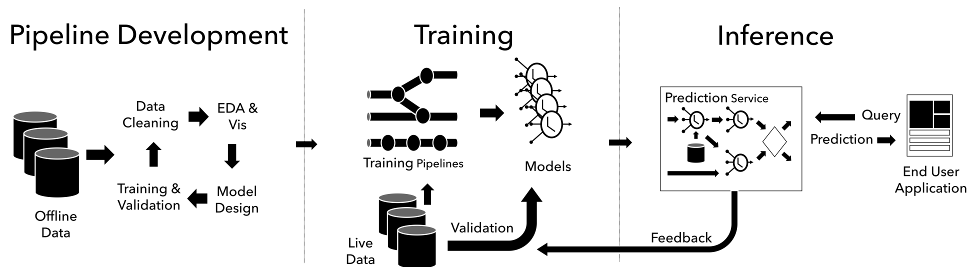

Figure 2.2: Architecture of a prediction-serving system (based on ref. [26]).

responsiveness and scalability, prediction-serving architectures may incorporate features such as batching, caching, GPU acceleration, and load balancing. More advanced implementations include lifecycle management capabilities, enabling model versioning, performance monitoring, and retraining can be considered as new data becomes available.

Prediction-serving has already become integral to many industrial domains, from e-commerce (recommendation and demand forecasting) and finance (fraud detection, credit scoring) to natural language processing (chatbots and large language models such as GPT or BERT). In these contexts, such systems provide low-latency inference that integrates seamlessly with user-facing applications, ensuring that machine learning models can operate continuously and reliably in production environments.

In contrast, adoption within scientific HPC remains limited. While machine learning has been successfully applied to problems such as molecular dynamics energy prediction [7, 31], turbulence modeling in computational fluid dynamics (CFD) [6], and outcomes of DFT simulations [45], these models are rarely deployed as persistent or accessible services. Instead, predictions typically remain confined to offline analysis, requiring manual intervention to reproduce and interpret results.

Within HPC workflows, however, prediction-serving systems have the potential to play a transformative role. In the context of QUANTUM ESPRESSO, for instance, such a system could provide runtime estimations before a simulation is launched. By querying the service with key input parameters—such as the number of bands, k-points, nodes, pools, or tasks per node—users can obtain an estimated time per SCF cycle and plan computational resources more efficiently.

In this thesis, we adopt this serving paradigm: machine learning models trained on QUANTUM ESPRESSO benchmarking data are integrated into a lightweight, web-based prediction system designed for runtime estimation and ease of access. This approach demonstrates how prediction-serving concepts can be effectively extended to the scientific HPC domain, bridging the gap

between machine learning inference and real-world computational performance optimization.

Table 1: Key runtime-affecting factors in QUANTUM ESPRESSO.

Category	Factors
Physical system descriptors	• Number of atoms and electrons • Number of bands • k-point mesh • Plane-wave cutoff energy • Number of plane-waves (G-vectors)
Parallelization parameters	• Number of nodes • Number of Cores • MPI processes • OpenMP threads • k-point parallelization
Algorithmic settings	• Diagonalization method. • FFT grid size • Mixing scheme • Convergence thresholds
Hardware characteristics	• CPU model and clock speed • Memory bandwidth • Interconnect (e.g., InfiniBand) • Storage I/O performance

3 DATA COLLECTION AND PREPROCESSING

Without data, you're just another person with an opinion.

— W. Edwards Deming

The foundation of any successful machine learning (ML) application lies in high-quality, well-prepared data. While contemporary research often focuses on complex model architectures and serving systems, the data collection and preprocessing stage remains a pivotal determinant of model performance. In fact, scholars increasingly advocate for a data-centric AI approach, emphasizing that even the most advanced algorithms cannot compensate for poor-quality data [41].

3.1 THE ROLE OF DATA IN MACHINE LEARNING

The importance of data in ML can be understood from three complementary perspectives: its foundational role, the recent data-centric trend, and its place in the ML application lifecycle.

- **Data Collection as a Foundation**: Data collection is the essential first step in any ML pipeline, laying the groundwork for everything that follows. Without it, modern ML systems cannot generate accurate predictions, automate tasks, or drive insights. "Garbage in, garbage out" remains a core principle, while well-collected, representative data directly correlates with better model performance [24].
- **Data-Centric AI Trend**: The field is shifting toward "data-centric AI", where the emphasis is on improving the quality and structure of datasets rather than tweaking models themselves [42].
- **The ML application Context**: Under the three-phase paradigm of ML applications (Model development, training, inference, as mentioned in chapter 2), the model development begins with gathering and preprocessing raw data, i.e., extracting features. Then it continues with selecting model architectures

(e.g., logistic regression, random forest, FCNN), and iterative hyperparameter tuning.

Together, these perspectives highlight the intrinsic importance of data in machine learning. Since this work focuses on designing a time-prediction application (a specific type of ML system), the present chapter is dedicated to data gathering and preprocessing within this context.

3.2 DATA COLLECTION FRAMEWORK

3.2.1 Automated Workflows with AiiDA

AiiDA (Automated Interactive Infrastructure and Database for Computational Science) is an open-source Python infrastructure designed to automate, manage, store, share, and ensure reproducibility of complex computational science workflows [18]. AiiDA's strength lies primarily in its ability to capture the complete trace and record of all data transformations, promoting full reproducibility [17]. Moreover, AiiDA supports high throughput, handling tens of thousands of processes per hour, while storing structured data in a database designed for efficient querying [18].
By leveraging AiiDA, large-scale QE calculations could be run in a structured and reproducible manner, with all inputs and outputs systematically stored.
We performed a large-scale benchmark campaign on the Leonardo-Booster cluster at the CINECA supercomputing centre. The purpose of this run was twofold: (i) to generate as much high-quality data as possible for offline training of our prediction-serving pipeline, and (ii) to stress-test the Leonardo batching system under extreme load conditions. This effort was carried out jointly by our group, the CINECA User Support team, and collaborators from the University of Trieste.

To construct the dataset for offline training of our development pipeline, we selected a diverse subset of bulk 3D materials structures from the Materials Cloud database [38]. Using an AiiDA-based workflow, we generated approximately 10,000 Quantum ESPRESSO input files with different structures and parallelisation setups. Each input corresponded to a distinct structural configuration combined with randomised—but physically meaningful—computational parameters[1]. Because this campaign had to be launched under significant time pressure, we adopted a "blind" input-generation

1 To have a dataset that reflects the effects of different parallelisation parameters, two additional samples ($MoTe_2$ and WTe_2) were considered with fixed QE inputs, and just parallelisation parameters were varied (about 20% of samples)

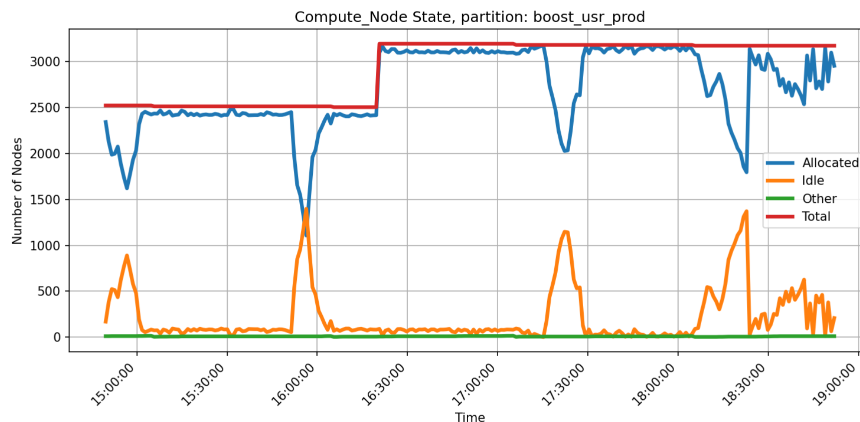

Figure 3.1: Evolution of allocated, idle, and total nodes over the measurement period.

strategy: instead of pre-screening the parallelisation parameters for each job, with analytical or ML-based models [29], we accepted the possibility of a significant failure rate. From a dataset-construction perspective, this was still considered acceptable, as even failed calculations provide useful information by indicating failed parameter combinations.
All calculations were executed on the GPU-accelerated Booster partition. For each input, we used a single batch job script with a fixed parallelisation layout based on the node configuration of Leonardo-Booster.

The overall campaign was executed for nearly five hours, during which an average of about 3,000 Booster nodes were continuously utilised. Figure 3.1 shows the cluster-level resource usage during this period, including the number of allocated, idle, and total nodes.
A detailed classification of job outcomes is shown in Fig. 3.2. Out of the 10,000 submitted runs, approximately 3,500 completed normally. Approximately 100 ended with communication-related failures, and around 1,500 reached the maximum SCF iteration limit without achieving convergence. Approximately 4,000 remaining calculations were terminated by the scheduler due to excessive memory usage during communications, where the exchange and redistribution of large data arrays exceeded the per-node memory limits. As we mentioned before, we did not employ automated resource-prediction models to tune allocations in the AiiDA workflow. Some groups at CINECA and collaborating institutions have developed ML predictors for time-to-solution (e.g., Pittino et al., PASC 2019 [29]). However, Pittino et al. compared deep-learning time predictors against a parametrised analytical performance model for runtime; we did not use that analytical model here.

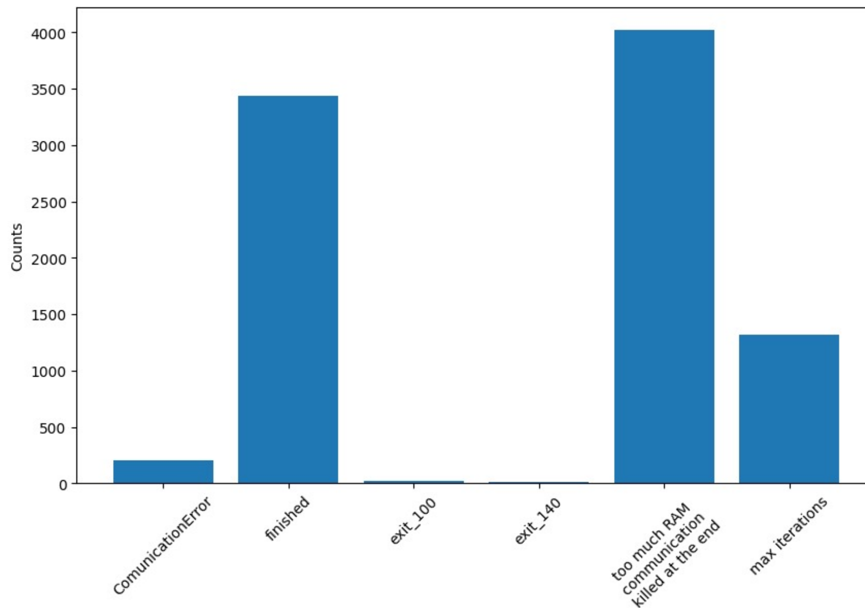

Figure 3.2: Distribution of QUANTUM ESPRESSO job outcomes. "too much RAM communication" refers to jobs terminated due to *insufficient memory in data exchange*.

3.2.2 Performance Benchmarking with JUBE

JUBE (Jülich Benchmarking Environment) [22] was utilised to add more raw data by enabling parametric job execution, systematic performance measurement, and aggregation of runtime metrics across different hardware configurations. Through JUBE, it was possible to explore the effect of varying computational parameters (e.g., number of nodes, tasks per node) on the QUANTUM ESPRESSO runtime, thereby generating the performance dataset used for training the prediction model.
JUBE uses an XML-based configuration system to define and automate benchmarking workflows. In the customised version developed and maintained at CINECA for the MaX project, the workflow is organised into four main XML components: `application.xml`, `platform.xml`, `workload.xml`, and `benchmark.xml`. The `application.xml` file defines application-specific details (such as QUANTUM ESPRESSO, SIESTA, or YAMBO) and how they are executed. The `platform.xml` file specifies the characteristics of the target supercomputing systems (e.g., Leonardo, Lumi, MareNostrum, Galileo100), including job submission commands, module environments, and scheduler settings. The `workload.xml` file contains the definitions of physical systems and their corresponding input parameters, while the `benchmark.xml` file orchestrates the overall benchmarking process, combining workloads, applications, and platforms to define the full parameter space of runs.

Parts of the workload.xml and benchmark.xml files are provided below to demonstrate how input variables and workflow logic are structured in practice. In the workload.xml file (Listing 1), variables such as the number of atoms (n_at), input/output file names, and runtime options are declared and dynamically substituted into Quantum ESPRESSO input templates. Meanwhile, the benchmark.xml file (Listing 2) coordinates these workloads with system-specific configurations defined in the platform.xml and application.xml files, generating job scripts automatically for submission through JUBE.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <parameterset name="testcaseset">
    <parameter name="n_at">5</parameter>
    <parameter name="casename">input_perovskite_$n_at</parameter>
    <parameter name="fin">$casename</parameter>
    <parameter name="dirin">inputfiles</parameter>
    <parameter name="dirout">'./out'</parameter>
    <parameter name="maxstep">3</parameter>
    ...
  </parameterset>
  <substituteset name="inputsub">
    <iofile in="$fin" out="$fin" />
    ...
</jube>
```

Listing 1: Part of workload.xml for the perovskite system in JUBE.

Similarly, the benchmark.xml file (partially shown below) defines how JUBE combines these workloads with platform and application configurations to generate job scripts and submit them automatically to the computing system.

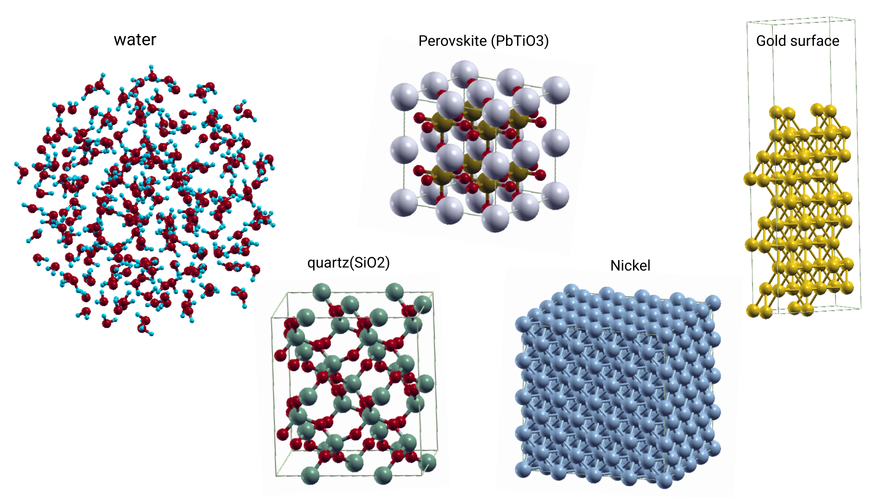

Figure 3.3: Visualisation of benchmark structures used in JUBE, including water, nickel, perovskite, quartz, and gold systems, each tested at multiple scales and parallel setups.

```
<?xml version="1.0" encoding="UTF-8"?>
<jube>
  <include>workload.xml</include>
  <parameterset name="benchmark_params">
    <parameter name="nodes">4</parameter>
    <parameter name="tasks_per_node">128</parameter>
    <parameter name="walltime">'00:30:00'</parameter>
  </parameterset>
  ...
</jube>
```

Listing 2: Part of a `benchmark.xml` demonstrating the inclusion of workload parameters.

After executing the JUBE workflow, a set of job scripts corresponding to each configuration is automatically generated and submitted to the scheduler (e.g., SLURM) through JUBE. Once all runs are completed, JUBE collects and aggregates runtime data, making it straightforward to analyze performance scaling and to construct datasets for further modeling and prediction.

The benchmarking includes representative materials of scientific and computational interest: a water cluster, bulk nickel, a perovskite ($PbTiO_3$), quartz (SiO_2), and a gold surface. While Figure 3.3 displays one representative structure for each material, the actual JUBE benchmarks covered multiple system sizes and various parallelization configurations to ensure comprehensive performance profiling.

Ultimately, using both JUBE and AiiDA, we collected 5,885 successful run samples for offline training during the development stage of the ML application.

FUTURE IMPROVEMENTS Although this dataset already provides strong predictive performance, several directions could further enhance the model. One possibility is to incorporate failed runs into the training set. This would allow the model not only to predict performance but also to identify combinations of computational parameters that are likely to lead to failures.

Another promising direction is to enrich the dataset with additional hardware descriptors. Expanding the diversity of hardware configurations—potentially by integrating results contributed by external users worldwide—would support the development of a more general and hardware-aware model. To enable this, a useful long-term improvement would be extending Quantum ESPRESSO so that its XML output includes more comprehensive hardware metadata, thereby providing the necessary inputs for models that leverage both computational and hardware features.

3.3 DATA PREPROCESSING

Raw data obtained from AiiDA and JUBE needed to be transformed into a consistent, structured format suitable for machine learning. The following steps were performed:

1. **Cleaning and Filtering:** Incomplete or failed runs were excluded, duplicate configurations were removed, and the files that have no smooth grid, or smooth grid equals dense grid, were omitted.

2. **Feature Engineering:** The dataset was enriched with scientifically meaningful descriptors derived from QUANTUM ESPRESSO outputs, such as the number of atoms, k-points, and convergence thresholds. System-level information, including the number of nodes and tasks per node, was also added. Derived metrics, such as the time per SCF cycle and system complexity, were computed to capture performance-relevant relationships.

3. **Adding Hardware Information as Metadata:** Hardware details were included manually for known clusters, although they can also be extracted automatically using third-party tools. For example, the IDENTIKEEP plugin was employed to retrieve such metadata during execution when available.

4. **Data Formatting and Storage:** The processed dataset was stored in a structured JSON format, ensuring compatibility with the machine learning pipeline and reproducibility by preserving both numerical values and metadata.

To recap, leveraging AiiDA for reproducible workflows and JUBE for systematic benchmarking, we constructed a rich, structured dataset on Leonardo Booster. We applied rigorous preprocessing (including cleaning, feature engineering, adding hardware metadata, and formatting) to ensure suitability for modeling.

4 MACHINE LEARNING MODELS

All models are wrong, but some are useful.

— George E. P. Box

In this chapter, we present the application and evaluation of several machine learning (ML) models for predicting the time-per-cycle of Self-Consistent Field (SCF) iterations in QUANTUM ESPRESSO. The four representative models considered are Linear Regression (LR), Kernel Ridge Regression (KRR), Random Forest (RF), and Fully Connected Neural Networks (FCNN). Detailed descriptions of these models, including their architectures, hyperparameters, training methodologies, and implementation details, are provided in Chapter 2, specifically: LR in Sections 2.2.1, KRR in 2.2.2, RF in 2.2.3, and FCNN in 2.2.4. The implementation of the custom FCNN model used in this study is provided in Appendix A.

4.1 PERFORMANCE COMPARISON

Model performance is evaluated using standard regression metrics: Mean Absolute Error (MAE), Root Mean Square Error (RMSE) [44], and the coefficient of determination (R^2). These metrics provide complementary perspectives on predictive accuracy and variance explained, allowing a comprehensive comparison across the different ML approaches. The results are summarized in Table 2.

Table 2: Performance of Different ML Models for Predicting SCF Time-per-Cycle

Model	$\downarrow$ **RMSE** ($\times 10^{-06}$)	$\downarrow$ **MAE** ($\times 10^{-07}$)	$\mathbf{R^2}$
LR	8.832	17.842	0.037
KRR	3.952	3.613	0.807
FCNN	3.850	2.712	0.817
RF	1.533	1.673	0.971

As expected, the LR model performs the weakest, with a high RMSE of 8.83×10^{-6} and a low coefficient of determination ($R^2 =$

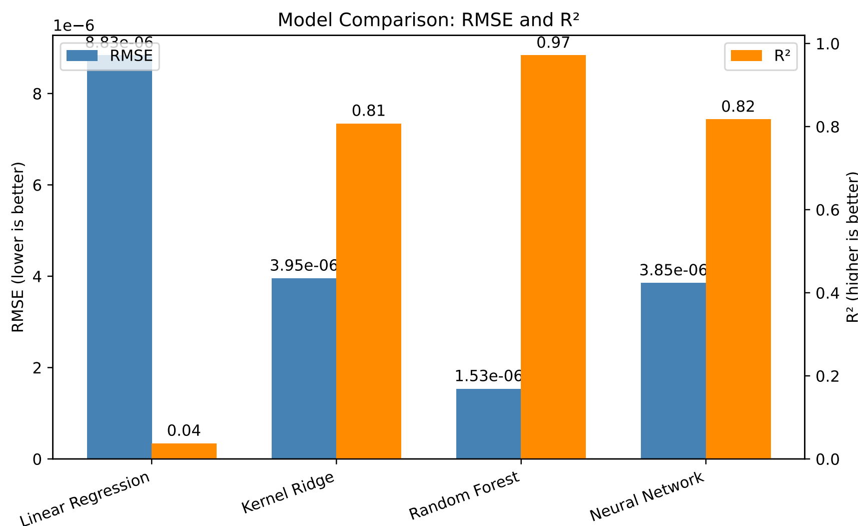

Figure 4.1: Comparison of prediction performance across models using RMSE (left axis, blue bars) and R^2 (right axis, orange bars).

0.037). This outcome aligns with the discussion in the Background chapter (Linear Regression subsection 2.2.1), where we noted that linear models can only succeed if the feature space is transformed such that the target–feature relationship becomes approximately linear. Since no such feature transformations were applied in this work, the LR model is unable to capture the complex, nonlinear dependencies that govern QUANTUM ESPRESSO runtime behavior.
In contrast, the KRR model achieves a substantial improvement by introducing nonlinear kernels, reducing both RMSE and MAE by more than half relative to LR and increasing R^2 to 0.807. This demonstrates the clear advantage of models that can natively represent nonlinear relationships without extensive manual feature engineering.
The FCNN achieves slightly better results than KRR, with marginally lower RMSE and MAE values (3.85×10^{-6} and 2.71×10^{-7}, respectively) and an R^2 of 0.817. This improvement indicates that the neural network captures additional nonlinearities and complex feature interactions, though the performance gain over KRR is moderate given the higher computational and training cost.

The RF model stands out as the best-performing approach, achieving the lowest errors (RMSE $= 1.53 \times 10^{-6}$, MAE $= 1.67 \times 10^{-7}$) and the highest coefficient of determination ($R^2 = 0.971$). These results demonstrate its strong generalization capability and robustness, making it the most suitable model among those tested for predicting the time per SCF cycle from QE input parameters and hardware features.

The trends illustrated in Figure 4.1 confirm the quantitative analysis from Table 2. RF clearly dominates in both accuracy and explanatory

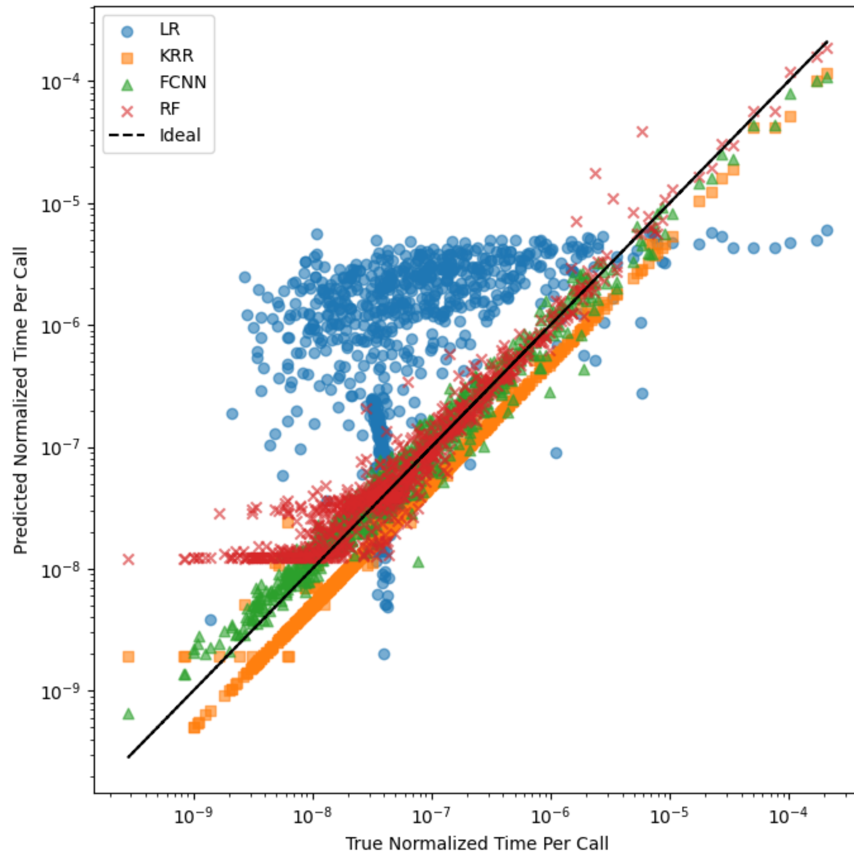

Figure 4.2: Comparison of real and predicted normalized time per call across models.

power, followed by the FCNN and KRR models, which perform comparably well but still fall short of the ensemble-based approach. LR, by contrast, fails to reproduce the underlying patterns, as reflected in its near-zero R^2. Overall, these results highlight the importance of nonlinear and ensemble learning methods in accurately modeling the complex relationships present in the benchmarking data.

To complement these metrics, scatter plots comparing the true normalized time per SCF call with the predicted values provide a more detailed view of model behavior across the parameter space (Figure 4.2).

This plot 4.2 visually demonstrates how predictions align with the ideal y=x trend. The diagonal line represents perfect predictions, while deviations indicate over- or underestimation. From these plots, several observations can be made:

- **LR** fails to reproduce the ideal trend across the full range of normalized times, particularly at both small and large values, reflecting its inability to capture the underlying non-linear dependencies.

- **KRR** generally predicts lower values than the ideal line, indicating systematic underestimation across the dataset.
- **RF** achieves the best aggregate metrics; however, at very small normalized time per call values ($\approx 10^{-8}$), predictions deviate slightly above the ideal line. This suggests that RF tends to overestimate the runtime for very fast SCF iterations.
- **FCNN** closely follows the ideal y=x line across the full range, demonstrating strong linearity and consistency in predictions, even at the extremes of normalized time per call.

These observations reinforce the numerical results while highlighting model-specific behaviors that may not be evident from RMSE or R^2 alone. In particular, although RF provides the lowest global error and highest R^2, the scatter plot shows that neural networks better preserve the fine-grained proportionality between true and predicted values for very small SCF runtimes. This is especially relevant for applications where precise estimation of short runtimes is critical, such as in fine-grained HPC scheduling or rapid preliminary simulations. In the following subsection, we go through the feature importance analysis to see why the RF deviates in small SCF runtimes.

Feature Importance Analysis

To further interpret the behavior of the Random Forest model, a feature importance analysis was performed based on the mean decrease in impurity criterion. This approach quantifies the relative contribution of each input variable to the model's predictive performance by measuring how much each feature reduces prediction error when used for decision splits across the ensemble of trees. The resulting importance values are normalized to sum to one and are shown in Figure 4.3.

The analysis reveals that the number of Kohn–Sham states (`n_ks`) is the dominant predictor of SCF time-per-cycle, followed by the total number of computational cores (`n_cores`), and the number of electrons (`n_el`). This hierarchy aligns with the underlying computational physics of Quantum ESPRESSO: The number of Kohn–Sham states directly governs the size of the Hamiltonian matrix and the number of eigenvalue problems to be solved at each SCF iteration, while the number of cores reflects the degree of parallelism and thus directly impacts execution time. The number of electrons correlates with system size and indirectly with basis-set size, contributing to computational cost. Conversely, parameters such as the number of atomic species (`n_species`) or the number of pseudopotential projectors (`n_betas`) show lower relative importance, indicating that

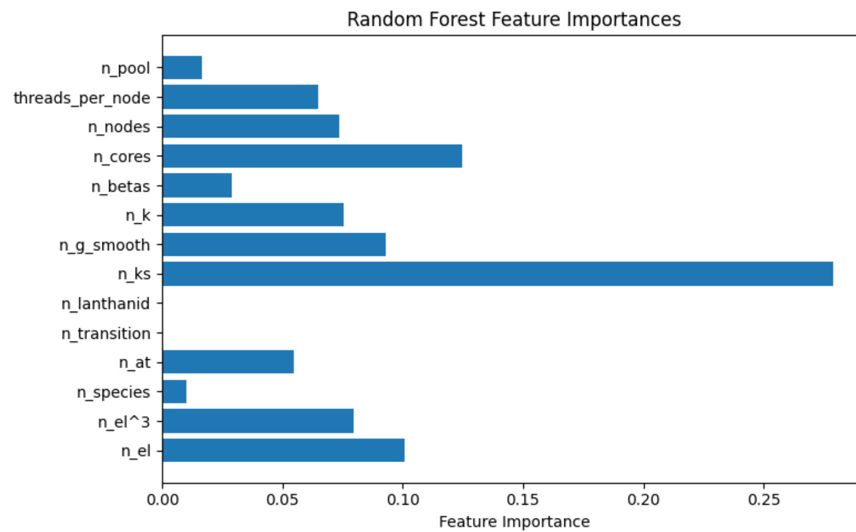

Figure 4.3: Feature importance for the Random Forest regression model.

they play a secondary role in determining runtime once the electronic and parallelization parameters are accounted for.

Overall, the feature importance distribution confirms that the Random Forest model captures physically meaningful relationships between computational input parameters and runtime behavior, providing both predictive power and interpretability.

Compared to the FCNN model—which acts as a highly flexible nonlinear approximator—the Random Forest provides an additional advantage of interpretability through feature importance, offering valuable physical insight into the dominant computational cost drivers in Quantum ESPRESSO simulations.

It is important to note that for these scatter plots, normalized time per call was used to allow comparison across heterogeneous datasets. This normalization accounts for the cubic scaling with the number of electrons, the dimensionality of the k-point mesh, and the number of cores. The Python code below shows this quantity as a column added to the pandas dataframe.

```
1  ...
2  column[complexity] = column['n_el^3'] * column['dims_nkpoints'] /
   ↪  column['n_cores']
3  ...
```

While the analysis uses normalized values for comparability, the final predictions can be transformed back to real execution times by applying this factor, ensuring that the results are meaningful for practical HPC resource allocation.

Taken together, the scatter plots complement the aggregated error metrics and provide an intuitive visualization of each model's predictive behavior. They demonstrate that, although RFs excel in global accuracy, neural networks provide the most faithful representation of runtime proportionality across the entire spectrum of SCF iteration times. This subtle understanding informs practical recommendations for model selection, suggesting that RF is preferable for overall accuracy and interpretability, whereas FCNN may be favored when precise proportionality and robustness across extreme values are required.

An important observation emerging from this study is the tension between interpretability and accuracy. Linear models and decision-tree-based methods provide insights that are immediately accessible to users and system administrators. For example, the monotonic effect of increasing the number of nodes or the saturating effect of k-point sampling is readily visible in their coefficients or feature importances. In contrast, neural networks, although significantly more accurate, act as black-box predictors, offering little in terms of direct interpretability. This interplay reflects a broader tension in the application of ML to HPC performance modeling: the choice between models that explain and models that predict. A natural approach would be to explore hybrid methods, where interpretable models provide guidance and validation, while neural networks deliver the final, high-fidelity predictions.

At the end, we can reach a general idea as: Each regression model exhibits distinct trade-offs. LR provides a simple and interpretable baseline but cannot model the nonlinear dependencies present in the data. KRR introduces flexibility through kernel functions, effectively capturing nonlinearities but at the expense of computational cost. The RF model combines robustness and interpretability, performing particularly well when the relationships between the features and the target variable are complex and non-linear. Finally, the FCNN offers high expressive power and competitive accuracy but requires careful tuning and remains less transparent than ensemble methods.

5 PREDICTION-SERVING SYSTEM

The best way to predict the future is to create it.

—- Alan Kay

5.1 DEFINITION AND CONTEXT

A prediction-serving system is a software infrastructure that enables trained ML models to produce real-time or batch predictions with low latency, high scalability, and integration capability via APIs or graphical interfaces. In other words, unlike the training phase, which is compute-intensive and often performed offline, the serving phase emphasizes low latency, scalability, and integration with external workflows [9].

In scientific and HPC applications, such systems significantly reduce the turnaround time of complex simulations. For instance, instead of executing full *ab initio* calculations, our system predicts key performance metrics (such as the time per self-consistent field (SCF) cycle) based on prior training data. This reduction of simulation runtime is our original goal.

5.2 ARCHITECTURE OF THE SYSTEM

The prediction-serving system implemented in this work follows a simple yet extensible architecture:

$$\text{Input} \rightarrow \text{Feature Extraction} \rightarrow \text{ML Model} \rightarrow \text{Prediction Output}$$

5.2.1 Input and Feature Extraction

Users provide a Quantum ESPRESSO input file together with the corresponding pseudopotentials and specify the hardware configuration (e.g., number of nodes, tasks per node, and threads). These raw inputs are automatically processed by a dedicated

feature-extraction module, which parses both the QE input and pseudopotential files to build a structured feature vector. The parser computes quantities such as: (i) number of electrons, atoms, and species, k-point mesh and symmetry reduction, grid dimensions (`n_g_smooth`) derived from energy cutoffs and lattice parameters, (ii) total valence and number of projectors extracted from UPF pseudopotentials.

The complete implementation of this parsing routine is provided in Appendix B, where the function `parse_qe_input()` and its supporting utilities are described in detail.

5.2.2 Machine Learning Model

The core of the prediction-serving system is a trained neural network that estimates the runtime per SCF iteration from the structured feature vector derived from the Quantum ESPRESSO input and pseudopotentials. Although the FCNN model is explained in Chapter 4, the key aspects of the model are summarized below for completeness:

- **Architecture:** The network consists of multiple dense layers with activation functions (e.g., Swish), regularization, and dropout to prevent overfitting. The input layer matches the dimensionality of the flattened feature vector (14 features), while the output layer predicts a single scalar value representing the runtime per SCF iteration.

- **Normalization and Denormalization:** Both input features and target values are internally normalized within the model to improve numerical stability and generalization. Specifically, the model applies a logarithmic transformation followed by standardization (zero mean and unit variance) during inference, and automatically reverses these operations (rescaling and exponentiation) to produce predictions in physical units. This ensures consistency between training and serving phases.

- **Training:** Separate models are trained for different target supercomputers (e.g., Leonardo Booster, Marconi100) to account for hardware-specific performance characteristics. Training uses the mean absolute error (MAE) as the loss function and the Adam optimizer with a tuned learning rate.

At prediction time, the backend selects the appropriate pre-trained model based on the requested supercomputer, ensuring accurate performance estimates for each hardware configuration.

5.2.3 Prediction Output

Once the neural network produces a prediction, the system returns the structured result. The predicted value, initially obtained in normalized log-space, is first denormalized within the model (as described in subsection 5.2.2 and chapter 2) to recover the physical scale of the runtime per call.

However, since the target values were also normalized by the system complexity factor, the predicted runtime must be further **denormalized to the scale of time** by multiplying by this factor, defined as explained in chapter 4. This operation converts the model output into the real runtime per SCF iteration, making it directly interpretable and comparable with benchmark data.

The backend then generates a JSON response that includes the predicted runtime in physical units together with the processed input features. Internally, the Flask server records detailed logs of parsing, preprocessing, and transformation stages, including the cleaned feature vectors and intermediate normalization steps. These logs, accessible only to developers through the server console, provide valuable support for debugging and reproducibility during model deployment and validation.
For end users, error handling is implemented through descriptive messages returned in the JSON response, notifying them of issues such as missing pseudopotentials, malformed input files, or configuration values outside the operational limits of the selected supercomputer. In this way, the system maintains transparency for users while ensuring robust traceability and maintainability on the backend.

5.2.4 Web Interface

The web interface provides a user-friendly layer for interacting with the prediction-serving system. It consists of two main components: frontend and backend.

Frontend

The frontend provides the user-facing interface of the prediction-serving system and facilitates interaction between the user and the backend services. It is implemented using Flask for server-side routing and JavaScript for client-side interactivity. Through the web interface, users can upload Quantum ESPRESSO input files along with the corresponding UPF pseudopotentials, and specify the relevant hardware configuration parameters, such as the number of nodes, tasks per node, and thread counts. Once the required information is submitted, the frontend communicates

with the backend through asynchronous HTTP requests[1], displays prediction results dynamically, and ensures a seamless and responsive user experience.

Backend Implementation

The backend is responsible for all server-side operations that support the prediction workflow. Its main functions include receiving and temporarily storing uploaded user files, validating input data according to the resource limits and configuration constraints of the selected supercomputer, and automatically selecting the appropriate prediction model.
It then parses the uploaded QUANTUM ESPRESSO input files and associated UPF pseudopotentials, building the corresponding mappings between atomic species and pseudopotentials. Subsequently, the backend invokes the feature-extraction module (`parse_qe_input()` and auxiliary routines) to transform the raw inputs into the structured numerical feature vector required by the neural network. This process includes normalization and conversion into the standardized input format expected by the trained model.
Once the feature vector is generated, the backend loads the corresponding pre-trained model, performs the prediction, and returns the results to the frontend as a structured JSON response.
A brief version of the backend prediction endpoint is provided below.

```
1  @app.route('/predict', methods=['POST'])
2  def predict():
3      model_name = request.args.get('model', 'Leonardo_Booster')
4      model_path = f"models/model_{model_name}.keras"
5      model = ann.TimePerCall.load(model_path)
6      input_file = request.files['input_file']
7      pseudos = request.files.getlist('pseudos')
8      tmpdir = "./tmp"; os.makedirs(tmpdir, exist_ok=True)
9      input_path = os.path.join(tmpdir, input_file.filename)
10     input_file.save(input_path)
11     pseudo_map = {}
12     for pf in pseudos:
13         path = os.path.join(tmpdir, pf.filename)
14         pf.save(path)
15         val, nproj = parse_upf(path)
16         pseudo_map[pf.filename] = {"valence": val, "nproj": nproj,
           ↪  "path": path}
17         ...
```

1 This “asynchronous” behavior is what makes the web interface feel smooth and interactive, rather than laggy or frozen while waiting for the server. It’s a core concept in modern web development used by frameworks like plain JavaScript with fetch() calls (also used here).

```
1   # --- Extract and flatten features ---
2   parsed = parse_qe_input(input_path, pseudo_map=pseudo_map,
    ↪  pseudos_dir=tmpdir)
3   data = flatten_for_nn(parsed, extra_inputs=request.form.to_dict())
4
5   df = pd.DataFrame([data]).astype(float)
6   prediction = model.predict_normed(df)
7   return jsonify({"prediction_time": round(float(prediction[0][0]),
    ↪  10)})
```

The complete implementation of the backend is detailed in Appendix B.

5.3 DEPLOYMENT

The system was deployed using a combination of open-source and cloud-compatible technologies designed for scalable and efficient deployment on cloud infrastructure[2].

ADA Cloud served as the hosting platform, providing secure HTTPS access through NGINX configuration.

The average response time of the system is negligible with respect to the average time of a typical SCF run by QE. The system also achieved a high level of usability by offering a simple web-based form, reducing the entry barrier for non-expert users.

This transformation is applied to the prediction-serving system webpage at `https://maxpredictor.cineca.it/` (shown in figure 5.1).

Since our prediction serving system is in an early stage, the current version has several limitations: For example, fine-tuning has not yet been applied. Once deployed, the model remains static and cannot adapt to new workloads or updated simulation data.
On the other hand, live data updates have not been implemented. There is no mechanism to automatically integrate results from new Quantum ESPRESSO runs into the training dataset, thereby limiting adaptability.
Deployment is also limited to a single model. Only one trained model is served at a time, which restricts comparisons across model versions.

2 Flask was chosen as the lightweight Python web framework for exposing the ML model through HTTP endpoints.

Predicting time: SCF

Select supercomputer: Leonardo Booster

Upload a Quantum ESPRESSO input file:

Choose file

Upload corresponding pseudopotential files (.UPF):

Choose files

Specify computational resources for prediction:

Number of cores | Number of nodes | Threads per node | n_pool

Upload QE Input & Predict

QE Prediction Guide

This tool uses a trained **Neural Network** model to estimate the **time per SCF call** for your **Quantum ESPRESSO** calculations. The prediction is based on your QE input file, pseudopotentials, and **system parameters**.

System parameters describe the computational setup used for your calculation. Please specify them according to the machine or configuration you plan to run on:

- **Number of cores** – total number of cores used in the calculation (MPI × OpenMP). It affects the overall parallelization and speedup.
- **Number of nodes** – number of compute nodes allocated for the job. Each node typically contains several cores and memory resources.
- **Threads per node** – number of OpenMP threads used per node. This defines the hybrid parallelization (MPI × OpenMP) setup.
- **n_pool** – number of `k-point` or band parallelization groups (`npool` in Quantum ESPRESSO). It distributes work among MPI pools for better performance in large systems.

For more information or to report any unexpected issue, please contact: **Mandana Safari** — m.safari@cineca.it

Figure 5.1: Prediction-serving system web interface for end-users.

5.4 FUTURE PERSPECTIVES FOR THE PREDICTION-SERVING SYSTEM

Several extensions are envisioned to further enhance the functionality and usability of the prediction-serving system. One promising direction is to incorporate automated exploration of hardware configurations by looping over combinations of parallelization parameters (such as the number of nodes, tasks per node, and threads) to generate predicted execution times for each setting. These results could be visualized interactively using Plotly, enabling users to easily identify the most efficient configuration for their specific QUANTUM ESPRESSO runs. Furthermore, integration with job schedulers on high-performance computing systems would allow seamless transition from prediction to job submission, bridging the gap between estimation and execution. In the longer term, the system could evolve toward a more mature and automated infrastructure aligned with modern Machine Learning Operations (MLOps) practices [20, 34, 39], enabling continuous model retraining, version control, and deployment monitoring in production environments.

6 CONCLUSIONS AND FUTURE WORK

This thesis has explored the use of machine learning methods to predict the execution time of self-consistent field iterations in Quantum ESPRESSO simulations. The motivation for this work stems from the significant computational costs associated with *ab initio* simulations and the challenges of anticipating performance across different input configurations and hardware environments.

Scientifically, the thesis presents a systematic study applying machine learning to runtime prediction in Quantum ESPRESSO, offering a systematic comparison of multiple algorithms and revealing how error metrics and scatter plot analyses complement one another in evaluating model performance.

Among the tested models, Random Forests achieved the lowest overall error and highest explanatory power (RMSE $= 1.53 \times 10^{-6}$, $R^2 = 0.971$), confirming their effectiveness as both accurate and interpretable predictors. Through a feature importance analysis, the Random Forest model provided meaningful physical insights: identifying the number of Kohn–Sham states, total cores, and electrons as the dominant factors influencing runtime. This interpretability bridges machine learning inference with computational physics, clarifying how problem size and parallelization parameters govern performance in Quantum ESPRESSO.

Fully Connected Neural Networks (FCNNs) also performed strongly, offering comparable accuracy (RMSE $\approx 3.85 \times 10^{-6}$, $R^2 \approx 0.817$) and exhibiting smooth, consistent predictions across the entire runtime spectrum. While Random Forests tended to slightly overestimate shorter runtimes, FCNNs maintained a uniform bias, suggesting their potential value in applications requiring stability and generalization across diverse execution regimes.

Kernel Ridge Regression achieved competitive intermediate results, effectively balancing bias and variance, whereas Linear Regression (limited by its linear assumptions) underperformed on nonlinear runtime dependencies. Together, these findings reveal a detailed performance profile: Random Forests excel when both predictive accuracy and interpretability are essential, while Neural Networks are preferable for deployment scenarios prioritizing robustness and continuity across runtime scales. Ultimately, the optimal model choice depends not only on statistical metrics but also on the practical goals

of HPC users, whether they emphasize explainability, global accuracy, or consistent predictive behavior in edge cases.

Practically, it delivers a functioning prediction-serving system that exemplifies how machine learning can support users and administrators in managing computational resources more efficiently. The web interface allows users to input Quantum ESPRESSO parameters and obtain real-time runtime estimates, illustrating how data-driven models can be integrated directly into scientific workflows. This prototype highlights the benefits of reducing trial-and-error in resource allocation, minimizing wasted computational time, and assisting users in making informed decisions about job submission.

At the same time, it emphasizes the importance of dataset coverage and model generalization. In other words, predictions remain most reliable within the parameter regimes represented in the training data, and future improvements will require continuous data collection and retraining.

Future perspectives. Several promising directions emerge for extending and enhancing this work. A natural next step involves expanding the dataset to encompass a broader spectrum of materials and computational tasks, such as phonon calculations or structural relaxations, thereby improving the robustness and transferability of the models. The web-based prediction-serving system could also be augmented with automated visualization capabilities: instead of providing a single prediction, the platform could systematically explore combinations of parallelization parameters (e.g., number of nodes, tasks per node, threads, and pools), generating a comprehensive performance landscape. Interactive visualization with tools such as Plotly would enable users to identify the most efficient configuration for their specific workload before job submission.

In addition, a major opportunity lies in incorporating hardware descriptors directly into the learning pipeline. This would require extending the Quantum ESPRESSO XML output to include detailed hardware metadata, allowing the model to account for architectural variability and generalize across diverse systems. Since insufficient memory was among the primary causes of failed runs, another valuable direction would be developing models capable of predicting memory footprint alongside runtime, enabling users to avoid memory-related failures.

Beyond these immediate extensions, integrating the prediction system with HPC job schedulers would create a seamless link between runtime estimation and actual resource allocation (an important step toward intelligent, data-driven job management). Incorporating continuous learning pipelines or adopting modern Machine Learning

Operations (MLOps) frameworks [20, 34, 39] could further enable automatic retraining as new benchmarking data become available, ensuring that models remain up to date and adaptive to evolving architectures. Finally, generalizing the current architecture to support other electronic-structure codes and additional supercomputing infrastructures would pave the way for a unified, scalable platform for performance prediction and benchmarking across heterogeneous environments.

CONCLUDING REMARKS. In conclusion, this thesis demonstrates that machine learning is not only a powerful tool for scientific discovery but also a practical ally in optimizing computational workflows. By combining predictive accuracy, interpretability, and real-world deployment, it contributes to a vision where HPC simulations are guided by data-driven insights into performance, ultimately enabling more efficient and sustainable use of computational resources in the exascale era.

A MACHINE LEARNING MODEL

This appendix presents the implementation of the core machine learning model used for predicting the time-per-cycle of QUANTUM ESPRESSO SCF iterations. The code defines the network architecture, training procedures, normalization schemes, and save/load functionality.

A.1 UTILITY FUNCTIONS

The function `train_test_indices` creates training and validation splits from a given dataset, while the `NumpyEncoder` enables saving `NumPy` arrays into JSON format for reproducibility.

```
def train_test_indices(df, split=0.8):
    # Create training and test set
    inds = np.random.choice(np.array(df.index), len(df), replace=False)
    ntr = int(len(df)*0.8)
    ind_tr = inds[:ntr]
    ind_val = inds[ntr:]
    return ind_tr, ind_val


class NumpyEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, np.ndarray):
            return obj.tolist()
        return json.JSONEncoder.default(self, obj)
```

Listing 3: Utility functions for dataset splitting and JSON encoding.

A.2 MODEL DEFINITION

The main predictive model is implemented in the `TimePerCall` class. The architecture is a feed-forward neural network with multiple dense layers, L1/L2 regularization, dropout, and a final regression output.

```
class TimePerCall():
    def __init__(self, activation='swish', l1=1e-5,
                 l2=1e-4, lr=0.00005, loss='mae', nvars=15):

        # Explicit Input layer
        inputs = keras.Input(shape=(nvars,))

        # Define the model architecture
        x = layers.Dense(30, activation=activation,
                         kernel_regularizer=regularizers.l1_l2(l1=l1,
                         ↪  l2=l2))(inputs)
        x = layers.Dropout(0.2)(x)
        x = layers.Dense(20, activation=activation,
                         kernel_regularizer=regularizers.l1_l2(l1=l1,
                         ↪  l2=l2))(x)
        x = layers.Dense(15, activation=activation,
                         kernel_regularizer=regularizers.l1_l2(l1=l1,
                         ↪  l2=l2))(x)
        x = layers.Dense(10, activation=activation,
                         kernel_regularizer=regularizers.l1_l2(l1=l1,
                         ↪  l2=l2))(x)
        outputs = layers.Dense(1)(x)

        # Build and compile the model
        self.model = keras.Model(inputs=inputs, outputs=outputs)
        self.optimizer = tf.keras.optimizers.Adam(learning_rate=lr)
        self.model.compile(loss=loss,
                           optimizer=self.optimizer,
                           metrics=[loss])
        self.eps = 1e-12
        self.mx, self.my = None, None
        self.sx, self.sy = None, None
```

Listing 4: Definition of the `TimePerCall` neural network model.

A.3 TRAINING AND PREDICTION

The training function supports optional plotting of the loss curve.

```
def train(self, X, Y, epochs=50, validation_split=0.2, plot=1,
    save_path="training_plot.png"):
    train_history = self.history = self.model.fit(
        X, Y,
        epochs=epochs, validation_split=validation_split,
            batch_size=20, verbose=1)
    total_training_samples = len(X)
    steps_per_epoch = self.history.params['steps']
    calculated_batch_size = total_training_samples // steps_per_epoch
    print(f"Calculated Batch Size: {calculated_batch_size}")

    if plot == 1:
        # Plot training and validation loss
        plt.figure(figsize=(8, 6))
        plt.plot(np.arange(epochs), train_history.history['loss'],
            label='Train Loss')
        plt.plot(np.arange(epochs), train_history.history['val_loss'],
            label='Validation Loss')
        plt.legend(fontsize=12)
        plt.xlabel('Epoch', fontsize=14)
        plt.ylabel('Loss', fontsize=14)
        plt.title('Training vs Validation Loss', fontsize=16)
        plt.grid(True)
        plt.savefig(save_path)
        plt.show()
def predict(self, X):
    return self.model.predict(X)

def train_normed(self, x, y,
                 epochs=50, validation_split=0.2, plot=1,
                    save_path="training_plot.png"):
    df_normx = self.normalize_x(x)
    df_normy = self.normalize_y(y)
    self.train(df_normx, df_normy, epochs, validation_split,
        plot=plot, save_path=save_path)

def predict_normed(self, x):
    Y_hat = self.predict(self.normalize_x(x))
    return self.denormalize_y(Y_hat)
```

Listing 5: Training function with loss visualization.

Normalization is performed in log-space by applying a logarithmic transformation followed by standardization (zero mean and unit variance) to improve numerical stability during training.

```
def normalize_x(self, x):
    logx = np.log(x + self.eps)
    if self.mx is None:
        self.mx = np.mean(logx)
    if self.sx is None:
        self.sx = np.std(logx) + self.eps
    return (logx - self.mx)/self.sx
def normalize_y(self, y):
    logy = np.log(y + self.eps)
    if self.my is None:
        self.my = np.mean(logy)
    if self.sy is None:
        self.sy = np.std(logy) + self.eps
    return (logy - self.my)/self.sy
def denormalize_y(self, y):
    normy = y*self.sy + self.my
    return np.exp(normy) - self.eps
```

Listing 6: Normalization and denormalization methods.

The class provides methods to save and reload trained models.

```
def save(self, filename):
    self.model.save(filename)
    params = {
        'eps': self.eps,
        'mx': float(self.mx),
        'my': self.my,
        'sx': self.sx.values,
        'sy': self.sy,
        'lr': self.lr,
        'model_name': filename}
    with open(filename + ".json", "w") as f:
        json.dump(params, f, cls=NumpyEncoder)
@classmethod
def load(cls, filename):
    model = TimePerCall()
    with open(filename + ".json") as f:
        params = json.load(f)
    model.model = keras.models.load_model(filename)
    model.eps = params['eps']
    model.mx = params['mx']
    model.sx = params['sx']
    model.my = params['my']
    model.sy = params['sy']
    model.lr = params['lr']
    return model
```

Listing 7: Save/load functionality for trained models.

B PREDICTION SERVING SYSTEM

This appendix presents the implementation of the prediction-serving system, which provides a REST API to query trained models. The system is built using Flask with CORS support to handle web-based requests, and dynamically loads models for inference.

B.1 FLASK APPLICATION SETUP

The serving application is based on Flask, extended with CORS for cross-origin support.

```
from flask import Flask, request, jsonify, render_template
import os
import sys
sys.path.append("../src")
import pandas as pd
import ann_model as ann
from qe_parser import parse_qe_input, parse_upf, flatten_for_nn
import joblib
import numpy as np
from flask_cors import CORS
import warnings
from pymatgen.io.espresso.utils import IbravUntestedWarning

app = Flask(__name__)
CORS(app)
#The root endpoint (/) serves the HTML template for the user interface.
@app.route('/')
def home():
    return render_template('index.html')
```

Listing 8: Initialization of the Flask application with CORS.

B.2 PREDICTION ENDPOINT

The `/predict` endpoint receives QE input and corresponding pseudopotentials, dynamically loads the appropriate model, applies preprocessing, and returns the predicted runtime.

```
1  @app.route('/predict', methods=['POST'])
2  def predict():
3      try:
4          model_name = request.args.get('model', 'Leonardo_Booster')
5          model_path = f"models/model_{model_name}.keras"
6          if not os.path.exists(model_path):
7              return jsonify({"error": f"Model '{model_name}' not found"}),
           ↪  400
8
9          model = ann.TimePerCall.load(model_path)
10
11         data = None
12         # Temporary directory for uploaded files
13         tmpdir = "./tmp"
14         os.makedirs(tmpdir, exist_ok=True)
15         # --- QE input file + pseudos uploaded ---
16         if 'input_file' in request.files:
17             input_file = request.files['input_file']
18
19             pseudos = request.files.getlist('pseudos')
20
21             input_path = os.path.join(tmpdir, input_file.filename)
22             input_file.save(input_path)
23             pseudo_paths = []
24             for pf in pseudos:
25                 pseudo_path = os.path.join(tmpdir, pf.filename)
26                 pf.save(pseudo_path)
27                 pseudo_paths.append(pseudo_path)
28
29             # Build pseudo_map automatically from pseudo files
30             pseudo_map = {}
31             for path in pseudo_paths:
32                 val, nproj = parse_upf(path)
33                 pseudo_map[os.path.basename(path)] = {
34                     "valence": val,
35                     "nproj": nproj,
36                     "path": path
37                 }
38             # Parse QE input using new version
39             parsed_features = parse_qe_input(input_path,
40                 pseudo_map=pseudo_map, pseudos_dir=tmpdir)
41             # --- Collect user-provided compute resources ---
42             n_cores = int(request.form.get("n_cores", 0))
43             n_nodes = int(request.form.get("n_nodes", 0))
44             threads_per_node = int(request.form.get("threads_per_node",
               ↪  0))
45             n_pool = int(request.form.get("n_pool", 0))
46             # Validate against supercomputer limits
47             limits = supercomputer_limits.get(model_name, {})
48             if n_cores > limits.get("max_cores", n_cores):
49                 return jsonify({"error": f"n_cores exceeds maximum for
                   ↪  {model_name}"}), 400
50             if n_nodes > limits.get("max_nodes", n_nodes):
51                 return jsonify({"error": f"n_nodes exceeds maximum for
                   ↪  {model_name}"}), 400
52             ...
```

```
        ...
            if threads_per_node > limits.get("max_threads", threads_per_node):
                return jsonify({"error": f"threads_per_node exceeds maximum for {model_name}"}), 400
            # Flatten for NN
            data = flatten_for_nn(parsed_features, extra_inputs={
                "n_cores": n_cores,
                "n_nodes": n_nodes,
                "threads_per_node": threads_per_node,
                "n_pool": n_pool
            })
        # --- Filter out non-feature fields ---
        excluded_fields = {"supercomputer", "model", "submit"}
        clean_data = {}
        for k, v in data.items():
            if k in excluded_fields:
                continue
            try:
                clean_data[k] = float(v)
            except (TypeError, ValueError):
                clean_data[k] = 0.0
        # --- Debug: print the cleaned features ---
        print("\n[DEBUG] Final features for NN input:")
        for k, v in sorted(clean_data.items()):
            print(f"  {k}: {v}")

        expected_cols = [
            "n_el", "n_el^3", "n_species", "n_at", "n_transition",
            "n_lanthanid", "n_ks", "n_g_smooth", "n_k", "n_betas",
            "n_cores", "n_nodes", "threads_per_node", "n_pool"
        ]
        # --- Prepare for prediction ---
        test_input = pd.DataFrame([clean_data])
        numeric_input = test_input.reindex(columns=expected_cols, fill_value=0.0).astype(float)
        prediction_normed = model.predict_normed(numeric_input)
        output = float(prediction_normed[0][0])
        return jsonify({
            "parsed_features": clean_data,
            "prediction_time": round(output, 10)
        })

    except Exception as e:
        return jsonify({"error": str(e)}), 400
        ...
```

The feature-extraction stage is responsible for converting the information contained in a QUANTUM ESPRESSO (QE) input file and its corresponding pseudopotentials into a structured numerical representation suitable for the neural network model. This process

is implemented by the function `parse_qe_input()`, which parses the uploaded QE input file, extracts physical and computational parameters, and computes derived quantities such as total valence electrons, number of bands, grid dimensions, and k-point symmetry reduction.

The main feature-extraction routine relies on several auxiliary functions, including:

- `build_species_to_pseudo_map_from_qeinput()`: builds the mapping between atomic species and their pseudopotential files (valence, projectors, and file paths).

- `parse_upf()`: parses a pseudopotential (UPF) file to extract the valence charge and number of projectors.

- `compute_ng_smooth_cubic()` and `compute_ng_smooth_general()`: compute the smooth grid dimensions (`n_g_smooth`) for cubic and general lattice geometries.

- `parse_kpoints_from_input()`: extracts the k-point sampling parameters from the QE input file.

- `flatten_for_nn(parsed, extra_inputs=None)`: flattens the parsed QE features (and optional user-provided metadata such as number of cores, nodes, and pools) into the final 14-dimensional feature vector used by the neural network.

The implementation of the main parser function is shown below.

```
def parse_qe_input(input_file, pseudo_map=None, pseudos_dir=None):
    """
    Parse QE input file (using PWin + custom k-point parser) plus
    ↪  optional pseudo files.
    Returns a dict with key features for NN prediction.
    """
    if not isinstance(input_file, str):
        raise ValueError(f"Expected input_file to be path str, got
        ↪  {type(input_file)}")

    # Parse input with PWin
    qe_in = PWin.from_file(input_file)
    cell = qe_in.structure

    # Build Structure
    lattice = Lattice(cell.lattice.matrix)
    species = [str(s) for s in cell.species]
    frac_coords = cell.frac_coords
    structure = Structure(lattice, species, frac_coords)

    species_to_pseudo = build_species_to_pseudo_map_from_qeinput(qe_in,
    ↪  input_file, pseudos_dir)

    # Now compute total_valence and total_betas by summing over structure
    ↪  sites
    total_valence = 0.0
    total_betas = 0
    missing_pseudos = defaultdict(int)

    for site in structure:
        sym = site.species_string
        pseudo_info = species_to_pseudo.get(sym)
        val, nproj = None, None

        if pseudo_info:
            val = pseudo_info.get("valence")
            nproj = pseudo_info.get("nproj")
            # fallback: if not present, parse directly from path
            if (val is None or nproj is None) and "path" in pseudo_info:
                val, nproj = parse_upf(pseudo_info["path"])
        else:
            # Try to match by element name in pseudo_map
            if pseudo_map:
                for k in pseudo_map.keys():
                    if k.lower().startswith(sym.lower()):
                        val = pseudo_map[k].get("valence")
                        nproj = pseudo_map[k].get("nproj")
                        break

        ...
```

Listing 9: Python function for feature extraction from QE inputs and pseudopotentials.

```
        ...
            # fallback: parse from pseudos_dir if available
            if (val is None or nproj is None) and pseudos_dir:
                for fname in os.listdir(pseudos_dir):
                    if fname.lower().startswith(sym.lower()):
                        full_path = os.path.join(pseudos_dir, fname)
                        val, nproj = parse_upf(full_path)
                        break

        if val is None or nproj is None:
            missing_pseudos[sym] += 1
        else:
            total_valence += float(val)
            total_betas += int(nproj)

    # Apply total charge if present
    total_charge = qe_in.system.get("tot_charge", 0) or 0
    try:
        total_valence = float(total_valence) - float(total_charge)
    except Exception:
        pass

    if missing_pseudos:
        missing_msg = "; ".join([f"{s}:{c}" for s, c in
        ↪   missing_pseudos.items()])
        print(f"[WARN] Some species lacked pseudo info: {missing_msg}. "
              f"Computed total_valence={total_valence},
              ↪   total_betas={total_betas}")

    n_el = total_valence
    n_el_cubed = float(n_el) ** 3
    ecutwfc = qe_in.system.get("ecutwfc", None)
    ecutrho = qe_in.system.get("ecutrho", None)
    n_g_smooth = None
    fft_dims = (0, 0, 0)
    if ecutrho and ecutwfc:
        try:
            # convert structure lattice to Bohr units if needed
            ang_to_bohr = 1.889726
            lattice_bohr = np.array(structure.lattice.matrix) *
            ↪   ang_to_bohr

            # choose cubic or general path
            if abs(np.linalg.det(structure.lattice.matrix)) > 1e-6:
        ...
```

Listing 10: Continue of python function for feature extraction from QE inputs and pseudopotentials.

```
    ...
            # decide if cubic by checking approximate orthogonality
             ↪ and equal edges
            lengths = structure.lattice.abc
            angles = structure.lattice.angles
            if max(angles) - min(angles) < 1e-3 and max(lengths) /
             ↪ min(lengths) < 1.01:
                n_g_smooth, fft_dims =
                 ↪ compute_ng_smooth_cubic(lengths[0] * ang_to_bohr,
                 ↪ ecutwfc, ecutrho)
            else:
                n_g_smooth, fft_dims =
                 ↪ compute_ng_smooth_general(lattice_bohr, ecutwfc,
                 ↪ ecutrho)
        else:
            print("[WARNING] Invalid lattice matrix, using fallback
             ↪ cubic estimate")
            n_g_smooth, fft_dims = compute_ng_smooth_cubic(10.0,
             ↪ ecutwfc, ecutrho)

    except Exception as e:
        print(f"[WARNING] Failed to compute n_g_smooth: {e}")
        n_g_smooth, fft_dims = 0, (0, 0, 0)
else:
    print("[INFORMATION] ecutrho or ecutwfc missing --skipping
     ↪ n_g_smooth computation")
    n_g_smooth, fft_dims = 0, (0, 0, 0)

# Spin
spin = qe_in.system.get("nspin", 1)

# nbnd estimation
if spin == 1:
    n_states = int(np.ceil(total_valence / 2))
else:
    n_states = int(np.ceil(total_valence))
nbnd = int(n_states * 1.1) + 5

# K-points
mode, mesh, shift = parse_kpoints_from_input(input_file)
if mode == "automatic" and mesh is not None:
    dataset = spglib.get_symmetry_dataset(
        (structure.lattice.matrix, structure.frac_coords,
         ↪ [site.specie.Z for site in structure])
    )
    irreducible_kpoints = len(
        spglib.get_ir_reciprocal_mesh(mesh, structure.lattice.matrix,
         ↪ is_shift=shift)[0]
    )
else:
    # fallback for gamma/explicit
    irreducible_kpoints = mesh[0] if mesh else 1
    ...
```

Listing 11: Continue of python function for feature extraction from QE inputs and pseudopotentials.

```python
    ...
# Return a dict of features
features = {
    "species_in_input": species,
    "nbnd_estimate": nbnd,
    "total_valence": n_el,
    "n_el^3": n_el_cubed,
    "total_betas": total_betas,
    "n_atoms": len(structure),
    "ecutwfc": ecutwfc,
    "ecutrho": ecutrho,
    "irreducible_kpoints": irreducible_kpoints,
    "nspin": spin,
    "n_g_smooth": n_g_smooth,
    "fft_dims": fft_dims,
    "pseudo_summary": pseudo_map,
    "kpoints_mode": mode,
    "kpoints_mesh": mesh,
    "kpoints_shift": shift,
    }

return features
```

Listing 12: Continue of python function for feature extraction from QE inputs and pseudopotentials.

B.3 html TEMPLATE (SIMPLIFIED)

The user interface allows input submission via a web form and displays the prediction.

```html
<!DOCTYPE html>
 ...
 <body>
    <div class="login" style="margin-bottom: 20px;">
      <h1 style="color: olive">Predicting time: SCF</h1>
      <form id="predictionForm">
          <label for="supercomputer">Select supercomputer:</label>
          <select name="supercomputer" id="supercomputer" required>
          <option value="Leonardo_Booster">Leonardo Booster</option>
        ...
```

Listing 13: Briefed HTML template for the prediction interface.

```
    ...
<h3>Upload a Quantum ESPRESSO input file:</h3>
<input type="file" id="qeFile"
    accept=".txt,.in,.inp,.pw,.qe,*/*"
    style="display:none;" />
<button type="button" id="qeBtn"
    class="custom-btn">Choose file</button>
<span id="qeStatus" style="margin-left:8px;"></span>
<button type="button" id="qeRemoveBtn"
    class="remove-btn" style="display:none;
    margin-left:8px;">x</button>
<h3>Upload corresponding pseudopotential files (.UPF):</h3>
<input type="file" id="pseudoFiles"
    accept=".UPF,.upf" multiple style="display:none;" />
<button type="button" id="pseudoBtn"
    class="custom-btn">Choose files</button>
<span id="pseudoStatus" style="margin-left:8px;"></span>
<ul id="fileList"></ul>
<h3>Specify computational resources for prediction:</h3>
<div class="form-row">
  <div class="form-group">
    <label for="n_cores">Number of cores</label>
    <input type="number" id="n_cores"
    name="n_cores" required min="1" />
  </div>
  <div class="form-group">
    <label for="n_nodes">Number of nodes</label>
    <input type="number" id="n_nodes"
            name="n_nodes" required min="1" />
  </div>
  <div class="form-group">
    <label for="threads_per_node">Threads per node</label>
    <input type="number" id="threads_per_node"
           name="threads_per_node" required min="1" />
  </div>
  <div class="form-group">
    <label for="n_pool">n_pool</label>
    <input type="number" id="n_pool" name="n_pool"
        required min="1" />
  </div>
</div>
   ...
```

Listing 14: Continue of briefed HTML template for the prediction interface.

```
        ...
<button type="button" onclick="uploadQE()"
    class="custom-btn">
  Upload QE Input & Predict
</button>
<p id="qeResult" style="margin-top: 10px;
    font-weight: bold;"></p>
      </form>
    </div>
<script>
        .... // JavaScript to upload input and
         //pseudoponetials and display the result
</script>
</body>
</html>
```

Listing 15: Continue of briefed HTML template for the prediction interface.

The JavaScript responsible for uploading QE input and Pseudopotentials in this framework will be:

```
<script>
function uploadQE() {
  const qeFile = selectedQEFile;
  if (!qeFile) {
    alert("Please upload a QE input file first.");
    return;
  }
  const formData = new FormData();
  formData.append("input_file", qeFile);
  for (let i = 0; i < selectedPseudos.length; i++) {
    formData.append("pseudos", selectedPseudos[i]);
  }
  // User-provided computational parameters
  const nCores = parseInt(document.querySelector("
        input[name='n_cores']").value);
  const nNodes = parseInt(document.querySelector("
        input[name='n_nodes']").value);
      ...
```

Listing 16: JavaScript for uploading QE input, Pseudopotentials, and displaying results.

```
    ...
const threadsPerNode = parseInt(document.querySelector("
        input[name='threads_per_node']").value);
const nPool = parseInt(document.querySelector("
        input[name='n_pool']").value);
formData.append("n_cores", nCores);
formData.append("n_nodes", nNodes);
formData.append("threads_per_node", threadsPerNode);
formData.append("n_pool", nPool);
const selectedModel = document.getElementById("
      supercomputer").value;

fetch(`/predict?model=${selectedModel}`, {
  method: "POST",
  body: formData,
})
  .then((res) => res.json())
  .then((data) => {
    console.log("Received this from backend:", data);

    if (data.error) {
      document.getElementById("qeResult").innerText =
           "Error: " + data.error;
      return;
    }

    const parsed = data.parsed_features || {};

    // Extract n_el and n_k from parsed data
    const n_el = parseFloat(parsed["n_el"]);
    const nkpoints = parseFloat(parsed["n_k"]);

    if (isNaN(n_el) || isNaN(nkpoints)) {
      console.error("Parsed features missing n_el or n_k");
      document.getElementById("qeResult").innerText =
        "Error: Missing n_el or n_k in parsed QE data.";
      return;
    }

    const predict_time = data.prediction_time;
    const complexity = Math.pow(n_el, 3) * nkpoints / nCores;
    const real_time = predict_time * complexity;
    ...
```

Listing 17: Continue of JavaScript for uploading QE input, Pseudopotentials, and displaying results.

```
      ...
      document.getElementById("qeResult").innerHTML = `
        <strong>Prediction Result from QE input</strong><br>
        <ul>
          <li><b>Supercomputer Model:</b> ${selectedModel}</li>
          <li><b>Neural Network Output:</b> Normalized time per
            call predicted by the model</li>
          <li><b>n_el (from QE+pseudos):</b> ${n_el}</li>
          <li><b>n_k (from QE input):</b> ${nkpoints}</li>
          <li><b>Calculated Computational Complexity:</b>
              ${complexity.toExponential(2)}</li>
          <li><b><u>Estimated Time per Call:</u></b>
            <span style="color: green;">${real_time.toExponential(2)}
                seconds</span></li>
          <li><b>Parsed Features:</b>
            <pre>${JSON.stringify(parsed, null, 2)}</pre>
          </li>
        </ul>
      `;
    })
    .catch((err) => {
      console.error("Upload error:", err);
      document.getElementById("qeResult").innerText =
        "An error occurred during QE input +
          pseudos prediction.";
    });
}
</script>
```

Listing 18: Continue of JavaScript for uploading QE input, Pseudopotentials, and displaying results.

Here, it is worth mentioning that the webpage has been designed with more features, such as `navbar` and proper `CSS` web designs plus `Help` section. We avoid mentioning then in the thesis to avoid extra information. If you are interested in more details, please visit the GitLab page or contact the authors.

C HPC AND CLOUD INTEGRATION

This appendix presents the scripts and configuration files used to integrate the machine learning workflow with high-performance computing (HPC) resources and to deploy the prediction serving system on Cineca's ADAcloud (IaaS). The integration demonstrates a complete pipeline: *training on the HPC cluster* and *deployment on the cloud infrastructure*.

C.1 TRAINING ON HPC WITH SLURM

Model training was performed on the Cineca cluster using SLURM job submission.

```
#!/bin/bash

#SBATCH --job-name=mp_main
#SBATCH --output=logs.out
#SBATCH --error=logs.err
#SBATCH --account=cin_staff
#SBATCH --partition=boost_usr_prod
#SBATCH --nodes=1
#SBATCH --gres=gpu:1
#SBATCH --cpus-per-task=1
#SBATCH --mem=3G
#SBATCH --time=00:10:00

module load python/3.11.7

source /leonardo/pub/userinternal/msafari1/NNenv/bin/activate

cd /leonardo_work/cin_staff/msafari/NN_predictor/

python pipeline.py --base_path ./data-leo --machine
↪ Leonardo-booster --epochs 80
```

Listing 19: Example SLURM job submission script for training the ML model on cluster.

C.2 DEPLOYMENT ON ADACLOUD

The trained models were deployed as a REST API service on `maxpredict.cineca.it`, hosted on Cineca's ADAcloud (IaaS). The deployment used a combination of `systemd` for process management and `NGINX` as a reverse proxy with HTTPS support.

C.2.1 Systemd Service

We used the configuration below for a `systemd` service that runs the Flask-based prediction server.

```
#/etc/systemd/system/gunicorn.service
[Unit]
Description=Gunicorn instance to serve maxpredict
After=network.target

[Service]
User=ubuntu
WorkingDirectory=/home/ubuntu
ExecStart=/home/ubuntu/VENV/NNenv/bin/gunicorn -w 4 -b
↪  127.0.0.1:8000 maxpredictor:app
Restart=always

[Install]
WantedBy=multi-user.target
```

C.2.2 NGINX Reverse Proxy

The NGINX configuration file used to proxy requests from the public domain to the local Flask application can be presented as below.

```
#/etc/nginx/sites-available/maxpredict
server {
    server_name maxpredictor.cineca.it;

    client_max_body_size 900M;

    location / {
        proxy_pass http://127.0.0.1:8000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }

    listen 443 ssl; # managed by Certbot
    ssl_certificate
    ↪ /etc/letsencrypt/live/maxpredictor.cineca.it/fullchain.pem;
    ↪ # managed by Certbot
    ssl_certificate_key
    ↪ /etc/letsencrypt/live/maxpredictor.cineca.it/privkey.pem; #
    ↪ managed by Certbot
    include /etc/letsencrypt/options-ssl-nginx.conf; # managed by
    ↪ Certbot
    ssl_dhparam /etc/letsencrypt/ssl-dhparams.pem; # managed by
    ↪ Certbot

}
server {
    if ($host = maxpredictor.cineca.it) {
        return 301 https://$host$request_uri;
    } # managed by Certbot


    listen 80;
    server_name maxpredictor.cineca.it;
    return 404; # managed by Certbot


}
```

C.3 INTEGRATION WORKFLOW

The overall workflow can be summarized as follows:

1. **Training:** Train the model on the Data of our HPC cluster (Leonardo_booster), export weights in `.keras` format.

2. **Model transfer:** Copy the trained model to the ADAcloud instance.

3. **Cloud serving:** Start the Flask service via `systemd`, with NGINX proxying requests from the public domain.

This workflow ensures that computationally intensive tasks are executed on the HPC cluster, while user-facing predictions are provided through a scalable and accessible cloud-based interface.

BIBLIOGRAPHY

[1] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*. 2016. arXiv: 1603.04467 [cs.DC]. URL: https://arxiv.org/abs/1603.04467.

[2] D.H. Bailey et al. "The Nas Parallel Benchmarks". In: *The International Journal of Supercomputing Applications* 5.3 (1991), pp. 63–73. DOI: 10.1177/109434209100500306. eprint: https://doi.org/10.1177/109434209100500306. URL: https://doi.org/10.1177/109434209100500306.

[3] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[4] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

[5] Alexandru Calotoiu et al. "Using automated performance modeling to find scalability bugs in complex codes". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: Association for Computing Machinery, 2013. ISBN: 9781450323789. DOI: 10.1145/2503210.2503277. URL: https://doi.org/10.1145/2503210.2503277.

[6] Clément Caron, Philippe Lauret, and Alain Bastide. "Machine Learning to Speed Up Computational Fluid Dynamics Engineering Simulations for Built Environments: A Review". In: *Building and Environment* 267 (2025), p. 112229. DOI: 10.1016/j.buildenv.2024.112229. URL: https://www.sciencedirect.com/science/article/pii/S0360132324010710.

[7] HPC Carpentry. *Benchmarking and Scaling – Running LAMMPS on HPC systems*. https://www.hpc-carpentry.org/tuning_lammps/03-benchmark-and-scaling/index.html. Accessed: 2025-09-04. 2019.

[8] Dan Crankshaw and Joseph Gonzalez. "Research for Practice: Prediction-Serving Systems". In: *Communications of the ACM* (2018). ACM Research for Practice column overviewing ML model serving systems.

[9] Daniel Crankshaw et al. "Clipper: A Low-Latency Online Prediction Serving System". In: *arXiv preprint arXiv:1612.03079*. 2016.

[10] David Culler et al. "LogP: towards a realistic model of parallel computation". In: *SIGPLAN Not.* 28.7 (July 1993), pp. 1–12. ISSN: 0362-1340. DOI: 10.1145/173284.155333. URL: https://doi.org/10.1145/173284.155333.

[11] J. Demmel et al. "Self-Adapting Linear Algebra Algorithms and Software". In: *Proceedings of the IEEE* 93.2 (2005), pp. 293–312. DOI: 10.1109/JPROC.2004.840848.

[12] M. Frigo and S.G. Johnson. "The Design and Implementation of FFTW3". In: *Proceedings of the IEEE* 93.2 (2005), pp. 216–231. DOI: 10.1109/JPROC.2004.840301.

[13] Paolo Giannozzi et al. "Quantum ESPRESSO toward the exascale". In: *The Journal of Chemical Physics* 152.15 (Apr. 2020), p. 154105. ISSN: 0021-9606. DOI: 10.1063/5.0005082. eprint: https://pubs.aip.org/aip/jcp/article-pdf/doi/10.1063/5.0005082/16721881/154105_1_online.pdf. URL: https://doi.org/10.1063/5.0005082.

[14] Paolo Giannozzi et al. "QUANTUM ESPRESSO: a modular and open-source software project for quantum simulations of materials". In: *Journal of Physics: Condensed Matter* 21.39 (Sept. 2009), p. 395502. DOI: 10.1088/0953-8984/21/39/395502. URL: https://doi.org/10.1088/0953-8984/21/39/395502.

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

[16] Trevor Hastie, Robert Tibshirani, Jerome Friedman, et al. *The elements of statistical learning*. 2009.

[17] Sebastiaan P. Huber. "Automated reproducible workflows and data provenance with AiiDA". In: *Nature Reviews Physics* 4.7 (2022), pp. 431–431. ISSN: 2522-5820. DOI: 10.1038/s42254-022-00463-1. URL: https://doi.org/10.1038/s42254-022-00463-1.

[18] Sebastiaan P. Huber et al. "AiiDA 1.0, a scalable computational infrastructure for automated reproducible workflows and data provenance". In: *Scientific Data* 7.1 (2020), p. 300. ISSN: 2052-4463. DOI: 10.1038/s41597-020-00638-4. URL: https://doi.org/10.1038/s41597-020-00638-4.

[19] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[20] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. "Machine Learning Operations (MLOps): Overview, Definition, and Architecture". In: *IEEE Access* 11 (2023), pp. 31866–31879. DOI: 10.1109/ACCESS.2023.3262138.

[21] Yunseong Lee et al. "PRETZEL: Opening the Black Box of Machine Learning Prediction Serving Systems". In: *arXiv preprint arXiv:1810.06115*. 2018.

[22] Sebastian Lührs et al. "JUBE – A Flexible, Application- and Platform-Independent Environment for Benchmarking". In: ISC High Performance, Frankfurt (Germany), 12 Jul 2015 - 16 Jul 2015. July 12, 2015. URL: https://juser.fz-juelich.de/record/202788.

[23] Preeti Malakar et al. "Benchmarking Machine Learning Methods for Performance Modeling of Scientific Applications". In: *Proceedings of the 9th International Workshop on Performance Modeling, Benchmarking, and Simulation of High-Performance Computer Systems (PMBS18)*. Held in conjunction with SC'18. Available at https://sc18.supercomputing.org/proceedings/workshops/workshop_pages/ws_pmbsf119.html. Argonne National Laboratory. Dallas, TX, USA, 2018.

[24] Sedir Mohammed et al. "The effects of data quality on machine learning performance on tabular data". In: *Information Systems* 132 (July 2025), p. 102549. ISSN: 0306-4379. DOI: 10.1016/j.is.2025.102549. URL: http://dx.doi.org/10.1016/j.is.2025.102549.

[25] D.C. Montgomery, E.A. Peck, and G.G. Vining. *Introduction to Linear Regression Analysis*. Wiley Series in Probability and Statistics. Wiley, 2012. ISBN: 9780470542811. URL: https://books.google.it/books?id=0yR4KUL4VDkC.

[26] NASA Applied Remote Sensing Training Program (ARSET). *ARSET - Fundamentals of Machine Learning for Earth Science*. https://appliedsciences.nasa.gov/get-involved/training/english/arset-fundamentals-machine-learning-earth-science. Accessed: 2024-12-22. 2023.

[27] Christopher Olston et al. "TensorFlow-Serving: Flexible, High-Performance ML Serving". In: *arXiv preprint arXiv:1712.06139*. 2017.

[28] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *J. Mach. Learn. Res.* 12.null (Nov. 2011), pp. 2825–2830. ISSN: 1532-4435.

[29] Federico Pittino et al. "Prediction of time-to-solution in material science simulations using deep learning". In: *Proceedings of the Platform for Advanced Scientific Computing Conference*. 2019, pp. 1–9.

[30] Prajit Ramachandran, Barret Zoph, and Quoc V. Le. *Searching for Activation Functions*. 2017. arXiv: 1710.05941 [cs.NE]. URL: https://arxiv.org/abs/1710.05941.

[31] Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian Processes for Machine Learning*. ISBN 026218253X. The MIT Press, 2006. URL: https://www.GaussianProcess.org/gpml.

[32] Craig Saunders, Alexander Gammerman, and Volodya Vovk. “Ridge Regression Learning Algorithm in Dual Variables”. In: *Proceedings of the Fifteenth International Conference on Machine Learning*. ICML ’98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 515–521. ISBN: 1558605568.

[33] Bernhard Schölkopf, Alexander J. Smola, and Klaus-Robert Müller. “Kernel principal component analysis”. In: *Advances in Kernel Methods: Support Vector Learning*. Cambridge, MA, USA: MIT Press, 1999, pp. 327–352. ISBN: 0262194163.

[34] Alex Serban et al. “Adoption and Effects of Software Engineering Best Practices in Machine Learning”. In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM ’20. Bari, Italy: Association for Computing Machinery, 2020. ISBN: 9781450375801. DOI: 10.1145/3382494.3410681. URL: https://doi.org/10.1145/3382494.3410681.

[35] John Shawe-Taylor and Nello Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.

[36] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15.56 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[37] Jingwei Sun et al. “Automated Performance Modeling of HPC Applications Using Machine Learning”. In: *IEEE Transactions on Computers* 69.5 (2020), pp. 749–763. DOI: 10.1109/TC.2020.2964767.

[38] Leopold Talirz et al. “Materials Cloud, a platform for open computational science”. In: *Scientific Data* 7.1 (2020), p. 299. ISSN: 2052-4463. DOI: 10.1038/s41597-020-00637-5. URL: https://doi.org/10.1038/s41597-020-00637-5.

[39] Damian A. Tamburri. “Sustainable MLOps: Trends and Challenges”. In: *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 2020, pp. 17–23. DOI: 10.1109/SYNASC51798.2020.00015.

[40] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. “Optimization of Collective Communication Operations in MPICH”. In: *The International Journal of High Performance Computing Applications* 19.1 (2005), pp. 49–66. DOI: 10.1177/1094342005051521. eprint: https://doi.org/10.1177/

1094342005051521. URL: https://doi.org/10.1177/1094342005051521.

[41] Steven Euijong Whang et al. *Data Collection and Quality Challenges in Deep Learning: A Data-Centric AI Perspective*. 2022. arXiv: 2112.06409 [cs.LG]. URL: https://arxiv.org/abs/2112.06409.

[42] Steven Euijong Whang et al. *Data Collection and Quality Challenges in Deep Learning: A Data-Centric AI Perspective*. 2022. arXiv: 2112.06409 [cs.LG]. URL: https://arxiv.org/abs/2112.06409.

[43] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785. URL: https://doi.org/10.1145/1498765.1498785.

[44] Cort J. Willmott and Kenji Matsuura. "Advantages of the mean absolute error (MAE) over the root mean square error (RMSE) in assessing average model performance". In: *Climate Research* 30 (2005), pp. 79–82. DOI: 10.3354/cr030079. URL: https://www.int-res.com/abstracts/cr/v30/cr030079.

[45] Zhi-Hua Zhou and Ji Feng. "Deep Forest: Towards An Alternative to Deep Neural Networks". In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 2017, pp. 3553–3559. DOI: 10.24963/ijcai.2017/497. URL: https://doi.org/10.24963/ijcai.2017/497.