| | |
|---|---|
| Project Title | Artificial Intelligence in Secure PRIvacy-preserving computing coNTinuum |
| Project Acronym | AI-SPRINT |
| Project Number | 101016577 |
| Type of project | RIA - Research and Innovation action |
| Topics | ICT-40-2020 - Cloud Computing: towards a smart cloud computing continuum (RIA) |
| Starting date of Project | 01 January 2021 |
| Duration of the project | 36 months |
| Website | wwwwww.ai-sprint-project.eu/ |

# D2.1 - First release and evaluation of the AI-SPRINT design tools

| | |
|---|---|
| Work Package | WP2 \| Design Time Tools |
| Task | T2.1 Programming Model Interface, T2.2 AI Models Architecture Design, Optimization and Redesign, T2.3 Performance Models, T2.4 Design Space Exploration |
| Lead author | Daniele Lezzi (BSC) |
| Contributors | Francesc Lordan, Davide Cirillo, Fatemeh Baghdadi (BSC), Danilo Ardagna, Hamta Sedghani, Federica Filippini, Francesco Lattari, Eugenio Lomurno, Nahuel Coliva, Giovanni Dispoto, Enrico Galimberti (Polimi), André Martin (TUD) |
| Peer reviewers | Miguel Caballer (UPV), Mahshid Mehrabi (TUD) |
| Version | V1.2 |
| Due Date | 31/12/2021 |
| Submission Date | 22/12/2021 |

**Dissemination Level**

| | |
|---|---|
| X | PU: Public |
| | CO: Confidential, only for members of the consortium (including the Commission) |
| | EU-RES. Classified Information: RESTREINT UE (Commission Decision 2005/444/EC) |
| | EU-CON. Classified Information: CONFIDENTIEL UE (Commission Decision 2005/444/EC) |
| | EU-SEC. Classified Information: SECRET UE (Commission Decision 2005/444/EC) |

AI-Sprint - Artificial Intelligence in Secure PRIvacy-preserving computing coNTinuum, has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement no. 101016577.

# Versioning History

| Revision | Date | Editors | Comments |
|---|---|---|---|
| 0.1 | 20/10/2021 | Daniele Lezzi | ToC definition |
| 0.2 | 24/11/2021 | Hamta Sedghani, Federica Filippini, Danilo Ardagna | Added section 4.6 |
| 0.3 | 24/11/2021 | Davide Cirillo | Added section 2 |
| 0.5 | 26/11/2021 | André Martin | Added 4.2.4 |
| 0.6 | 26/11/2021 | Daniele Lezzi | General edits |
| 0.7 | 1/12/2021 | Federica Filippini, Hamta Sedghani | Added sections 4.4, 4.7.2, 4.7.4 |
| 0.8 | 3/12/2021 | Daniele Lezzi, Danilo Ardagna | Extended Executive Summary. General edits |
| 0.9 | 6/12/2021 | Federica Filippini, Enrico Galimberti | Edits in section 4.8.2 |
| 1.0 | 6/12/2021 | Daniele Lezzi | General formatting |
| 1.1 | 14/12/2021 | Daniele Lezzi, Danilo Ardagna | Addressed reviewers' comments |
| 1.2 | 17/12/2021 | Daniele Lezzi | Final version |

# Glossary of terms

| Item | Description |
|---|---|
| **AF** | Atrial Fibrillation |
| **AI** | Artificial Intelligence |
| **CSVM** | Cascade Support Vector Machine |
| **DAG** | Direct Acyclic Graph |
| **DNN** | Deep Neural Network |
| **ECG** | Electrocardiogram |
| **IM** | Infrastructure Manager |
| **K8S** | Kubernetes |
| **ML** | Machine Learning |
| **OSCAR** | Open-source Serverless Computing for Data-processing Applications |
| **QoS** | Quality of Service |

# Keywords

Artificial Intelligence; Edge Computing; Computing Continuum; Programming Models.

# Disclaimer

# Executive Summary

The aim of the AI-SPRINT project is to implement a design and runtime framework to accelerate the development of AI applications whose components are spread across the edge-cloud computing continuum. AI-SPRINT tools will allow trading-off application performance (in terms of end-to-end latency or throughput), energy efficiency, and AI models accuracy while providing security and privacy guarantees.

This document, in particular, describes the first release and evaluation of the design tools of the AI-SPRINT platform, while the second release and evaluation are due at M24.

The first release includes **programming abstractions** that hide the communications across components and transparently implement the parallelization of the compute-intensive part of the application; **quality annotations** to enrich applications with constraints on performance and security; **performance models** to support AI components execution time prediction (both for inference and training tasks); **AI Models Network Architecture Search** providing solutions to enable developers with limited Machine Learning (ML) expertise to train high-quality models specific to their needs also in terms of Quality of Service (QoS) requirements; **Applications design space exploration** tools to evaluate multiple alternative candidate deployments for complex applications involving many components.

The main results of this release include:

- The implementation of a prototype application for the AI-SPRINT Healthcare use case that adopts a parallel version of the Cascade Support Vector Machine (CSVM) algorithm to train a model that classifies a dataset to identify atrial fibrillations possibly related to the heart stroke. This implementation adopts the programming model of AI-SPRINT using PyCOMPSs and the dislib distributed ML library, reaching an average f1-score of 0.667 in discriminating atrial fibrillation from normal recordings using the time-frequency features of a balanced selection of the PhysioNet CinC Challenge 2017 available data with shuffled 80/20 training and test sets splits repeated 10 times.
- The design of QoS annotations corresponding to performance and security constraints. The AI-SPRINT API will include parsing tools which will extract the decorated functions from the code of the application and produce semi-automatically[1] the input required by the AI-SPRINT toolchain (including design, deployment and monitoring).
- Tools for the definition of performance models and profiling tools based on a ML library, that allows to predict the execution time of inference and training jobs, through feature selection, and hyperparameter tuning.
- A component for automatic neural architecture search (POPNAS) to automate the search for the best Deep Neural Network (DNN) architecture for a given classification/regression task. Also leveraging the performance models, the algorithm searches for the best network configuration to achieve the higher accuracy in the lowest possible time by searching the Pareto front of the time-accuracy trade-off.
- A tool (SPACE4AI-D) for the automatic design space exploration in order to minimize the execution cost of the AI application while providing response time guarantees. The output of this tool determines the optimal component placement, resource selection and the optimal number of nodes/VMs which helps the developer to find the optimal placement.

For each of the above-mentioned components, an evaluation section is also provided to discuss preliminary results on the adopted technologies and to evaluate the integration of the components according to the project's milestones.

[1] The input files needed to run the AI-SPRINT tools pipeline will require, in some scenarios, some edits by the AI developers/architects or sysadmins.

## Table of Contents

## List of Figures

## List of Tables

# 1. Introduction

## 1.1 Scope of the document

The aim of the AI-SPRINT "Artificial intelligence in Secure PRIvacy-preserving computing coNTinuum" project is to develop a platform composed of design and runtime management tools to seamlessly design, partition and operate Artificial Intelligence (AI) applications among the current plethora of cloud-based solutions and AI-based sensor devices (i.e., devices with intelligence and data processing capabilities), providing resource efficiency, performance, data privacy, and security guarantees. This document provides a detailed description of the first release and evaluation of the design tools, including the software assets developed in the first year of the project and the description of a prototype application from one of the use cases used to evaluate these developments.

## 1.2 Target Audience

The release and evaluation of the design tools (initial and final version) is intended for internal use, although it is publicly available. The target audience is the AI-SPRINT technical team including all partners involved in the delivery of work packages 2,3 and 4, but also it serves as reference for the developers of the three use cases of the project.

## 1.3 Structure of document

This document includes four main parts:

- The **Example Applications** introduce two prototype applications, one developed in the context of the healthcare use case, that allows the training of a ML model using the design tools, while the other is used to describe the quality annotations.
- The **Architectural Overview of the Design Tools** provides a high-level description of the main components to design AI applications and to provide quality annotations that will be used by the runtime environment.
- The **First release of the Design Tools: Components and Evaluation** section provides the details of each software component of the Design Tools together with the description of the activities to enhance them to fulfil the AI-SPRINT use cases requirements. An evaluation of each component is also provided.
- The **Towards the Integrated framework for the Design Time Tools** section provides a description of the roadmap for the integration of the components of the Design Tools.

# 2. Example applications

The implementation of the components of this first release of the design tools has been driven by the requirements analysis of the use cases, whose implementation plan and evaluation are due at M12 and M24, respectively. Anyway, to guide the discussion on the technology choices in this document, we used two applications to i) evaluate the implementation of a Personalised Healthcare using the Programming Abstractions component and ii) to demonstrate the definition of QoS constraints.

## 2.1 Parallel CSVM classification

### Motivation

The AI-SPRINT Personalised Healthcare use case concerns the collection, analysis and modeling of digital data and lifestyle information from subjects who suffered a stroke. The digital data will be collected from a wearable device (a smartwatch or band) equipped with biometric sensors, namely photoplethysmography (PPG) and seismocardiography (ECG). PPG and ECG are generally used to perform heart rate (HR) and heart rhythm measurements. In particular, when coupled with PPG, which measures changes in the microvascular blood volume reflecting pulse waves, a single-lead ECG, which is produced by placing a contralateral finger on a negative electrode on the side of the device while its back serves as a positive electrode, can diagnose arrhythmias that are relevant for stroke, such as atrial fibrillation (AF). Given the relevance of ECG in stroke monitoring, this example application is focused on AF classification from single-lead ECG recordings.

### Dataset and classification approach

Since, due to COVID-19 pandemic, our supplier delayed the delivery of the wearable device, the first year research focussed on the PhysioNet [Goldberger2000] dataset. PhysioNet is a repository of freely-available medical research data, managed by the MIT Laboratory for Computational Physiology. Among many freely available datasets in PhysioNet, the database of the Computing in Cardiology (CinC) Challenge 2017 on AF classification (https://physionet.org/content/challenge-2017/1.0.0/) contains a total of 12,186 single-lead ECG recordings donated by AliveCor, an AI company producing ECG hardware. The data, collected from individuals at rest, is available as Matlab V4 files (each including a .mat file containing the ECG and a .hea file containing the waveform information) and split into training and test data sets. At the time of the challenge, the training set contained 8,528 recordings lasting from 9 to 61 seconds and the test set contained 3,658 recordings of similar lengths. The classes represented in the database are (1) Normal (5154 recordings), (2) AF (771 recordings), (3) Other rhythms (2557 recordings), and (4) Nosy recordings (46 recordings). By achieving a F1-score of 0.79 in 5-fold cross validation on the AF class [Datta2017], a classifier based on the Cascade Support Vector Machines (CSVM) algorithm was among the winners of the CinC Challenge 2017. Currently, the test set is unavailable to the public and labels have been updated. As of December 2021 and concerning the classes of interest for this example application, the database is composed of 5050 Normal and 738 AF recordings.

CSVM is an algorithm for SVM that can be parallelized efficiently and scales to very large problems. Conceptually, CSVM consists in splitting the data into subsets that are optimised separately with multiple SVMs, whose partial results are then combined and filtered again into a 'cascade' of SVMs until the global optimum (minimum of the loss function in the training set) is reached. As the kernel matrices are smaller than a regular SVM, CSVM exhibits several computational advantages as it requires less memory and its training can be spread over multiple processors with minimal communication overhead. It has been demonstrated that a single pass through the 'cascade' provides good generalisation, although convergence to the global optimum is guaranteed with multiple passes [Graf2004].

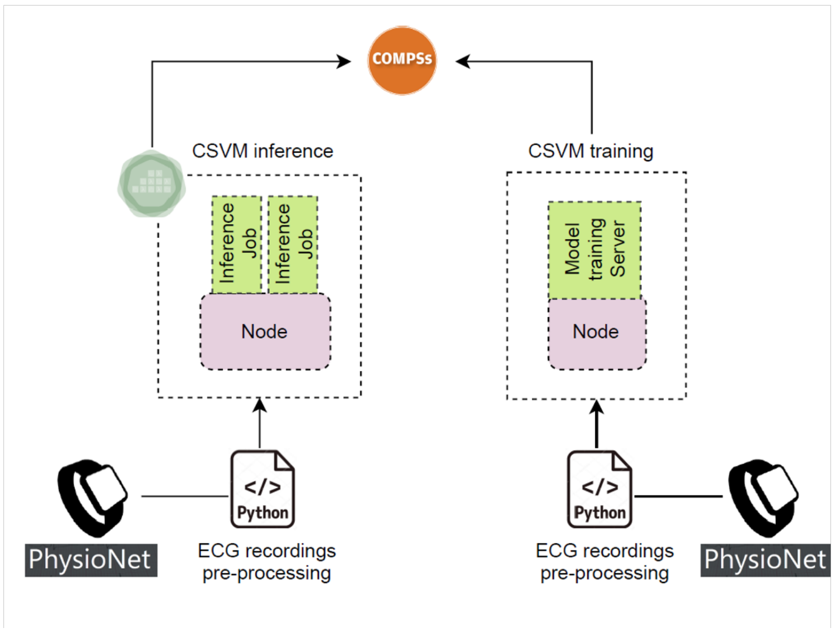


*Figure 2.1 - Architecture of the healthcare application*

A binary classifier using the CSVM algorithm is conveniently available in the dislib library (https://dislib.bsc.es/en/latest/dislib.classification.csvm.html). Thus, given the scope of the example application, we transformed the original multiclass problem of the PhysioNet CinC Challenge 2017 into one binary problem, namely the AF class versus a balanced representation of the Normal class.

### Time-frequency features extraction

ECG are commonly analysed in the time-domain by experienced physicians. However, as pathological conditions may not always be obvious in the time-domain analysis, frequency-domain techniques, such as Fourier transform (FT), have been introduced [Clayton1993]. Nevertheless, ECG signals belong to the family of multicomponent nonstationary signals [Wood1996], for which accurate time-varying spectral estimates can be extremely difficult to obtain. Thus, combining the time-domain and frequency-domain can improve the interpretation of the signals and the detection of complex arrhythmias [Mahmoud2006]. For this reason, we implemented a feature extraction procedure to pre-process the PhysioNet CinC Challenge 2017 dataset before using it to train the dislib CSVM binary classifier. The features extracted, after uniforming the length of the recordings by padding with zeros for a total of 61 seconds, comprise the frequency component, the time component, and the amplitude for each wave (data point).

## 2.2 Mask Detection

Figure 2.2 depicts the workflow of the mask-detection example application which is considered as a reference example in WP3; more details are provided in the *AI-SPRINT Deliverable D3.1 - First release and evaluation of the runtime environment*. The application is composed of two main components: "*Anon and split*" and "*Mask detector*". The former is in charge of anonymizing the video frames by blurring the detected faces, while the latter performs the detection of the masks. The implementation of the application can be found at: examples/mask-detector-workflow · master · AI-SPRINT / SCAR · GitLab (polimi.it).

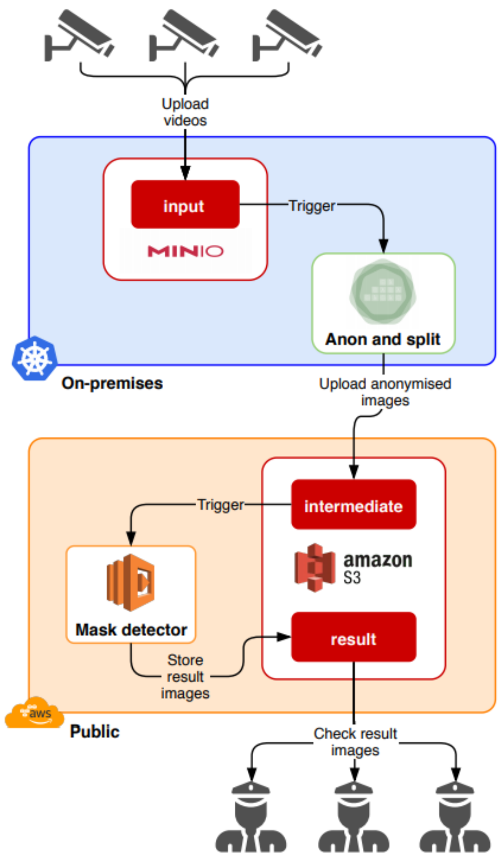


*Figure 2.2 - Workflow of the mask detection example application*

# 3. Architectural overview of the Design Tools

The AI-SPRINT architecture overview was described in *Deliverable 1.3 Initial design of the architecture*. This section summarises components of the design time tools. For each individual block, we describe its role and relation within the global architecture. An overview of the global architecture is shown in Figure 3.1.

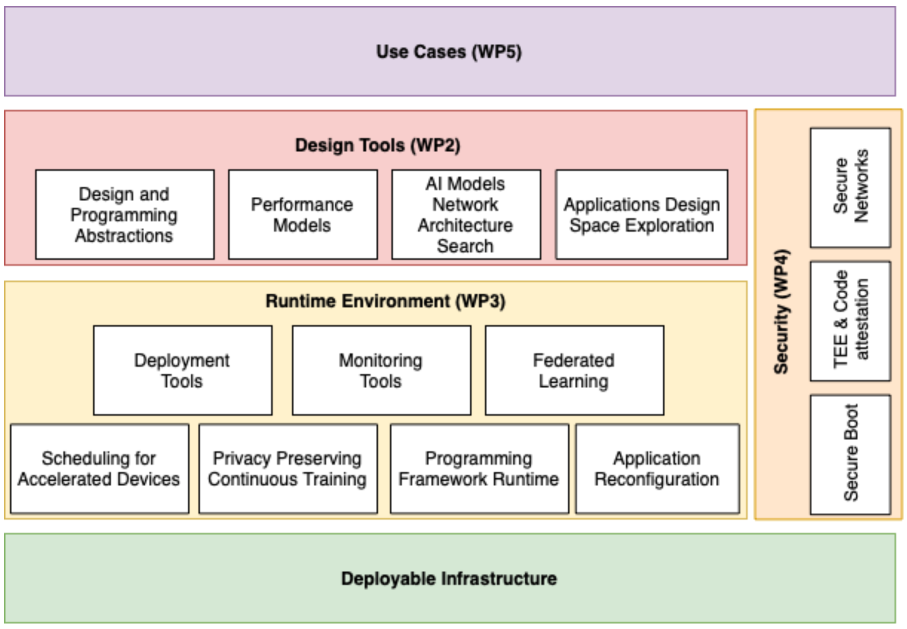

Use Cases (WP5)
Design Tools (WP2)
Design and Programming Abstractions
Performance Models
AI Models Network Architecture Search
Applications Design Space Exploration
Runtime Environment (WP3)
Deployment Tools
Monitoring Tools
Federated Learning
Scheduling for Accelerated Devices
Privacy Preserving Continuous Training
Programming Framework Runtime
Application Reconfiguration
Security (WP4)
Secure Networks
TEE & Code attestation
Secure Boot
Deployable Infrastructure

*Figure 3.1 - AI-SPRINT Architecture Overview*

In this deliverable, we focus on the development and evaluation of Design Tools developed in WP2:

- *Design and programming abstractions*: hide the communications across components and transparently implement the parallelization of the compute-intensive part of the application, possibly exploiting specialised resources (e.g., GPUs and AI enabled sensors). Applications are also enriched with quality annotations (e.g., data flow rates, application latency, energy constraints) to express performance, accuracy, privacy, and security constraints.
- *Performance models*: automate the AI application performance profiling and identify the performance model (based mainly on Machine Learning) providing the highest performance prediction accuracy. Also, support performance ML model selection and hyper-parameters tuning considering the target deployment of AI-based sensors.
- *AI models network architecture search*: provide solutions for developers with limited ML expertise to train high-quality models specific to their needs also in terms of Quality of Service (QoS) requirements. Also, build a learning as a service solution, that starting from a training set with labelled training examples (images or temporal data series which are of interest for use cases) will automatically identify the most accurate deep neural network which provides execution time guarantees.
- *Applications design space exploration*: perform and automate design space exploration in order to minimise the execution cost of the AI application while providing response time QoS guarantees by implementing several heuristic algorithms (e.g., random greedy, local search, tabu search, etc.). The tool considers AI applications with different candidate deployments, which include different DNN partitions for each component and different resource candidates and aims to select the optimal deployment and resource candidate while taking into account resource contention.

Table 3.1 summarises the WP-level responsibilities of the different components of the Design Tools, the lead maintainer and the major contributors. The complete table for the rest of the components is available in *AI-SPRINT Deliverable D1.3 Initial design of the architecture*.

| Tool | WP | Task | Lead Maintainer | Major contributors |
|---|---|---|---|---|
| **Design and Programming Abstractions** | 2 | T2.1 | BSC | TUD, POLIMI |
| **Performance Models** | 2 | T2.3 | POLIMI | BSC, UPV |
| **AI Neural Architecture Search** | 2 | T2.2 | POLIMI | BSC, TUD |
| **Application Design Space Exploration** | 2 | T2.4 | POLIMI | BSC, UPV |

*Table 3.1 - WP-level responsibilities for the design tools*

The objective of the AI-SPRINT Design Tools is to provide a layer that abstracts the applications from the underlying computing resources, being these edge resources or cloud servers in such a way that the application developer only needs to focus on the actual algorithm and application logic. On the other hand, interfaces for easing the integration of the AI applications with the runtime system will be developed.

Several software components have been adopted in the AI-SPRINT platform and will be integrated to support the requirements identified by the AI-SPRINT use cases and described in deliverable *D1.2 Requirements analysis*.

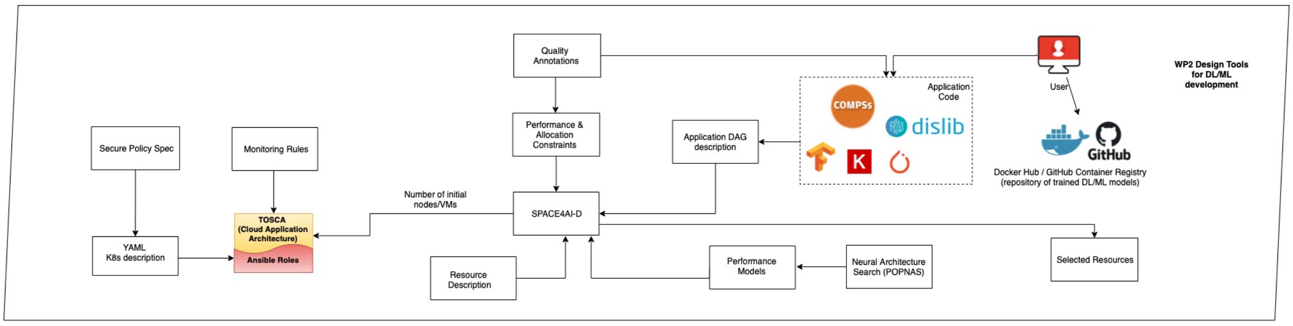

Quality Annotations
Performance & Allocation Constraints
Application DAG description
Application Code
COMPSs
dislib
User
WP2 Design Tools for DL/ML development
GitHub
Docker Hub / GitHub Container Registry (repository of trained DL/ML models)
Secure Policy Spec
Monitoring Rules
Number of initial nodes/VMs
SPACE4AI-D
TOSCA (Cloud Application Architecture)
Ansible Roles
YAML K8s description
Resource Description
Performance Models
Neural Architecture Search (POPNAS)
Selected Resources

*Figure 3.2 - AI-SPRINT Architecture detail of Design Tools relations*

Figure 3.2 describes the main interactions among the components of the Design Tools components.

The design and composition of ML, DL and AI applications is based on the **COMPSs/PyCOMPSs** [Lordan2014] programming model, that enables the development of complex and dynamic workflows, composed of pure computational parts, classical data analytics and ML/DL methods. Several ML algorithms implemented with PyCOMPSs are already available as a Distributed Computing Library (**dislib**) [Alvarez2019] inspired by Scikit-learn, which eases the task of developing applications providing a common interface in all algorithms. The main benefit of the PyCOMPSs programming model together with the dislib library is the reduction of the execution time of the training and inference processes by exploiting the inherent data parallelism of the input data. To do so, the input dataset is distributed over the different targeted nodes and each node oversees training the model with the assigned dataset part. Later, the different models trained in isolation are combined, thus requiring exchanging either the model (parameters) or the parameter's gradients between the nodes. The user (application developer role) can save the trained model and store it in a repository and then load it for inference tasks. COMPSs also allows the integration with popular frameworks executing PyTorch, TensorFlow and Keras tasks as external processes.

The developed code in AI-SPRINT is enriched with high-level **annotations for QoS constraints** and code dependencies, with performance parameters for the allocation of tasks to computing continuum resources and with security and privacy annotations for data allocation and processing. A *code-parsing* tool (whose implementation will start at M13) parses the code of the application searching for the decorated functions. The output of the code parsing is a list of annotations together with the corresponding parameters. Additionally, annotations to guide the partitioning of specific components based on Deep Neural Networks are also provide.

Once an AI application is designed, **Performance Models** can help to anticipate the performance of the application components before the production deployment or throughout revision cycles, under different configurations and deployment settings at the full computing continuum. The AI-SPRINT performance modelling approach is mainly based on ML, in particular on the a-MLLibrary[2] library enhanced to improve the hyperparameters tuning phase. The **NAS (Neural Architecture Search) module** (aka POPNAS) takes into account desired QoS both from a machine learning perspective (e.g., accuracy, precision, recall, etc.) and a performance perspective (e.g., latency, memory, power consumption, etc.).

The **SPACE4AI-D** tool (*System PerformAnce and Cost Evaluation on Cloud for AI applications Design*) tackles the component placement problem and resource selection in the computing continuum at design time, dealing with different AI application requirements in order to effectively orchestrate heterogeneous edge and cloud resources.

Monitoring rules includes a built-in set of metrics which describe statistics of the Kubernetes cluster (for Pod and nodes) hosting the target AI applications. More details can be found in the AI-SPRINT deliverable *D3.2 First release and evaluation of the monitoring system*. Means for specifying the Pods/nodes to be monitored and the metrics time granularity, will be developed.

[2] https://github.com/a-MLLibrary/a-MLLibrary

# 4. First release of the Design Tools: Components and Evaluation

This section describes the components of the Design Tools layer according to the Software Design Specification (SDS) standard (IEEE Standard 1016). Each component is described in terms of its external interfaces and dependencies with other components.

First, we provide a description of the methodology in the design of the tools, and the conceptualization of the relevant entities shared by multiple AI-SPRINT roles and tools. For each of the components we provide a description of the main functionalities and the high level architecture. Then, a more detailed description of the actual release is provided, describing the activities around the developments in the first year of the project. A specific sub section is dedicated to the analysis of the performance evaluation of the tools.

## 4.1 Template description of components

The following template will be used as the structure to provide the information for each component involved in the runtime environment. The template is included here to make this document self-contained. A similar description is also reported in AI-SPRINT deliverables *D3.1 First release and evaluation of the runtime environment*, and *D4.1 Initial release and evaluation of the security tools.*

| | |
|---|---|
| **Identification** | The unique name for the component and its location in the system |
| **Type** | A module, a subprogram, a data file, a control procedure, a class, etc. |
| **Purpose** | Function and performance requirements implemented by the design component, including derived requirements. Derived requirements are not explicitly stated in the SRS, but are implied or adjunct to formally stated SDS requirements. |
| **Function** | What the component does, the transformation process, the specific inputs that are processed, the algorithms that are used, the outputs that are produced, where the data items are stored, and which data items are modified. |
| **High level Architecture** | The internal structure of the component, its constituents, and the functional requirements satisfied by each part. |
| **Dependencies** | How the component's function and performance relate to other components. How this component is used by other components. The other components that use this component. Interaction details such as timing, interaction conditions (such as order of execution and data sharing), and responsibility for creation, duplication, use, storage, and elimination of components. |
| **Interfaces** | Detailed descriptions of all external and internal interfaces as well as of any mechanisms for communicating through messages, parameters, or common data areas. All error messages and error codes should be identified. All screen formats, interactive messages, and other user interface components (originally defined in the SRS) should be given here. |
| **Data** | For the data internal to the component, describes the representation method, initial values, use, semantics, and format. This information will probably be recorded in the data dictionary. |

| | |
|---|---|
| **Needed improvement** | Description of the needed improvements of this tool with regards the AI-SPRINT project, in order to fulfil the user requirements and to build the runtime environment |
| **Implemented Improvements for the First Release** | A description of the implemented improvements in the service to achieve the first release of the runtime environment. |
| **Release Version & Repository** | The software version released and the repository from where it can be downloaded. |

## 4.2 Design methodology and shared concepts

In this section, we describe the overall approach to the conceptualization of the main entities relevant for the project and shared by multiple AI-SPRINT roles and tools. Figure 4.1 describes with a class diagram the AI-SPRINT entities, their attributes, and the relationships among them. Here, we cover the design aspects of the applications, the design space exploration, the resources definition and the deployment options that are then enacted by the runtime tools.

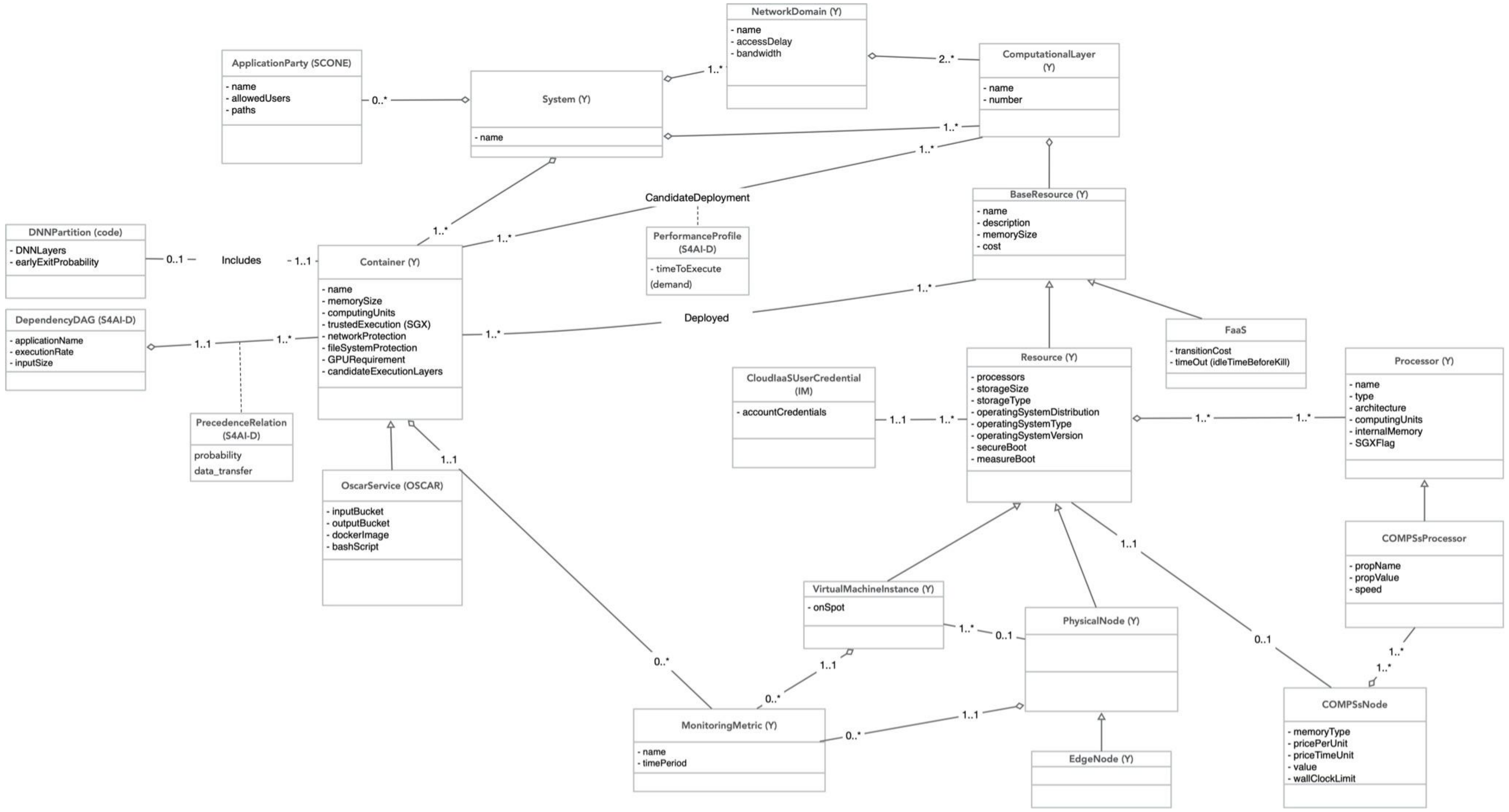


*Figure 4.1 - AI-SPRINT Roles and actions related to the generation and consumption of data*

The common approach for the usage of the AI-SPRINT toolchain is the definition of YAML files to describe the information needed by multiple tools that will be translated by an ad-hoc code parser into the specific input files required by individual tools. Examples of YAML files will be reported in the following sections within the evaluation section and appendices. To rely on the AI-SPRINT toolchain, AI-SPRINT developers/application architects/application managers/AI experts need to define mainly a System YAML file and a Monitoring rule YAML file listing the metrics to be gathered at resource level with their time granularity. Most of the attributes are self-explanatory. We describe here in detail the attributes requiring a clarification.

A system is characterised by a set of Network Domains that alternatively includes at least two computational layers consisting of one or more candidate resources. Each Network Domain has specific properties related to the technology of the underlying connection like bandwidth and access delay. All the computational layers located in the same Network Domain can communicate together under the Network Domain's connection properties. In AI-SPRINT, the components of an application are deployed on different computing layers in a hierarchical way, following the OpenFog Reference Architecture (RA) [OpenFog]. As an example, in Figure 4.2 layer 1 includes IoT and/or AI-enabled sensors, layer 2 includes edge servers, PCs and /or Raspberry Pi devices, layer 3 includes the cloud, while layer 4 is a Function as a Service (FaaS). According to the specific use case, some devices (e.g., smartphones) can be considered at layer 1 or 2, which usually are identified with the term *edge*.

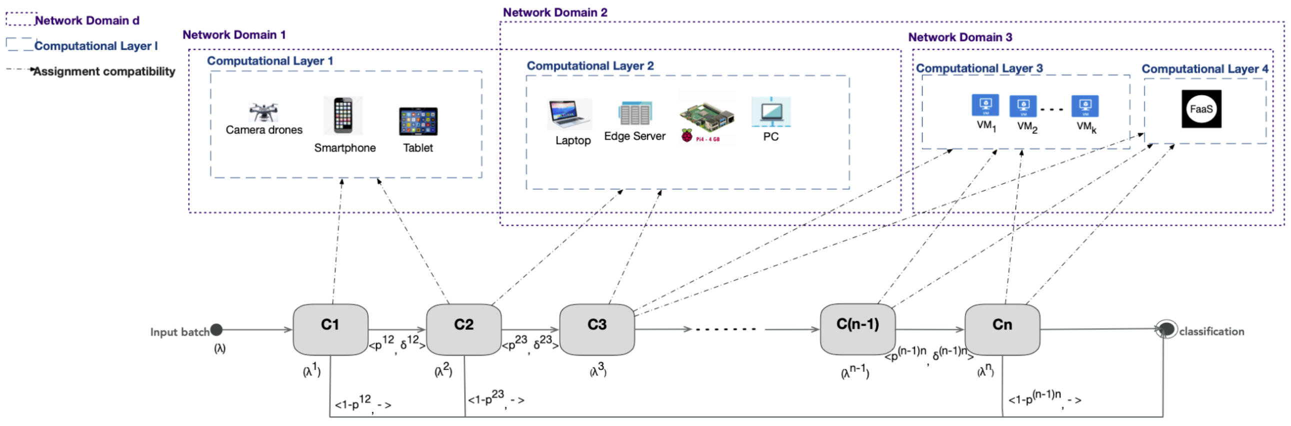


*Figure 4.2 - Example of AI-SPRINT system description*

Base resources include attributes common to every resource in the computing continuum and are then classified in Resources (further detailed in VirtualMachines, PhysicalNodes, and EdgeNodes) and FaaS. As an example, a COMPSsNode is characterised additionally by the type of processor, the number of cores for each processor, the internal cache and other user defined properties. If PyCOMPSs is adopted to implement an application, a COMPSs parser transforms the System YAML description into the XML resource representation used by the PyCOMPSs runtime scheduler with empty attributes that will be edited by the application manager in a second stage (implementing a semi-automatic transformation).

Furthermore, resources are annotated if they support secure boot, i.e., if the BIOS provides and is enabled for secure booting as well as if the operating system image used on the physical or virtual node supports measured boot in order to fulfil the security constraints laid out by the application developer or user (see also *AI-SPRINT Deliverable 4.1 Initial release and evaluation of the security tools*).

Container attributes are intended as requirements for the container. So, for example, if the trustedExecution flag is set to true the container can be orchestrated only on a resource providing a processor with SGXFlag true (see Figure 4.1) and this information will be passed to the Infrastructure Manager (IM) to deploy a K8s cluster with SGX support.

Figure 4.3 depicts the actions performed by each role to generate and consume the information required by the tools. The AI-SPRINT roles have been defined in deliverable *D1.2 Requirements Analysis*.

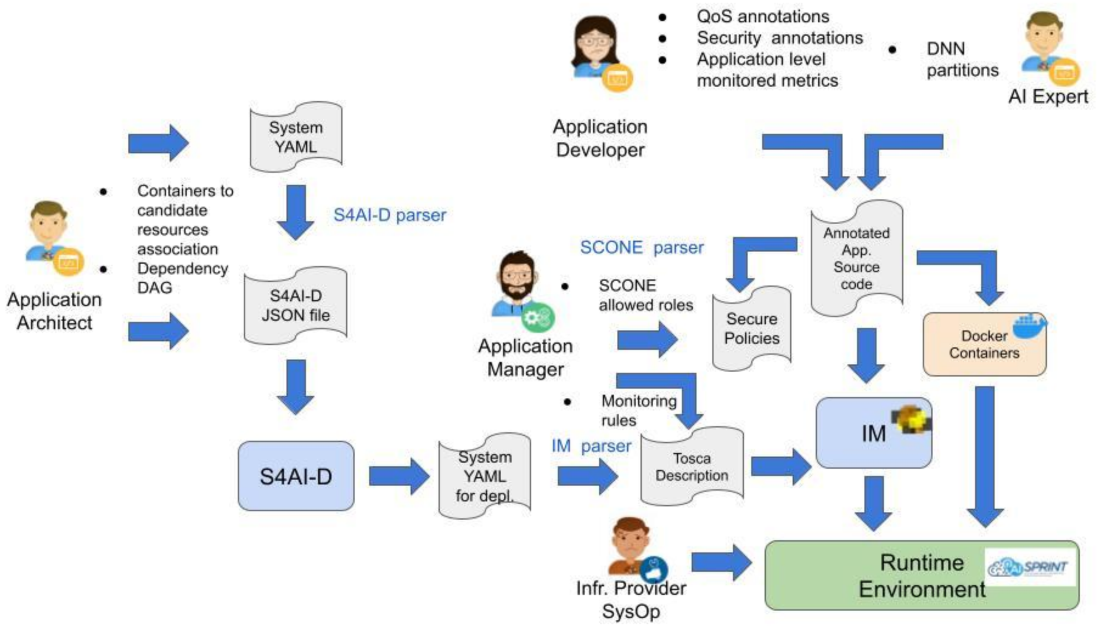


*Figure 4.3 - AI-SPRINT Roles and actions related to the generation and consumption of data*

## Application architect

- Defines the *System YAML* that is parsed by SPACE4AI-D to generate a system description for the deployment, with the objective of minimising the costs given the constraints; this description will be then used by IM to perform the initial deployment.
- Defines the associations among the Containers, the OSCAR Services and the candidate resources.
- Defines the dependency Directed Acyclic Graph (DAG) of the containers which indicates the sequence of execution of the containers. The nodes of the DAG represent the different components which, possibly, are DNNs implemented as Python functions running in containers. The DAG includes a single entry point, characterised by the input exogenous workload λ (expressed in terms of requests/sec, see also Figure 4.2). Each edge connecting two components is labelled with a pair: *transition probability* and *data size* transferred by the first component to the other one.

## Application developer

- Provides quality annotations in the code through Python decorators and PyCOMPSs annotations, resource constraints and security constraints.
- Defines DNN partitions.
- Provides application level monitored metrics (more details in *AI-SPRINT Deliverable D3.2 First release and evaluation of the monitoring system*).

## AI Expert

- Contributes to the definition of the DNN partitions.

## Application manager

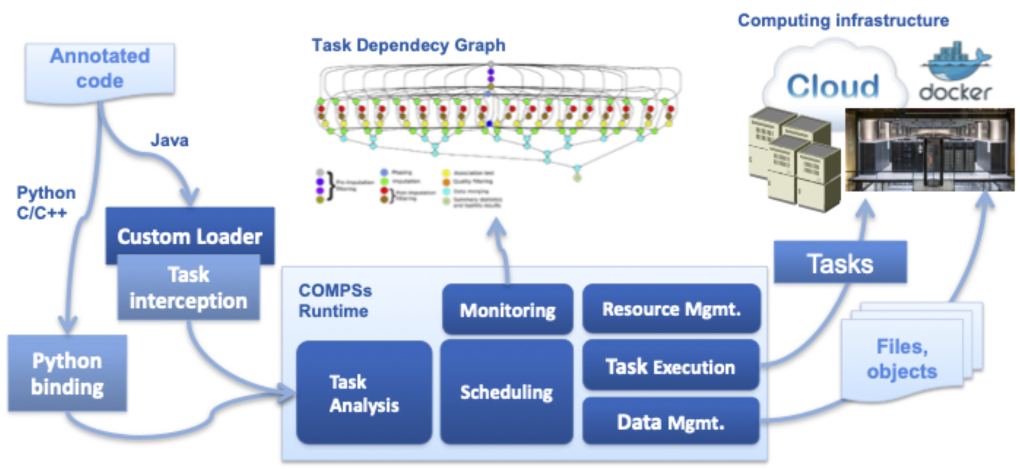
- Defines resource level monitoring metrics in a YAML file.

- Completes the output *System YAML* file obtained by SPACE4AI-D to generate an initial version of a TOSCA description that will be submitted to the IM for deployment.
- Completes the security tools input files, automatically generated initially by the SCONE parser from code decorators.

## 4.3 Design and Programming Abstractions

The objective of the AI-SPRINT Design Tools is to provide a layer that abstracts the applications from the underlying computing resources, being these edge resources or cloud servers. The programming model is provided by PyCOMPSs while the quality annotations, which are currently at a definition stage, will be supported by decorators and an ad-hoc code parser.

### *4.3.1 PyCOMPSs*

| | |
|---|---|
| **Identification** | PyCOMPSs |
| **Type** | Subprogram |
| **Purpose** | Automatic parallelization of the application leveraging the user provided information through input dependency annotations in the code. |
| **Function** | PyCOMPSs provides a sequential programming model to develop AI/ML applications, hiding the complexity of the underlying infrastructure. The programming model includes the definition of the tasks and of the constraints (through Python annotations) to drive the scheduling phase at execution time. Annotations are extended to allow predicating on components performance and to specify constraints on the target deployment. |
| **High level Architecture** | PyCOMPSs is part of the design tools layer.<br>The application developer provides a sequential Python script whose functions are annotated through decorators; these annotations are used by the runtime to run those parts of code as asynchronous parallel tasks code. When executed, the user code (the annotated part) is intercepted and analysed by the runtime, generating an execution graph.<br> Annotated code<br>Java<br>Python C/C++<br>Custom Loader<br>Task interception<br>Python binding<br>Task Dependecy Graph<br>COMPSs Runtime<br>Monitoring<br>Resource Mgmt.<br>Task Analysis<br>Scheduling<br>Task Execution<br>Data Mgmt.<br>Computing infrastructure<br>Cloud<br>docker<br>Tasks<br>Files, objects |

| | |
|---|---|
| **Dependencies** | In the programming phase PyCOMPSs has no dependencies on any other component of the platform. |
| **Interfaces** | The interfaces to the programming model are the annotations provided by the AI application developer in the code to describe the type of parameters and constraints on the resources. PyCOMPSs also provides a set of APIs to control the flow of the applications (fault tolerance and synchronisation points). |
| **Data** | PyCOMPSs processes the information provided by the user through Python decorators and generates a dependency graph, used by SPACE4AI-D, and a set of log files that can be used to produce performance models. |
| **Needed improvement** | Extend the programming interface to include QoS constraints in order to improve the scheduling phase of the runtime. |
| **Implemented Improvements for the First Release** | Support for HTTP tasks to allow calls to external functions published as a service.<br>Support for optional parameters and default values in tasks. |
| **Release Version & Repository** | The software is available at https://github.com/bsc-wdc/compss and in the AI-SPRINT GitLab repository https://gitlab.polimi.it/ai-sprint/compss with daily updates. |

### Detailed Description of Activities for the First Release

An objective for the development environment for AI-SPRINT user applications is that low-level details are hidden from the developer, thus making the programming of edge-cloud applications as simple as possible. To this end, we will use the PyCOMPSs programming model, since it prevents the programmer from explicitly dealing with distributed computing issues on the heterogeneous resources that conform the AI-SPRINT infrastructure. In this way, technical details such as communications, networking, resource management or data management are hidden from developers, who can just focus on the functionality they want to implement.

The PyCOMPSs programming model focuses on simplicity and is based on the idea of programming sequential and executing distributed and in parallel. Sequential programming is much more simple than parallel/distributed programming and enables the application developer to focus on the semantics of the problem. The parallelism in PyCOMPSs is exploited at task level, and for this the syntax provides a simple interface to identify tasks. Depending on the programming language (Python, Java or C/C++) tasks are identified by the programmers in a slightly different way, but basically the programmer needs to identify those methods or functions of the application that should be tasks and executed in parallel and indicate the directionality of their parameters. The directionality of the parameters can be "input" when a parameter is read, "output" when a parameter is written or "inout" when it is read and written by the task. This information is used by the runtime to determine the data dependencies.

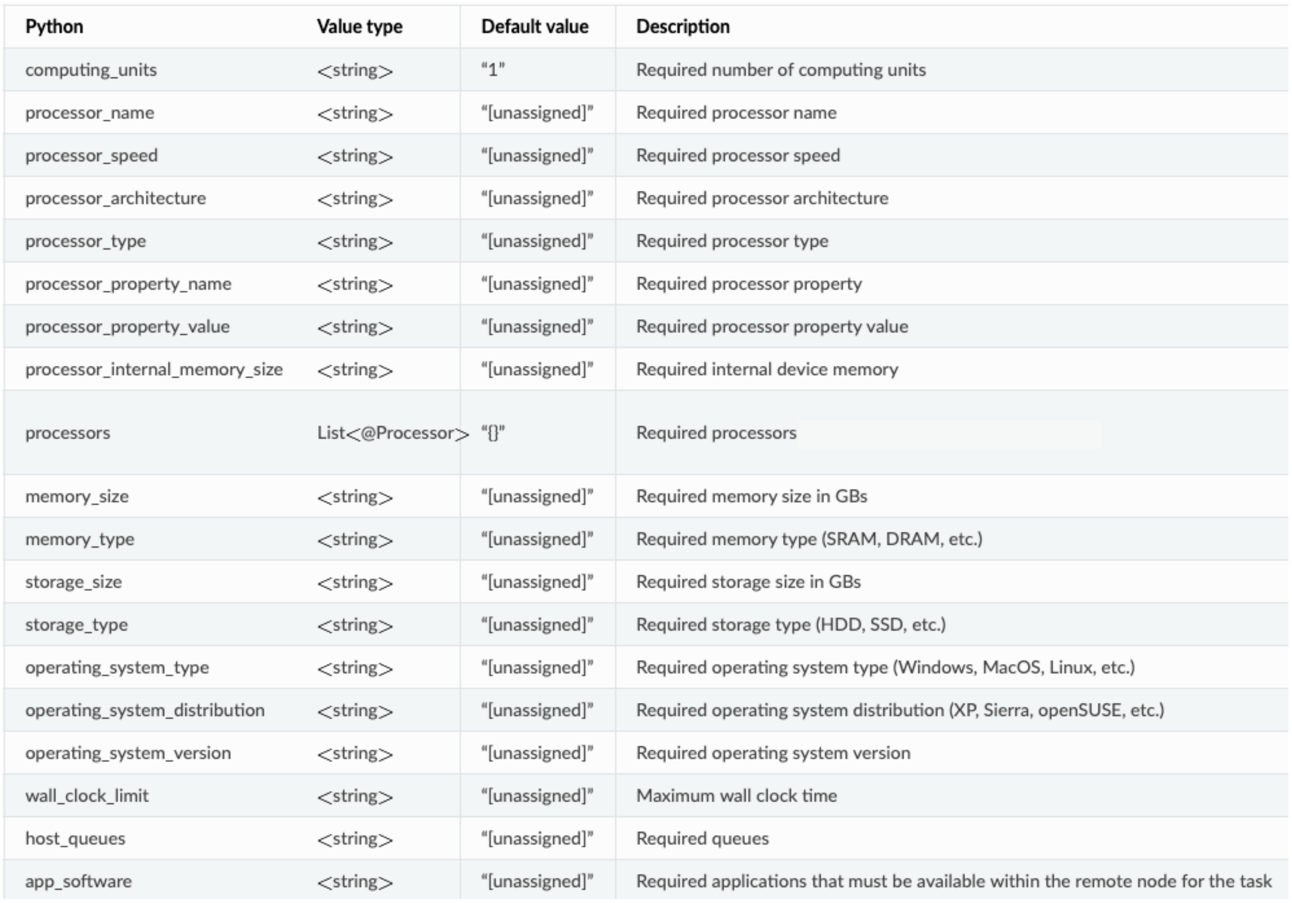
The PyCOMPSs programming model provides a set of Python decorators that allow the user to identify the function/methods to be considered as tasks and a small API for synchronisation. PyCOMPSs also features a runtime that is able to identify the data dependencies that exist among tasks and to extract the parallelism between them building a data dependency graph of tasks. The runtime is also responsible of managing their execution across distributed infrastructures (e.g., Grids, Clusters, Clouds, and Container manager clusters) – scheduling them and performing the necessary data transfers when needed – guaranteeing that the result is the same as if the application was executed sequentially.

The main decorator that PyCOMPSs provides to identify that a function/method has to be considered as a task is the *@task* decorator. This decorator can be placed on top of any function, instance method or class method and it is used to identify the function's input/output parameters and return peculiarities.

Moreover, PyCOMPSs provides a set of decorators to identify that the execution of a binary file is considered as a task. PyCOMPSs support the invocation of three types of binaries: simple binaries, MPI binaries, and OMPSs[3]. To this end, the *@binary, @mpi*, and *@ompss* decorators respectively must be placed on top of the @task decorator. The users must also specify the binary to execute, as well as the specific parameters for MPI and OmpSs invocation (i.e., the number of nodes to use).

Besides, PyCOMPSs also supports the task's constraint definition. To this end, it provides the *@constraint* decorator, which also needs to be placed on top of the stack of decorators. Constraints are also used to let the developers provide hints on the fault tolerance at task level thus allowing to discard parts of a workflow that do not lead to relevant results or that fail for some reason, without affecting the main application.

Figure 4.4 provides the complete list of arguments for the constraint decorator.



| Python | Value type | Default value | Description |
|---|---|---|---|
| computing_units | <string> | "1" | Required number of computing units |
| processor_name | <string> | "[unassigned]" | Required processor name |
| processor_speed | <string> | "[unassigned]" | Required processor speed |
| processor_architecture | <string> | "[unassigned]" | Required processor architecture |
| processor_type | <string> | "[unassigned]" | Required processor type |
| processor_property_name | <string> | "[unassigned]" | Required processor property |
| processor_property_value | <string> | "[unassigned]" | Required processor property value |
| processor_internal_memory_size | <string> | "[unassigned]" | Required internal device memory |
| processors | List<@Processor> | "{}" | Required processors |
| memory_size | <string> | "[unassigned]" | Required memory size in GBs |
| memory_type | <string> | "[unassigned]" | Required memory type (SRAM, DRAM, etc.) |
| storage_size | <string> | "[unassigned]" | Required storage size in GBs |
| storage_type | <string> | "[unassigned]" | Required storage type (HDD, SSD, etc.) |
| operating_system_type | <string> | "[unassigned]" | Required operating system type (Windows, MacOS, Linux, etc.) |
| operating_system_distribution | <string> | "[unassigned]" | Required operating system distribution (XP, Sierra, openSUSE, etc.) |
| operating_system_version | <string> | "[unassigned]" | Required operating system version |
| wall_clock_limit | <string> | "[unassigned]" | Maximum wall clock time |
| host_queues | <string> | "[unassigned]" | Required queues |
| app_software | <string> | "[unassigned]" | Required applications that must be available within the remote node for the task |

[3] https://pm.bsc.es/ompss

| Annotation | Value type | Default value | Description |
|---|---|---|---|
| processorType | <string> | "CPU" | Required processor type (e.g. CPU or GPU) |
| computingUnits | <string> | "1" | Required number of computing units |
| name | <string> | "[unassigned]" | Required processor name |
| speed | <string> | "[unassigned]" | Required processor speed |
| architecture | <string> | "[unassigned]" | Required processor architecture |
| propertyName | <string> | "[unassigned]" | Required processor property |
| propertyValue | <string> | "[unassigned]" | Required processor property value |
| internalMemorySize | <string> | "[unassigned]" | Required internal device memory |

*Figure 4.4 - Arguments of the @constraint decorator in PyCOMPSs*

In the last release of PyCOMPSs, we have extended the parameters and the decorators set with new ones that are relevant for the AI-SPRINT design tools. The user can define the timeout of a task within the @task decorator with the *time_out=<TIME_IN_SECONDS>* hint. The runtime will cancel the task if the time to execute the task exceeds the time defined by the user.

The *@http* decorator (see Figure 4.5) can be used for the tasks to be executed on a remote Web Service via HTTP requests. This decorator, together with the definition of the resource(s) in resources and project files of the runtime allows it to execute GET/POST requests to the service URL. Moreover, Python parameters can be added to the request query as shown in the example (between double curly brackets).

```
from pycompss.api.task import task
from pycompss.api.http import http

@http(service_name="service_1", request="GET",
      resource="get_length/{{message}}")
@task(returns=int)
def an_example(message):
    pass
```

*Figure 4.5 – Example of the use of the @http decorator in PyCOMPSs*

For POST requests (Figure 4.6) it is possible to send a parameter as the request body by adding it to the payload argument. In this case, the payload type can also be specified ('application/json' by default). If the parameter is a FILE type, then the content of the file is read at runtime in the PyCOMPSs master node and added to the request as request body.

```
from pycompss.api.task import task
from pycompss.api.http import http

@http(service_name="service_1", request="POST", resource="post_json/",
      payload="{{payload}}", payload_type="application/json")
@task(returns=str)
def post_with_param(payload):
    pass
```

*Figure 4.6 - @http decorator with payload arguments*

For the cases where the response body is a JSON formatted string, PyCOMPSs' HTTP decorator allows response string formatting by defining the return values within the produces parameter. In Figure 4.7, the return value of the task would be extracted from 'length' key of the JSON response string:

```
from pycompss.api.task import task
from pycompss.api.http import http

@http(service_name="service_1", request="GET",
      resource="produce_format/{{message}}",
      produces="{'length':'{{return_0}}'}")
@task(returns=int)
def an_example(message):
    pass
```

*Figure 4.7 - Example of response type for a JSON string*

It is also possible to take advantage of INOUT python dicts within HTTP tasks. In this case, string updates can be used to update the INOUT dict (Figure 4.8).

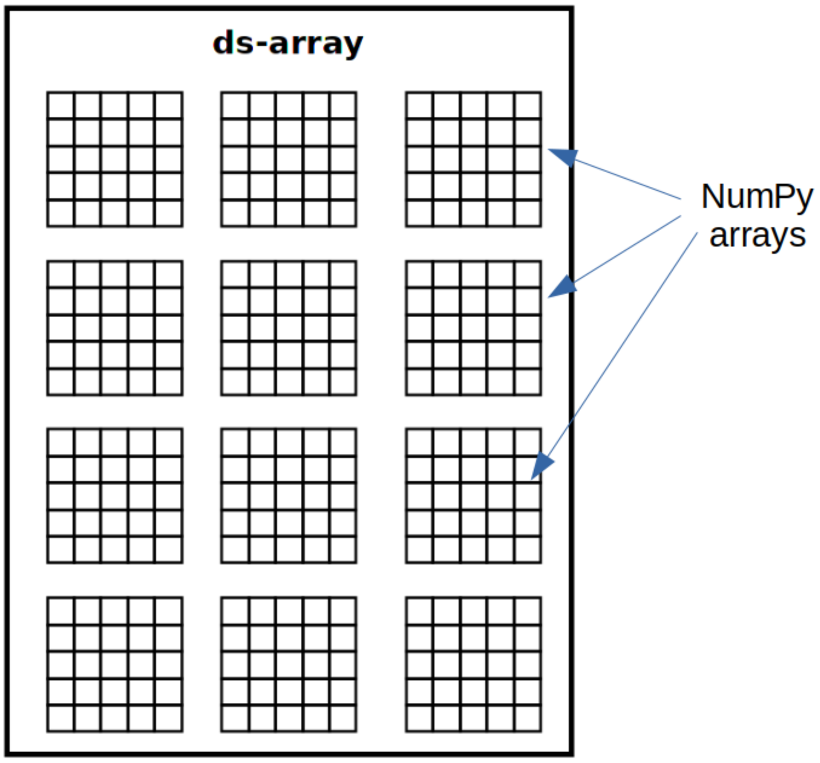
```
@http(service_name="service_1", request="GET",
      resource="produce_format/test",
      produces="{'length':'{{return_0}}','child_json':{'depth_1':'one',
'message':'{{param}}'}}",
      updates='{{event}}.some_key = {{param}}')
@task(event=INOUT)
def http_updates(event):
    """
    """
    pass
```

*Figure 4.8 - @http decorator with INOUT arguments of the tasks*

In the example above, 'some_key' key of the INOUT dict param will be updated according to the response. Please note that the {{param}} is defined inside produces. In other words, parameters that are defined inside *produces* string can be used to update INOUT dicts.

### 4.3.2 dislib

| | |
|---|---|
| **Identification** | dislib - distributed machine learning library |
| **Type** | Library |
| **Purpose** | Provide parallelised Machine Learning algorithms |
| **Function** | The Distributed Computing Library (dislib) is a Python library built on top of PyCOMPSs that provides distributed mathematical and machine learning algorithms through an easy-to-use interface. dislib abstracts Python developers from all the parallelisation |

| | |
|---|---|
| | details and allows them to build large-scale machine learning workflows in a completely sequential and effortless manner. |
| **High level Architecture** | dislib is a subcomponent of the design tools layer. dislib is a collection of PyCOMPSs applications that exposes two main interfaces to developers: 1) a distributed data structure called distributed array (ds-array), and 2) an estimator-based API. A ds-array is a 2-dimensional matrix divided in blocks that are stored across different computers. Ds-arrays offer a similar API to NumPy [Numpy2011], which is one of the most popular numerical libraries for Python. The difference between NumPy arrays and ds-arrays is that ds-arrays are internally parallelised and use distributed memory. This means that ds-arrays can store and process much larger data than NumPy arrays.<br> <br>The typical dislib application consists of the following steps:<br>1) Load data into a ds-array<br>2) Instantiate an estimator object with parameters<br>3) Fit the estimator with the loaded data<br>4) Retrieve information from the estimator or make predictions on new data<br>Each machine learning model in dislib is enclosed in a class implementing the estimator interface, where an estimator is anything that learns from data that implements two main methods: *fit* and *predict*. |
| **Dependencies** | dislib is based on PyCOMPSs |
| **Interfaces** | dislib provides a scikit-learn inspired interface for the different algorithms (i.e., fit, predict, etc.). This makes dislib's interface easy to learn to users already familiar with scikit-learn, and allows a smooth transition of existing codes from scikit-learn to dislib. |
| **Data** | dislib processes data through the distributed array abstraction: A built-in 2-dimensional array (sharded both by row and by column) that can be operated in |

| | |
|---|---|
| | parallel, and that is used as the main input for the different algorithms. Distributed arrays store samples and labels in a distributed way that works as a regular Python object from the user point of view.<br>Methods for loading data from files in common formats, such as CSV and LibSVM are also provided. |
| **Needed improvement** | Add methods to save a trained model and load it for inference tasks.<br>Extend the CSVM algorithm with multi-level classification.<br>Include annotations in dislib for QoS scheduling. |
| **Implemented Improvements for the First Release** | First prototype of save and load mechanisms. |
| **Release Version & Repository** | The software is available at https://github.com/bsc-wdc/dislib and from the AI-SPRINT GitLab https://gitlab.polimi.it/ai-sprint/dislib |

### Detailed Description of Activities for the First Release

Inspired by scikit-learn, dislib provides an estimator-based interface that improves productivity by making algorithms easy to use and interchangeable. This interface also makes programming with dislib very easy to scientists already familiar with scikit-learn. dislib leverages the distributed data structure (ds-array) that can be operated as a regular Python object. The combination of this data structure and the estimator-based interface makes dislib a distributed version of scikit-learn, where communications, data transfers, and parallelism are automatically handled behind the scenes by the PyCOMPSs runtime.

In dislib all the ML methods are provided as scikit-learn estimator objects. An estimator is anything that learns from data given certain parameters. dislib estimators implement the same API as scikit-learn, which is mainly based on the fit and predict operators.

The typical workflow in dislib consists of the following steps:

- Reading input data into a ds-array
- Creating an estimator object
- Fitting the estimator with the input data
- Getting information from the model's estimator or applying the model to new data

```
from dislib.data import load_txt_file
x = load_txt_file("data.csv", block size=…)
kmeans = KMeans(n_clusters=10)
kmeans.fit(x)
print(kmeans.centers)
```

*Figure 4.9 - dislib training example code*

Figure 4.9 shows an example of use of the dislib to train a K-means model and to print the coordinate of the resulting centers. It is worth noting that, although the code of a dislib application looks completely sequential, all dislib algorithms and operations are parallelised by using PyCOMPSs. The annotations as tasks definition and constraints are already defined in the dislib source code. The QoS and privacy annotations developed by the project have to be included in the source code of the library in the next releases.

### Interface for ds-array management

Distributed arrays (ds-arrays) are the main data structure used in dislib. In essence, a ds-array is a matrix divided into blocks that are stored remotely. Each block of a ds-array is a NumPy array or a SciPy CSR[4] matrix, depending on the kind of data used to create the ds-array. dislib provides an API similar to NumPy to work with ds-arrays in a completely sequential way. However, ds-arrays are not stored in the local memory. This means that ds-arrays can store much more data than regular NumPy arrays. The degree of parallelisation with PyCOMPSs is controlled using the array's block size. Block size defines the number of rows and columns of each block in a ds-array. Sometimes a ds-array cannot be completely split into uniform blocks of a given size. In these cases, some blocks of the ds-array will be slightly smaller than the defined block size.

dislib provides a set of routines to create ds-arrays from scratch or using existing data. The API reference contains the full list of available routines. For example, *random_array* can be used to create a ds-array with random data. Another way of creating a ds-array is by reading data from a file. dislib supports common data formats, such as CSV and SVMLight, using load_txt_file and load_svmlight_file.

In this first release we have implemented a classificator to support the development of the personalised healthcare use case. The CascadeSVM estimator implements a version of support vector machines that parallelises training by using a cascade structure. The algorithm (Figure 4.10) splits the input data into N subsets, trains each subset independently, merges the computed support vectors of each subset two by two, and trains again each merged group of support vectors. One iteration of the algorithm finishes when a single group of support vectors remains. The final support vectors are then merged with the original subsets, and the process is repeated for a fixed number of iterations or until a convergence criterion is met. The fitting process of the CascadeSVM estimator creates the first layer of the cascade with the different row blocks of the input ds-array. This means that the estimator creates one task per row block at the first layer, and then creates the rest of the tasks in the cascade. Each of these tasks use scikit-learn's SVC internally for training and load a row block in memory. The maximum amount of parallelism of the fitting process is thus limited by the number of row blocks in the input ds-array. In addition to this, the scalability of the estimator is limited by the reduction phase of the cascade.

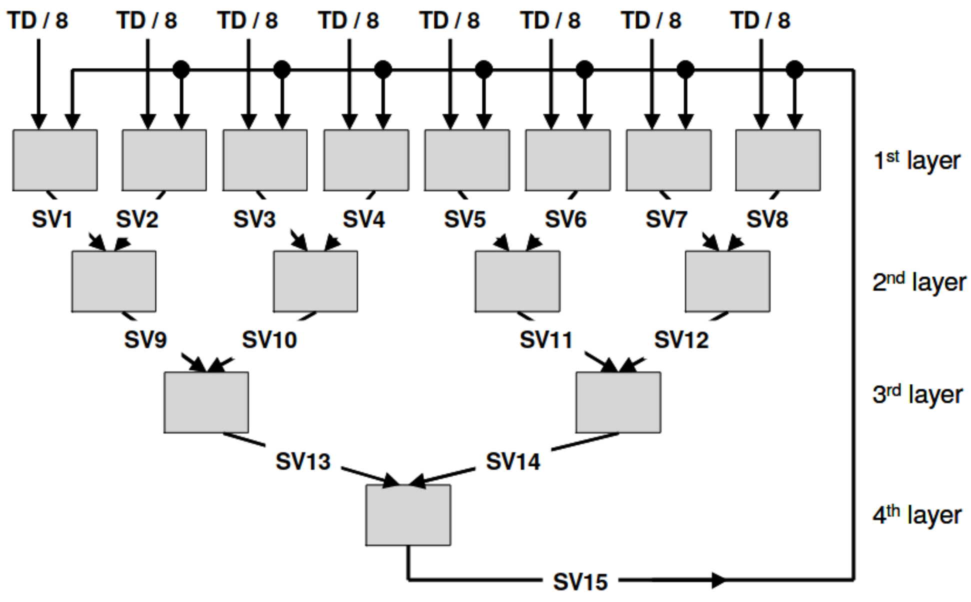


*Figure 4.10 - Cascade Support Vector Machine*

## 4.4 Quality Annotations

The aim of the Quality Annotations is to provide the user with a means to specify, through code decorators, constraints that could be evaluated at runtime to drive the scheduling and the security policies.

[4] https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html#scipy.sparse.csr_matrix

| | |
|---|---|
| **Identification** | Quality Annotations |
| **Type** | A set of modules and parsers |
| **Purpose** | Provide annotation for application components |
| **Function** | The module allows users to annotate application components to specify quality and security constraints and a YAML file to describe the system architecture. To this purpose, the module makes use of Python decorators, which wrap the functions of the components involved in the constraint. Furthermore, decorators compute runtime information that is communicated to the Monitoring Subsystem (see *AI-SPRINT Deliverable D3.2 First release and evaluation of the monitoring system*). An initial version of the secure policy will be automatically generated by a SCONE parser. The YAML system description will be transformed by a SPACE4AI-D parser to generate the initial JSON file which will be completed by the Application architect. The module provides also an additional annotation to enable users to define a partitionable DNN, i.e., a network that can be partitioned across multiple devices. |
| **High level Architecture** | Annotations are implemented as Python decorators, with a structure represented by the pseudo-code reported below. The *wrapper* is the core function of the decorator, which is in charge of executing the annotated function (*func*) together with additional AI-SPRINT code for monitoring purpose. The function implementing the decorator, i.e., *decorator*, is returned by the *annotation* function. This mechanism of nested functions allows the use of decorators with arguments. Indeed, the *annotation* is associated with a set of arguments, i.e., the *annotation_args*, which represent the parameters of the corresponding quality constraint.<br><br>`# Quality annotation (decorator)`<br>`def annotation(*annotation_args):`<br>`  def decorator(func):`<br>`    @functools.wraps(func)`<br>`    def wrapper(*args, **kwargs):`<br>`      # Compute runtime information (pre)`<br>`      ...`<br>`      # Call wrapped function`<br>`      func(*args, **kwargs)`<br>`      # Compute runtime information (post)`<br>`      ...`<br>`      # Communicate with the monitoring tool`<br>`      ...`<br>`    return wrapper`<br>`  return decorator` |
| **Dependencies** | A quality annotation depends on *functools* Python module, that allows to act on or return other functions. |
| **Interfaces** | The provided interface is the annotation itself, which is used by the developer to annotate the desired function through the '@annotation' string placed on top of the function definition. |

| | |
|---|---|
| **Data** | The annotation takes as input a set of parameters that define specifics about the constraint. |
| **Needed improvement** | Quality annotations corresponding to time constraints, i.e., 'execTime' annotation and security annotations have been designed, but need to be implemented and integrated in the SDK, together with the function to communicate to the Monitoring Subsystem. |
| **Implemented Improvements for the First Release** | The initial work focused on the definition of the quality annotations and analysis of the dependencies among the different design time and runtime components. Decorators and parsers are not yet available but will be implemented by M18 and be part of the first integrated release of the AI-SPRINT framework. The next releases of the design tools will also include an annotation (and a tool) for defining partitionable DNNs (i.e., DNNs which can be split and executed at multiple layers of the computing continuum). |
| **Release Version & Repository** | N.A. |

### *4.4.1 Detailed Description of Activities for the First Release*

#### Annotations for QoS constraints

One of the objectives of the design environment in AI-SPRINT is to provide an abstract layer between the developed application to be deployed and the computing continuum, by masking the management of the available resources to developers. This translates into a simplification from the perspective of an application developer, who can concentrate mainly on the implementation of the algorithm itself rather than on the complex mechanism behind the deployment process. At the same time, the goal is to give enough control to the developer during the implementation to constrain the allocation of resources in the cloud continuum at design time. The idea is to give developers the capability of defining QoS and security constraints that must be satisfied by the designed deployment. To this purpose, the AI-SPRINT SDK will be equipped with a set of programming abstractions that AI-SPRINT users can exploit to define performance parameters at the application level.

Abstractions in AI-SPRINT will be implemented as Python decorators, which can be used to *annotate* the components composing the developed applications. At design time, each component will be placed on a specific resource, selected from the set of candidates, by considering the constraint defined through the annotations, if provided by the user. In the specific, quality annotations make it possible to define *time constraints* on the application components, i.e., the user can specify a required execution time of the components. The set of candidate resources will be then skimmed, by selecting those that meet the desired time requirements. In the following, annotations for time constraints are presented in detail. Figure 2.2 will be used as a guideline during the dissertation.

#### Time Constraints Annotations

As already introduced, AI-SPRINT will give developers the capability of associating a desired execution time to the application components. For instance, by looking at the workflow of the example in Figure 2.2, we could require that the "*Anon and split*" component must be executed in a maximum time of 0.5 seconds at runtime. In the proposed AI-SPRINT view, this will translate into annotating the component with a Python decorator named *execTime,* which will take as argument the maximum execution time required by the user. The decorator provides a wrapper of the annotated component, allowing executing additional Python code each time the component is executed. Specifically, the *execTime* decorator will extract information about the runtime of the component, i.e., it will compute the starting and ending time before and after the execution of the component, respectively, and the overall execution time. A possible implementation is given in Figure 4.11. The *func* argument is the decorated function and *to_monitoring_tool* is a function that will be part of the AI-SPRINT API that will store the execution time information to InfluxDB[5] database, to be used at runtime by the monitoring tool (for additional details see D3.2). The arguments of the decorator are *time_thr*, i.e., the required upper bound for the execution time, and *prev_components.* The role of the latter will be clarified later in the document.

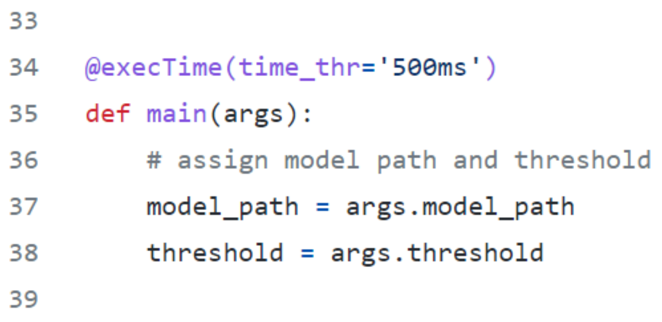
```python
import functools
import time

def execTime(time_thr=None, prev_components=None):
  def decorator_execTime(func):
      """
        Compute the execution time and
        notify the monitoring tool.
      """
      @functools.wraps(func)
      def wrapper_execTime(*args, **kwargs):
          start_time = time.time()
          value = func(*args, **kwargs)
          end_time = time.time()
          run_time = end_time - start_time
          to_monitoring_tool(func.__name__, run_time, start_time, end_time)
          return value
      return wrapper_execTime
  return decorator_execTime
```

*Figure 4.11 - Example implementation of the execTime decorator*

Annotating a component means wrapping the corresponding *main* function in the user code with the decorator. For instance, in the example of putting a time constraint on the execution of the anonymization component, it means annotating the *main* function in the *auto_blur_image.py* script[6]. In the following Figure 4.12, a code snippet shows how the main function of the anonymization component should be annotated in the considered example (maximum execution time of 0.5 seconds).

[5] https://www.influxdata.com/

[6] https://gitlab.polimi.it/ai-sprint/scar/-/blob/master/examples/mask-detector-workflow/blurry-faces/src/auto_blur_image.py



```python

@execTime(time_thr='500ms')
def main(args):
    # assign model path and threshold
    model_path = args.model_path
    threshold = args.threshold

```

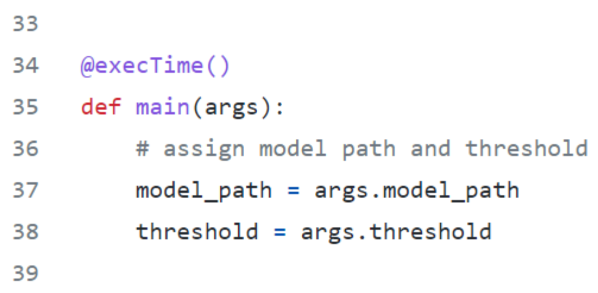
*Figure 4.12 - Code snippet from the auto_blur_image.py script in which the main function has been decorated with execTime decorator to add a time constraint to the function*

Time constraints can be categorised into *local* and *global* time constraints. Local time constraints refer to single components, like the example proposed in Figure 4.12, while global time constraints can be associated with a group of consecutive components. Considering again the current mask detection example application, the user could require, for instance, that the consecutive execution of both the anonymization and mask detector components must take at most 1 second. In order to define a global constraint at the level of the code a new strategy is introduced. Defining as *path* a group of consecutive components involved in a constraint, the current proposal for defining a global time constraint is to employ the following annotation mechanism:

1. Decorate all the functions in the path except the last one with the *execTime* decorator with no arguments, i.e., the *time_thr* is None (default value). These annotations only serve to mark these functions as part of the global constraint.
2. Decorate the last function with the *execTime* decorator with two arguments: the *time_thr*, which, in this case, does not refer to the required execution time of the single component but it refers to the required execution time of the whole path. Finally, an additional argument of the annotation comes into play, i.e., the *prev_components*, which is the list of names of the previous components in the constrained path. The last annotated component must have a non-empty *prev_components*, otherwise the constraint will be considered as a local one.

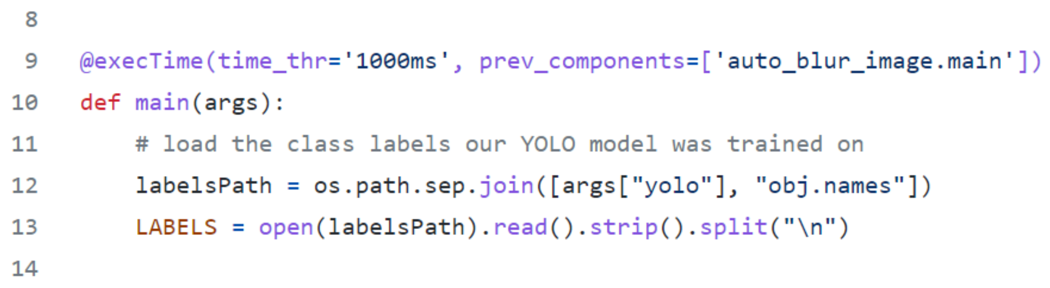
In the considered example, in which we supposed a required execution time of 1 second for the whole application, the developer must decorate the *main* function of the *auto_blur_image.py* script with *@execTime()* with no arguments, since it is involved in the global constraint and it is not the last component in the path. Then, the main function in the *mask-detector-image.py* script[7], which is the last component in the path, must be annotated by defining both the *time_thr* and the *prev_components.* In Figure 4.13, the code snippets from the *auto_blur_image.py* (a) and *mask-detector-image.py* (b) files are reported, extended with the annotations specific for the considered global constraint in the example.

[7] https://gitlab.polimi.it/ai-sprint/scar/-/blob/master/examples/mask-detector-workflow/mask-detector/mask-detector-image.py



```python

@execTime()
def main(args):
    # assign model path and threshold
    model_path = args.model_path
    threshold = args.threshold

```

(a)



```python

@execTime(time_thr='1000ms', prev_components=['auto_blur_image.main'])
def main(args):
    # load the class labels our YOLO model was trained on
    labelsPath = os.path.sep.join([args["yolo"], "obj.names"])
    LABELS = open(labelsPath).read().strip().split("\n")

```

(b)

*Figure 4.13 - Code snippets from the auto_blur_image.py (a) and mask-detector-image.py (b) scripts in which the main functions have been annotated considering a global time constraint of 1 second*

Time-constraints decorators can only compute and store the execution time of the function being wrapped. Nevertheless, in the case of global constraints we need to compute the execution time of the whole constrained path at runtime. The proposed solution is to delegate in OSCAR, which is in charge of orchestrating the application components for the inference task, to trace the computation through the paths defined by the global constraints. This is done by integrating ad-hoc monitoring services, which are in charge of tracing the computation of a specific file, from its upload to the end of the application, and of storing the runtime information to InfluxDB for monitoring. Tracing the computation of a specific file through the deployed application is not easy due to the asynchronous execution of the components. To this purpose, monitoring services are integrated with a mechanism of fingerprinting, through which the file is intercepted after the upload, and it is marked with a unique identifier.

### Parsing the annotations

Once the components have been annotated by the application developer, we need to collect and store this information for later use in the AI-SPRINT framework. To this purpose, the AI-SPRINT API will include a *code-parsing* tool, which will parse the code of the application searching for the decorated functions. The output of the code parsing is a list of annotations together with the corresponding parameters. This information will be routed to different tools in the AI-SPRINT framework, depending on the kind of annotation, e.g., time constraints will be used by OSCAR for monitoring local/global constraints or by the SPACE4AI-D, at design time, to place the components on the candidate resources.

### Security constraints

In addition to giving the developer the option to define a desired execution time, developers can furthermore annotate tasks to receive certain security guarantees while executing. For this, we define the following constraints that can be used to annotate a *main* function of a Python script:

```
@security(trustedExecution=true,network=true,filesystem=true)
```

We will now describe the different properties and meaning of the above annotations.

The `trustedExecution` translates to the use of **trusted execution environments** (TEE). Hence, the process that is executing the task will run in a TEE such as Intel SGX. This will also lead to the fact that such tasks will only be scheduled on processes that run on nodes that provide the necessary hardware support. If no such node is available, the task will run on a node that provides at least **secure** and **measured** boot mechanisms such that the used operating system and kernel can be trusted (for additional details, see *AI-SPRINT Deliverable D4.1 Initial release and evaluation of the security tools*).

The `network` flag ensures that all TCP connections will be wrapped using SCONE's **network shielding layer**. Alternatively, service meshes such as Istio[8] that provide secure communication will be deployed along the processes on a Kubernetes cluster.

Similar as with network encryption, the `filesystem` flag ensures that all files written and read by the process executing the tasks are encrypted. This is achieved by intercepting the system calls through SCONE in a similar fashion as for the network shielding.

The security annotations are parsed prior to the deployment and execution of the respective process and used to establish the appropriate configuration needed to ensure these properties. More details about the configuration, such as how these annotations will be used to form security policies is provided in D4.1 *"First release and evaluation of the AI-SPRINT security tools"*.

### Deep Neural Network Partitioning Annotation

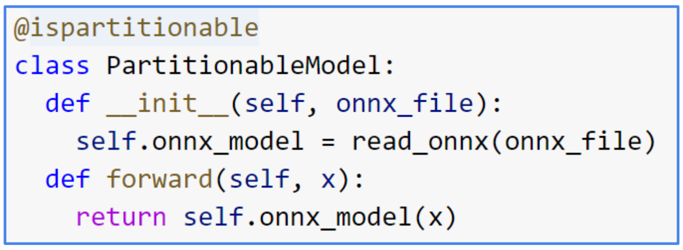
SPACE4AI-D (see Section 4.7) provides the optimal components placement among the candidate resources in the computing continuum. Additionally, for each component multiple partitions may exist, since one of the features offered by the AI-SPRINT is the possibility of partitioning DNNs across different resources. The goal of the design tool is also to find the optimal DNN partitioning which guarantees the memory and QoS constraints.

AI-SPRINT will enable users to inform SPACE4AI-D if a particular DNN used in the developed application can be partitioned. To this purpose an additional annotation is proposed, named *ispartitionable*, which is implemented as a Python decorator, as in the case of the previously introduced annotations. This decorator will be used by the developers to annotate directly in the application code any DL model that can be partitioned, i.e., whose layers can be deployed on different devices. During the parsing of the application code this annotation is recognized and the design tool, i.e., SPACE4AI-D, is informed of the possibility of partitioning the corresponding annotated models. Finally, the candidate deployments for the network partitions are obtained based on the provided memory and time constraints.

In order to consider an homogeneous representation among the possible implementations of DL models, and also among different DL frameworks, we require the developers to represent the partitionable DNNs through the Open Neural Network Exchange (ONNX[9]) format, which serves as an intermediate model representation providing a powerful sharing mechanism among frameworks. The ONNX format allows to directly obtain the computational graph of the DL model, where the nodes are the layers of the DNN, thus providing the DAG representation required by the SPACE4AI-D tool to define the candidate deployments. In order to obtain the needed representation the annotation tool will require the employed DL models to be implemented as Python classes exhibiting a simple interface, which allows to read and run the model from an ONNX file. An

[8] https://istio.io/
[9] https://onnx.ai/

example implementation of such a Python class (as pseudo-code) is provided in Figure 4.14, already annotated with the *ispartitionable* decorator. The class provides two methods, i.e., the _*_init_*_ function and the *forward* function. The structure of the class is very similar to the standard implementation of models in the most common DL frameworks. The _*_init_*_ function will take the complete path to the ONNX file as input, i.e., *onnx_file*, representing the entire DNN computation. The model is then read from its ONNX representation and saved as an attribute of the class. Finally, the *forward* function will be executed at runtime to compute the model output given the provided input *x*. In order to exploit the partitioning feature provided by AI-SPRINT, the developers are simply required to provide the employed DNNs as instances of the *PartitionableModel*. At design time, the SPACE4AI-D tool is informed of the presence of a partitionable model, by parsing the code annotations, and uses the provided ONNX format to compute the partitions and the candidate deployments.



```
@ispartitionable
class PartitionableModel:
  def __init__(self, onnx_file):
    self.onnx_model = read_onnx(onnx_file)
  def forward(self, x):
    return self.onnx_model(x)
```

*Figure 4.14 - Example implementation (as pesudo-code) of a partitionable DL model, which is further annotated to signal the design tool that the model is partitionable*

## 4.5 Performance Models

This section introduces performance models based on a-MLLibrary machine learning library that aims to predict the execution time of AI application components under different configurations.

| | |
|---|---|
| **Identification** | Performance models |
| **Type** | Subprogram |
| **Purpose** | Provide to the application manager/AI-SPRINT runtime components means to choose an appropriate configuration to: 1) avoid applications performance violations, 2) avoid under or overestimation of the utilisation of the continuum resources, and 3) predict the execution time of Deep Learning components on a target configuration. The final goal is to provide solutions for selecting the most appropriate system configuration for executing the required application component/inference pipeline/training job to fulfill QoS requirements while minimising operational costs. |
| **Function** | Performance models are based on the *a-MLLibrary* machine learning library initially developed within the ATMOSPHERE project and currently maintained and extended by the EuroHPC LIGATE project (https://www.ligateproject.eu). This library supports feature selection, hyperparameter tuning, and model selection and can generate the most accurate regression model for a specific task. Using the generated regression model, the execution time of inference components or pipelines or training jobs can be predicted. |
| **High level Architecture** | The tool is a separate component. The current implementation is based on Python and relies on standard Python libraries (mainly scikit, pandas and hyperopt). |
| **Dependencies** | The tool requires as input the execution time of the inference component/training job |

| | |
|---|---|
| | under different configurations. This information can be obtained by the AI-SPRINT monitoring infrastructure or by code instrumentation. The tool provides as output the machine learning models as pickle[10] files. The other tools relying on the performance models are SPACE4AI-D, SPACE4AI-R, the Job scheduler and the POPNAS component. |
| **Interfaces** | Command line tool. Detailed user guide available at https://gitlab.polimi.it/ai-sprint/a-mllibrary. |
| **Data** | The execution time of the inference component/training job under study under different configurations are provided as CSV files. Moreover, an input Python file specifies, at high level, the set of methods to be evaluated for identifying the best model, the set of hyper-parameters, the set of features and, possibly, how to perform features augmentation (e.g., compute the inverse of some features, perform product of features up to a given degree).<br>The tool provides as output the machine learning models as pickle files and detailed information on the accuracy achieved by the candidate performance models, the set of features selected and the hyper-parameters setting for the most accurate models. |
| **Needed improvement** | The core of the *a-MLLibrary* is already available. Furthermore, a performance profiling tool, *a-GPUBench*, based on TensorFlow 2.0 able to automate the execution of the training of the application under study under different configurations has been developed to gather the data needed to train the models (see Appendix D). |
| **Implemented Improvements for the First Release** | Profiling tools for the inference of an application based on OSCAR are under development. Furthermore, for the training, Pytorch will be also supported. |
| **Release Version & Repository** | The software of *a-MLLibrary* is available at: https://github.com/a-MLLibrary/a-MLLibrary, and https://gitlab.polimi.it/ai-sprint/a-mllibrary, while the performance profiling tool (for the TensorFlow jobs training) *a-GPUBench* is available at https://gitlab.polimi.it/ai-sprint/a-gpubench |

### *4.5.1 Detailed Description of Activities for the First Release*

Once an AI application is designed, performance models can help to anticipate the performance of the application components before the production deployment or throughout revision cycles. The goal of AI-SPRINT is to develop, on one side, ML models able to predict AI application components performance under different configurations and deployment settings at the full computing continuum stack and, on the other side, a set of tools that will automate AI application components profiling to gather the training data required by the ML models. Figure 4.15 illustrates the machine learning-based methodology AI-SPRINT is developing to identify models able to predict the performance of the analysed applications when executed on different hardware (characterised, e.g., by a different number or type of GPUs) or software settings (e.g., different number of training iterations or number of inner modules of a ResNet deep learning model). As a first step (see Figure 4.15(a)), we must profile the AI component under study on some reference machines and under multiple configuration settings.

[10] https://docs.python.org/3/library/pickle.html

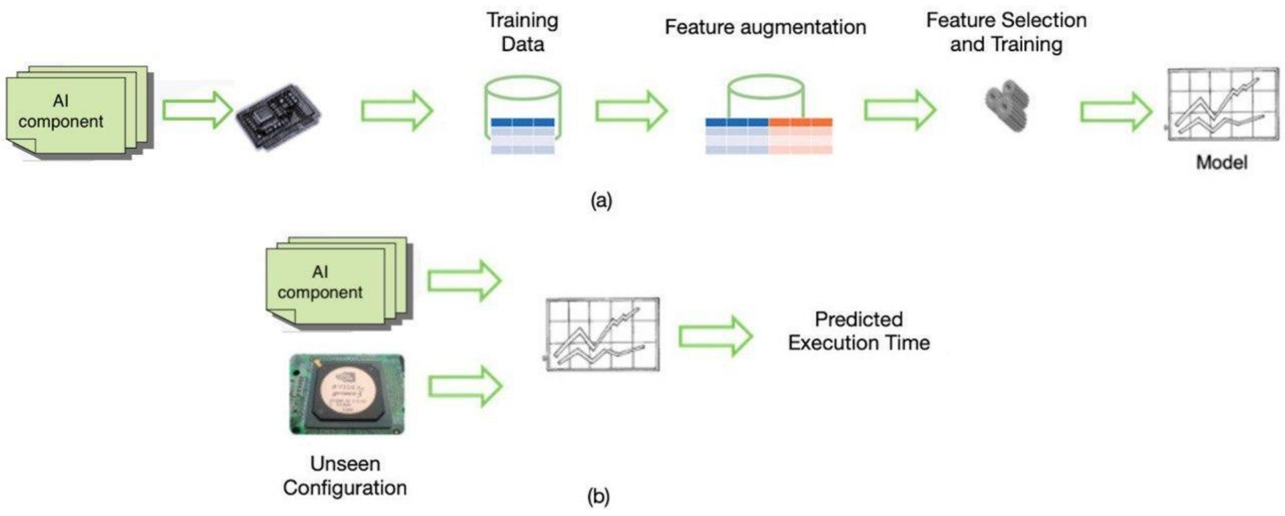


*Figure 4.15 - Performance Modeling Methodology based on Machine Learning (model building (a) and prediction (b) step)*

The aim of the profiling is to measure the AI component execution time by varying some hardware or software settings (e.g., the batch size and number of iterations of a training job, or the memory assigned to a Function as a Service (FaaS) component). Once the relevant training dataset for the performance regressors has been collected, the building of the model is performed by considering a relevant set of features that differ according to the target application and deployment.

As a second step (see Figure 4.15(b)), we use the performance regressor models to predict the AI component execution time on unseen configurations. The general aim of ML is to infer the input/output relationships that map the AI component and system characteristics onto the target performance indicators through statistical models, without requiring knowledge of internal system details. It is noteworthy that ML provides good interpolation capabilities, i.e., it can predict values in areas of the features space that have been sufficiently observed during the training phase [Didona2014]. In our approach, we also test extrapolation capabilities, i.e., we investigate if performance regressor models can predict values in regions of the parameters space not sufficiently explored. Testing the performance prediction models' generalisation capabilities is of the utmost importance and has a significant practical implication: generalisable models enable accurate performance prediction in conditions that users may not be able, or willing, to empirically investigate.

The AI-SPRINT approach is built on a machine learning library called *a-MLLibrary,* which was initially developed in the ATMOSPHERE project. This high-level library helps us to know how to perform feature selection, hyper-parameters tuning, etc., and we can predict the execution time of AI components and feed this estimated time to SPACE4AI-D, SPACE4AI-R, the POPNAS component, and the job scheduling tools.

AI-SPRINT is developing two kinds of models: the first class of models is a *gray box per-layer model,* which aims at predicting the performance of an AI component based on a DNNs, layer by layer. The second class of models is a *black-box method* which aims at modelling the execution time for training and inference of DNNs according to the features and historical data. The end-to-end model is usually accurate and allows us to predict different configurations also in terms of hardware while the per-layer model cannot be generalised on the hardware side, but it is capable of being generalised on the application side which means if we train per-layer model on some reference applications, we can predict the execution time of the applications that have never seen before without the need of profiling.

We describe both the models in the following and then compare the end-to-end with the per-layer-model in the performance evaluation section for DNN job training. POPNAS, SPACE4AI-D, SPACE4AI-R tools rely on per-layer models while the job scheduling tool relies on end-to-end models. Finally, for PyCOMPSs and dislib applications which provide general AI models (not limited to DNN) and mainly rely on CPUs and distributed servers, we will develop ad-hoc models whose main feature is the total number of available cores.

### Per-layer model

All Convolutional Neural Networks (CNNs) comprise a number of layers belonging to a limited collection of basic categories. Building upon this observation, we propose to learn several regression models, in order to characterise common layer types. In this way, it is possible to estimate the performance of a wide range of CNNs even without previous experience with the specific structure, just relying on these low-level layer models. Our approach is based on two basic assumptions, in order to make the problem easily tractable and improve model generality. When working with GPUs, applications attain their best performance when they fully leverage the data parallel architecture, hence we expect CNN designers, as well as users, to tune networks accordingly. Such a consideration means that, mostly, the execution of different layers will not overlap, whence follows that layer running time predictions can just be summed to obtain an estimate of the overall execution time:

$$\hat{t}^{CNN} = I \sum_{l \in L} \hat{t}_l \tag{1}$$

where, in case of job training, $I$ is the total number of training iterations. The second aspect to take care of is the choice of features to feed into the regression models. A simplistic idea could be using all the various hyper-parameters as features, but this would make for a difficult to interpret and hardly generalizable formulation. On the other hand, we propose to summarise all the relevant characteristics in a single feature: layers computational complexity in terms of simple primitives available on GPGPUs, a good metric for layer workload. To exemplify the derivation of computational complexity from network hyper-parameters, here we discuss the method for convolutional layers. The convolutional part operates on 3D tensors. Let us denote with $C$ the number of channels, $H$ the number of rows or height, $W$ the number of columns or width. The amount of zero padding on each side of the matrices is $P$, whilst the stride is $S$. Subscripts distinguish properties of the input, output, and filters, e.g., $H_{in}$ , $H_{out}$ , $H_f$. Cardinalities are used as a shorthand for index sets, as in $i \in H_{in}$.

Some layers just apply predetermined operations, possibly depending on hyper-parameters under users' control. In contrast, convolutional and fully connected layers have a set of learnable weights that evolve during the training phase via back propagation. The number of weights depends on their hyper-parameters. Each output channel is obtained by convolving a different filter with the input tensor; hence the count of learnable parameters is given by:

$$(H_f W_f C_{in} + 1)\, C_{out}$$

Convolution entails multiplying filters of size $H_f W_f C_{in}$ elementwise with input activations and producing as output the sum of all these partial products and an additive bias, hence there are $C_{out}$ filter-bias pairs that contribute all the entries in the tensor plus one coefficient.

The 3D tensors involved in CNNs contain all the partial values, called activations, obtained via the incremental transformations operated by filters. In practice, layers take an input tensor and apply a filter to its entries, thus yielding an output tensor with a possibly different layout. It is possible to compute output dimensions given layer hyper-parameters, specifically filter sides, padding around the edges, and stride. Tensor sizes are relevant because they appear in the formulas for computational complexity, since every output activation comes from one of the several applications of the filter to its inputs: it is common to consider complexity per pixel, as the overall layer operations count is always directly proportional to $H_{out}\ W_{out}$. In particular, continuing our example, convolutional layers can be formalised as the following expression for all $(i, j, k) \in H_{out} \times W_{out} \times C_{out}$:

$$y_{ijk} = b_k + \sum_{t \in H_f} \sum_{u \in W_f} \sum_{l \in C_{in}} w_{tulk} x_{\underline{i}\underline{j}l}$$

where $\underline{i} = \varphi(i,t)$ and $\underline{j} = \psi(j,u)$, while $x$ and $y$ are, respectively, input and output activations. $\varphi$ and $\psi$ associate output and filter indices to the input needed to convolve each activation and their specific functional forms depend on the CNN and its hyper-parameters, in particular padding and stride, but they do not affect the derivation of complexity. Overall, you multiply all the weights times the input activations and accumulate the products on the bias once per output channel, hence convolutional layers require $O(H_f W_f C_{in} C_{out})$ operations per output pixel. Similar considerations enable determining the computational complexity for back propagation. Without spelling out the full details for space limitations, note that propagating deltas to biases requires one operation per channel, whilst doing so for parameters and inputs costs twice as much as the forward pass, because it fundamentally amounts to following the convolution backwards once for weights and once for activations. All in all, each output pixel requires $O((2H_f W_f C_{in} + 1)C_{out})$.

| Layer | Forward | Backward |
|---|---|---|
| Conv | $H_f W_f C_{in} C_{out}$ | $(2H_f W_f C_{in} + 1)C_{out}$ |
| FC | $H_{in} W_{in} C_{in} C_{out}$ | $2H_{in} W_{in} C_{in} C_{out}$ |
| Loss | $4C_{out} - 1$ | $C_{out} + 1$ |
| Norm | $5C_{out} + C_{in} - 2$ | $8C_{out} + C_{in} - 1$ |
| Pool | $H_f W_f C_{out}$ | $(H_f W_f + 1)C_{out}$ |
| ReLU | $3C_{out}$ | $4C_{out}$ |

*Table 4.1 - Operations per output pixel*

Table 4.1 shows formulas for all the kinds of layers, with the computational complexity per output pixel for both the forward and backward passes. Now, using these formulas it is possible to build a dataset where the operations count is associated with the measured execution times of both passes: given this data, we can build a series of models where computational complexity is the only explanatory variable. For every layer category and direction we learn, following the theory for linear regression, a model of the form:

$$t_l = \beta_{0l} + \beta_{1l} c_l + \varepsilon_l$$

is considered, where $t_l$ is the execution time, $c_l$ the complexity, and $\varepsilon_l \sim N(0, \sigma_l^2)$ random errors. With the estimated coefficients $\hat{\beta}$ it is possible to predict both forward and back propagation time, then all the relevant contributions are added to obtain the time taken for one iteration. Multiplying by the overall number of iterations (in case of job training) and summing the terms due to each layer, as in (1), yields a prediction for the full run. In particular, it is possible to predict both training and inference execution times, depending on whether back propagation terms are included in the sum.

The choice of using only computational complexity as independent variable confers a lot of generality, allowing to learn a set of models on data coming from a limited selection of CNNs and to apply it nonetheless to different networks. Anyhow, not adding explicitly any contribution related to hardware in the formulation makes every trained model specific to the deployment where data is extracted.

The experimental setup and result adopted for training the per-layer performance models is presented in Section 4.8.2. In the same section, we report the results of an additional analysis, performed by integrating this model with the Neural Architecture search presented in the next section.

### End-to-end model

The per-layer approach adopts each layer's computational complexity to estimate the layer forward or backward pass execution times. This technique is quite general in its applicability, however the prediction errors tend to increase as more complex networks are considered, since its generality entails some approximations. In the case of a working deployment, it is quite natural to trade off some generality for lower prediction errors, whence the end-to-end method laid out in the following. The basic idea is to extract from historical data, particularly logs of previous runs or traces collected by the monitoring platform, the execution time of the network in its entirety, so as to build a dataset associating these timings to, e.g., batch sizes and number of iterations. Then it is possible to apply regression to a sample in order to obtain a model specialised for the particular CNN and deployment under consideration, but capable of predicting performance with high accuracy.

Deep learning practice usually involves several alternating phases of CNN training and testing. The former iteratively feeds the network with labelled image batches, so that its parameters can change following the direction of the back propagated gradient, whilst the latter evaluates the CNN’s evolving quality in terms of more human readable metrics, rather than the loss function used for training, but without contributing to the learning of weights and biases. For example, generally training is performed minimising a loss function that may be SVM-like or based on cross entropy, but the stopping criterion is likely expressed in terms of classification accuracy or F-score, for unbalanced datasets. Since training involves back propagation, but testing does not, it is necessary to characterize two different models.

The experimental setup and its validation adopted for training the end-to-end performance models is presented in Section 4.8.2.

### PyCOMPSs and OSCAR performance models

The performance prediction of PyCOMPSs and OSCAR applications, which rely mainly on CPUs rather than on GPUs, are currently under development. We consider parallel, distributed systems such that the main feature is given by the number of available cores C.

Models are built by considering the features proposed in [Venkataraman2016] and further analysed in [Maros2019]. Specifically, since the input data size was not available, we focused on the number of CPU cores C used for the computation. We performed feature augmentation by considering also its logarithm log(C), which encodes the cost of reducing operations in parallel frameworks. Moreover, we considered the inverse of the number of cores, to capture any contribution that may come in this way. For the PyCOMPSs application scenario, the inverse of log(C), as well as the crossover terms of all features, up to the second degree, were added when applying Sequential Forward feature Selection (SFS), so as to capture their contribution limiting the overall size of the model. For the OSCAR application performance models, we considered instead the crossover terms of all features, up to the second degree, without applying SFS due to time constraints. We report in Section 4.8.2 the validation performed by focusing on the CSVM and mask detection use cases.

## 4.6 AI Neural Architecture Search

| | |
|---|---|
| **Identification** | POPNAS (Pareto Optimal Progressive Neural Architecture Search) |
| **Type** | Subprogram |
| **Purpose** | Perform an automatic neural architecture search with time-accuracy trade off and Pareto efficiency property. |
| **Function** | POPNAS receives as input a set of configuration parameters and an annotated |

| | |
|---|---|
| | dataset, e.g., images with labels. The goal is to look for the best neural network architecture for the given classification/regression task. The algorithm searches for the best network configuration to achieve the highest accuracy in the lowest possible time by searching the Pareto front of the time-accuracy trade-off. Architectures on the Pareto front are then proposed as possible candidates to select among. |
| **High level Architecture** | POPNAS is a separate component. The current implementation is based on Python and relies on standard Python libraries. POPNAS approach searches for stackable cells which can be stacked effectively to solve the assigned task. To achieve this multiple cell candidates are generated and evaluated by means of training or by means of performance models (to speed up the evaluation process). |
| **Dependencies** | In its current implementation, POPNAS depends on the a-MLlibrary models for the prediction of performance, and on Keras and Tensorflow2 frameworks for the design and the training of the neural architectures. |
| **Interfaces** | The software is currently accessible via command line interface. |
| **Data** | The input data includes the path to a labelled dataset of images (current implementation is tailored to image classification) and a list of training and cells hyperparameters. Cell hyperparameters define the search space of the algorithm and thus the potential cells which can be generated. |
| **Needed improvement** | The second version of the algorithm is already under development. The major improvements of this new version include: <br>● Skip connections among networks blocks and cells <br>● Time prediction improvements thanks to a new feature set <br>● Time prediction improvements through the usage of more complex machine learning models and better features encoding <br>● Pruning of equivalent blocks and cells from the search space to reduce the total search time and improve Pareto variety <br>● Exploration steps consisting in training networks with input and operator values not used or very rarely used in the Pareto front to exit local minima and increase Pareto variety <br>● Additional quality dimensions (e.g., memory requirement of the networks) to partition the DNN across multiple devices in the computing continuum <br>● Implementation and evaluation of early exits to partition the DNN across multiple devices in the computing continuum |
| **Implemented Improvements for the First Release** | The current version of the tool has been developed from scratch within the AI-SPRINT project. This is the first release. |
| **Release Version & Repository** | https://gitlab.polimi.it/ai-sprint/popnas |

### 4.6.1 *Detailed Description of Activities for the First Release*

Neural Architecture Search (NAS [Zoph2016]) is the process to automate searching for the best Deep Neural Network (DNN) architecture for a given task. By exploiting different heuristics and AI techniques, NAS has achieved the state-of-the-art in image classification among other auto machine learning strategies and human being handcrafted architectures.

Almost all the NAS techniques rely on three fundamental steps:

- Definition of a search space: the set of all the possible neural networks to build;
- Search strategy: the algorithm to explore the search space;
- Evaluation strategy: the criterion to evaluate and rank the explored models.

While both the evaluation strategy and the search space have essential importance in the performance and computational costs of autonomous generated models, the literature is often divided according to the most appropriate exploration strategy to adopt, e.g., reinforcement learning, gradient-based optimization, evolutionary algorithm, and bayesian optimization. Despite the progress achieved, the computational costs of these techniques remain too expensive in most scenarios. Indeed, it is necessary to frequently update deep learning architecture in many cases, and the required time can become a fundamental discriminating factor.

Pareto-Optimal Progressive Neural Architecture Search (POPNAS [Lomurno2021]) is a NAS method that, starting from the Progressive Neural Architecture Search (PNAS [Liu2018]) technique, manages the trade-off between time and accuracy via Pareto efficiency. With this technique, it is possible to obtain competitive performance results and massive reductions in model search time with respect to PNAS.

The goal of POPNAS is to keep all the PNAS algorithm advantages while dealing with time constraints to speed up the whole research and to achieve similar accuracy performance. In order to do that, a new time regressor is required, which jointly works with the controller. As shown in Figure 4.16, at each iteration, after models expansion, one predictor, named controller, has to evaluate the accuracy of children architectures, as it is done in PNAS, while another predictor, named regressor, has to predict their training time to achieve the Pareto efficiency simultaneously.

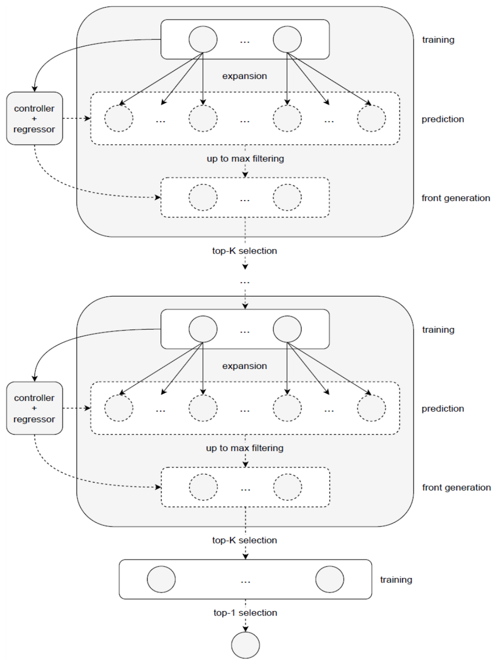


*Figure 4.16 - POPNAS overall schema*

The aim of POPNAS is to search for the most accurate cell structure among those with the lowest training time, pruning out the cells that take more time but have the same accuracy. A cell is a structure composed of blocks as shown in Figure 4.17, i.e., the binary operations searched by the algorithm.

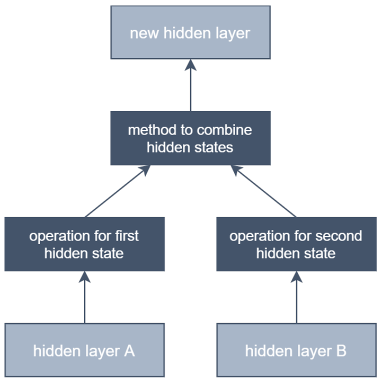

new hidden layer
method to combine hidden states
operation for first hidden state
operation for second hidden state
hidden layer A
hidden layer B

*Figure 4.17 - Representation of a POPNAS general block*

The list of these operations (as in PNAS) includes:

- 3x3 Average Pooling
- 3x3 Max Pooling
- Identity
- 3x3 Convolution
- 3x3 Depthwise Separable Convolution
- 5x5 Depthwise Separable Convolution
- 7x7 Depthwise Separable Convolution
- 1x7 Convolution followed by 7x1 Convolution

A neural network built via POPNAS is a stack of properly connected cells. In the beginning, all the possible cells with only one block are generated. Proceeding with the iterations, the number of blocks is expanded and both the accuracy predictor, i.e., the controller, and time predictor, i.e., the regressor, are updated. They are used to build the Pareto front for iteration i+1 having the information until iteration i. From the Pareto front, only the best K models are selected, trained, used to update the predictors and sent to the next expansion step.

The algorithm is described in detail in [Lomurno2021] while numerical results are reported in Section 4.8.3 for the CIFAR10 dataset together with examples of the architectures designed with the tool.

## 4.7 Application Design Space Exploration (SPACE4AI-D)

In this section, we introduce SPACE4AI-D which is a design time tool and aims to provide an optimal component placement and resource selection for AI applications in the computing continuum.

| | |
|---|---|
| **Identification** | SPACE4AI-D |
| **Type** | Subprogram |
| **Purpose** | Perform and automate design space exploration in order to minimise the execution cost of the AI application while providing response time guarantees.mplements several heuristic algorithms (i.e.., random greedy, local search, tabu search, etc.). |
| **Function** | SPACE4AI-D tool receives resource description, performance model, performance constraints and application DAG as an input and finds the minimum cost solution for component placement and resource selection problem while guaranteeing performance requirements (namely, requirements on the maximum admissible response times of single components or sequences of components) using, in the current release, a random greedy algorithm. The output of this tool determines the |

| | |
|---|---|
| | optimal component placement, resource selection and the optimal number of nodes/VMs which helps the developer to find the optimal placement. |
| **High level Architecture** | SPACE4AI-D tool is a separate component. The current implementation is based on Python and relies on standard Python libraries. Future extensions will consider the Solid library (https://github.com/100/Solid) for the implementation of more advanced heuristics. |
| **Dependencies** | SPACE4AI-D depends on the performance models (a-MLlibrary), and requires performance constraints, resource descriptions and the DAG of AI applications that will be obtained by parsing the application code. The tool generates the optimal solution for the component placement and resource selection. A parser will be developed to transform the output to the TOSCA standard format to be provided as input to the virtual infrastructure provision module (IM, see AI-SPRINT Deliverable *D3.1 First release and evaluation of the runtime environment*). |
| **Interfaces** | The software is a command line tool that accepts some input files including the performance model and constraints, the candidate resource descriptions and the application DAG in JSON format. |
| **Data** | The input data includes the performance model (pickle file obtained from the a-MLlibrary) and constraints, resource descriptions and application DAG. The input JSON file will be obtained by an external parser from the YAML descriptions discussed in Section 4.4.1. |
| **Needed improvement** | The tool needs to be validated on real systems. Furthermore, the code parser and output file translations into TOSCA format need to be developed. Additional heuristics (on-going work to be included in the next release is evaluating the effectiveness of the tabu search algorithm) will be developed and included in the M24 release. |
| **Implemented Improvements for the First Release** | The current version of the tool has been developed from scratch within the AI-SPRINT project. A random greedy algorithm has been implemented to find the optimal solution for the joint component placement and resource selection problem. At each iteration, a solution is generated based on the system description and then the solution feasibility is tested. The candidate feasible solution with minimum cost will is provided as output. |
| **Release Version & Repository** | The first version released is available at https://gitlab.polimi.it/ai-sprint/space4ai-d |

### 4.7.1 Detailed Description of Activities for the First Release

In SPACE4AI-D framework, AI applications are modelled as DAGs, see Figure 4.18, whose nodes represent the different components. These are DNNs implemented as Python functions running in Docker containers that can be deployed in edge devices, cloud Virtual Machines (VMs) or according to the Function as a Service (FaaS) paradigm. For the sake of simplicity, we assume that the DAG includes a single entry point, characterised by the input exogenous workload λ (expressed in terms of requests/sec), and a single exit point. We assume that the inter-arrival time of requests, i.e., $\frac{1}{\lambda}$ is exponentially distributed. We denote the set of components by $I$. The directed edge connecting components *i* and *k* is labelled with $< p^{ik}, \delta^{ik} >$, where $p^{ik}$ is a transition probability, and $\delta^{ik}$ denotes the size of data sent from *i* to *k*.

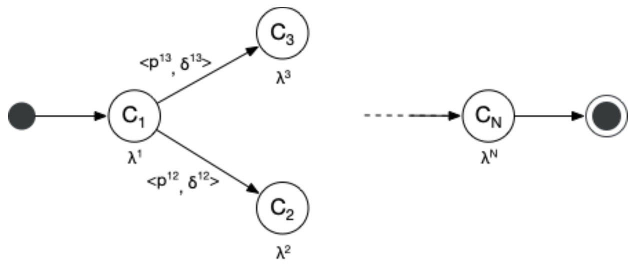
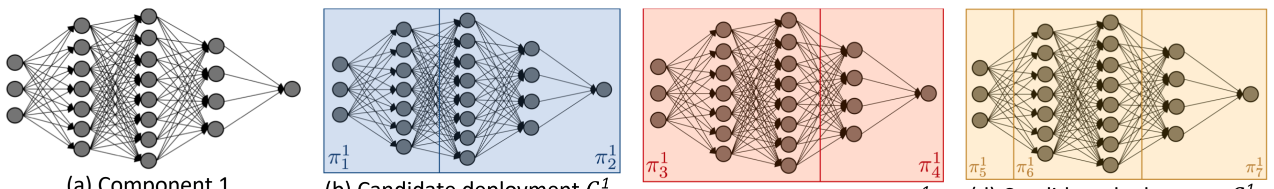


*Figure 4.18 - Directed acyclic graph for components*

Furthermore, components can be characterised by multiple candidate deployments. Each element in a candidate deployment is a partition of the corresponding DNN. We denote by $C^i$ the set of all candidate deployments for component $i \in I$. Each element $c_s^i \in C^i$ is defined as$c_s^i = \{\pi_h^i\}_{h \in H_s^i}$, where $\pi_h^i$ denotes the DNN partition. The set $H_s^i$ is defined as the set of indices $h$ of all the partitions $\pi_h^i$ in the candidate deployment $c_s^i$. An example of an AI application component (denoted by i=1) with its candidate deployments is reported in Figure 4.19. Three alternative deployments are available: $c_1^1$ and $c_2^1$, characterised by two partitions and denoted by $c_1^1 = \{\pi_1^1, \pi_2^1\}$ and $c_2^1 = \{\pi_3^1, \pi_4^1\}$, respectively, and $c_3^1$, characterised by three partitions and denoted by $c_3^1 = \{\pi_5^1, \pi_6^1, \pi_7^1\}$. In this setting, we will define $H_1^1 = \{1,2\}$, $H_2^1 = \{3,4\}$ and $H_3^1 = \{5,6,7\}$.

Note that, in some cases, one of the candidate deployments may correspond to the complete DNN identifying the component. The corresponding set $c_s^i$ would therefore contain a unique partition $\pi_h^i$.



(a) Component 1 (b) Candidate deployment $C_1^1$ (c) Candidate deployment $C_2^1$ (d) Candidate deployment $C_3^1$

*Figure 4.19 - Example of AI application component with its candidate deployments*

Together with $p^{ik}$, which is related to the transition from component $i$ to component $k$, we introduce an additional parameter $p_{h\xi}^i$, which defines the probability that partition $\pi_\xi^i$is executed just after partition $\pi_h^i$. Mechanisms as early stopping [Yao2007] entail that not all component partitions are necessarily executed, which dictates the necessity of defining the probability of actually moving from one to the other. Similarly, we define $\delta_{h\xi}^i$ as the amount of data transferred from partition $\pi_h^i$ to partition $\pi_\xi^i$. Moreover, each $\pi_h^i$ is characterised by a memory requirement (expressed in MB), and by a total load $\lambda_h^i$, which depends on $\lambda$ and on the transition probabilities related to all predecessors of the partition.

For simplicity, we consider DAGs including only sequential execution and branches, since, as in [Ardagna2007], we assume that loops are unfolded (or peeled) while parallel execution is not supported for the time being. We define execution paths as sequences of application components from the entry point to the exit point of the DAG, while a path $P$ denotes a set of consecutive components included in an execution path.

The main performance metric we consider in our system is the response time. QoS requirements may be imposed on both the response time of single components (*local constraints*), and on the response time of all components included in a path (*global constraints*), see Section 4.4.1. QoS requirements may be imposed on both the response time of single components *local constraints*, and on the response time of all components included in a path *global constraints*.

### Resources general model

Computing continuum resources include edge devices, cloud Virtual Machines (VMs) and Function as a Service (FaaS) configurations (OSCAR is considered as the target environment for supporting functions, see AI-SPRINT Deliverable *D3.1 First release and evaluation of the runtime environment*). Each resource is characterised by a maximum memory capacity and is included in a different computational layer. In our model, the first layer includes local devices generating data (such as drones, in the AI-SPRINT Maintenance and Inspection use case). The second layer is often located in the edge and may include smartphones, PC or edge servers with a higher computational power. Cloud layers include VMs coming from a single cloud provider catalogue (however, our approach can be easily extended to consider multiple cloud providers). The VMs selected at a given layer are homogeneous and evenly share the workload due to the execution of one or multiple application components. We consider VMs characterised by a single GPU, if available: costs and inference performance scale linearly with the number of GPUs [Sivaraman2018], therefore such assumption allows to improve the availability of the whole system. Finally, we consider all FaaS configurations to be in the same layer because functions run on independent containers. The same component can be associated with multiple FaaS configurations characterised by different memory settings. The response time of all the executed components is computed as follows:

- We characterise the demanding time to run a component on edge or cloud resources without resource contention (i.e., when a node executes a single request, see [Lazowska1984]).
- We model edge devices and VM instances as individual M/G/1 (single server multiple class) queues [Tadakamalla2021] to cope with resource contention.
- We compute the average execution time for each component on a given FaaS configuration starting from the execution times of hot and cold requests, the expiration threshold and the arrival rate of the configuration by relying on the tool proposed in [Mahmoudi2020].
- We consider several network domains connecting edge devices with each other and with the remote cloud back-end. Resource layers are included in, possibly, multiple network domains, associated with a given technology characterised by access time and bandwidth.
- We include in the global execution time of each path the network delay due to data transmissions, depending (see, e.g., [Mahmoudi2020]) on the amount of data transferred, the network bandwidth and the access delay of the network domain. We neglect the network delay in the cloud since all VMs and FaaS instances are executed in the same data center.
- Specific components (partition) may also be characterised by an ad-hoc performance model identified by the methodology described in Section 4.5.1.

According to the results reported in [Tadakamalla2021] and [Mahmoudi2020] and the preliminary results reported in Section 4.8.2, response times of components deployed at each layer can be estimated with a percentage error between 10% and 30%, which is acceptable for design-time purposes [Lazowska1984].

Finally, a compatibility matrix $A$ is introduced to show which devices can be used to execute each partition. Specifically, $a^i_{hj}$ is 1 if $\pi^i_h$ can be executed on device $j$, 0 otherwise.

### System costs

Resources in the computing continuum are characterised by different costs:

- Costs of edge devices are estimated, for the single run of the target application, amortising the investment cost along the lifetime horizon of the device and dividing the yearly management costs by the number of times the application is run over a year.
- Cloud VMs costs are hourly costs [AzureVMpricing], while FaaS costs are expressed in GB-second [AWS], and they depend on the memory size, the functions duration, and the total number of invocations.

- An additional *transition cost* can be required by FaaS providers (e.g., [AWSStepFunctionsPricing]) to account for the message passing and coordination. Some third-party frameworks, e.g., AI-SPRINT OSCAR, avoid transition costs by supporting the orchestration through an architectural component.

A summary of the problem formulation and the proposed random greedy algorithm are reported in Appendix B.

## 4.8 Performance evaluation

In this section, we provide a description of the activities to evaluate the results of the tests on the software components described in the previous sections.

### *4.8.1 Parallel CSVM classification*

In order to evaluate the performance of the prototype example application, as described in Section 2.1 - Parallel CSVM classification, we executed a set of runs in the MareNostrum IV supercomputer.

The code performs a training of the model using the dislib implementation of the CSVM algorithm and then calculates the score returning the mean accuracy on a given test data and labels. The input dataset is loaded from the PhysioNet repository files into ds-array objects as training set and labels. The data is split by dislib in blocks of 500x500 thus generating 631 tasks managed by PyCOMPSs (Figure 4.20). Figure 4.21 depicts the Python script used for the tests, while Figure 4.22 contains a part of the internal implementation of the *train()* and *score()* functions in dislib, using PyCOMPSs annotations.

The code to extract the time-frequency features from the PhysioNetc CinC Challenge 2017 dataset and train the dislib CSVM binary classifier is available at https://doi.org/10.5281/zenodo.5734309.

```
# Parameters
dataDir = 'training2017/'
FS = 300
WINDOW_SIZE = 60*FS

parser = argparse.ArgumentParser()
args = parser.parse_args()

## Loading time serie signals
files = sorted(glob.glob(dataDir+"*.mat"))
trainset = np.zeros((len(files),WINDOW_SIZE))
count = 0
for f in files:
    record = f[:-4]
    record = record[-6:]
    # Loading
    mat_data = scipy.io.loadmat(f[:-4] + ".mat")
    #print('Loading record {}'.format(record))
    data = mat_data['val'].squeeze()
    # Preprocessing
    #print('Preprocessing record {}'.format(record))
    data = np.nan_to_num(data) # removing NaNs and Infs
    data = data - np.mean(data)
    data = data/np.std(data)
    trainset[count,:min(WINDOW_SIZE,len(data))] = data[:min(WINDOW_SIZE,len(data))].T # padding sequence
    count += 1


## Loading labels
import csv
csvfile = list(csv.reader(open(dataDir+'REFERENCE.csv')))
traintarget = np.zeros((trainset.shape[0],1))
classes = ['A','N','O','~']
for row in range(len(csvfile)):
    #traintarget[row, 0] = classes.index(csvfile[row][1])
    traintarget[row, 0] = 0 if classes.index(csvfile[row][1]) == 0 else 1
```

```
trainset = np.concatenate((trainset, trainset), axis=0)
traintarget = np.concatenate((traintarget, traintarget), axis=0)

load_time = time.time()
print("Load time", load_time - start_time)

print("trainset.shape", trainset.shape)
print("traintarget.shape", traintarget.shape)

x = ds.array(trainset, block_size=(500, 500))
y = ds.array(traintarget, block_size=(500, 1))

csvm = CascadeSVM(kernel='rbf', c=1, gamma='auto', tol=1e-2, random_state=seed)
csvm.fit(x, y)

compss_barrier()
fit_time = time.time()
print("Fit time", fit_time - load_time)
out = [csvm.iterations, csvm.converged, load_time, fit_time]

out.append(compss_wait_on(csvm.score(x, y)))
print("Test time", time.time() - fit_time)
print("Score: ", out)
```

*Figure 4.20 - Script for the execution of CSVM with PyCOMPSs and dislib*

```
@constraint(computing_units="${ComputingUnits}")
@task(x_list={Type: COLLECTION_IN, Depth: 2},
      y_list={Type: COLLECTION_IN, Depth: 2},
      id_list={Type: COLLECTION_IN, Depth: 2},
      returns=4)
def _train(x_list, y_list, id_list, random_state, **params):
    x, y, ids = _merge(x_list, y_list, id_list)

    clf = SVC(random_state=random_state, **params)
    clf.fit(X=x, y=y.ravel())

    sup = x[clf.support_]
    start, end = 0, 0
    sv = []

    for xi in x_list[0]:
        end += xi.shape[1]
        sv.append(sup[:, start:end])
        start = end

    sv_labels = y[clf.support_]
    sv_ids = ids[clf.support_]

    return sv, sv_labels, sv_ids, clf
```

```
@constraint(computing_units="${ComputingUnits}")
@task(x_list={Type: COLLECTION_IN, Depth: 2},
      y_list={Type: COLLECTION_IN, Depth: 2}, returns=tuple)
def _score(x_list, y_list, clf):
    x = Array._merge_blocks(x_list)
    y = Array._merge_blocks(y_list)

    y_pred = clf.predict(x)
    equal = np.equal(y_pred, y.ravel())

    return np.sum(equal), x.shape[0]
```

*Figure 4.21 - Internal implementation of the training and score tasks in dislib*

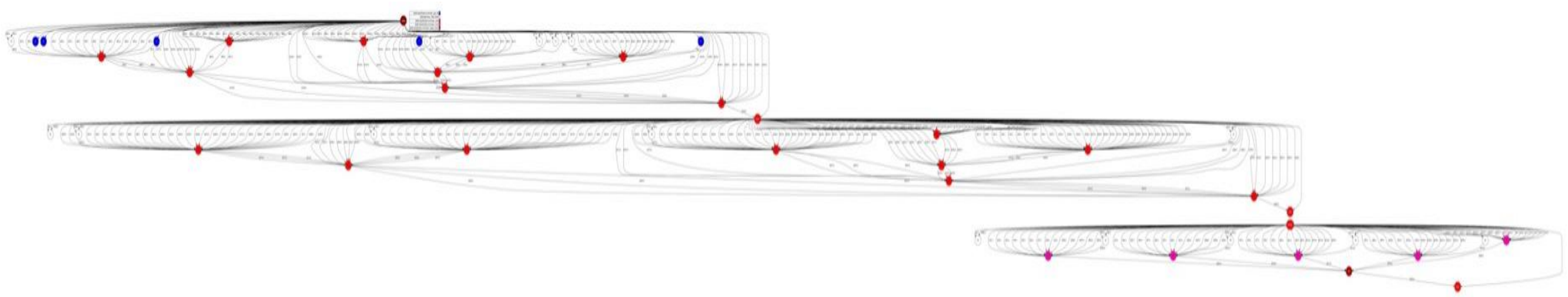


*Figure 4.22 - Execution graph of the CSVM algorithm*

To conduct the experiment in Marenostrum, a 3,456-node (48 servers of 72 nodes) supercomputer where each node has two 24-core Intel Xeon Platinum 8160 and 98 GB of main memory, was used. Each node hosts the execution of 6 tasks, each using 8 cores. The logs of these executions are available at: https://doi.org/10.5281/zenodo.5734190. Figure 4.23 represents the results of these tests highlighting that, for this specific configuration, we can achieve performance improvements thanks to the PyCOMPSs parallelisation, up to 192 cores.

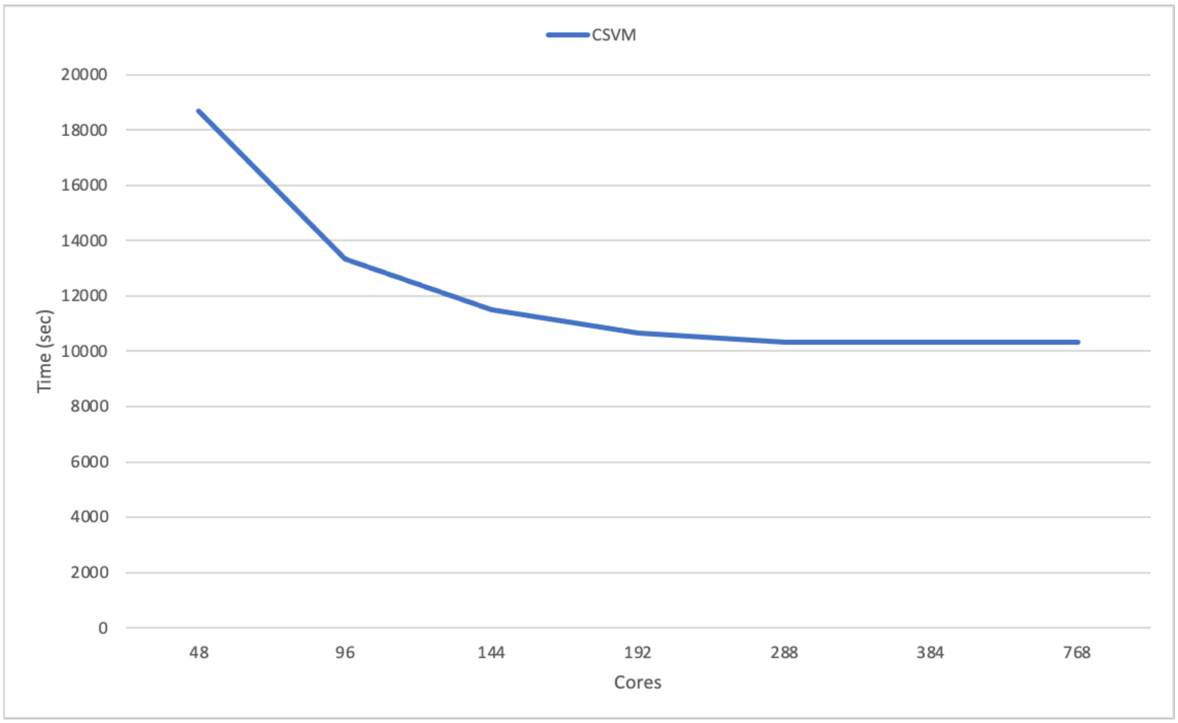


*Figure 4.23 - Results of the execution of CSVM algorithm in the Marenostrum cluster*

## 4.8.2 Performance Models

In this section, the results obtained in building end-to-end and per-layer performance models for DNN based AI applications and for PyCOMPSs will be presented. In all experiments, the accuracy of models is evaluated considering their mean absolute percentage error (MAPE) on the test set:

$$MAPE = \frac{100\%}{S} \sum_{k=1}^{S} \left| \frac{y_k - \hat{y}_k}{y_k} \right| \quad (1)$$

Where $S$ is the number of samples in the test set, $y_k$ is the execution time measured on the operational system and $\hat{y}_k$ is the predicted execution time from the learned model. It is worth noting that, in order to have models suitable to be exploited in what-if performance analyses and capacity planning, their prediction error should be small. As from the common practice [Lin2013], we will consider a model accurate if its MAPE is lower than 30%.

### End-to-end performance model

In this section, first we describe the experimental setup and then present the performance evaluation results of the end-to-end models with a particular focus on job training.

The features we considered can be classified in two different classes:

1) Features parameters describing the deep learning job training process independently from the hardware on which it will be run such as the number of performed iterations ($I$) and the number of training examples processed during an iteration ($B$) i.e., the batch size.

2) Features describing the characteristics of the underlying hardware such as:

- $P$: The computational power of the used GPUs, measured in single precision GFlops/second.
- $G$: The number of GPUs used.
- $T$: The number of CPU threads used to load in parallel the training examples of a mini-batch to GPU memory.
- $D$: The disk delay, measured as the time required to load 120 kB files of 192 kB from disk into the memory of a GPU.

To increase the accuracy of the performance regressor at the profiling stage, we use the two following approaches to augmenting the feature set:

- *The reciprocals of the features:* There are some metrics that more appropriately should appear in their reciprocals: for example, $G$ and $T$ are likely to produce terms that depend on their inverse [Gianniti2018b]. In this deliverable, the feature set is augmented with all the reciprocals of the original features, so as to cater both for those parameters that intuitively contribute in this way and for possible second-order effects, which may be harder to anticipate.
- *Using interaction terms:* In line with this method, the feature set is also extended with crossover terms. Each term can contain any combination of original features and reciprocals under the constraint that every monomial has at most degree 1 in every variable. This constraint comes from the previously mentioned choice of not using any higher degree powers. Additionally, combined terms cannot contain both a feature and its reciprocal, so as to avoid adding redundant information and/or introducing numerical issues.

To reduce the model size of the performance regressor and limit the overfitting, feature selection is also performed by applying the Sequential Forward feature Selection (SFS) [Ferri1994] and Draper and Smith [Draper1966] method, which jointly performs forward and backward feature selection.

SFS [Ferri1994] is a greedy search algorithm which aims at automatically selecting, among the *d* available features, the *k* (*k* < *d*) that are more relevant for the tackled problem. Applying feature selection allows first of all to improve computational efficiency, and, moreover, to reduce the generalisation error by removing irrelevant features or noise. The SFS algorithm takes as input the whole feature space (whose dimension is *d*) and the dimension of the target space *k*. Starting from an empty feature set, it proceeds iteratively by adding as a new feature the one that optimises a given criterion function, until the number of features *k* is reached.

Draper and Smith [Draper1966] outline an approach that tries to merge the basic forward selection and backward elimination schemes, thus achieving the best of both worlds with a linear model. On one side, backward elimination starts with the full set of features and iteratively drops the least significant term from the regression equation, eventually stopping when all the remaining coefficients are significant at a given confidence level. In a dual fashion, forward selection starts with an empty model and iteratively adds the most promising features one at a time, until the latest addition results to be not significant enough. In both cases, the decision is made based on the p-value of single coefficient t-tests, where at each iteration the least significant (respectively, last added) feature is compared to a predefined threshold. Since both methods are somewhat impaired by their greedy approach, Draper and Smith suggest starting with an empty model and proceed in combined steps that try both to add the most promising feature and to remove the least significant, until neither succeeds at the preliminarily set up confidence levels.

For what concern extrapolation analyses, the most obvious scenario we investigated is the extrapolation on the number of iterations, $I$, which simply corresponds to training an NN for more epochs. Such information helps in scheduling jobs on a shared infrastructure or can be used, conversely, to tune the number of epochs so as to fit a given time window. Similarly, extrapolating on the number of inner modules of a DNN, $N$, allows for fine- tuning the network depth of the architectures that can be parametrized in this sense, such as ResNet and VGG. Another set of features can have an even higher economic impact, as their interpretation is linked with hardware changes in the deployment of interest. For instance, since it is common practice to use the largest batch size that fits on the GPUs at hand, in order to reach the optimal parallelism, extrapolating on $B$ corresponds to using different GPUs with larger memory. Along the same lines, extrapolation on computational power $P$ means switching to GPUs with a higher nominal speed in GFlops/second. In the end, extrapolating on the number of data loading threads $T$ or of GPUs $G$ relates to the installation of more, respectively, CPUs or graphics cards.

Here, we describe in detail the experimental setup for collecting data and the set of conducted deep net profiling experiments. Experiment data and our scripts source code for models training and testing are available at https://doi.org/10.5281/zenodo.5327342, and at https://doi.org/10.5281/zenodo.5735476. To enforce the generality of the proposed approach, different open-source frameworks and multiple NNs have been considered. The adopted frameworks are PyTorch 0.3.1 [Paszke2017, PyTorch] and TensorFlow 1.8.0 [Abadi2016, TensorFlow], while the trained CNNs are AlexNet [Krizhevsky2012], ResNet-50 [Krizhevsky2012], and VGG-19 [He2015], whose implementations are already available within the considered frameworks. The experiments have been run on four different types of machines, whose characteristics are summarised in Table 4.2. VGG-19 cannot be trained on the in-house server 1 since its parameters do not fit in GPU memory. To test the generalisation capabilities of the performance models when the number of inner modules is varied, ResNet has been considered as a reference deep network and the number of inner modules has been varied between 1 and 10.

| Machine Type | (v)CPUs | Mem. [GB] | GPU type | Computational Power [GFlops/s] | N. GPUs |
|---|---|---|---|---|---|
| Azure Standard_NC6 | 6 | 56 | K80 | 5591 | 1 |
| Azure Standard_NC12 | 12 | 112 | K80 | 5591 | 2 |
| Azure Standard_NC24 | 24 | 224 | K80 | 5591 | 4 |
| Azure Standard_NV6 | 6 | 56 | M60 | 7365 | 1 |
| Azure Standard_NV12 | 12 | 112 | M60 | 7365 | 2 |
| Azure Standard_NV24 | 24 | 224 | M60 | 7365 | 4 |
| In-house server 1 | 20 | 48 | Quadro P600 | 1195 | 2 |
| In-house server 2 | 40 | 256 | GTX 1080Ti | 11339 | 8 |

*Table 4.2 - Characteristics of the target machines*

The addressed scenario for CNNs is the classification of images belonging to the ImageNet [Deng2009] database. The images are cropped to 32x32 pixels when used as input of the ResNet with a variable number of inner modules, and to 224x224 in all the other scenarios. To speed up the experimental campaign, the set of analysed images is a subset composed of around 120,000 items evenly partitioned into 100 classes.

As in other research studies, each experiment has been run immediately after preliminary one-epoch-long experiments and the execution time of the first 20 iterations has been excluded from each experiment's total time, because they can take significantly longer to complete than the subsequent iterations (see, e.g., [Hadjis2016]). It is worth noting that their removal does not affect the significance of the trained performance models, since in real scenarios the first 20 iterations are negligible with respect to the full training, which runs for several thousands of iterations. To remove other possible "warm-up" effects, each experiment performed on cloud VMs has been repeated at least three times and the data about its first run discarded. In this type of scenario, indeed, the whole first epoch can have a significant time delay caused by the retrieval of the training examples used during training, which may not be immediately available on the VM's local disk. Even in this case, the impossibility of correctly estimating the first epoch of a training procedure is not a significant limitation, since this is only a limited fraction of the overall deep net training process. Moreover, to reduce the overall execution time of the experimental campaign, the upper bound of the number of epochs has been set to three. To verify that the data collected on the first three epochs can be effectively used to build general models able to predict the execution time of real training processes with thousands of epochs, some ad-hoc long-running experiments have been performed. In particular, for each framework, for each network, and for each type of GPU but the GTX 1080Ti, a long experiment (at least 24 hours) has been run. Long runs on GTX 1080Ti could not be performed because of the limited availability of in-house server 2.

To increase the number of available samples, for each run on the target system, timing data is also collected after the 25%, 50%, and 75% of iterations, respectively. In this way, four data points are extracted from each run. Finally, to mitigate the effects of system perturbation (for example, running of operating system services) on the collected data and to improve the accuracy of the generated models, all the experiments whose execution time is shorter than 10 seconds have been removed from the data sets.

Different ranges of batch size have been considered for different networks and for different machines, since GPU devices with larger memory support training with larger batch sizes. A strong scale approach has been adopted: the batch size identifies the overall number of training examples processed, possibly across multiple GPUs, during an iteration [PyTorch]. The largest batch size is 8,192 for AlexNet, 512 for ResNet-50, and VGG-19, when running on 8 GTX 1080Ti. Several values of numbers of CPU threads have been analysed for the AlexNet experiments, while only up to 8 CPU threads have been used in the training of ResNet-50, and VGG-19 since incrementing the thread numbers did not significantly influence the forward or backward times of these NNs. The overall number of deep net training runs is about 10400 with an overall execution time of more than 5500 hours. For the sake of completeness, the size of the training set for each investigated scenario is reported in Appendix A. Note that the reported numbers refer to available samples. The number of corresponding runs is roughly one fourth, i.e., for each experiment four samples are generated as previously described. Some of the generated samples, however, have been discarded because they are shorter than 10 seconds.

In order to verify that the execution time of individual epochs is stable so that the models built by using data on a few epochs can be effectively used to predict performance of long-running deep net training, we performed a set of experiments. Table 4.3 presents the difference between the average execution time of an iteration in the whole long-running experiment and the average execution time of one iteration computed either on the first epoch or on the first three epochs. Because of limited time slots accessibility, results on the in-house server 2 could not be collected for multiple day executions. On the in-house server 1 the difference between the time of the initial epochs and the average execution time of epochs for long experiments is very small (less than 1%). Results obtained on the Azure VMs are characterised by a larger difference, but in all the scenarios the difference is below 10%. Moreover, when the number of initial epochs is 3, the difference is reduced to less than 8%, so short experiments data can be effectively used to train models to predict real scenario deep net training execution times. It is worth noting that possible "warm-up" issues have been removed by the preliminary one-epoch-long experiments.

| **Network** | **Framework** | **N. Initial Epochs** | **GPU Type** | | |
|---|---|---|---|---|---|
| | | | **P600** | **K80** | **M60** |
| AlexNet | PyTorch | 1 | 0.73 | 7.42 | 9.25 |
| | | 3 | 0.18 | 1.52 | 1.38 |
| | TensorFlow | 1 | 0.18 | 7.14 | 4.76 |
| | | 3 | 0.12 | 7.40 | 4.80 |

| | | | | | |
|---|---|---|---|---|---|
| ResNet-50 | PyTorch | 1 | 0.08 | 1.02 | 0.24 |
| | | 3 | 0.49 | 0.20 | 0.01 |
| | TensorFlow | 1 | 0.33 | 0.27 | 0.62 |
| | | 3 | 0.31 | 0.09 | 0.82 |
| VGG-19 | PyTorch | 1 | - | 2.71 | 0.78 |
| | | 3 | - | 1.50 | 0.22 |
| | TensorFlow | 1 | - | 4.00 | 3.62 |
| | | 3 | - | 3.91 | 3.72 |

*Table 4.3 - Relative percentage difference between average iteration time of initial epochs and average iteration time over the whole long running experiment*

By analysing individual deep nets, it can be noticed that the difference for ResNet-50 and VGG-19 is not larger than 4%, while, on the contrary, AlexNet differences are greater, up to 9.25%. These results show how AlexNet, despite its smaller complexity (i.e., smaller number of parameters), has a more irregular performance behaviour with respect to other networks. This is caused by the impact of data loading on AlexNet, which contributes significantly to the overall training time. Since data loading is more subject to system perturbation than computation, the larger is its delay contribution, the larger the variance in the epochs execution time.

The performance of the mentioned Deep Learning applications have been evaluated by considering the XGBoost Machine Learning model [Chen2016], coupled with Sequential Forward Selection (SFS). Hyperparameter selection has been performed by relying on HyperOpt [HyperOptGitHub], an open-source Python library based on Bayesian optimization algorithms over awkward search spaces, which may include real-valued, discrete, and conditional dimension. The space of the hyperparameters is defined in Table 4.4. The maximum number of evaluations performed by HyperOpt has been set to 10.

| Parameter | Value | Type |
|---|---|---|
| min_child_weight | 1 | Fixed |
| gamma | loguniform(0.1, 10) | Prior probability |
| n_estimators | 1000 | Fixed |
| learning_rate | loguniform(0.01,1) | Prior probability |
| max_depth | 100 | Fixed |

*Table 4.4 - Parameters required by the XGBoost method*

Whenever the performance obtained by XGBoost was not accurate enough, we considered the Draper and Smith method, setting the probabilities to add or remove a feature to 0.05 and 0.1, respectively, and the maximum number of iterations to 100.

Feature augmentation has been performed in both cases by including the reciprocal of the available features and the crossover terms, in order to capture second-order effects which may be harder to anticipate.

### Experimental result

The results of the experiments described in the previous section are reported in the following.

Hold-out models were trained by considering all the samples collected by the previously described experiments, with a given network on different numbers and types of GPUs. This set of samples has been randomly splitted into a training set, including the 80% of the collected data, and a test set including the remaining 20%. A summary of the results obtained when exploiting different ML models is reported in Table 4.5. In particular, both XGBoost and Draper and Smith methods are tested in two different scenarios: in the first one, we let the model include all the relevant features. In the second one, we limit (through SFS, in the case of XGBoost) the number of features to 5.

| Network | Framework | ML Model (maximum number of used features) | | | |
|---|---|---|---|---|---|
| | | XGBoost (all) | XGBoost + SFS (5) | Draper & Smith (all) | Draper & Smith (5) |
| AlexNet | PyTorch | 3.90 | 7.58 | 10.11 | 17.15 |
| | TensorFlow | 3.69 | 6.50 | 11.92 | 20.35 |
| ResNet-50 | PyTorch | 4.99 | 6.30 | 10.13 | 16.56 |
| | TensorFlow | 1.43 | 28.11 | 9.99 | 16.94 |
| VGG-19 | PyTorch | 2.28 | 4.47 | 5.21 | 14.02 |
| | TensorFlow | 2.25 | 2.98 | 3.10 | 8.14 |

*Table 4.5 - MAPE [%] on test set for different models*

We can notice that the results obtained exploiting PyTorch and TensorFlow are quite similar: estimating the training time for the AlexNet network is more difficult than the other, due to the fact that, since it is a very simple network, the data loading process is more significant in determining the overall training time. The regularity of the ResNet-50 and VGG-19 networks usually results in very accurate performance estimation models. The relation between the prediction error and the number of features in the model is characterised in a sample scenario, adopting the Draper and Smith method, in Figure 4.24. It is worth noting that there are very significant improvements in the accuracy of the model up to the sixth added feature. After that, adding more features improves the model accuracy only slightly. Nevertheless, considering all the selected features instead of a limited number still results in a significant accuracy improvement.

As an example, Table 4.6 reports the list of features selected by SFS to build the best XGBoost model for the different networks and frameworks under study, together with the corresponding weights.

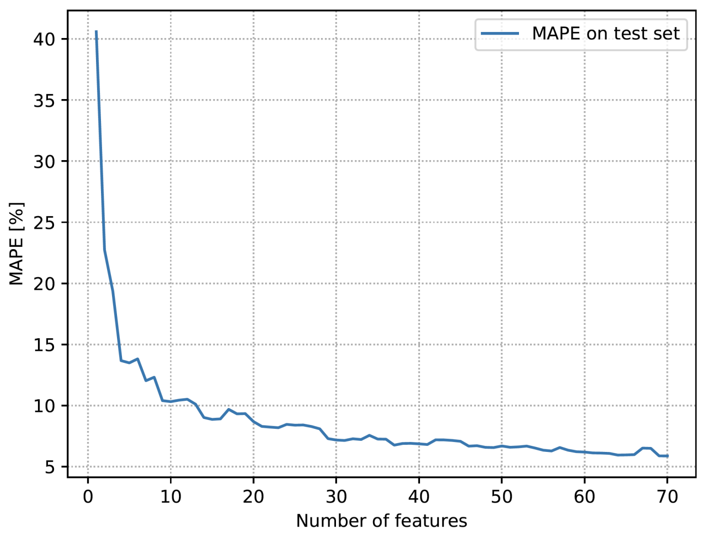


*Figure 4.24 - MAPE when varying the number of features*

| Network | Framework | Features (corresponding weights) |
|---|---|---|
| AlexNet | PyTorch | TP (0.434485), IB (0.399988), PI (0.165527) |
| | TensorFlow | ID (0.506602), TD (0.493398) |
| ResNet-50 | PyTorch | I/D (1.0) |
| | TensorFlow | PI (0.508897), TP (0.491103) |
| VGG-19 | PyTorch | ID (0.649914), PG (0.350086) |
| | TensorFlow | PI (1.0) |

*Table 4.6 - Relevant features and corresponding weights for the XGBoost model using SFS*

Additional tests were performed to measure the extrapolation capability of the produced models. These are particularly crucial when building a performance prediction model, since its main goal is to predict an application execution time in unforeseen scenarios, for which it is not feasible to collect profiling data. These may include both the exploitation of new hardware (e.g., a new VM type in the cloud) and the training of a new NN version. The results that will be presented in the following were collected by considering performance models built with the aim of predicting the execution time of experiments characterised by a feature whose value is larger than all the values of the same feature in the training data. Note that, in all scenarios except the extrapolation on the neural network depth (*N*), the models were built using the Draper and Smith method. A comparison with XGBoost has been added when considering the extrapolation on *N*.

- *Extrapolation on the batch size (B):* we considered performance models built through the Draper and Smith method, where the overall largest batch size value is twice larger than the largest one included in the training test. Considering such a value of B corresponds to predicting the NN training time on GPUs whose available memory is twice larger than the already characterised devices. The extrapolation accuracy is reported in Table 4.7 and Table 4.8.

- *Extrapolation on the GPU number (G):* the available targets, reported in Table 4.9, consider a maximum number of GPUs equal to 4 for K80 and M60, and to 8 for GTX 1080Ti. Therefore, in the first two cases the model training has been performed considering experiments with 1, 2 and 3 GPUs, and predicting the value when the number of GPUs is equal to 4. In the third case, in turn, the training set included experiments with 1, 2 and 4 GPUs, while the 8-GPUs experiments were used as the test set. Note that the P600 GPUs were not considered in this scenario, since the only available data included 1 or 2 GPUs, which is not enough to perform predictions. The accuracy is reported in Table 4.9. It can be noticed that the error margin is reasonable in all scenarios, even if it is larger when exploiting the TensorFlow framework, showing how the effects of the implemented interactions between GPUs are more difficult to model compared to PyTorch.
- *Extrapolation on the computational power (P):* in this scenario available data is used to estimate the NN performance on more powerful unseen GPUs. In particular, we built the training set with the data collected from experiments run on P600, K80, and M60 GPUs, while we evaluated the results on the data collected on the GTX 1080Ti. Notice that different architectures are characterised not only by a different computational power (P), but also by a different disk speed (D). In these experiments, we retrieved its value by performing a short profiling run, measuring the load time. The results, reported in Table 4.10, show how the information collected on less powerful GPUs can be effectively used to predict NN training time on better performing hardware without requiring expensive experimental campaigns.
- *Extrapolation on the neural network depth (N):* in this set of experiments, we varied the number of inner modules *N* in a ResNet network between 1 and 10. Specifically, we built the training set by including experiments performed with *N* varying between 1 and *n*, while the test set is made by experiments with N between *n*+1 and 10. We reported in Table 4.11 the results obtained by performance models built by considering n equal to 4, 5, 6, and 8. As expected, the larger the maximum number of inner modules in the training set, the better the extrapolation capability of the model, and so its accuracy. Nevertheless, even by limiting the training set to the experiments with up to 5 inner modules (hence half of the maximum value, like in the other extrapolation scenarios), the MAPE of the models is small enough to guarantee that the user can easily evaluate the effect on performance of incrementing the number of inner modules in the ResNet. Note that the results obtained with the Draper and Smith model at 1 GPU were computed by considering in the feature augmentation process crossover terms up to degree 3 (instead of 2 as considered in all the other scenarios).

| Network | Framework | GPU type and number | | | | | |
|---|---|---|---|---|---|---|---|
| | | P600 | | K80 | | | |
| | | 1 | 2 | 1 | 2 | 3 | 4 |
| AlexNet | PyTorch | 11.12 | 5.33 | 1.74 | 3.33 | 1.81 | 0.66 |
| | TensorFlow | 9.83 | 10.04 | 2.30 | 2.61 | 4.28 | 2.82 |
| ResNet-50 | PyTorch | 10.64 | 11.97 | 0.76 | 7.83 | 3.09 | 4.53 |
| | TensorFlow | 10.64 | 11.97 | 10.25 | 1.27 | 1.84 | 6.83 |
| VGG-19 | PyTorch | - | - | 13.88 | 21.71 | 27.63 | 9.65 |
| | TensorFlow | - | - | 18.20 | 0.92 | 1.16 | 10.58 |

*Table 4.7 - MAPE [%] on test set for batch size (B) extrapolation (P600 and K80 GPUs)*

| Network | Framework | GPU type and number | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | M60 | | | | GTX 1080Ti | | | |
| | | 1 | 2 | 3 | 4 | 1 | 2 | 4 | 8 |
| AlexNet | PyTorch | 7.53 | 14.00 | 6.58 | 16.73 | 0.43 | 1.62 | 1.15 | 4.16 |
| | TensorFlow | 7.19 | 6.36 | 6.91 | 6.96 | 4.06 | 5.36 | 1.14 | 1.12 |
| ResNet-50 | PyTorch | 3.60 | 20.04 | 9.58 | 4.64 | 12.62 | 11.93 | 20.63 | 4.29 |
| | TensorFlow | 2.08 | 2.79 | 3.07 | 21.49 | 0.68 | 6.44 | 1.43 | 12.06 |
| VGG-19 | PyTorch | 10.85 | 18.18 | 13.81 | 8.01 | 24.98 | 17.40 | 2.93 | 14.06 |
| | TensorFlow | 7.34 | 5.06 | 2.74 | 6.92 | 22.88 | 6.37 | 24.12 | 23.56 |

*Table 4.8 - MAPE [%] on test set for batch size (B) extrapolation (M60 and GTX 1080Ti GPUs)*

| Network | Framework | GPU type | | |
|---|---|---|---|---|
| | | K80 | M60 | GTX 1080Ti |
| AlexNet | PyTorch | 7.21 | 14.45 | 4.98 |
| | TensorFlow | 24.75 | 17.27 | 8.77 |
| ResNet-50 | PyTorch | 9.13 | 9.04 | 11.76 |
| | TensorFlow | 24.58 | 18.29 | 6.54 |
| VGG-19 | PyTorch | 11.78 | 15.98 | 24.13 |
| | TensorFlow | 8.84 | 13.52 | 13.65 |

*Table 4.9 - MAPE [%] on test set for GPU number (G) extrapolation*

| Network | Framework | Used features | |
|---|---|---|---|
| | | All | 5 |
| AlexNet | PyTorch | 7.27 | 27.10 |
| | TensorFlow | 5.08 | 5.08 |
| ResNet-50 | PyTorch | 18.09 | 20.74 |
| | TensorFlow | 20.23 | 19.82 |

*Table 4.10 - MAPE [%] on test set for computational power (P) extrapolation*

| Model | Max number of inner modules | GPU type and number M60 | | |
|---|---|---|---|---|
| | | 1 | 2 | 4 |
| Draper & Smith | 4 | 21.02 | 27.95 | 17.40 |
| | 5 | 19.52 | 25.11 | 16.75 |
| | 6 | 31.56 | 20.40 | 16.63 |
| | 8 | 22.26 | 7.93 | 15.99 |
| XGBoost | 4 | - | 0.67 | 0.59 |
| | 5 | - | 0.60 | 0.55 |
| | 6 | - | 0.53 | 0.51 |
| | 8 | - | 0.44 | 0.27 |

*Table 4.11 -MAPE [%] on test set for network depth (N) extrapolation*

According to the results we obtained, since the MAPE is always lower than 30% we have already achieved the KPI2.2 (Accuracy of performance estimation models for individual components, evaluated as mean of the average absolute percentage error <=30) foreseen at M24.

### Per-layer performance model

We used our proposed per-layer model (see Section 4.5.1), to predict the inference time of DNN components partitioned (see Figure 4.25) on a computing continuum. As it is shown in Figure 4.26, the Mobile Edge acquires an image and starts computation of the first layers of the Convolutional Neural Network. The Cloud Server receives through an internet connection the pre-computed results and completes the computation returning the image classification.

Due to the depth and large number of layers in VGG16 and VGG19, they are representatives of complex DNNs and an edge device is usually not able to process them locally, hence, they need to be partitioned along the computing continua.

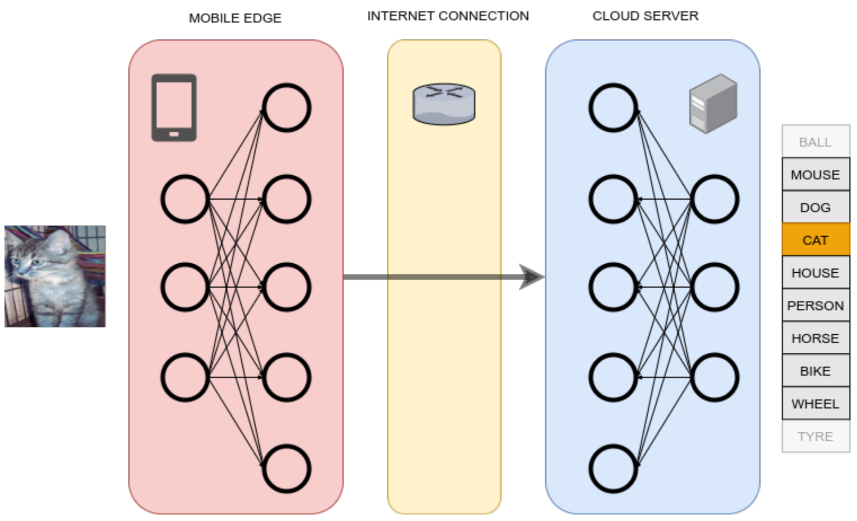


*Figure 4.25 - DNN component partitioning*

The problem we faced is the following: given the computational time required by the first $j$ convolutional layers in a Deep model predict two different quantities:

- The time required by the next layer $j + 1$ (we will call it Next-Layer Analysis).
- The time required by each of the next convolutional layers up to the end of the CNN (we will call it All-Layers Analysis).

These types of predictions aim at profiling the CNN and gaining insights on the accuracy that we can obtain on forecasting the computational times. We focused on CNN layers only since, in the preliminary analyses we performed, they are responsible for about 90% of the inference execution time in most of the relevant architectures proposed for image recognition [Gianniti2018b].

After next and all layers analyses, we trained our models considering an entire VGG16 network as the training set and we considered VGG19 as the test set, investigating the generalisation capabilities of our models (we will call it VGG16 to VGG19 Analysis).

Notice that we will present the analyses in increasing order of difficulty, starting with the single-layer prediction, passing through the all-layer one (in which the errors of different layers sum up) and finally verifying whether we have enough data to predict a similar, but more complex, CNN, that is, VGG19.

In order to perform such analyses, we considered various machine learning algorithms (namely, XGBoost, Ridge Regression and Draper and Smith), and performed Sequential Forward feature Selection (SFS) and feature augmentation (i.e., introducing additional features by combining the initial ones with products and inverses).

### Experimental setup

The data were collected in three different scenarios, listed, in the following, from the one with the lowest computational power in the edge device to the most powerful one and reported in Figure 4.26.

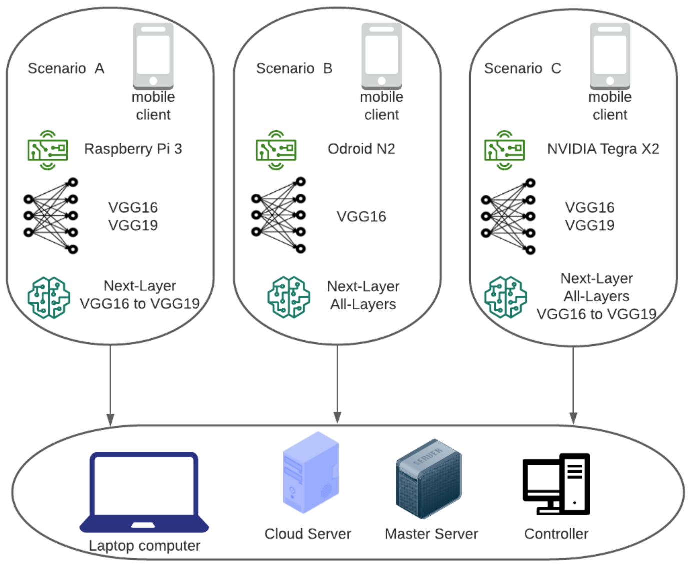


*Figure 4.26 - Experimental scenarios on devices with different computational powers.*

To simulate the cloud, we used in all scenarios a low-end laptop PC. Although this is not representative of real-life settings, it simulates quite well the behaviour of a busy high-end cloud server that is experiencing a peak of load. Its main features are summarised in Table 4.12, but notice that, since we are only focused on the mobile-side performance prediction, this will not affect the data on which we are going to perform analyses.

| Feature | Details |
|---|---|
| Processor | Intel i5-7300U@2.60GHz |
| Memory | 16GB LPDDR4@2133Hz |
| Storage | 240 GB SSD SATA III |
| OS | Ubuntu 18.04 LTS |
| Tensorflow Version | Tensorflow 2.0 |

*Table 4.12 - Main technical features of the laptop pc that simulated the cloud*

Each scenario was evaluated in each possible split between Edge and Cloud (i.e., 22 for VGG16 and 25 for VGG19) using as input, for each test, 256 images from the ImageNet test set.

In the first scenario we tried to simulate a poorly performant video camera connected to a Raspberry Pi 3 that executes object recognition in order to spot an intruder; it also performs a partial computation of the data coming from its sensors before sending them to the cloud, therefore exploiting the computing continuum. The device that has been used in such a scenario is a Raspberry Pi 3, whose technical features are summarised in Table 4.13.

| Device | Raspberry Pi 3 Model B |
|---|---|
| Processor | Broadcom BC,2837 (4x Cortex-A53 64-bit SoC@1.4GHz) |
| Memory | 1GB LPDDR2 SDRAM |
| Storage | 32 GB Micro SDHC (R 95MB/s, W 20MB/s) Raspbian Buster Linux Kernel |
| OS | v4.20 |
| Tensorflow Version | Tensorflow Lite Interpreter 2.10 |

*Table 4.13 - Main technical features of the first edge device*

The second scenario aims at evaluating the performance of a mid-end mobile device, such as a smartphone, that is carrying out face recognition or image processing for some purpose. The employed device is an Odroid N2, whose technical features are summarised in Table 4.14.

| Device | Odroid N2 |
|---|---|
| Processor | Amlogic S992X (4x Cortex-A73 CPU@1.8GHz + 2x Cortex-A53@1.9GHz) |
| Memory | 4GB LPDDR4 RAM@1320MHz |
| Storage | 128 GB Micro SDHC (R 95MB/s, W 20MB/s) |
| OS | Ubuntu 18.04 Linux Kernel v4.20 |
| Tensorflow Version | Tensorflow Lite Interpreter 2.10 |

*Table 4.14 - Main technical features of the second edge device*

The third and last scenario employs an NVIDIA Tegra X2 to simulate the behaviour of a high-performance mobile device capable of receiving images from different sources. Its technical features are summarised in Table 4.15.

| Device | NVIDIA Tegra X2 |
|---|---|
| CPU | Custom 64-bit processor (2x NVIDIA Denver2 ARMv8 + 4x ARMv8 ARM Cortex-A57) |
| GPU | NVIDIA Pascal architecture with 256 CUDA cores |
| Memory | 8GB LPDDR4 SDRAM |
| Storage | 32 GB eMMC 5.1 (R 250MB/s, W 125MB/s) |
| OS | Ubuntu 18.04 Linux Kernel v4.9 |
| Tensorflow Version | Tensorflow-GPU 1.4 |

*Table 4.15 - Main technical features of the third edge device*

**Experimental results**

In this section, first, we deal with the data preparation and algorithms employed and then we will show the actual preliminary results obtained by the regressors in the three analyses performed (namely Next-Layer, All-Layers and VGG16 to VGG19 Analysis). Moreover, we will outline the SFS analysis results.

In the system under test, we instrumented the inference application code and measured the time needed $T(j)$ by the Edge device to perform the computation up to layer $j$ of image $i$ and the time required by the Cloud device to complete its processing according to the structure of the reference network. However, we are only interested in the former edge device processing time, that breaks down to:

$$T(j) = \sum_{k=2}^{j} \left(T_{read}(input_k) + T_{comp}(layer_k) + T_{write}(output_k)\right) \quad (2)$$

where:

- $T_{read}(input_k)$ is the time required to read the input that comes from the $k-1$ layer.
- $T_{comp}(layer_k)$ is the time required for the computation of layer $k$.
- $T_{write}(output_k)$ is the time required to write the results of layer $k$.

Notice that the sum starts at layer 2 because the first layer represents the raw input.

Once we have obtained this decomposition, we can say that the time $T(j)$ depends on the following features:

- Size of the input;
- Number of operations performed by the layer, depending on the layer type and computed as discussed in Table 4.1 (Section 4.5.1).
- Size of the output.

In particular, since we will only try to predict convolutional layer's time, the number of operations of a layer k is proportional to:

$$n_{ops(k)} = H_f W_f C_{in} C_{out} \ * pixels \qquad (3)$$

as proposed in [Gianniti2018b], where $H_f$ and $W_f$ are the height and width of the filter respectively, while $C_{in}$ and $C_{out}$ are the number of channels in input/output and $pixels$ is the number of pixels of the analysed input.

Due to the limitations of Tensorflow, which do not allow to profile the time of a single layer of the CNN, $T_{comp}(layer_k)$ (i.e., the computational time required by each layer) has been calculated as:

$$T_{comp}(layer_k) = T(k) - E[T(k-1)] - T_{read}(input_k) - T_{write}(output_k) \qquad (4)$$

Where $E[T(k-1)]$ is the empirical mean of the time required by the computation up until the previous layer.

However, we choose as target variable $TLT(k)$, that is the Total Layer Time, from the following formula:

$$TLT(k) = T_{comp}(layer_k) + T_{read}(input_k) + T_{write}(output_k) = T(k) - E[T(k-1)] \quad (5)$$

due to the difficulties in measuring the times required to read/write the memory. The final dataset is therefore summarised in Table 4.16.

| Feature | Description |
|---|---|
| Layer Number | Incremental number of the layer (only for technical purposes, not a feature) |
| Height | Height of the filter |
| Width | Width of the filter |
| Input Channels | Number of channels of the input |
| Output Channels | Number of channels of the output |
| FLOPS | $n_{ops}$ as in Formula (3) |
| TLT | Total Layer Time as in Formula (5) |

*Table 4.16 - Dataset summary*

### Data Preprocessing

Unfortunately, the application under study in many cases and across different edge devices settings, experienced a cold start and, more importantly, the operating system affected the data with random variance, due to its other tasks that were going on in parallel with the experiments. For these reasons, outliers filtering was necessary (for example some per layer time computed through Formula (5) were negative). The method we used here is a distance-based approach [Ben-Gal2005]: for every layer, a proximity index of every point was created, which took into consideration its 5 closest points in the dataset. The closer those points were, the higher the index was.

In particular, every data point $x_i$ had such proximity index $p(x_i)$ computed as:

$$p(x_i) = \sum_{j \in N(x_i)} \frac{1}{d(x_i, x_j)}$$

Where $N(x_i)$is the set of the 5 closest neighbours of $x_i$ and $d(x_i, x_j)$ is their euclidean distance.

Eventually, we kept only the points that contributed the most to the total sum of the indexes $\sum_i \quad p(x_i)$, keeping points up until reaching 99% of it. This method allowed us to obtain fairly dense clouds of data points for each layer.

Another important thing to notice is the optimization issue linked to the batch dimension: due to its non-homogeneity, we decided to consider only data that came from experiments that had unitary batch size.

We also performed feature enhancement and Sequential Forward feature Selection (SFS); in particular:

- each feature was normalised, and we performed feature augmentation by considering also the inverse and the crossover terms up to the second degree;
- SFS kept, at most, the best 8 features.

We decided to test three different ML algorithms: XGBoost, Ridge Regression (for both of them performing, in some scenarios, also SFS), and the Draper and Smith method. In Table 4.17, we summarise the hyperparameters explored in the training phase: to select the best ones when a prior probability is specified, a 5-fold cross-validation procedure was performed by relying on the HyperOpt framework. The Draper and Smith model was defined such that the probability of adding or removing a feature was 0.05 and 0.1, respectively.

| Algorithm | Hyperparameter Name | Values | Type |
|---|---|---|---|
| XGBoost | min_child_weight<br>gamma<br>n_estimators<br>learning_rate<br>max_depth | 1<br>loguniform(0.1, 10)<br>1000<br>loguniform(0.01,1)<br>100 | Fixed<br>Prior probability<br>Fixed<br>Prior probability<br>Fixed |
| Ridge Regression | alpha | loguniform(0.01,10) | Prior probability |

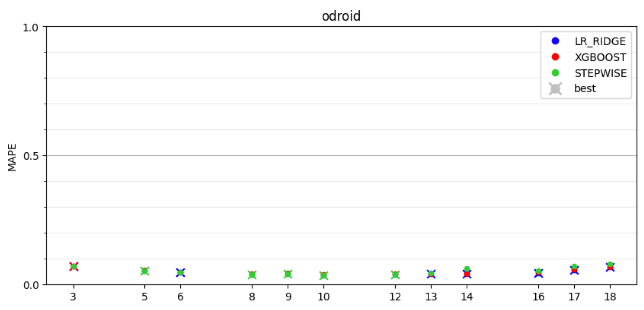
*Table 4.17 - Hyperparameters summary*

*VGG16: Next-Layer Analysis*

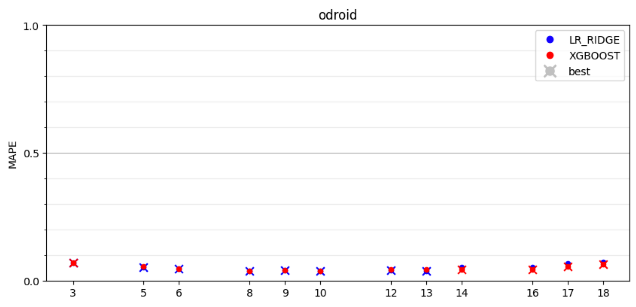
The goal of a next-layer analysis is to predict the time required to run layer *j* + 1, through a model that was trained considering experiments running up to layer *j*. Specifically, we considered a training set including layers between 3 and 9 (the convolutional layers), while the test set includes layer 10.

We report in the following the results obtained on the cross validation set (Figure 4.27) and the test set (Figure 4.28), on the three devices described in the previous section. Note that, in the plots, the different colours represent the different ML models we have considered. A cross characterises the best model among the alternatives, while results are represented as a triangle if the corresponding MAPE exceeds 100%.

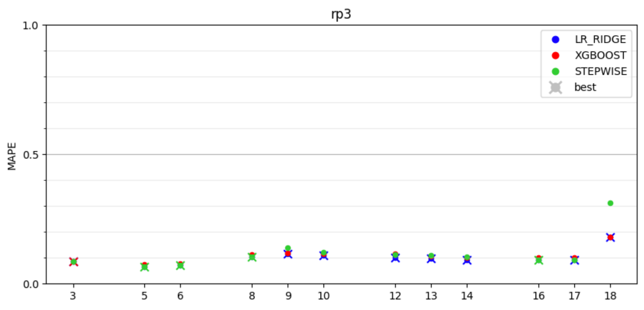
Note that the Draper and Smith model is always referred to in the following as the *Stepwise* model.



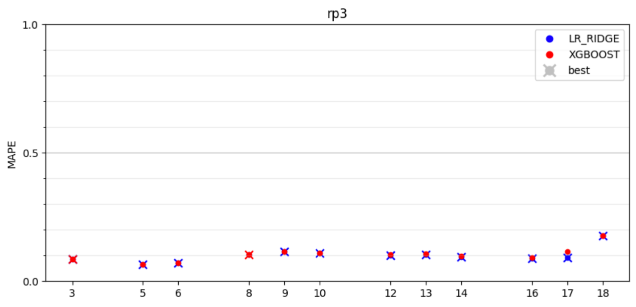
Odroid N2: Cross-validation MAPE without SFS



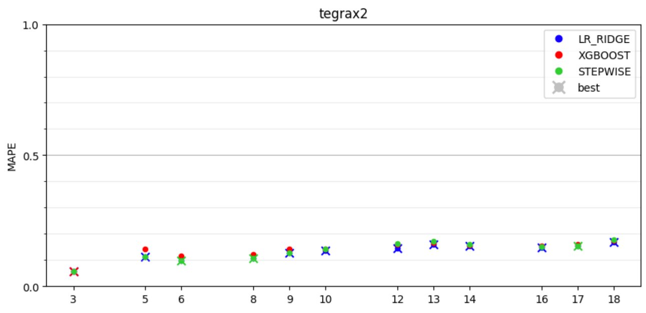
Odroid N2: Cross-validation MAPE with SFS



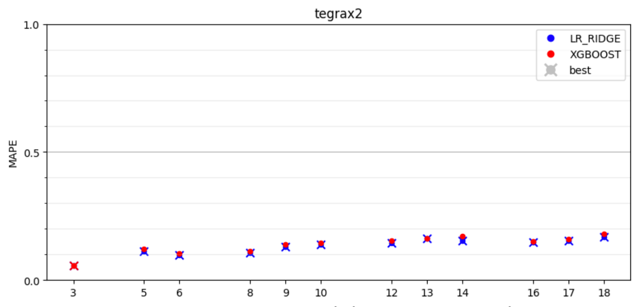
Raspberry Pi 3: Cross-validation MAPE without SFS



Raspberry Pi 3: Cross-validation MAPE with SFS

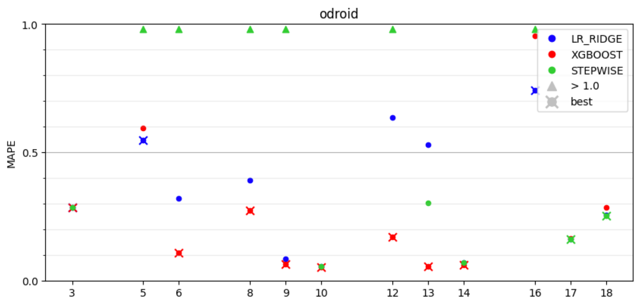


NVIDIA Tegra X2: Cross-validation MAPE without SFS

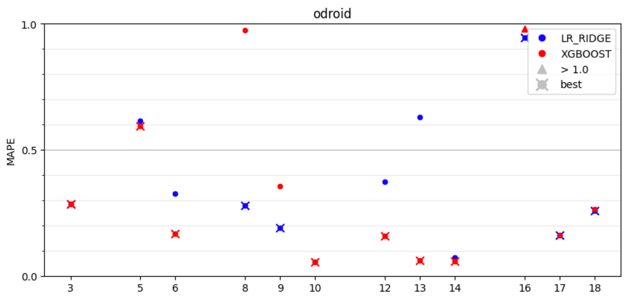


NVIDIA Tegra X2: Cross-validation MAPE with SFS

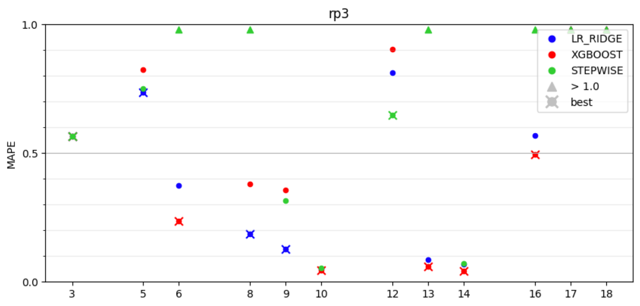
*Figure 4.27 - Cross validation MAPE with the different models on all the considered devices*



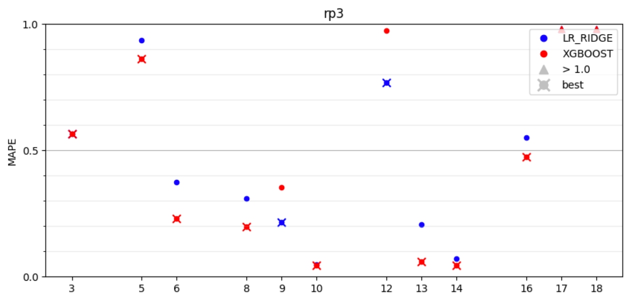
Odroid N2: MAPE on the test set without SFS



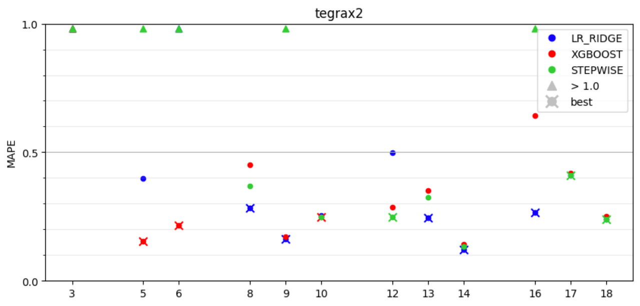
Odroid N2: MAPE on the test set with SFS



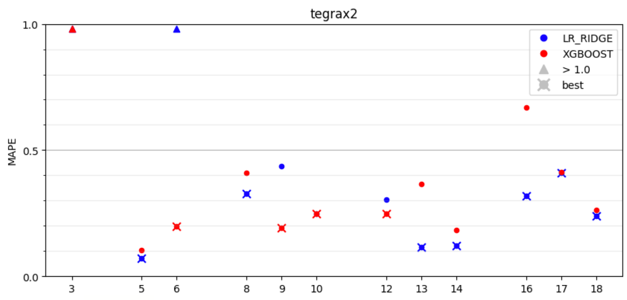
Raspberry Pi 3: MAPE on the test set without SFS



Raspberry Pi 3: MAPE on the test set with SFS



NVIDIA Tegra X2: MAPE on the test set without SFS



NVIDIA Tegra X2: MAPE on the test set with SFS

*Figure 4.28 - MAPE on the test set with the different models on all the considered devices*

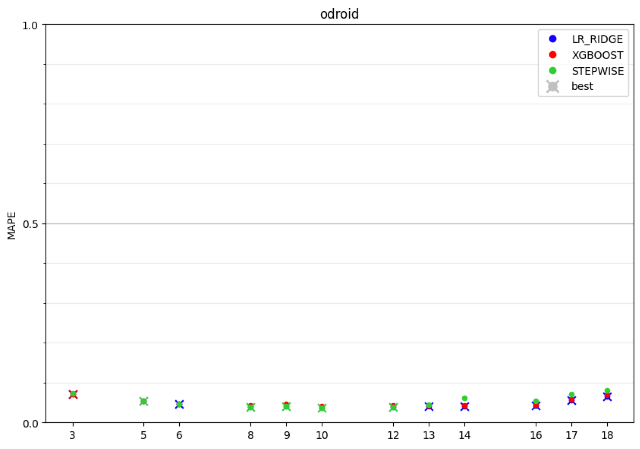
When analysing the MAPE on the test set (Figure 4.28), it is relevant to notice that the Raspberry Pi 3 dataset was very noisy. This contributed to an overall poor scoring of the different algorithms, with a MAPE sometimes above 50%. Since the cross-validation errors are significantly lower (Figure 4.27), this outlines a poor generalisation capability for all the considered models. A similar behaviour can be observed for the Odroid N2 dataset, while results are generally better for NVIDIA Tegra X2 device, where the MAPE on the test set is usually between 10 and 50%.

*VGG16: All Layers Analysis*

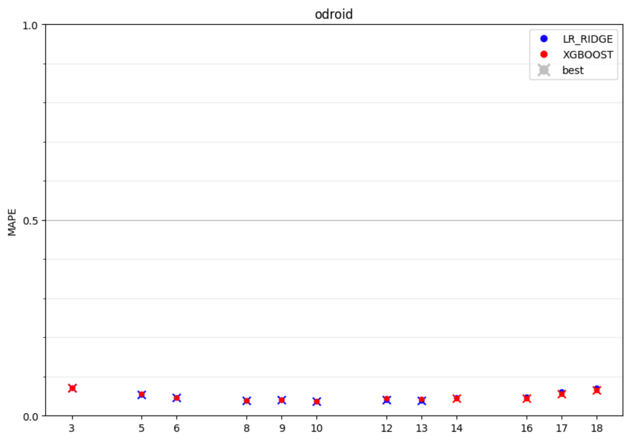
The goal of the all-layers analysis is to predict the time required to run all layers from *j* + 1, through a model that was trained considering experiments running up to layer *j*. Specifically, we considered a training set including layers between 3 and 9 (the convolutional layers), while the test set includes layers 10 to 18.

We report in the following the results obtained on the cross validation set (Figure 4.29) and the test set (Figure 4.30), on Odroid N2 and NVIDIA Tegra X2 devices described in the previous section. The Raspberry PI 3 was not considered because the amount of collected data was not enough to perform the analysis. Note that, in the plots, the different colours represent the different ML models we have considered. A cross characterises the best model among the alternatives, while results are represented as a triangle if the corresponding MAPE exceeds 100%.

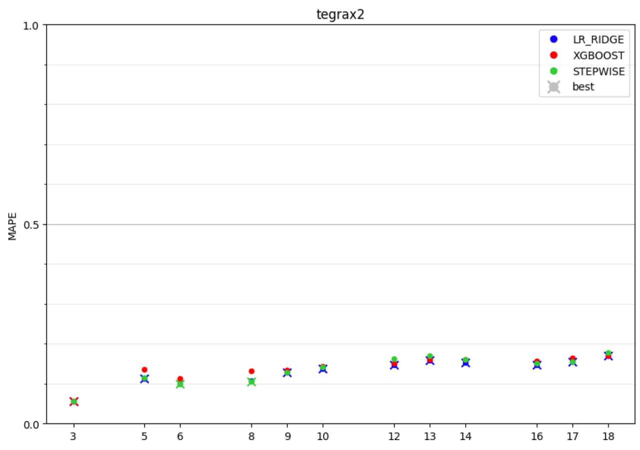
Note that the Draper and Smith model is always referred to in the following as the *Stepwise* model.



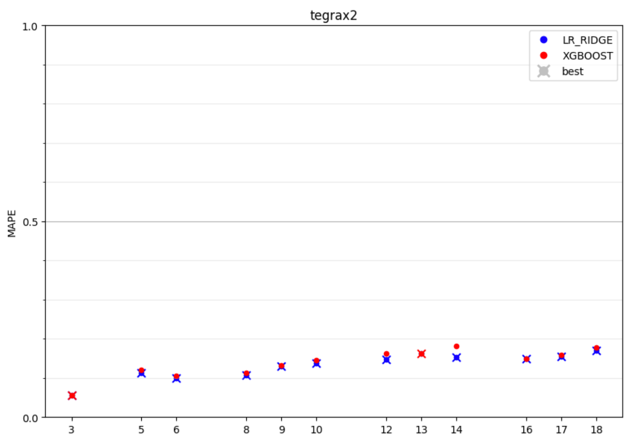
Odroid N2: Cross-validation MAPE without SFS



Odroid N2: Cross-validation MAPE with SFS

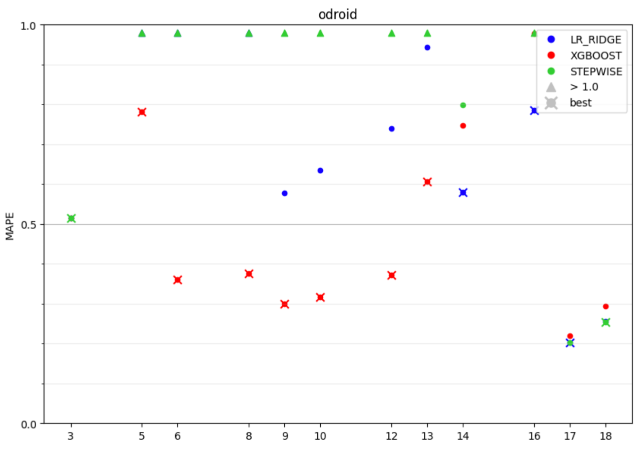


NVIDIA Tegra X2: Cross-validation MAPE without SFS

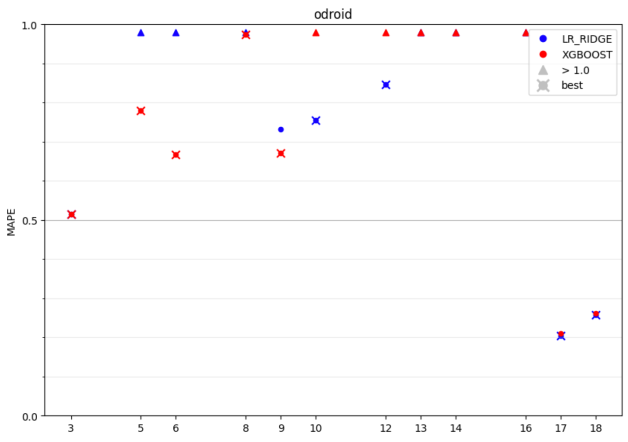


NVIDIA Tegra X2: Cross-validation MAPE with SFS

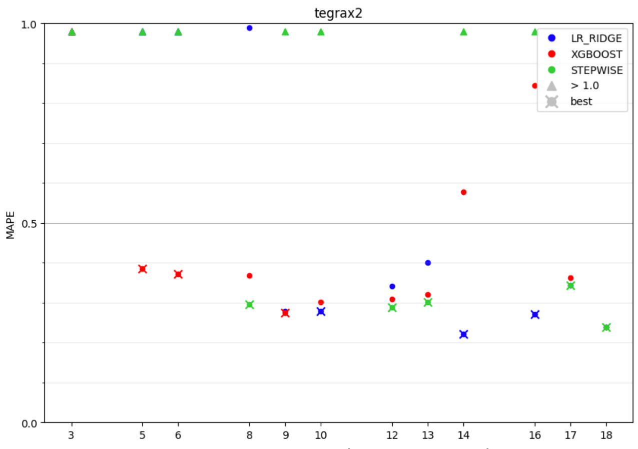
*Figure 4.29 - Cross validation MAPE with the different models on all the considered devices*



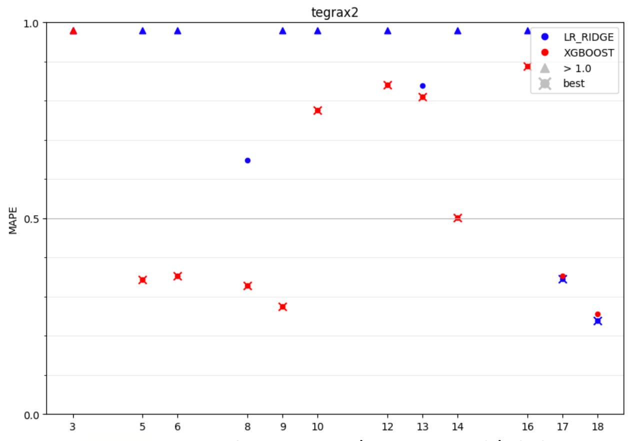
Odroid N2: MAPE on the test set without SFS



Odroid N2: MAPE on the test set with SFS



NVIDIA Tegra X2: MAPE on the test set without SFS



NVIDIA Tegra X2: MAPE on the test set with SFS

*Figure 4.30 - MAPE on the test set with the different models on all the considered devices*

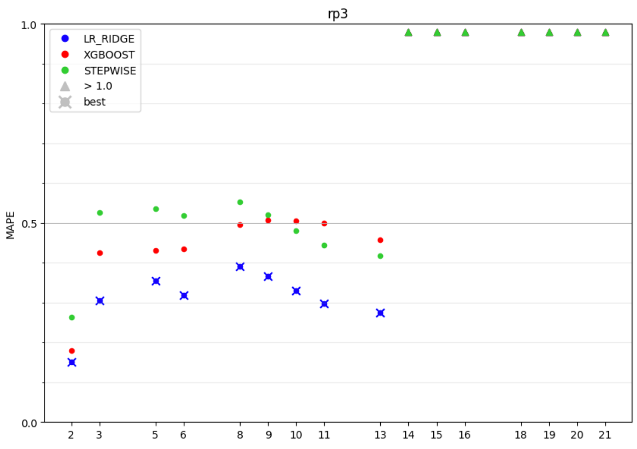
The results in Figure 4.30 show that XGBoost performs generally better than the other analysed models, obtaining worse results, though, when paired with SFS. It is relevant to notice that the three models, however, obtain similar performance, in terms of identifying the best model, when the NVIDIA Tegra X2 device is considered (without performing SFS). In the considered scenarios, the best model achieves a MAPE in the range 20-50%.

*VGG16 to VGG19 Analysis*

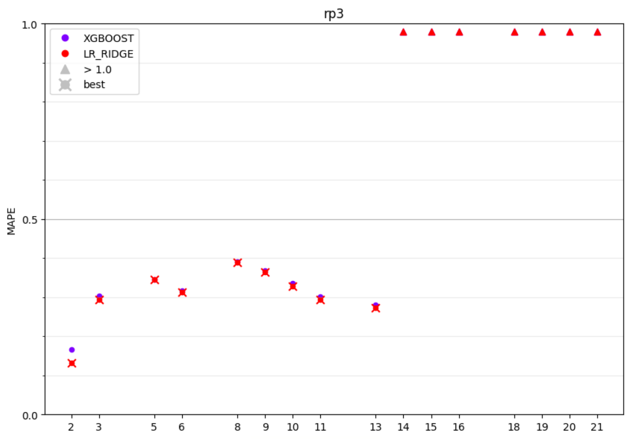
The tests described in this section aimed at evaluating how the different ML models were able to predict the performance of VGG19 network when trained on experiments performed with the VGG16 network, thus testing the generalisation capability of the models on a different (even if similar) CNN. Specifically, a single layer of VGG19 was considered as a target for the prediction.

The extrapolation MAPE obtained when analysing the Raspberry Pi 3 and NVIDIA Tegra X2 devices is reported in Figure 4.31.

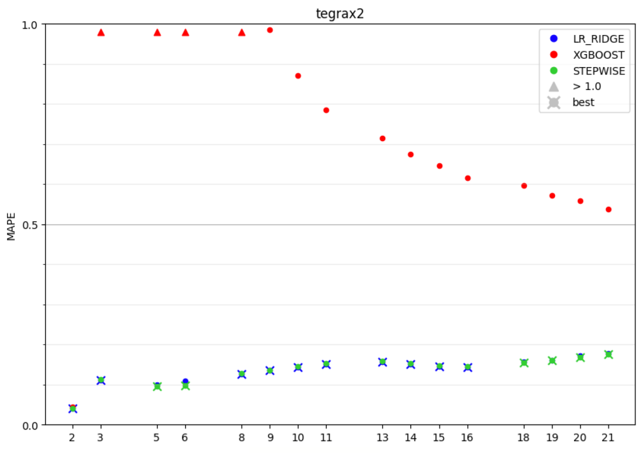
Note that the Draper and Smith model is always referred to in the following as the *Stepwise* model.



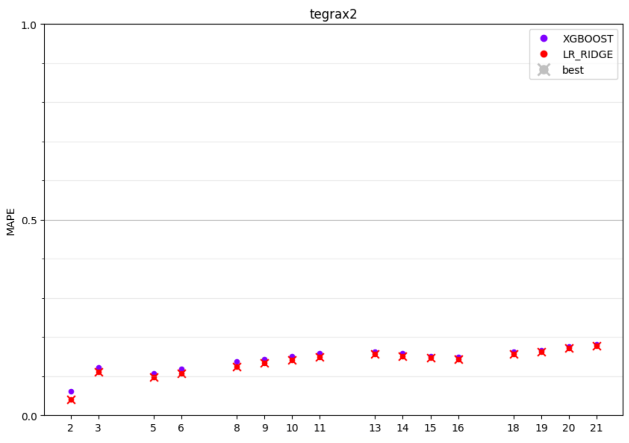
Raspberry Pi 3: Extrapolation MAPE without SFS



Raspberry Pi 3: Extrapolation MAPE with SFS



NVIDIA Tegra X2: Extrapolation MAPE without SFS



NVIDIA Tegra X2: Extrapolation MAPE with SFS

*Figure 4.31 - Extrapolation MAPE with the different models on all the considered devices*

In this scenario, we can observe that the XGBoost model achieves very poor performance when SFS is not applied, particularly for the NVIDIA Tegra X2 dataset (for which, instead, the performance of Ridge Regression and the Draper and Smith model are almost the same).

Overall, the adoption of ML based performance models in this scenario is promising, achieving MAPE for the best model always lower than 20% on Odroid and NVIDIA Tegra X2.

The detailed results are available at: https://doi.org/10.5281/zenodo.5735476.

## Per-layer performance model in Neural Architecture Search

In this section, we present the result of integrating a-MLLibrary ML models with the Neural Architecture Search component POPNAS (described in Section 4.6). We have investigated different models through the use of the a-MLLibrary. Experiments were conducted on a subset of CIFAR-10. In particular, since CIFAR-10 has five batches of 10,000 images each, we used a randomly selected single batch, splitting it into a training set of 9,000 images and a validation set of 1,000 images. Experiments were performed on an NVIDIA Tesla V100 SXM2, with 16GB VRAM. As a performance model regressor we chose Ridge Regression, XGBoost [Chen2016], and NNLS (Non-negative least square). Ridge regressor was trained with $\alpha$ set to 0.1. For NNLS, we experimented both with and without the fit_intercept set to True. For XGBoost, we considered 1 and 3 as child-weight thresholds to stop the tree splitting if exceeded; gamma, that regularises the information across the trees, allowing the node addition only if the associated gain is larger or equal to the given value, equal to 0 and 1; numbers of tree regressors: 50, 100, 150 and 250. We selected as possible learning rate 0.01, 0.05, and 0.1. The trees' maximum depth has been chosen equal to 1, 2, 3, 5, 9, and 13. The a-MLLibrary performed a 5-fold cross-validation to find, through a grid search strategy, the best hyperparameter settings for each regressor. In our experiments, we trained time-regressors on data available at iteration b, and evaluated them at iteration b+1, before the updating procedure. In the ablation study, we first compared the selected methods and then we have progressively pruned the less performing approaches, assessing the impact of the proposed optimization strategies: input removal, which removes the information related to the block's input, not necessary to solve the problem, and focuses only on the operations type, static re-index in which each operator is associated to an integer value ranging from 1 to the size of the operator set, and dynamic re-index in which a heuristic re-index method that takes into account the distance between indices, is considered. As a reference model, we considered *block sum* which predicts the training time as the sum of the time taken by the cell obtained in the previous iteration.

In the first plain comparison, it can be deduced from Figure 4.32 that block sum shows better results only at the first iteration, while it tends to gradually underestimate the training time from the second iteration onwards (each cluster of data points is representative of candidate networks with increasing number of blocks with b=2, 3, 4). This consideration allows us to deduce that the training time increase induced by a block addition is not entirely linear, but it introduces a bias dependent on the number of blocks.

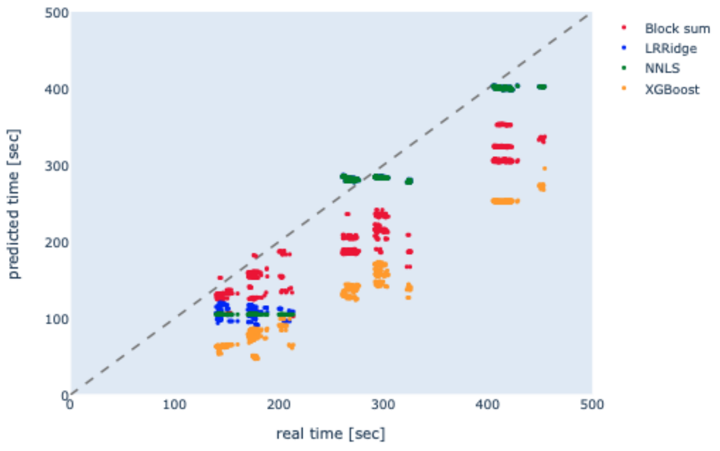


*Figure 4.32 - Performance of time-regressors. The identified clusters are related to the blocks iterations*

**Static and Dynamic Re-index:** In this step, we present the comparison between static and dynamic re-indexes on NNLS algorithm which provided better results with respect to XGBoost and Ridge in most of the considered scenarios. We trained the POPNAS algorithm according to the previous setup choices. The benefits of dynamic re-index can be immediately noticed as shown in Table 4.18. In fact, with this technique we notice an overall lower average relative error of 0.2129, compared to static re-index, which instead achieves an error of 0.2380. Moreover, NNLS with static re-index has a limited tendency to underestimate the training time. In Table 4.18, we can witness the better performance of dynamic re-index through iterations, reaching an improvement of 0.05 at the second one, compared to static re-index. Even if the dynamic re-index did not grant an almost perfect accuracy, it seems to halve the MAPE, reducing it from 0.6044 to 0.3490, at the expense of a lower MAPE increasing from 0.0171 to 0.0813.

| | Avg MAPE | Max MAPE | Min MAPE |
|---|---|---|---|
| **Static** | 0.2380 | 0.6043 | 0.0171 |
| **Dynamic** | 0.2129 | 0.3490 | 0.0813 |

*Table 4.18 - Average, maximum and minimum MAPE of NNLS, with static re-index and dynamic re-index*

The detailed results are discussed in [Lomurno2021].

## PyCOMPSs Performance Models Accuracy

The data collected from the execution of the CSVM PyCOMPSs application (see Section 4.8.1) were used to generate and evaluate a set of Machine Learning models (exploiting XGBoost, Ridge Regression, Decision Tree, Random Forest and SVR), with and without Sequential Forward feature Selection. The features provided to these models are discussed in Section 4.5.1, while the hyperparameters tested for each model are reported in Table 4.19. The hyperparameter tuning operations were performed by relying on the HyperOpt framework [HyperOptGithub], setting the maximum number of evaluations to 10. The MAPE obtained with the different models is reported in Table 4.20.

Two additional sets of tests have been performed, in order to assess the interpolation and the extrapolation capabilities of our models, i.e., to test whether they are able to predict values in areas of the features space that have been sufficiently observed during the training phase or, vice versa, in regions of the parameters space not sufficiently explored. Specifically, both interpolation and extrapolation are tested with respect to the most relevant feature, i.e., the number of available cores. In the first scenario, we considered a training set made by the execution times collected at 48, 144, 288, and 768 cores, while we predicted the execution time at 96, 192, and 384 cores. In the second scenario, instead, we built the training set considering a number of cores going from 48 to 288, and we predicted the execution time when exploiting a higher number of cores, namely 384 and 768. The corresponding results are reported in Table 4.21 and Table 4.22, respectively. Moreover, we included two plots reporting the real and predicted values when exploiting the interpolation and extrapolation best models in Figure 4.33.

The detailed results are available at: https://doi.org/10.5281/zenodo.5760964.

| Algorithm | Hyperparameter Name | Values | Type |
|---|---|---|---|
| XGBoost | min_child_weight<br>gamma<br>n_estimators<br>learning_rate<br>max_depth | 1<br>loguniform(0.1, 10)<br>1000<br>loguniform(0.01,1)<br>100 | Fixed<br>Prior probability<br>Fixed<br>Prior probability<br>Fixed |
| Ridge Regression | alpha | loguniform(0.01,10) | Prior probability |
| Decision Tree | criterion<br>max_depth<br>max_features<br>min_samples_split<br>min_samples_leaf | mse<br>3<br>auto<br>loguniform(0.01,1)<br>loguniform(0.01,0.5) | Fixed<br>Fixed<br>Fixed<br>Prior probability<br>Prior probability |
| Random Forest | n_estimators<br>criterion<br>max_depth<br>max_features<br>min_samples_split<br>min_samples_leaf | 5<br>mse<br>quniform(3,6,1)<br>auto<br>loguniform(0.01,1)<br>1 | Fixed<br>Fixed<br>Prior probability<br>Fixed<br>Prior probability<br>Fixed |
| SVR | C<br>epsilon<br>gamma<br>kernel<br>degree | loguniform(0.001,1)<br>loguniform(0.01,1)<br>1e-7<br>linear<br>2 | Prior probability<br>Prior probability<br>Fixed<br>Fixed<br>Fixed |

*Table 4.19 - Hyperparameters summary*

| Algorithm | SFS | MAPE |
|---|---|---|
| XGBoost | no | **0.32** |
| | yes | **0.32** |
| Ridge Regression | no | 2.48 |
| | yes | 14.71 |
| Decision Tree | no | 6.94 |
| | yes | 3.90 |
| Random Forest | no | 2.06 |
| | yes | 3.15 |
| SVR | no | 10.99 |
| | yes | 13.37 |

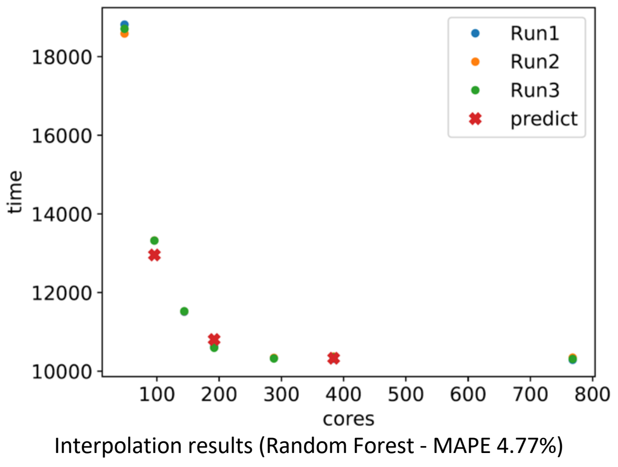
*Table 4.20 - MAPE [%] on the cross validation*

| Algorithm | SFS | MAPE |
|---|---|---|
| XGBoost | no | 7.29 |
| | yes | 7.29 |
| Ridge Regression | no | 4.94 |
| | yes | 14.42 |
| Decision Tree | no | 15.04 |
| | yes | 10.87 |
| Random Forest | no | 5.90 |
| | yes | **4.77** |
| SVR | no | 18.23 |
| | yes | 6.84 |

*Table 4.21 - MAPE [%] on interpolation*

| Algorithm | SFS | MAPE |
|---|---|---|
| XGBoost | No | **0.23** |
| | Yes | **0.23** |
| Ridge Regression | no | 205.09 |
| | yes | 134.09 |
| Decision Tree | no | 5.28 |
| | yes | 1.59 |
| Random Forest | no | 8.16 |
| | yes | 1.43 |
| SVR | no | 138.83 |
| | yes | 192.45 |

*Table 4.22 - MAPE [%] on extrapolation*



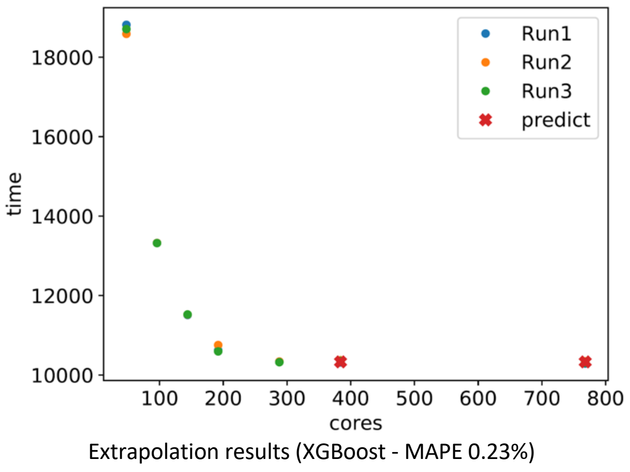


*Figure 4.33 - Interpolation and extrapolation results with the best models*

According to the results we obtained, since the MAPE is always lower than 30% we have already achieved the KPI2.2 (Accuracy of performance estimation models for individual components, evaluated as mean of the average absolute percentage error <=20) foreseen at M30.

### Mask Detection Application Performance Models Accuracy

Performance models for the mask detection use case application (see Section 2.2) were generated starting from two sets of experiments, involving either the *Blurry Faces* or the *Mask Detector* task. Each of them required to process a group of 10 videos lasting 15 minutes. The complete processing of this group of videos is referred to as a job.

The experiments were run first of all on private cloud Virtual Machines configurations, by varying the number of nodes between 1 and 5, and by setting the number of containers accordingly, to gather data for 1, 4, 8, 12, 16, and 20 cores. Each experiment has been repeated three times. Moreover, we measured the total runtime of the full application, involving both the Blurry Faces and the Mask Detector task, in order to check whether the performance models built for the two separate phases can be used to predict the execution time of the full application.

A similar list of experiments has been performed, for the Blurry Faces task only, by exploiting a cluster of Raspberry Pi devices. Specifically, we collected data by varying the number of nodes between 1 and 3, and by exploiting 1 or 4 containers per node. This allowed us to generate tests with 1, 2, 3, 4, 8, and 12 cores. Each experiment has been repeated twice, and the data set has been enlarged by adding the average runtime for each number of cores. Notice that, in this scenario, the K8s master node is co-located with the workers in the cluster, which increases the noise of the collected runtimes.

The data collected from the execution of the Mask Detector use case application were used to generate and evaluate a set of Machine Learning models (exploiting XGBoost, Ridge Regression, Decision Tree, Random Forest and SVR. The features provided to these models are discussed in Section 4.5.1, while the hyperparameters tested for each model are the same considered for the scenario described in the previous section and are reported in Table 4.19. The hyperparameter tuning operations were performed by relying on the HyperOpt framework [HyperOptGithub], setting the maximum number of evaluations to 10. The MAPE obtained with the different models is reported in Table 4.23.

| Algorithm | MAPE (Blur Faces task) | MAPE (Mask Detection task) |
|---|---|---|
| XGBoost | **1.3** | **0.69** |
| Ridge Regression | 3.27 | 9.29 |
| Decision Tree | 10.21 | 35.89 |
| Random Forest | 43.36 | 29.2 |
| SVR | 42.42 | 61.57 |

*Table 4.23 - MAPE [%] on the cross validation of Mask Detection application*

Two additional sets of tests have been performed, in order to assess the interpolation and the extrapolation capabilities of our models, i.e., to test whether they are able to predict values in areas of the features space that have been sufficiently observed during the training phase or, vice versa, in regions of the parameters space not sufficiently explored. Specifically, both interpolation and extrapolation are tested with respect to the most relevant feature, i.e., the number of available cores. In the interpolation scenario, we considered a training set made by the execution times collected at 1, 4, 12, and 20 cores, while we predicted the execution time at 8 and 16 cores. In the extrapolation scenario, instead, we built the training set considering a number of cores going from 1 to 16, and we predicted the execution time when exploiting a higher number of cores, namely 20. The corresponding results are reported in Table 4.24 and Table 4.25, respectively. Moreover, we included two plots reporting the real and predicted values when exploiting the interpolation and extrapolation best models in Figure 4.34 and Figure 4.35 for the Blurry Faces and Mask Detector tasks considering the VM setup, respectively.

| Algorithm | MAPE (Blur Faces task) | MAPE (Mask Detection task) |
|---|---|---|
| XGBoost | **3.02** | 21.73 |
| Ridge Regression | 23.45 | **14.91** |
| Decision Tree | 14.71 | 30.53 |
| Random Forest | 33.36 | 65.17 |
| SVR | 70.23 | 70.14 |

*Table 4.24 - MAPE [%] on interpolation*

| Algorithm | MAPE (Blur Faces task) | MAPE (Mask Detection task) |
|---|---|---|
| XGBoost | **1.7** | 40.1 |
| Ridge Regression | 16.83 | 267.43 |
| Decision Tree | 72.54 | **40.06** |
| Random Forest | 37.21 | 56.45 |
| SVR | 129.52 | 258.15 |

*Table 4.25 - MAPE [%] on extrapolation*

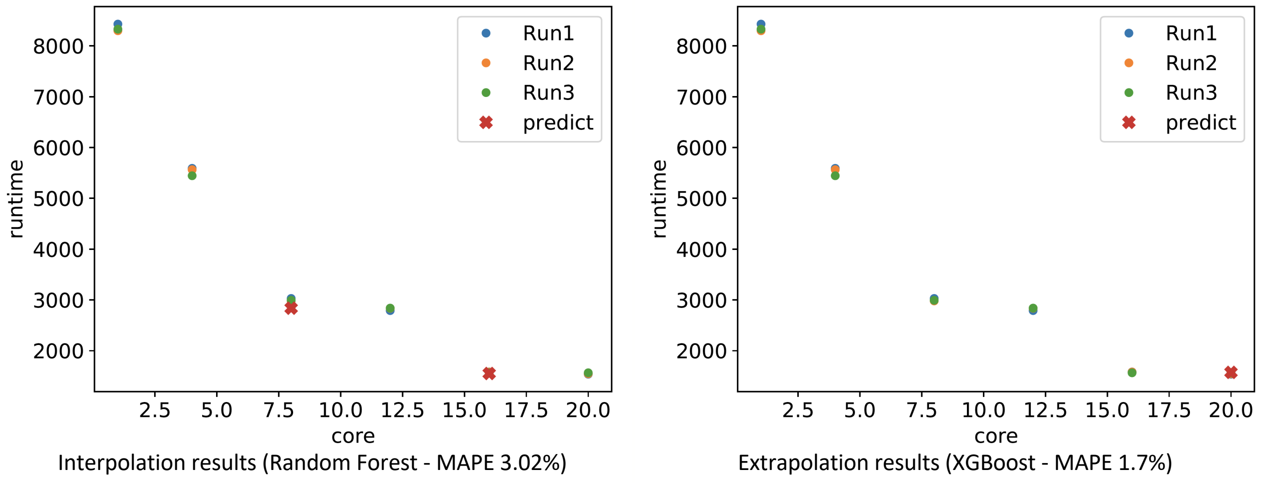


*Figure 4.34 - Interpolation and extrapolation results with the best models (Blur Faces task)*

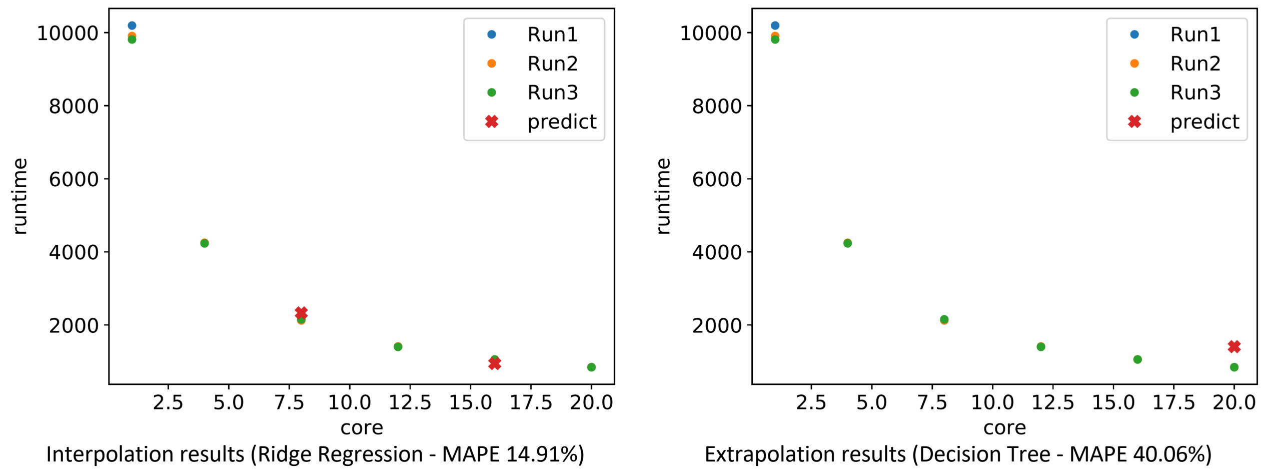


*Figure 4.35 - Interpolation and extrapolation results with the best models (Mask Detection task)*

The interpolation and extrapolation best models have been used to predict the runtime of the whole mask detection application, involving both the Blurry Faces and the Mask Detector tasks. Specifically, a MAPE of 9.46% is obtained when combining the XGBoost model (for Blurry Faces) and Ridge Regression model (for Mask Detector) to predict the runtime of the complete application in the interpolation scenario. The MAPE of the extrapolation scenario is instead about 40%, in line with the accuracy obtained by the Decision Tree model for the Mask Detector task. These results already are in line with the KPI K2.3 (Accuracy of performance estimation models for distributed applications, evaluated as mean of the average absolute percentage error) foreseen at M24.
The plots reporting the real and predicted values for interpolation and extrapolation best models are reported in Figure 4.36.

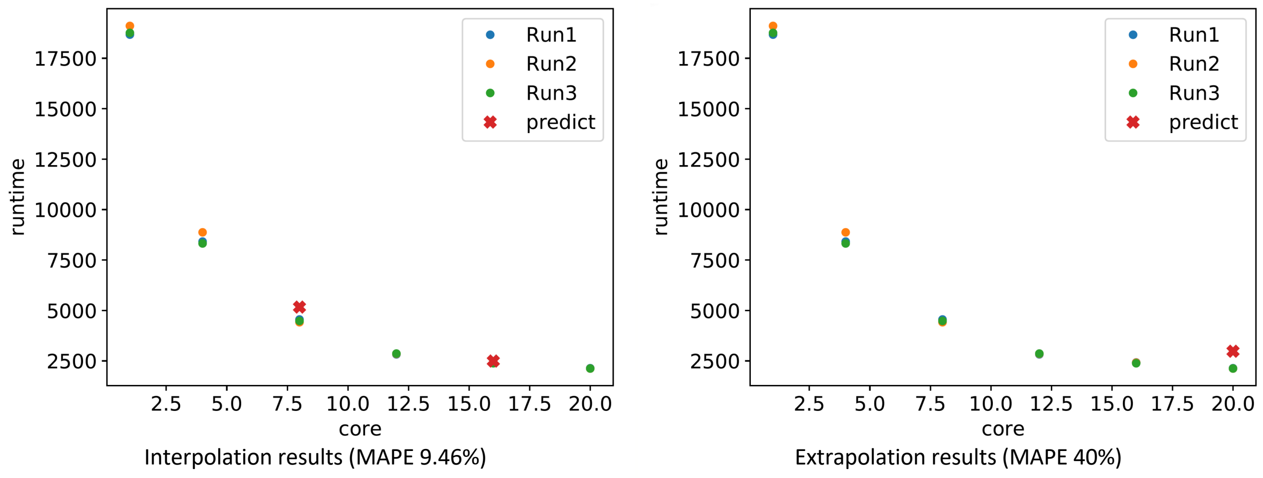


*Figure 4.36 - Interpolation and extrapolation results when predicting the runtime of the whole application*

Finally, an additional set of tests has been performed for the Blurry Faces task executed on a Raspberry Pi cluster. Since these experiments required a significant time to run, data were collected only for two instances per each number of cores. Therefore, to enlarge the dataset, we performed data augmentation by considering the average execution time at each number of cores.
In this scenario, we considered for interpolation a training set made by the execution times collected at 1, 2, 3, 4, and 12 cores, while we predicted the execution time at 8 cores. For extrapolation, instead, we built the training set considering a number of cores going from 1 to 8, and we predicted the execution time when exploiting a higher number of cores, namely 12. The corresponding MAPE values are reported in Table 4.26, while two plots reporting the real and predicted values when exploiting the interpolation and extrapolation best models are reported in Figure 4.37.

It is relevant to notice that our model achieves a good interpolation capability, even if the number of training data is quite low. On the other hand, the extrapolation results are negatively affected by the fact that the training data are noisy and limited. Extrapolation capabilities will be investigated during the second year of the project considering also the use of accelerators on board of Raspberry Pi devices.
The detailed results are available at https://doi.org/10.5281/zenodo.5784215.

| Algorithm | MAPE (cross validation) | MAPE (interpolation) | MAPE (extrapolation) |
|---|---|---|---|
| XGBoost | **0.35** | 33.5 | 50.46 |
| Ridge Regression | 6.17 | 27.82 | 223.69 |
| Decision Tree | 12.87 | 32.04 | **50.46** |
| Random Forest | 39.99 | **16.86** | 73.09 |
| SVR | 34.94 | 34.3 | 80.57 |

*Table 4.26 - MAPE [%] for experiments executed on a Raspberry Pi cluster*

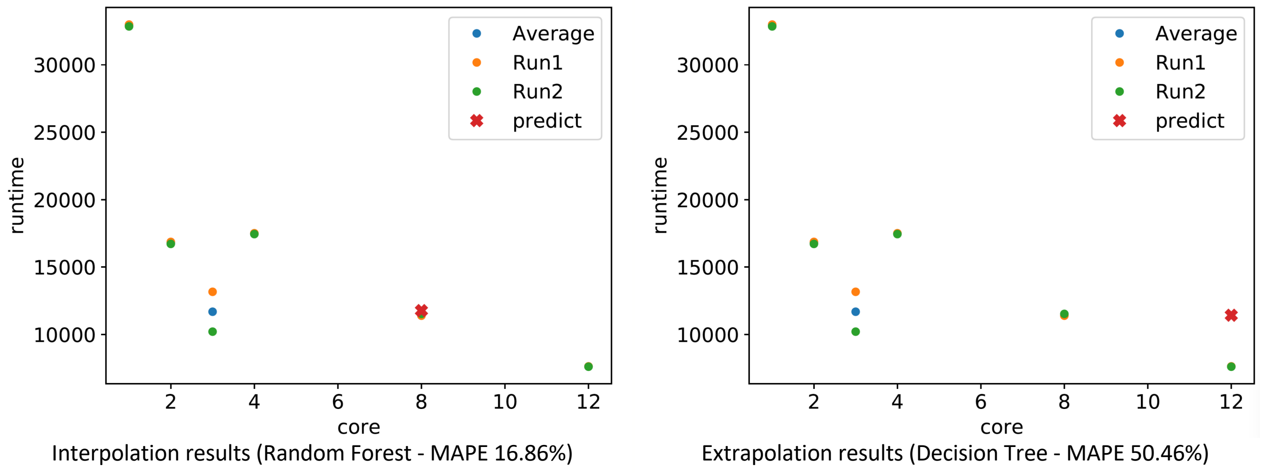


*Figure 4.37 - Interpolation and extrapolation results with the best models (Blur Faces task, RPi's cluster)*

### 4.8.3 AI Neural Architecture Search

After several ablation studies, the first POPNAS version has been released with the following characteristics:

- Number of maximum blocks: 5
- Number of cells: 8
- Number of best models per iteration (K): 256
- Controller algorithm: LSTM neural network (as in PNAS)
- Regressor algorithm: Non-Negative Least Squares
- Skip connections: False
- Learning rate: 0.01
- Dataset: CIFAR10 (first batch)

The experiments have been run on an NVIDIA GeForce GTX1080Ti. From a fair comparison between the same run of PNAS and POPNAS, despite an accuracy drop of 9.6%, the algorithm achieved a search boosting of 204.9% for the whole research, and a training time boosting for the best network of 173.7%. The best cell configuration that composes the top-1 found network is shown in Figure 4.38.

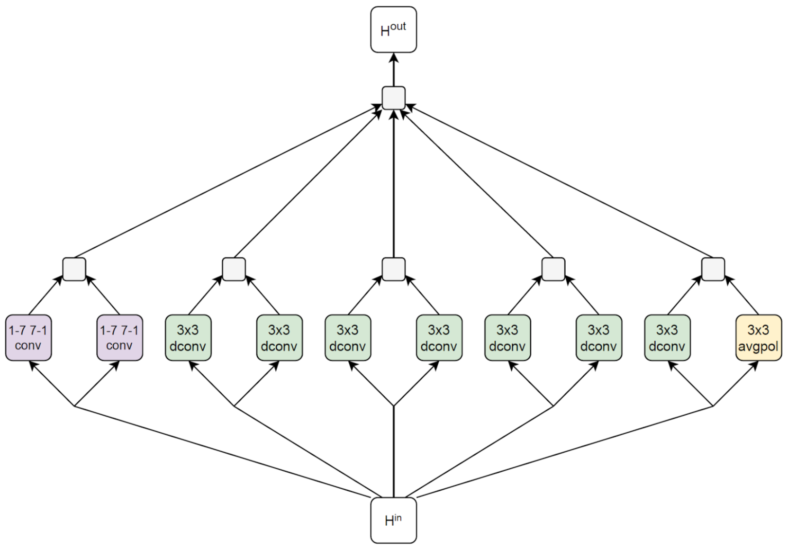


*Figure 4.38 - Schema of the best cell found by POPNAS to classify CIFAR10 (first batch)*

The architecture achieved an average accuracy of 74.33% with 38 minutes of training time, and 10 days, 10 hours and 1 minute of search time. These results, which in the first version of POPNAS are intended as an effective time-accuracy trade-off demonstration, are in line with the research hypothesis.

In the second version of POPNAS, which is under development with high performance goals, the accuracy gap with respect to PNAS is attacked and we plan to fill it almost completely while preserving the time speed up as much as possible.

Detailed results are discussed in [Lomurno2021].

### 4.8.4 *Application Design Space Exploration*

In order to validate the SPACE4AI-D tool, we consider two scenarios. The first scenario is inspired by the AI-SPRINT Maintenance and Inspection Use Case and considers an application to support drone inspections of wind farms while the second one considers the mask detection application described in WP3.

#### Wind farm use case

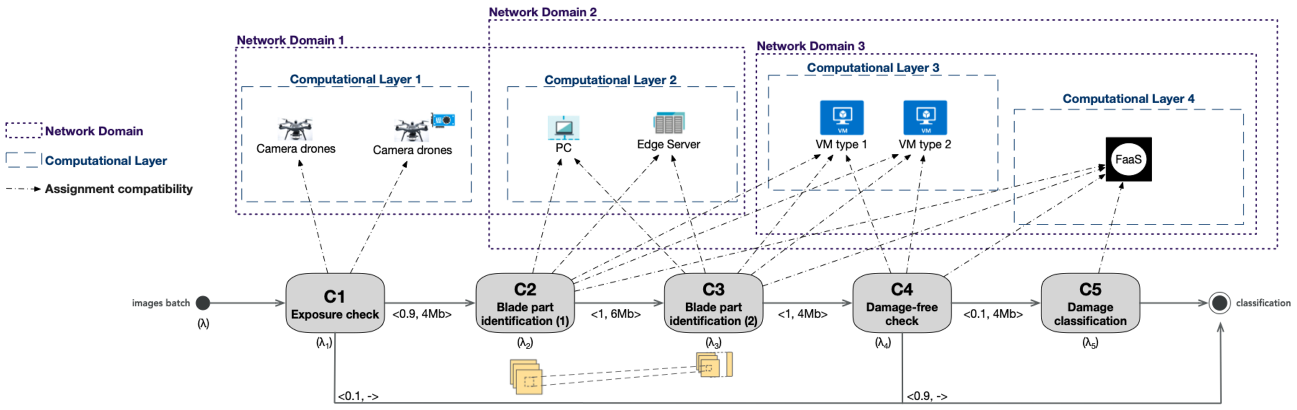


*Figure 4.39 - A use case of identifying wind turbine blade damage*

In this use case, the identification of damages in wind turbine blades is performed in the computing continuum, based on images collected by drones. The application software is characterised by multiple components, consisting of DNNs that can be deployed and executed locally (on the drone, or on operators' PCs, or on local edge servers in the operators' van) or remotely in the cloud (in a VM or through the FaaS paradigm). The set of components is illustrated in Figure 4.39. They can be deployed overall on four layers, and the dotted arrows connecting each component to the different resources denote the corresponding compatibility (i.e., the dotted arrows correspond to the ones of the compatibility matrix $A$).

As an initial step, a drone with an entry level or mid-range computation board (SPACE4AI-D determines which configuration should be bought) controlled remotely by a human operator, takes pictures of the wind turbine. These are composed of three blades, and pictures must be collected, for each blade, from four different angles to account for different types of damages; therefore, a huge amount of data is collected.

Images are processed in batches which define the incoming workload $\lambda$. Each batch is subject to an *exposure check*, $C_1$, which determines if the image quality is sufficient for further processing. If not, the component triggers the acquisition of new images. This improves the efficiency of the whole inspection process since it allows it to immediately react to the need of further data acquisition.

All well-exposed pictures are inspected by a sequence of two components ($C_2$, $C_3$) which collectively implement a complex, computer vision-based application whose goal is to monitor the inspection campaign and guarantee that this covers the complete site. They take as inputs the images processed by $C_1$ and a model of the wind farm, and, positioning the pictures on the farm itself, guide the operator to identify the next element to be examined. In particular, $C_2$ is responsible for the preliminary analysis of the images like identifying if it is a part of the blade or not, while $C_3$ identifies the parts and the position of the image elements on the blade.

These components, especially $C_3$, require a significant amount of computing and storage power. However, executing them at the edge may help in reducing the time needed to complete the maintenance and inspection process, if further data is needed to better identify the damages.

Images are then processed by two additional AI modules. $C_4$ is responsible for a *damage-free check*, i.e., of assessing whether the inspected part is damaged or not (this may possibly require the acquisition of new images). Depending on the situation, it may happen that a high percentage of the acquired pictures is clear, namely it does not show damage. Finally, $C_5$ is responsible for classifying the damage. These last steps are characterised by heavy computation requirements, therefore they are always performed in the cloud. Cloud resources are based on VMs and on the FaaS paradigm. FaaS includes different function configurations with different memory allocated and only one component can be run on each container with the specified function configuration.

The four aforementioned computational layers, namely the one involving camera drones, the one of edge resources, and those including Virtual Machines and FaaS, respectively, belong to three different network domains. In particular, drones and all edge resources communicate through a Wi-Fi network. Virtual Machines and the FaaS configurations are connected via an optical fiber network, while the information is transferred from edge to cloud resources through a 4G or 5G network.

For what concerns the candidate deployments, $C_1$ has a single one partition deployment, $C_2$, $C_4$ and $C_5$ have two deployments with one and two partitions while $C_3$ has three deployments with one, two and three partitions. The transition probabilities $p^{ik}$ and the amount of data $\delta^{ik}$ transferred between components are reported in Figure 4.39.

This scenario is characterised by both local and global QoS constraints. In particular, we prescribe that component $C_5$ must have a maximum response time of 2.5s, while we enforce that the global response time of the first four components does not exceed 2s.

## Mask Detection use case

As the second use case, we consider the WP3 Mask Detection application described in Section 2.2. The set of components is illustrated in Figure 4.40.

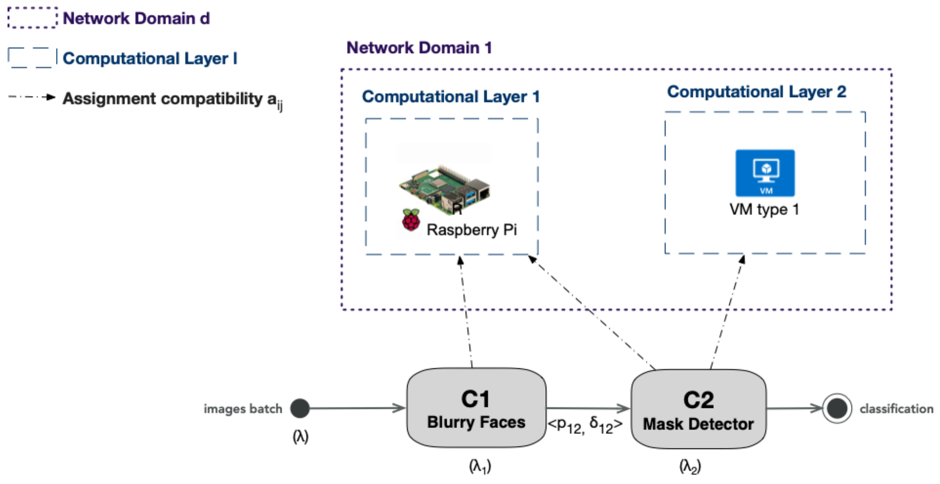


*Figure 4.40 - Mask detection use case*

In this small-scale setup, we consider a Raspberry Pi PIs 4 Model B with 4G memory and 4 computational units based on ARM64 with overall 4MB internal memory on the edge side, and a private cloud with one VM type with 8G memory and 4 computational units based on IntelX86 with 64MB internal memory on the cloud side. The application includes two components. The first one, *Blurry Faces* can run only on Raspberry Pi while the second one, *Mask Detector*, which performs the detection of the masks, can run both on Raspberry Pi and private cloud VMs. The System YAML specification and the corresponding JSON file used as the input of SPACE4AI-D tool, related to the mask detection use case, are reported and described in Appendix C.

## Experimental results under wind farm inspection use case

To evaluate the performance of the SPACE4AI-D tool under the wind farm inspection use case, we considered 5 application components, with the corresponding transition probabilities and amount of transferred data as shown in Figure 4.39. The components can be placed across four computational layers, defined as follows:

- *Edge Resources* are included in two computational layers. The first hosts a drone with an entry-level compute board (cost: 4.55 $/h), and one with a middle-level compute board (cost: 6.82 $/h). The second layer includes a PC and a GPU-based edge server (costs: 4.55 $/h and 9.1 $/h, respectively). Drones costs have been determined considering initial costs of 1000 $ and 1500 $, amortised over 2 years, and assuming that the application is executed 110 times per year. The initial costs of PC and edge server are of 1500 $ and 3000 $, respectively, amortised over 3 years (and the same number of executions on the field).
- *Cloud Resources* are all included in the third computational layer. We have considered G3 instances selected from the Amazon EC2 catalogue [AmazonPricing], powered by NVIDIA Tesla M60 GPUs equipped either with 4 vCPUs and 30.5GB of RAM (with a cost of 0.75 $/h), or with 16 vCPUs and 122 GB of RAM (with a cost of 1.14 $/h).
- *FaaS Resources* are selected from the AWS Lambda catalogue and are all included in the last computational layer. Their cost depends on the running component: the first configuration has a memory size of 4GB and an hourly cost of 0.06, 0.54, 0.16 and 0.96 $/h when used to run components from $C_2$ to $C_5$, respectively. The second configuration has a memory size of 6GB; it is used only to run component $C_5$, with a cost of 0.83 $/h. The expiration time is set to 10 minutes, as discussed in [Mahmoudi2020].

The first network domain is characterised by an access delay of 10 ms and a bandwidth of 150 Mb/s. For the second network domain, we have a fast 5G network, with bandwidth equal to 4 Gb/s and the access delay is fixed to 1 ms. The bandwidth of the third network domain has been set to 100 Gb/s, while the access delay is 0, according to the results reported in [BenchmarkingAmazon].

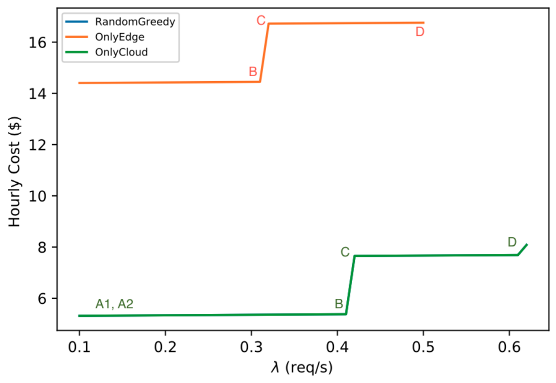
In order to evaluate SPACE4AI-D performance, two experiments have been performed. In the first experiment, we compared our proposed algorithm with an exhaustive search that has only cloud or only edge devices to limit the exploration space while in the second experiment, we compared the proposed algorithm with an open-source Python library, called *HyperOpt* [HyperOptGitHub]. In all experiments, we consider each application component (which is a Deep Neural Network (DNN)) with some candidate deployments of the DNN and each deployment includes one or more partitions as we showed in Figure 4.39. The tool should select one of the deployments and also choose a resource to run each partition of the deployment by applying a Random Greedy algorithm, in order to minimize the cost while guaranteeing response time constraints. The detailed demands of the components and partitions used in both experiments (in the comparison with the exhaustive search and HyperOpt) is reported in Appendix B (Table B.1).

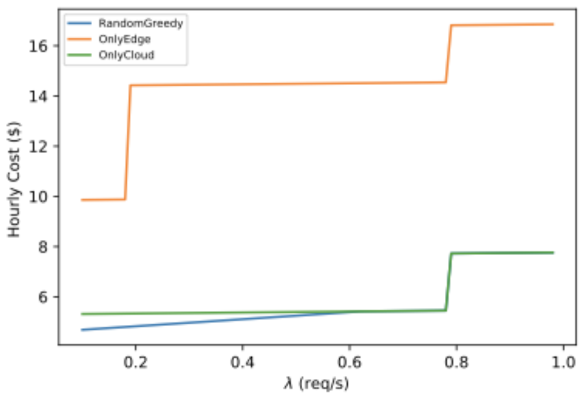
### Comparison with exhaustive search

We have varied λ between 0.1 and 1 req/s, with step 0.01, to account for different workload scenarios. For the second network domain, we have tested different settings: a 4G network, with a bandwidth of 20 Mb/s, and a slow or fast 5G network, with bandwidth equal to 2 or 4 Gb/s (the access delay is fixed to 1 ms). We have imposed a local constraint on component $C_5$, that must have a response time below 2.5 s, and a global constraint on path $P_1 = \{C_1, C_2, C_3, C_4\}$, corresponding to a maximum response time of 3 s. Finally, the proposed greedy algorithm performs 1000 iterations (*MaxIter*= 1000).

The proposed solution (mentioned by RandomGreedy label in the figure, in which the components are free to place on edge, cloud and FaaS, according to the compatibility matrix) is compared with the results of an exhaustive search, where (to limit the execution time) some components are restricted to run on edge or on the remote cloud. In particular, in the *OnlyCloud* scenario, components $C_2$ and $C_3$ can run only on the cloud VMs, while they can run only on the edge in the *OnlyEdge* scenario.

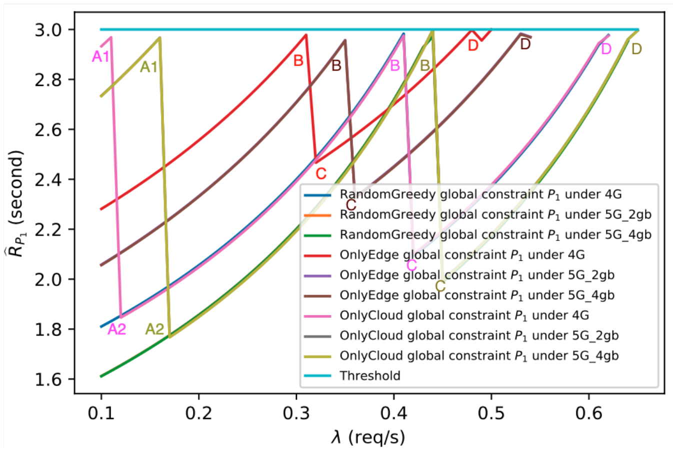
The cost analysis, shown in Figure 4.41(a) and Figure 4.41(b), is related to 5G@4Gb/s and it is slightly translated under other network settings. In all scenarios, when the incoming load is low, the best solutions use the entry-level compute board drone, Asus Zenbook 13 UX325EA and the G3 4 vCPUs VM, which are slower and cheaper than the alternative configurations. Then, as λ increases, the response time along the path $P_1$ gets closer to the global constraint threshold incurring in a QoS violation (see Figure 4.41(c)). Therefore, the solution selects the G3 VM equipped with 16 vCPUs (and Microsoft Surface Studio 2 LAM-00005 for OnlyEdge scenario), which reduces the response time (points B in Figure 4.41(c)). If λ is further increased, the best solution steps back to G3 equipped with 4 vCPUs but selects the drone with the mid-range compute board (points C in Figure 4.41(c)).





(a): Cost variation of best solutions with increasing λ in 5G@4Gb/s scenario.

(b): Cost variation of best solutions with increasing λ for the constraints equal to 100s and 5G@4Gb/s scenario.



(c): Response time variation of path $P_1$ with increasing λ in three network scenarios. Performance overlap under the two 5G settings.

*Figure 4.41 - Experimental result for the comparison with exhaustive search*

Finally, the best solution adopts the faster and more expensive 16 vCPUs VM (and Microsoft Surface Studio 2 LAM-00005 for OnlyEdge scenario) (points D in Figure 4.41(c)). When λ is about 0.6 req/s in the 4G scenario or 0.64 in the 5G scenarios both in RandomGreedy and OnlyCloud (and about 0.48 req/s in the 4G scenario or 0.5 in the 5G scenario in OnlyEdge), the response time of the path $P_1$ meets the threshold, and no feasible solution can be found. As it can be noticed in Figure 4.41(a) and Figure 4.41(c), our random greedy algorithm and *OnlyCloud* identify the same solutions. The only difference between random greedy and OnlyCloud happens when λ is low (points A1 in Figure 4.41(c)), where they select different deployments but the same devices (so the costs are the same (points A1 and A2 in Figure 4.41(a))). When λ increases (points A2 in Figure 4.41(c)), OnlyCloud gets close to the violation and selects the same deployment as random greedy. Indeed, according to components demands, resource costs and constraints, the *OnlyCloud* solution is the optimal solution because cloud VMs are both faster and cheaper than the available edge devices. This demonstrates that our algorithm converges to the global optimal solution. Vice versa, the *OnlyEdge* solution is expensive and slow and it is not even feasible for λ from 0.15 and 0.26 req/s in 4G and 5G scenarios, respectively (see Figure 4.41(a) and Figure 4.41(c)).

Note that, component $C_5$, which can run only on FaaS, is deployed on the high-end function configurations with 6GB of memory. This seems counter-intuitive because such configuration has a higher time-unit cost, but this deployment is proved to be more convenient, since the execution time of the component is decreased (and so the overall cost).

Costs increase linearly when λ ranges between 0.1 and 0.41 req/s, and between 0.42 and 0.62 req/s for the calls to AWS Lambda functions. Finally, in Figure 4.41(b), we increase the maximum response time of both local and global constraints to 100s in order to analyse how the cost changes under relaxed QoS requirements. In this scenario, the random greedy solution is equal or cheaper than the *OnlyCloud* by λ=0.62 req/s since it has more degrees of freedom and it can execute components $C_2$ , $C_3$ and $C_4$ on the FaaS configurations, which are slower but cheaper than VMs. But from λ=0.63 to 1 req/s, the cost of FaaS will be greater than clouds because of increasing the load, hence the solution selects the cloud VM and the cost will be the same as the OnlyCloud scenario.

## Comparison with HyperOpt

In this experiment, we compare the cost and performance of the Random Greedy algorithm with the solution obtained by *HyperOpt* [HyperOptGitHub], an open-source Python library. HyperOpt is based on Bayesian optimization algorithms over awkward search spaces, which may include real-valued, discrete, and conditional dimensions. To make the Random Greedy performance analysis more robust, we consider also a *Hybrid Method* that exploits HyperOpt to improve an initial result provided by the Random Greedy algorithm (hence the Hybrid Method always obtains results at least as good as the Random Greedy). To quantitatively evaluate the different approaches, we define the percentage gain (denoted as *Cost ratio*) as follows:

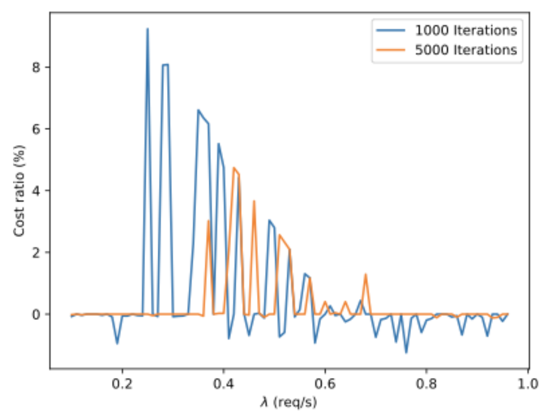
$$Cost\ ratio = \left(\frac{OtherMethodCost - RandomGreedyCost}{RandomGreedyCost}\right) \times 100$$

Where *OtherMethodCost* can denote the cost of HyperOpt or the Hybrid Method. We performed each experiment twice with *MaxIter* = 1000 and *MaxIter* = 5000. Note that, in each setting HyperOpt and the Hybrid Method always perform the same number of iterations of the Random Greedy.

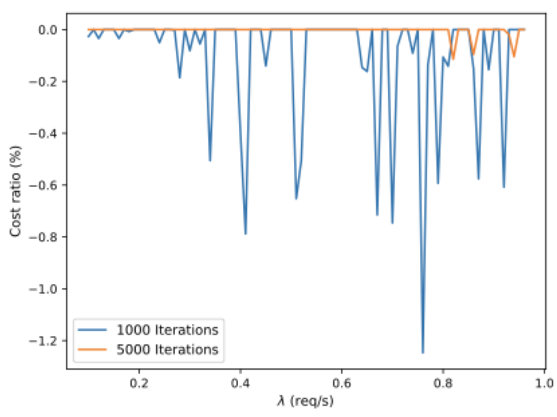
Note that the demanding time table is the same as the one used for comparison with exhaustive search (Appendix D). We run the experiments for both light constraints (local and global constraints are equal to 20s) and strict constraints (the local constraint is equal to 2.4s and the global constraint is equal to 2.9s), with λ ranging in [0.1, 1] req/s with step 0.01 req/s. The methods cannot find any feasible solution for λ > 0.63 req/s in the strict constraints scenario and λ > 0.98 req/s in the light constraints scenario.

The cost comparison under light constraints is reported in Figure 4.42(a) and Figure 4.42(b), while Figure 4.42(c) shows the average running time of each method varying the number of iterations. The same is reported for the strict constraints setting in Figure 4.42(d), Figure 4.42(e) and Figure 4.42(f), respectively.

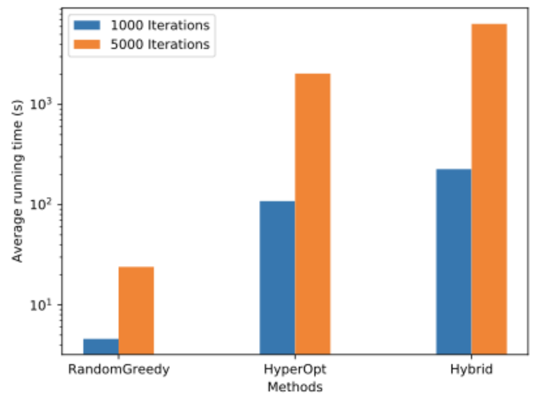
In order to fairly compare the performance, we run both Random Greedy and HyperOpt on one single core. However, note that SPACE4AI-D is based on a multi-threaded implementation that can leverage multi-cores. Unfortunately, HyperOpt is not able to exploit multiprocessing because it is a Bayesian sequential model-based optimizer. It can benefit from multi-cores only if run within Spark but in our single machine 40-cores setting it was not possible to improve its performance.



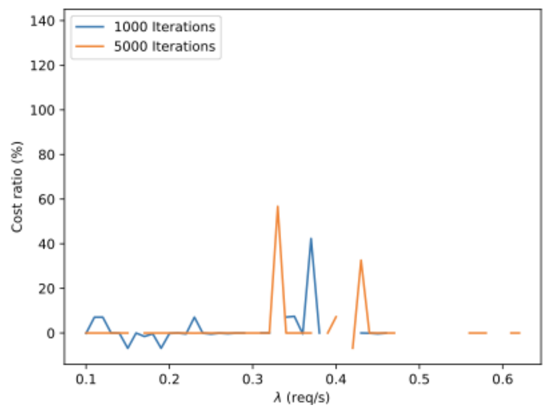
(a): Cost ratio of random greedy with respect to HyperOpt for light constraints.



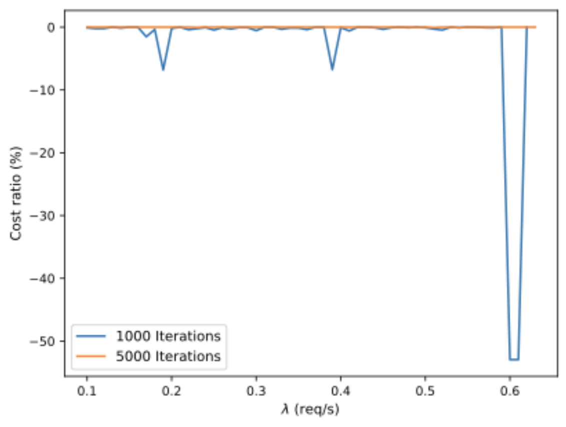
(b): Cost ratio of Random Greedy with respect to the Hybrid Method for light constraints.



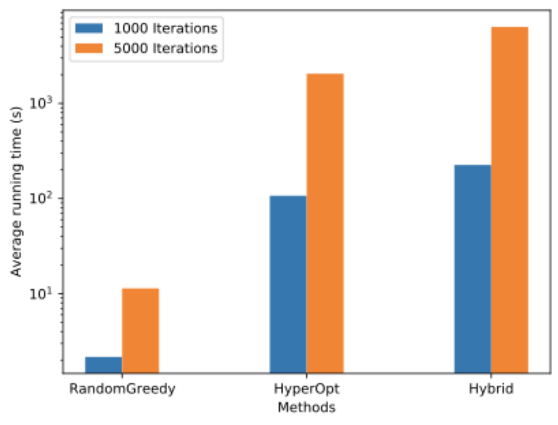
(c): Average running time of methods for light constraints.



(d): Cost ratio of random greedy with respect to HyperOpt for strict constraints.



(e): Cost ratio of Random Greedy with respect to the Hybrid Method for strict constraints.



(f): Average running time of methods for strict constraints.

*Figure 4.42 - Experimental result for the comparison with HyperOpt*

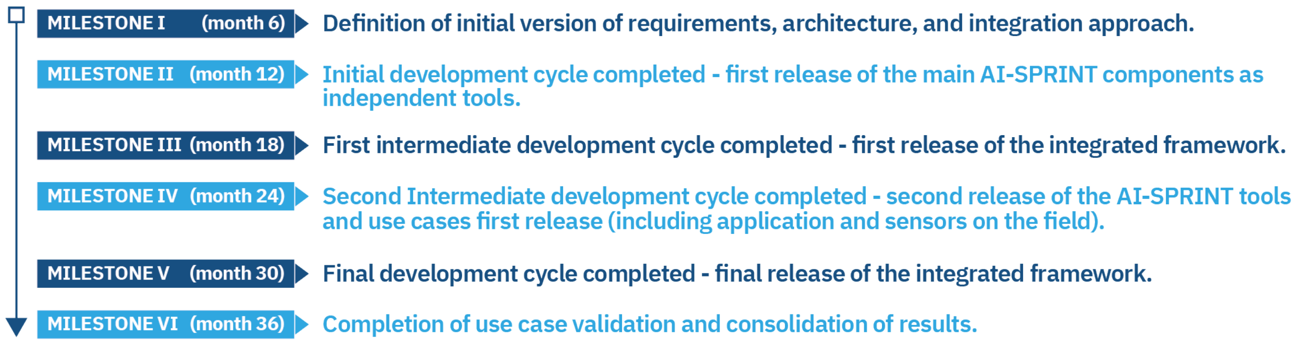
As it can be observed in Figure 4.42(a), Random Greedy costs are up to 10% lower than those obtained by HyperOpt, while losses are infrequent about up to 3% only for the 1,000 iterations scenario. The Hybrid Method (Figure 4.42(b)) improves Random Greedy less than 0.2% only in a few points for 5,000 iterations, and it obtains small gains (lower than 1.5%) at 1000 iterations. Under strict constraints, Random Greedy method always finds a feasible solution for $\lambda \leq 0.63$ req/s, this does not hold for HyperOpt which, even when running for 5000 iterations, it cannot find any feasible solution for about 20% of scenarios (disconnected points in Figure 4.42(d)). However, it can be noticed that 5000 iterations are barely enough for Random Greedy, since in one case it loses about 6% against HyperOpt (Figure 4.42(d)) and the Hybrid Method (Figure 4.42(e)).

Note that, as reported in Figure 4.42(c) and Figure 4.42(f), even if we run Random Greedy for 5,000 iterations, the required time is at least one order of magnitude lower than the other methods running for 1,000 iterations.

The scalability analysis result is presented in Appendix D. The data set relative to all the wind farm use case analyses is available at https://doi.org/10.5281/zenodo.5789377.

# 5. Towards the Integrated framework for the Design Time Tools

The integration plan of AI-SPRINT aims at harmonising the definition of the requirements with the design and development phase and ensuring that scientific and technical activities comply with the use cases definition. In WP1, a specific task is devoted to these integration activities and has also a design side, though it is confined at the level of architectural design. Figure 5.1 depicts the validation strategy of AI-SPRINT through milestones defined in the work plan of the project.



**MILESTONE I (month 6)** Definition of initial version of requirements, architecture, and integration approach.

**MILESTONE II (month 12)** Initial development cycle completed - first release of the main AI-SPRINT components as independent tools.

**MILESTONE III (month 18)** First intermediate development cycle completed - first release of the integrated framework.

**MILESTONE IV (month 24)** Second Intermediate development cycle completed - second release of the AI-SPRINT tools and use cases first release (including application and sensors on the field).

**MILESTONE V (month 30)** Final development cycle completed - final release of the integrated framework.

**MILESTONE VI (month 36)** Completion of use case validation and consolidation of results.

*Figure 5.1 - Milestones for components development*

This deliverable realises the Milestone II which releases the first version of the individual components, according to the development plan described in *AI-SPRINT Deliverable D1.2 Requirements Analysis*. In particular, this document describes the:

- Development of the first release of the programming models including the definition of quality annotations to specify security policy and performance constraints.
- Initial release of the Performance modelling tools (profiling of inference on edge devices and training on GPU clusters). Initial evaluation of the accuracy of performance models on benchmarking applications.

At M18 we will release the integrated framework which will include the integration of WP2 components with the runtime environment, in particular it will report on the initial integration of design abstractions and quality annotations with the runtime environment and on the mapping of the quality annotations related to performance constraints to monitoring rules.

# 6. Conclusions

This document has provided the description of the activities for the development of the components that build the first release of the design tools of the AI-SPRINT platform, allowing to reach the Milestone II at M12.

The document describes in detail the functionalities of each component and the activities to enhance the software to enable the integration of the components and to fulfil the use cases requirements. These activities have been also verified through the presentation of example applications that will be part of the use cases implementations.

The second version of this deliverable will be provided at M24 in *D2.3 Second release and evaluation of the AI-SPRINT design tools*. Another deliverable, *D2.2 Initial AI-SPRINT design and runtime tools integration* at M18 will provide further details on the integration of the design tools components with the runtime environment and with the security tools. The final release will be provided at M30 in *D2.4 - Final release and evaluation of the AI-SPRINT design tools* and in *D2.5Final AI-SPRINT design and runtime tools integration*

# Appendix A - Samples description

This appendix presents the number of samples that have been used to train each of the models presented in Section 4.8.2. Table A.1 reports the data about the hold-out scenario, while the remaining tables details the size of the training dataset for extrapolation experiments. For example, Table A.2 shows how the number of samples of the training set of the extrapolation experiment on the batch AlexNet implemented on PyTorch considering 1 and 2 P600 is 72 and 204 respectively. The number of samples in training data used in GPU number extrapolation models, computational power extrapolation models and Network depth extrapolation models are reported in Table A.3, Table A.4 and Table A.5, respectively.

| Network | Framework | Training set size |
|---|---|---|
| AlexNet | PyTorch | 1147 |
| | TensorFlow | 1147 |
| ResNet-50 | PyTorch | 1152 |
| | TensorFlow | 1152 |
| VGG-19 | PyTorch | 1152 |
| | TensorFlow | 1152 |

*Table A.1 - Number of samples in training data used in interpolation models*

| Network | Framework | GPU Type and Number | | | | | |
|---|---|---|---|---|---|---|---|
| | | P600 | | K80 | | | |
| | | 1 | 2 | 1 | 2 | 3 | 4 |
| AlexNet | PyTorch | 72 | 204 | 24 | 20 | 24 | 24 |
| | TensorFlow | 48 | 48 | 24 | 24 | 16 | 24 |
| ResNet-50 | PyTorch | 28 | 44 | 24 | 24 | 24 | 24 |
| | TensorFlow | 28 | 44 | 48 | 24 | 24 | 16 |
| VGG-19 | PyTorch | - | - | 16 | 64 | 48 | 24 |
| | TensorFlow | - | - | 16 | 32 | 24 | 24 |

*Table A.2 - Number of samples in training data used in batch size extrapolation models*

| Network | Framework | GPU Type and Number | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | M60 | | | | GTX 1080Ti | | | |
| | | 1 | 2 | 3 | 4 | 1 | 2 | 4 | 8 |
| AlexNet | PyTorch | 39 | 21 | 36 | 21 | 11 | 16 | 17 | 47 |
| | TensorFlow | 18 | 36 | 18 | 18 | 12 | 12 | 12 | 12 |
| ResNet-50 | PyTorch | 12 | 24 | 48 | 72 | 24 | 24 | 24 | 24 |
| | TensorFlow | 48 | 72 | 48 | 72 | 12 | 12 | 12 | 12 |
| VGG-19 | PyTorch | 20 | 32 | 24 | 29 | 24 | 24 | 24 | 24 |
| | TensorFlow | 20 | 30 | 24 | 28 | 12 | 12 | 12 | 12 |

| Network | Framework | GPU Type | | |
|---|---|---|---|---|
| | | K800 | M60 | GTX 1080Ti |
| AlexNet | PyTorch | 384 | 543 | 947 |
| | TensorFlow | 288 | 744 | 231 |
| ResNet-50 | PyTorch | 192 | 96 | 96 |
| | TensorFlow | 312 | 768 | 144 |
| VGG-19 | PyTorch | 216 | 144 | 84 |
| | TensorFlow | 288 | 288 | 96 |

*Table A.3 - Number of samples in training data used in GPU number extrapolation models*

| Network | Framework | Training set size |
|---|---|---|
| AlexNet | PyTorch | 58 |
| | TensorFlow | 52 |
| ResNet-50 | PyTorch | 65 |
| | TensorFlow | 52 |

*Table A.4 - Number of samples in training data used in computational power extrapolation models*

| Network | Framework | Max N | GPU Type and Number M60 | | |
|---|---|---|---|---|---|
| | | | 1 | 2 | 4 |
| ResNet | PyTorch | 4 | 680 | 131 | 181 |
| ResNet | TensorFlow | 5 | 935 | 182 | 233 |
| ResNet | PyTorch | 6 | 1200 | 240 | 289 |
| ResNet | TensorFlow | 8 | 1757 | 360 | 407 |

*Table A.5 - Number of samples in training data used in Network depth extrapolation models*

# Appendix B - SPACE4AI-D Optimization, implementation and scalability

**SPACE4AI-D Optimization Problem Formulation and Random Greedy Algorithm implementation**

In this section, we provide an overview of how we modelled the applications component placement and resource selection problem on heterogeneous edge and cloud resources within the SPACE4AI-D tool and its corresponding randomised greedy solution.

We developed a Mixed Integer Non-Linear Programming (MINLP) optimization formulation, aiming at minimizing the deployment cost at design time, while satisfying local and global QoS requirements. For space limits, we focus here only on the objective function of our model, while the complete formulation is available as an unpublished technical report at [TechnicalReport].

The main goal of SPACE4AI-D is to determine which kind of resource we should select at each computational layer (that includes which hardware to buy on the edge or, e.g., which type/flavour of VM to use in the cloud), whether each component's partition(s) should be deployed on the given resource, and, in this case, if the assignment is compatible with: 1) memory constraints, used to determine the maximum number of components' partitions that can be co-located in each device, and 2) QoS requirements.

We denote by $J$ the set of all resources in the computing continuum. To define the assignment decisions, namely, to characterise which resources we are selecting at each computational layer and how components are assigned to the available devices, we introduce the following variables:

- $y_{hj}^{i}$, which, for all $i \in I$, for all $c_s^i \in C^i$ and for each $h \in H_s^i$ and $j \in J$, is equal to 1 if partition $\pi_h^i$is deployed on device $j$,
- $x_j$ , which is 1 if device $j \in J$, is used in the final deployment,
- $\hat{y}_{hj}^{i}$, which, for cloud resources, denotes the number of VMs of type $j$ assigned to any partition $\pi_h^i$.

Edge devices are characterised by the estimated amortised costs for the single run of the target application, cloud VMs are characterised by hourly costs, and FaaS configurations costs are expressed in GB-second. In order to compute them, we denote with $T$the overall time an application is active for a single run and we assume that $T$ is equal or less than one hour.

If we denote by $J_{\mathcal{E}}$the subset of $J$ storing all the available edge devices, the corresponding execution cost can be defined as follows:

$$C_E = \sum_{j \in J_{\mathcal{E}}} c_j^E x_j$$

where $c_j^E$ is the amortised cost of the edge device $j \in J_{\mathcal{E}}$ .

The total execution cost on cloud VMs, whose set will be denoted by $J_c \subset J$ , can instead be computed as:

$$C_C = \sum_{j \in J_c} c_j^C \underline{y_j}$$

Where $c_j^C$ and $\underline{y_j} = max_{(i,h)} \hat{y}_{hj}^{i}$ denote the hourly cost and the maximum number of running VMs of type $j \in J_c$, respectively.

Finally, $J_F$ denote the set of all FaaS configurations, and let $c_{hj}^{F,i}$ be the GB-second unit cost for executing $\pi_h^i$on the function configuration $j \in J_F$. FaaS total costs depend on the memory size, the functions

duration, and the total number of invocations. The execution cost of the function layer will be as follows:

$$C_F = \sum_{i \in I} \sum_{s: c_s^i \in C^i} \sum_{h \in H_s^i} \sum_{j \in J_F} c_{hj}^{F,i} d_{hj}^{i,hot} y_{hj}^{i} \lambda_h^i T$$

where $d_{hj}^{i,hot}$denotes the execution time of a hot request of partition $\pi_h^i$on function configuration$j \in J_F$.

Note that the cost of the used memory is embedded in $c_{hj}^{F,i}$. Indeed, $d_{hj}^{i,hot}$ is inversely proportional to the memory $m_h^i$ allocated to the partition $\pi_h^i$(see [Lin2021]).

According to some FaaS providers (see, e.g., *AWS Step Functions* [AWSStepFunctionsPricing] and *Azure Logic Apps* [AzureLogicApps]), we need to introduce a *state transition cost*, denoted here with $c^T$, to model the additional charge for the message passing and coordination between two successive functions. If, however, the orchestration is supported by an architectural component (see, e.g., *SCAR* and *OSCAR* [Risco2021]), the state transition cost is set to $c^T = 0$. Without loss of generality, we can thus formulate the transition cost as:

$$C_T = \sum_{i \in I} \sum_{s: c_s^i \in C^i} \sum_{h \in H_s^i} \sum_{j \in J_F} c^T y_{hj}^{i} \lambda_h^i T$$

Therefore, the objective function of our problem, which corresponds to the minimization of all these operational costs, is as follows:

$$min \;\; C_E + C_C + C_F + C_T$$

subject to assignment compatibility, memory and QoS constraints and to the selection of a single device at each layer.

Due to the M/G/1 models that we used, the problem becomes a NP-hard MINLP problem. In the following, we describe the heuristic algorithm, based on a randomised greedy method, we developed to solve it (see Figure B.1).

The algorithm receives as input the compatibility matrix $A$, the application DAG description with the performance demand, candidate device costs, local and global constraints, and the maximum number of iterations to be performed. First, we initialize the best solution and corresponding cost to infinity. At each iteration, we set matrices $x$, $y$ and $\hat{y}$ to zero (line 4), we randomly pick a device at each layer (line 5) and a deployment for each component, and we randomly assign each partition of the selected deployment to the selected devices according to the compatibility matrix $A$ (lines 6-11). For each VM type $j$, we randomly chose the number of nodes between 1 and $n_j$, i.e., the maximum number of instances (line 12). This generates a solution $< x, y, \hat{y} >$ that satisfies the compatibility constraints. Then, we check the feasibility of memory constraints (line 13), and QoS constraints (line 14). If possible, we reduce the maximum number of selected VMs (line 15), preserving the feasibility of the current solution. At line 18, we check if the current solution improves the *BestSolution*, which is updated accordingly (lines 19-21). The best solution found, if any, is returned at lines 23-27.

**Algorithm 1** Random greedy algorithm

```
 1: Input: $\mathcal{I}$, $\mathcal{H}$, $\mathcal{J}$, DAG, $\mathbf{A}$, components demands, QoS constraints, system
    costs, MaxIter
 2: Initialization: $BestSolution \leftarrow \emptyset$, $BestCost \leftarrow \infty$
 3: for $m = 1, \ldots,$ MaxIter do
 4:     $\mathbf{x} \leftarrow [0], \mathbf{y} \leftarrow [0], \widehat{\mathbf{y}} \leftarrow [0]$
 5:     Randomly pick a node $j$ at each layer; set $x_j \leftarrow 1$
 6:     for $i \in \mathcal{I}$ do
 7:         Randomly pick a deployment $c_s^i \in \mathcal{C}^i$ of component $i$
 8:         for $h \in \mathcal{H}_s^i$ do
 9:             Randomly pick $j$ s.t. $x_j = 1$ and $a_{hj}^i = 1$; set $y_{hj}^i \leftarrow 1$
10:         end for
11:     end for
12:     $\widehat{y}_{hj}^i \leftarrow random[1, n_j] \cdot y_{hj}^i \;\; \forall i \in \mathcal{I}, \; \forall h \in \mathcal{H}_s^i, \; \forall$ VM $j$
13:     if memory constraints are fulfilled then
14:         if local and global constraints are fulfilled then
15:             ReduceVMClusterSize($j$) for each VM $j$ s.t. $x_j = 1$
16:         end if
17:     end if
18:     if $\langle \mathbf{x}, \mathbf{y}, \widehat{\mathbf{y}} \rangle$ is feasible and $cost(\langle \mathbf{x}, \mathbf{y}, \widehat{\mathbf{y}} \rangle) < BestCost$ then
19:         $BestSolution \leftarrow \langle \mathbf{x}, \mathbf{y}, \widehat{\mathbf{y}} \rangle$
20:         $BestCost \leftarrow cost(\langle \mathbf{x}, \mathbf{y}, \widehat{\mathbf{y}} \rangle)$
21:     end if
22: end for
23: if $BestSolution \neq \emptyset$ then
24:     return $BestSolution$
25: else
26:     No feasible solution found
27: end if
```

*Figure B.1 - Randomized greedy algorithm implemented in SPACE4AI-D*

The detailed demands of the components and partitions on the compatible resources used in the experiments, is reported in Table B.1.

| Resource Type | Demands[s] | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $C_1$ | $C_2$ | | | $C_3$ | | | | | | $C_4$ | | | $C_5$ | | |
| | $h_1$ | $h_1$ | $h_2$ | $h_3$ | $h_1$ | $h_2$ | $h_3$ | $h_4$ | $h_5$ | $h_6$ | $h_1$ | $h_2$ | $h_3$ | $h_1$ | $h_2$ | $h_3$ |
| Drone (Low-end Board) | 1.260 | 1.200 | 0.700 | 0.650 | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ |
| Drone (Mid-range Board) | 1.000 | 0.790 | 0.420 | 0.410 | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ |
| PC | _ | 1.000 | 0.550 | 0.530 | 5.000 | 2.800 | 2.700 | 2.000 | 1.900 | 1.800 | _ | _ | _ | _ | _ | _ |
| Edge Server | _ | 0.050 | 0.030 | 0.028 | 0.500 | 0.270 | 0.260 | 0.200 | 0.190 | 0.180 | _ | _ | _ | _ | _ | _ |
| G3 - 4vCPUs VM | _ | 0.012 | 0.007 | 0.006 | 0.112 | 0.560 | 0.550 | 0.040 | 0.040 | 0.040 | 0.033 | 0.021 | 0.020 | _ | _ | _ |
| G3 - 16vCPUs VM | _ | 0.010 | 0.006 | 0.005 | 0.090 | 0.050 | 0.045 | 0.032 | 0.031 | 0.030 | 0.027 | 0.017 | 0.013 | _ | _ | _ |
| FaaS - 4Gb (warm) | _ | 0.25 | 0.200 | 0.190 | 2.250 | 1.500 | 1.250 | 1.100 | 1.000 | 1.000 | 0.675 | 0.400 | 0.370 | 4.000 | 2.200 | 2.000 |
| FaaS - 4Gb (cold) | _ | 0.400 | 0.300 | 0.290 | 2.700 | 2.000 | 2.000 | 2.000 | 2.000 | 2.000 | 1.000 | 0.800 | 0.800 | 4.800 | 2.800 | 2.800 |
| FaaS - 6Gb (warm) | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | 2.300 | 1.300 | 1.100 |
| FaaS - 6Gb (cold) | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | _ | 2.760 | 2.000 | 2.000 |

*Table B.1 - Demands of components and partitions on the compatible resources*

## Scalability Analysis

To evaluate the scalability of our approach, we considered four different scenarios at different scales as reported in Table B.2. In the following, we report the average achieved by considering 10 random instances for each scenario. We randomly selected between 1 and 3 deployments for each component and between 1 and 4 partitions for each deployment. We also selected randomly, between 3 and 5, the maximum number of VMs of each type, while service demands were generated randomly in the range of [1, 2]s for drones, [1, 5]s for edge resources, [0.5, 2]s for VMs (as in [Elgamal2018]), and [2, 5]s for cold and warm FaaS requests (as in [Manner2018]). The iteration running time depends on the solution feasibility. For example in the largest case (15 components), the total time of an iteration that found a feasible solution is about 366 ms, where 2% of the total time is spent in creating the new solution and 98% in checking its feasibility, while the total time of an iteration that could not find a feasible solution is about 5 ms, where only 26% of the total time is spent in checking the feasibility of the solution (the iteration terminates as soon as any constraint is violated). Hence, in the following scalability analysis, in order to consider the worst case in terms of average running time, we set the local and global constraint thresholds to random larger numbers in the intervals shown in Table B.2 for all instances of the scenarios. Indeed, since each iteration terminates as soon as any constraint is violated, strict thresholds imply more frequent violations and also entail lower running times on average.

| Scenario | #Components | Local constraints | Global constraints |
|---|---|---|---|
| 1 | 5 | [120, 150] | [200, 250] |
| 2 | 7 | [150, 200] | [350, 400] |
| 3 | 10 | [300, 350] | [350, 400] |
| 4 | 15 | [350, 400] | [450, 500] |

*Table B.2 - The range of constraints in scalability analysis*

As it is shown in Table B.3(a), we considered problem instances including up to 15 components, 31 candidate nodes, 4 local and 4 global constraints. We set λ = 0.11 req/s and we replicated the experiment twice for *MaxIter* = 500 and *MaxIter* = 5000. The average execution time and feasibility percentage across 10 instances is reported in Table B.3(b).

Note that the maximum execution times (in the worst case as we said above) for the Random Greedy are about 19s and 4 min with 500 and 5000 iterations, while HyperOpt takes about 3.2 min and 3 hours, respectively. Moreover, HyperOpt cannot find a feasible solution for all instances even for very slight constraints (feasibility percentage about 97% for 500 iterations) while as we saw in the use case analysis, the feasibility percentage is about 60% for 5000 iterations.

Our approach can thus obtain significant improvements (with a factor of 51 and 46 for small and large scale, respectively) compared with HyperOpt in terms of average running time, which makes it suitable to tackle the component placement problem at design time.

(a): Scalability parameters

| Scenario | #Components | #Nodes in Computational Layers (CL) | | | | | | | | #Local, global constraints |
|---|---|---|---|---|---|---|---|---|---|---|
| | | $CL_1$ | $CL_2$ | $CL_3$ | $CL_4$ | $CL_5$ | $CL_6$ | $CL_7$ | $CL_8$ | |
| 1 | 5 | Drone:2 | Edge: 4 | VMs: 4 | VMs: 4 | FaaS: 2 | - | - | - | 1, 1 |
| 2 | 7 | Drone:2 | Edge: 4 | Edge: 4 | VMs: 4 | VMs: 4 | FaaS: 5 | - | - | 2,2 |
| 3 | 10 | Drone:2 | Edge: 4 | Edge: 4 | VMs: 4 | VMs: 4 | VMs: 4 | FaaS: 5 | - | 3,3 |
| 4 | 15 | Drone:2 | Edge: 4 | Edge: 4 | VMs: 4 | VMs: 4 | VMs: 4 | VMs: 4 | FaaS: 5 | 4,4 |

(b): Scalability performance evaluation

| Scenario | Avg. Exec. time for 500 iterations (s), feasibility percentage | | | Avg. Exec. time for 5000 iterations (s), feasibility percentage | | |
|---|---|---|---|---|---|---|
| | *RandomGreedy* | *HyperOpt* | *HybridMethod* | *RandomGreedy* | *HyperOpt* | *HybridMethod* |
| 1 | 6.8, 100 | 107.4, 100 | 132.4, 100 | 69.2, 100 | 3584.9, 100 | 11258.0, 100 |
| 2 | 19.5, 100 | 114.0, 100 | 139.1, 100 | 200.9, 100 | 5279.3, 100 | 14071.1, 100 |
| 3 | 21.9, 100 | 148.9, 100 | 157.8, 100 | 239.1, 100 | 7205.0, 100 | 15767.4, 100 |
| 4 | 19.3, 100 | 192.0, 90 | 193.9, 100 | 241.4, 100 | 11057.8, 100 | 16602.1, 100 |

*Table B.3 - Scalability analysis*

# Appendix C - YAML files description

Here, we report the initial YAML file describing the mask detection use case (Section 2.2) that will be processed by the SPACE4AI-D tool. A parser will convert the YAML file to the input json file required by SPACE4AI-D.
Our system is described, in this file, in two main sections. The first section is denoted as NetworkDomains and it defines several network domains with different network communication properties connecting devices with each other. Resource computational layers are included in, possibly, multiple network domains, associated with a given technology characterised by the access time (AccessDelay) and bandwidth (Bandwidth). Indeed, each computational layer (an item in ComputationalLayers) includes a list of candidate resources (Resources) such that one of them can be selected to run the application components. Each candidate resource is characterised by some properties, such as name, description, a list of processors (processors), etc.
The second section defines a list of containers (Containers). Each partition (or component) runs on the available resources as a container characterised by some properties like memorySize, computingUnits, and so on. The candidateExecutionLayers field shows the computational layers which include the compatible resources for each container. In the mask detection use case, we defined two containers c1 and c2.
The unit of measurement for some parameters such as storage, memory, cost, time (delay) and bandwidth are Giga Byte, Mega Byte, dollar per hour ($/h), hour and Megabits per second (Mbps), respectively, in the YAML file.

**YAML input file**

```
System:
  name: Mask Detection Application
  NetworkDomains:
    ND1:
      name: Network Domain 1
      AccessDelay: 0.00000277
      Bandwidth: 10000
      ComputationalLayers:
        - computationalLayer1:
            name: Edge Layer
            number: 1
            Resources:
              - resource1:
                  name: RaspPi
                  description: Raspberry PIs 4 Model B
                  cost: 0.0375
                  memorySize: 4096
                  operatingSystemDistribution: Raspbian
                  operatingSystemType: Linux
                  operatingSystemVersion: 10
                  secureBoot: string
                  measureBoot: string
                  storageSize: 64GB
                  storageType: flash
                  processors:
                    - processor1:
                        name: BCM2711
                        type: Cortex-A72
                        architecture: ARM64
                        computingUnits: 4
                        internalMemory: 4
                        SGXFlag: False
        - computationalLayer2:
            name: Cloud Layer
            number: 2
            Resources:
              - resource1:
                  name: VM1
                  description: AWS g3s.xlarge
                  cost: 0.75
                  memorySize: 8192
                  storageSize: 44
                  storageType: SSD
                  operatingSystemDistribution: Ubuntu
                  operatingSystemType: Linux
                  operatingSystemVersion: 20.04
                  secureBoot: False
                  measureBoot: False
                  onSpot: False
                  processors:
                    - processor1:
                        name: Xeon
                        type: SkyLake
                        architecture: IntelX86
                        computingUnits: 4
                        internalMemory: 64
                        SGXFlag: False

Containers:
  - container1:
      name: c1
      memorySize: 1024
      computingUnits: 1
      trustedExecution: False
      networkProtection: False
      fileSystemProtection: False
      GPURequirement: False
      candidateExecutionLayers: [1]
  - container2:
      name: c2
      memorySize: 2048
      computingUnits: 1
      trustedExecution: False
      networkProtection: False
      fileSystemProtection: False
      GPURequirement: False
      candidateExecutionLayers: [1, 2]
```

**JSON file description**

As already mentioned, the SPACE4AI-D tool requires an input file in JSON format. This will contain some information provided in the YAML file (such as the resources and system descriptions), which will be converted into the JSON format by a suitable parser. However, additional information is required, e.g., the application DAG, etc., and these should be provided by the Application Architect. In the following, we provide and describe a sample JSON file, where all data that are not provided within the YAML file and need to be added are in bold.

Each partition sends a certain amount of data (denoted by data_size in the JSON file) to its successor in the application DAG. This, denoted by next in the JSON file. The early_exit_probability field denotes the probability of early stopping described in Section 4.7.1.

We define the CompatibilityMatrix as a matrix that shows which devices can be used to run each partition. It can be derived by the field candidateExecutionLayers from the YAML file. For example "c1":{ "h1":["RasPi"] means that container c1 can run on computationalLayer1, which includes RasPi. Vice versa "c2":{ "h1":["RasPi","VM1"] } indicates that the mask detection container can run both on computationalLayer1 and computationalLayer2. supported by the RasPi and VM1, respectively. Moreover, the DemandMatrix field represents the demanding time to run a partition on the compatible resources; this needs to be provided by the Application Architect after performing the application profiling.

We denote local and global constraints by LocalConstraints and GlobalConstraints, respectively. The former are used to specify the maximum response time of single components, through the local_res_time field, while the latter set a name for a sequence of components, namely a *path* (denoted, e.g., by p1 in the JSON file), and fix a response time threshold for the path by global_res_time. This information can be obtained by parsing the code decorators (see Section 4.4.1).

The application DAG specified in the DirectedAcyclicGraph field, includes the name of the components, a list with the names of all the successive components (if we have a branch in the DAG, the list length will be greater than one), and a list of transition probabilities (`transition_probability`) between the current component and its successors. Finally, the `Lambda` field denotes the exogenous input workload.

**JSON input file:**

```
{
  "Components":{ "c1":{ "s1":{ "h1":{ "memory":1024, "next":"c2", "early_exit_probability":0,"data_size":4500}
                   }
              },
            "c2":{ "s1":{ "h1":{ "memory":2048, "next":"", "early_exit_probability":0, "data_size":0}
                  }             }
            },
  "EdgeResources":{ "computationallayer1":{ "RasPi":{ "description":"Raspberry PIs 4 Model B","number":1,
                                   "cost":0.0375, "memory":4096}
                             }
             },
  "CloudResources":{ "computationallayer2":{ "VM1":{ "description":"AWS g3s.xlarge", "number":1,
                                  "cost":0.75, "memory":8192}
                              }
              },
  "CompatibilityMatrix":{ "c1":{ "h1":["RasPi"]},
                  "c2":{ "h1":["RasPi","VM1"] }
                      },
  "DemandMatrix":{ "c1":{ "h1":{ "RasPi": 17474.5 }
                },
             "c2":{ "h1":{ "RasPi": 13900, "VM1": 4242.3}
                 }
            },
  "Lambda": 0.000046,
  "LocalConstraints":{ "c1":{ "local_res_time": 89100 }
              },
  "GlobalConstraints":{ "p1":{ "components":["c1","c2"], "global_res_time": 94350}
               },
  "NetworkTechnology":{ "ND1":{ "computationallayers":["computationallayer1","computationallayer2"],
                    "AccessDelay":0.00000277, "Bandwidth":10000}
               },
  "DirectedAcyclicGraph":{ "c1":{ "next":["c2"], "transition_probability":[1]}
                },
  "Time":1
}
```

**YAML output file**

After executing SPACE4AI-D an output YAML file according to the format reported below (where the container to resource assignment is performed) will be generated. In this specific case, according to the workload and constraints reported in the input JSON file, container 1 is associated with the Raspberry Pi (the only choice) while container c2 is assigned to VM1 to avoid the global constraint violation. In this scenario, the search space is small and the tool could find the best solution even with 5 iterations (0.002 s).

```
System:
  name: Mask Detection Application
  NetworkDomains:
    ND1:
      name: Network Domain 1
      AccessDelay: 0.00000277
      Bandwidth: 10000
      ComputationalLayers:
        - computationalLayer1:
            name: Edge Layer
            number: 1
            Resources:
              - resource1:
                  name: RaspPi
                  description: Raspberry PIs 4 Model B
                  cost: 0.0375
                  memorySize: 4096
                  operatingSystemDistribution: Raspbian
                  operatingSystemType: Linux
                  operatingSystemVersion: 10
                  secureBoot: string
                  measureBoot: string
                  storageSize: 64GB
                  storageType: flash
                  processors:
                    - processor1:
                        name: BCM2711
                        type: Cortex-A72
                        architecture: ARM64
                        computingUnits: 4
                        internalMemory: 4
                        SGXFlag: False
        - computationalLayer2:
            name: Cloud Layer
            number: 2
            Resources:
              - resource1:
                  name: VM1
                  description: AWS g3s.xlarge
                  cost: 0.75
                  memorySize: 8192
                  storageSize: 44
                  storageType: SSD
                  operatingSystemDistribution: Ubuntu
                  operatingSystemType: Linux
                  operatingSystemVersion: 20.04
                  secureBoot: False
                  measureBoot: False
                  onSpot: False
                  processors:
                    - processor1:
                        name: Xeon
                        type: SkyLake
                        architecture: IntelX86
                        computingUnits: 4
                        internalMemory: 64
                        SGXFlag: False

  Containers:
    - container1:
        name: c1
        memorySize: 1024
        computingUnits: 1
        trustedExecution: False
        networkProtection: False
        fileSystemProtection: False
        GPURequirement: False
        deploymentLayer: 1
    - container2:
        name: c2
        memorySize: 2048
        computingUnits: 1
        trustedExecution: False
        networkProtection: False
        fileSystemProtection: False
        GPURequirement: False
        deploymentLayer: 2
```

# Appendix D - Performance benchmarking of Deep Learning training applications

Predicting the execution time of DL training applications is important in order to effectively estimate the operational costs (energy cost in private clouds and resources-renting cost in public clouds) of inference or training tasks. As the results reported in the previous sections demonstrated, performance model regressors allow us to achieve accurate results but require an initial profiling of the target application. To reduce the profiling effort, we have evolved the initial version of the a-GPUBench framework initially developed within the ATMOSPHERE project[11], to automatically train several Deep Neural Networks (e.g., Convolutional Neural Networks or Recurrent Neural Networks) and evaluate their performance varying the model hyperparameters, pursuing the following objectives:

- Migrating the existing framework from Tensorflow 1 to Tensorflow 2,
- Integrate the new version in a containerized run-time environment.

The deployment diagram of the new, containerized version is reported in Figure D.1. It relies on two, possibly different, machines, denoted as *host machine* and *target machine*, respectively, such that the latter can be reached from the former through a *ssh* connection (note that, if the host and the target machine coincide, the connection is performed via localhost).

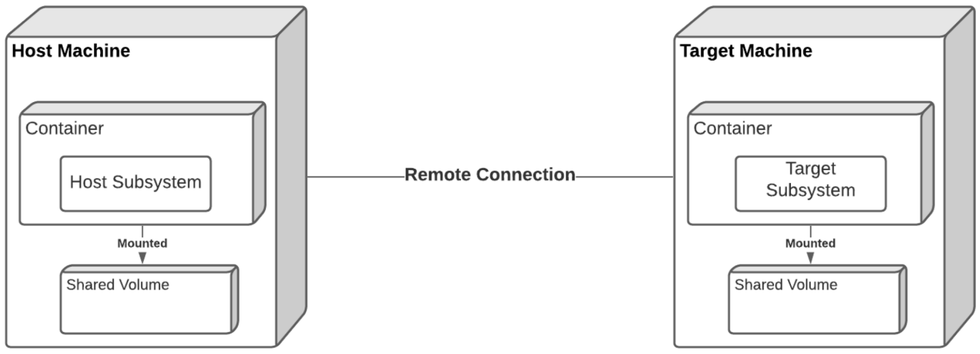


*Figure D.1 - Deployment diagram*

Each machine hosts a Docker container, which supports the execution of all modules in the *host subsystem* or the *target subsystem*, as described in the following. Specifically, the host subsystem is used first of all to set the parameters to be sent to the target machine, such as the configuration file related to the specific application that will be executed (reporting, e.g., the path to the training dataset, the maximum number of epochs, the batch size and other hyperparameters). The available modules are:

- *Experiment Frontend*: it checks the parameters that are passed and calls the provider where the training session is executed, setting the application that we are interested in benchmarking. The current version only supports Tensorflow applications, while the extension to PyTorch applications is ongoing.
- *Provider Controller*: it contains all the available providers on which the training phase can be executed. So far, it is possible to run the training phase on a *local* provider, which identifies a scenario where the host machine and the target machine coincide, and an *inhouse* provider, meaning that the training is executed on a remote machine in a private cloud.

[11] https://github.com/eubr-atmosphere/a-GPUBench

- *Post-training Analysis*: it contains all the scripts used to process the data retrieved from the training session. In particular, these produce a report concerning the execution time of the different phases and the CPU or GPU load.

The target subsystem is responsible for building and executing the training pipeline. It contains the following modules:

- *Local Experiment Frontend*: it prepares the directories necessary to output all the training logs, configures parameters such as the requested application and the hardware to be profiled (GPU and/or CPU), and starts the training session. When this ends, it optionally sends an email to inform that the training session is concluded and sends the training information to a log server.
- *Training Frontend*: it contains the scripts that the Local Experiment Frontend uses in order to start the training. In particular, it loads the configuration file where all the relevant information is specified (network type, dataset, training hyperparameters, etc.). Then, it calls the script that executes the training.
- *Nets Controller*: it contains the definition of the trainable Neural Networks (NNs) and a factory class that is used as an interface to set some hyperparameters and eventually modify the NN structure. This component can be extended with all the possible NNs. So far, AlexNet V2 [Krizhevsky2014], Vgg16 and Vgg19 [Simonyan2015], ResNet50 [He2015] and VQA [Agrawal2016] networks are available.
- *Dataset Controller*: it is composed by two main parts:
  - Dataset Factory: it contains the modules that build the training and validation datasets, using the files available in the file system.
  - *Dataset Setup*: it contains the script used during the setup phase of the target machine. In particular, this script downloads the dataset from the web, performs a preprocessing to build the validation set used for model evaluation, and converts the dataset into TFRecords files.
- *Preprocessing Controller*: it contains the preprocessing function used on the dataset. These are specific for the different NNs and are used to perform some image transformation (e.g., rescaling, subtracting the mean from the pixels, etc.).
- *Internet Service Interface*: it represents the Internet interface used by the Dataset Setup module and by the Local Experiment Frontend when it is required to copy the training files to a logserver.
- Mail Service Interface: it represents the mail interface used by the Training Frontend to notify the user about the training completion (if required).

A user guide with all the relevant information for executing the training and application profiling is reported in the README.md file of the framework, which is available at [a-GPUBench2021] as mentioned in Section 4.5.

# References

[Agrawal2016] A. Agrawal, J. Lu, S. Antol, M. Mitchell, C. L. Zitnick, D. Batra, D. Parikh. VQA: Visual Question Answering, 2016. https://arxiv.org/abs/1505.00468 .

[a-GPUBench2021] https://gitlab.polimi.it/ai-sprint/a-gpubench .

[Chen2016] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '16). Association for Computing Machinery, New York, NY, USA, 785–794. DOI:https://doi.org/10.1145/2939672.2939785 .

[Didona2014] D. Didona, P. Romano. On Bootstrapping Machine Learning Performance Predictors via Analytical Models. CoRR arXiv:1410.5102v1.

[Draper1966] NR. Draper, H. Smith. Applied Regression Analysis. Wiley, 1966.

[Ferri1994] F. J. Ferri, P. Pudil, M. Hatef, J. Kittler (1994). "Comparative study of techniques for large-scale feature selection." Pattern Recognition in Practice IV : 403-413.

[Gianniti2018] E. Gianniti, M. Ciavotta, D. Ardagna. Optimizing quality-aware big data applications in the cloud. IEEE Transactions on Cloud Computing, vol. 9, n. 2, page: 737-752, 2018. DOI: 10.1109/TCC.2018.2874944.

[Maros2019] A. Maros et al., "Machine Learning for Performance Prediction of Spark Cloud Applications," 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), 2019, pp. 99-106, doi: 10.1109/CLOUD.2019.00028.

[Paszke2017] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, A. Lerer. Automatic differentiation in PyTorch. In: 31st Conf. Neural Information Processing Systems (NIPS 17).

[PyTorch] PyTorch, Tensors and dynamic neural networks in Python with strong GPU acceleration. URL: https://pytorch.org

[TensorFlow] TensorFlow, An open source machine learning framework for everyone. URL www.tensorflow.org

[Abadi2016] M. Abadi, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. 2016. https://arxiv.org/abs/1603.04467

[Krizhevsky2012] A. Krizhevsky, I. Sutskever, GE. Hinton. ImageNet classification with deep convolutional neural networks. In: Proc. 25th Int'l Conf. Neural Information Processing Systems (NIPS 12), vol 1, pp 1097–1105, 2012.

[Krizhevsky2014] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks, 2014. https://arxiv.org/abs/1404.5997 .

[He2015] K. He, X. Zhang, S. Ren, J. Sun. Deep residual learning for image recognition, 2015. https://arxiv.org/abs/1512.03385 .

[Simonyan2015] K. Simonyan, A. Zisserman. Very deep convolutional networks for large-scale image recognition, 2015. https://arxiv.org/abs/1409.1556 .

[Deng2009] J. Deng, W. Dong, R. Socher, L. Li, K. Li, L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In: IEEE Conference on Computer Vision and Pattern Recognition, IEEE, pp 248–255, 2009.

[Ben-Gal2005] I. Ben-Gal. Outlier detection. In O. Maimon and L. Rockach, editors,DataMining and Knowledge Discovery Handbook: A Complete Guide for Practitioners and Researchers, chapter 1. Kluwer Academic Publishers, 2005.

[Mozilla] Foundation M Common voice. URL https://voice.mozilla.org/

[Hadjis2016] S.Hadjis, C. Zhang, I. Mitliagkas, C. Ré. Omnivore: An optimizer for multi-device deep learning on CPUs and GPUs, 2016. https://arxiv.org/abs/1606.04487

[PyTorch] PyTorch, Tensors and dynamic neural networks in Python with strong GPU acceleration. URL https://pytorch.org

[Lin2013] M. Lin, Q. Chen, S. Yan. Network in network, 2013. https://arxiv.org/abs/1312.4400

[Yao2007] Y. Yao, L. Rosasco, A. Caponnetto. On early stopping in gradient descent learning. Constr. Approx, page 289-315, 2007.

[Ardagna2007] D. Ardagna, B. Pernici. Adaptive Service Composition in Flexible Processes. IEEE Trans. on Software Engineering, vol. 33, n. 6, page: 369-384, 2007.

[Gianniti2018b] E. Gianniti, L. Zhang, D. Ardagna. Performance Prediction of GPU-Based Deep Learning Applications. 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2018.

[Sivaraman2018] H. Sivaraman, U. Kurkure, L. Vu. Task Assignment in a Virtualized GPU Enabled Cloud. IEEE HPCS, 2018.

[Lazowska1984] E. D. Lazowska, J. Zahorjan, G. S. Graham, K. C. Sevcik. Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall, Inc., 1984.

[Tadakamalla2021] U. Tadakamalla, D. A. Menasce. Autonomic Resource Management for Fog Computing. IEEE Transaction of Cloud Computing, 2021. DOI: 10.1109/TCC.2021.3064629.

[Mahmoudi2020] N. Mahmoudi, H. Khazaei. Performance Modeling of Serverless Computing Platforms. IEEE Transaction of Cloud Computing, 2020.

[AzureVMpricing] Pricing calculator. https://azure.microsoft.com/en-us/pricing/calculator/

[AWS] AWS. https://aws.amazon.com/

[AWSStepFunctionsPricing] AWS Step Functions Pricing. https://aws.amazon.com/step-functions/pricing/

[Risco2021] S. Risco, G. Moltò, D. M. Naranjo, I. Blanquer. Serverless Workflows for Containerised Applications in the Cloud Continuum. Journal of Grid Computing, vol. 19, n. 30, page 1-18, 2021.

[Technical Report] H. Sedghani, F. Filippini, D. Ardagna. SPACE4-AI: A Design-time Tool for AI applications Resource Selection in Computing Continua (Technical Report). https://www.dropbox.com/s/swce5b3iut4qhsz/Optimization_in_Fog_and_Cloud_Technical_Report.pdf?dl=0

[AmazonPricing] Amazon EC2 On-Demand Pricing. https://aws.amazon.com/ec2/pricing/on-demand/

[BenchmarkingAmazon] Benchmarking Amazon VPC. https://aws.amazon.com/premiumsupport/knowledge-center/network-throughput-benchmark-linux-ec2

[HyperOptGitHub] Hyperopt: Distributed Hyperparameter Optimization. https://github.com/hyperopt/hyperopt

[Elgamal2018] T. Elgamal, A. Sandur, K. Nahrstedt, G. Agha. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. IEEE/ACM SEC, 2018.

[Manner2018] J. Manner, M. Endre, T. Heckel, G. Wirtz. Cold Start Influencing Factors in Function as a Service. ACM/IEEE UCC Companion, page 181–188, 2018.

[Lin2021] C. Lin, H. Khazaei.Modeling and Optimization of Performance and Cost of Serverless Applications. IEEE Trans. on Parallel and Distributed Systems, vol. 32, n.3, page:615 - 632, 2021.

[AzureLogicApps] Azure Logic Apps. https://azure.microsoft.com/en-us/services/logic-apps/

[Goldberger2000] Goldberger, A., Amaral, L., Glass, L., Hausdorff, J., Ivanov, P. C., Mark, R., ... & Stanley, H. E. (2000). PhysioBank, PhysioToolkit, and PhysioNet: Components of a new research resource for complex physiologic signals. Circulation [Online]. 101 (23), pp. e215–e220.

[Datta2017] S. Datta et al., "Identifying normal, AF and other abnormal ECG rhythms using a cascaded binary classifier," 2017 Computing in Cardiology (CinC), 2017, pp. 1-4, doi: 10.22489/CinC.2017.173-154.

[Graf2004] Hans Peter Graf, Eric Cosatto, Leon Bottou, Igor Durdanovic, and Vladimir Vapnik. 2004. Parallel support vector machines: the cascade SVM. In Proceedings of the 17th International Conference on Neural Information Processing Systems (NIPS'04). MIT Press, Cambridge, MA, USA, 521–528.

[Clayton1993] R. H. Clayton and A. Murray, "Estimation of the ECG signal spectrum during ventricular fibrillation using the fast Fourier transform and maximum entropy methods," Proceedings of Computers in Cardiology Conference, 1993, pp. 867-870, doi: 10.1109/CIC.1993.378299.

[Wood1996] J. C. Wood and D. T. Barry, "Time-frequency analysis of skeletal muscle and cardiac vibrations," in Proceedings of the IEEE, vol. 84, no. 9, pp. 1281-1294, Sept. 1996, doi: 10.1109/5.535246.

[Mahmoud2006] Mahmoud SS, Fang Q, Davidović DM, Cosic I. A time-frequency approach for the analysis of normal and arrhythmia cardiac signals. Conf Proc IEEE Eng Med Biol Soc. 2006;Suppl:6509-6512. doi:10.1109/IEMBS.2006.260882

[Lomurno2021] Lomurno, E., Samele, S., Matteucci, M., & Ardagna, D. (2021, July). Pareto-optimal progressive neural architecture search. In Proceedings of the Genetic and Evolutionary Computation Conference Companion (pp. 1726-1734).

[Liu2018] Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L. J., ... & Murphy, K. (2018). Progressive neural architecture search. In Proceedings of the European conference on computer vision (ECCV) (pp. 19-34).

[Zoph2016] Zoph, B., & Le, Q. V. (2016). Neural architecture search with reinforcement learning. arXiv preprint arXiv:1611.01578.

[Lordan2014] Lordan, Francesc, et al. "Servicess: An interoperable programming framework for the cloud." Journal of grid computing 12.1 (2014): 67-91.

[Alvarez2019] J. Álvarez Cid-Fuentes, S. Solà, P. Álvarez, A. Castro-Ginard, and R. M. Badia, "dislib: Large Scale High Performance Machine Learning in Python," in Proceedings of the 15th International Conference on eScience, 2019, pp. 96-105

[Venkataraman2016] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica, "Ernest: Efficient performance prediction for large-scale advanced analytics." in NSDI, 2016, pp. 363–378.

[OpenFog] OpenFog Consortium Architecture Working Group. Openfog reference architecture for fog computing [opfra001.020817]. Techni- cal Report, 2017 p. 1–162. Accessed 2 December 2021.

[NumPy2011 ] S. van der Walt, S. C. Colbert, and G. Varoquaux, "The NumPy Array: A Structure for Efficient Numerical Computation," Computing in Science & Engineering, vol. 13, no. 2, p. 22, 2011.