ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

ARCHIVIO ISTITUZIONALE
DELLA RICERCA

Alma Mater Studiorum Università di Bologna
Archivio istituzionale della ricerca

HADA: An automated tool for hardware dimensioning of AI applications

This is the final peer-reviewed author's accepted manuscript (postprint) of the following publication:

Published Version:
De Filippo, A., Borghesi, A., Boscarino, A., Milano, M. (2022). HADA: An automated tool for hardware dimensioning of AI applications. KNOWLEDGE-BASED SYSTEMS, 251, 1-18 [10.1016/j.knosys.2022.109199].

Availability:
This version is available at: https://hdl.handle.net/11585/894554 since: 2022-11-16

Published:
DOI: http://doi.org/10.1016/j.knosys.2022.109199

Terms of use:

Some rights reserved. The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/).
When citing, please refer to the published version.

(Article begins on next page)

12 October 2024

This is the final peer-reviewed accepted manuscript of:

De Filippo, A., Borghesi, A., Boscarino, A., & Milano, M. (2022). HADA: An automated tool for hardware dimensioning of AI applications. Knowledge-Based Systems, 251

The final published version is available online at **https://dx.doi.org/10.1016/j.knosys.2022.109199**

Rights / License:

The terms and conditions for the reuse of this version of the manuscript are specified in the publishing policy. For all terms of use and more information see the publisher's website.

This item was downloaded from IRIS Università di Bologna (https://cris.unibo.it/)

When citing, please refer to the published version.

HADA: an Automated Tool for Hardware Dimensioning of AI Applications

Allegra De Filippo[a,b], Andrea Borghesi[a,b], Andrea Boscarino[a], Michela Milano[a,b]

[a] *Department of Computer Science and Engineering, University of Bologna, Italy*
[b] *Alma Mater Research Center for Human-Centered Artificial Intelligence, University of Bologna, Italy*

Abstract

In recent years, the uptake of Artificial Intelligence (AI) in industry is increasing. For many AI techniques, like Deep Learning, optimization, planning, etc., computational and storage requirements are significant. The problem of determining what is the right hardware (HW on premise or on the cloud) architecture and its dimensioning for AI algorithms is still crucial. Searching for the optimal solution is often challenging, as it is not trivial to anticipate the behaviour of an algorithm on diverse architectures. This is especially true if the AI application must respect quality-of-service constraints or budgets. In this scenario, having an automated decision support tool to match algorithms, user constraints and HW resources would be a great advantage for companies and practitioners working with AI applications.

In this paper, we tackle this challenge with an approach that relies on the Empirical Model Learning paradigm, based on the integration of Machine Learning (ML) models into an optimization problem. The key idea is to integrate domain knowledge held by experts with data-driven models that learn the relationships between HW requirements and AI algorithm performances. In particular, the approach starts with benchmarking multiple AI algorithms on different HW resources, generating data used to train ML models; then, optimization is used to find the best HW configuration that respects user-defined constraints (e.g., budget, time, solution quality).

Email addresses: `allegra.defilippo@unibo.it` (Allegra De Filippo), `andrea.borghesi3@unibo.it` (Andrea Borghesi), `andrea.boscarino2@unibo.it` (Andrea Boscarino), `michela.milano@unibo.it` (Michela Milano)

In the experimental evaluation we validate our approach on a complex problem, namely online algorithms for energy systems, an area characterized by uncertainty and tight HW and real-time constraints. Results show the effectiveness of our approach and its flexibility: we can train the ML models only once and reuse them in the optimization model to tackle a variety of problems, determined by different data instances and user-defined constraints.

Keywords: Empirical Model Learning, Hardware Dimensioning, Anticipatory Algorithms, Efficient Energy Management, Machine Learning, Constrained Optimization

---

1. Introduction

Artificial Intelligence (AI) applications are now the focus for many companies and industries. Moreover, interfacing with increasingly computationally intensive AI applications leads to the need of determining the right hardware (HW) architecture and its configuration to run them under required performances and budget limits. This problem is called *hardware dimensioning*. This represents a challenging and complex task for many companies that need assistance from AI experts in this task. Algorithm developers need to answer questions such as: (1) *if an AI algorithm has to run under real-time constraints, respecting some budget constraints and guaranteeing a certain solution quality, which HW architecture should be used?*, (2) *given a certain set of (possibly heterogeneous) HW resources, what is the best algorithm to solve a problem respecting user-defined constraints?*. These are not trivial questions which require domain and AI experts knowledge to be solved, owing to the complexity of knowing beforehand the behaviour of an algorithm on different HW architectures and evaluating the effect of all possible choices (HW and configurations). Moreover, even AI experts might not know exactly how to dimension the HW resources, typically resorting to over-provisioning (requesting more resources than those strictly needed) and/or using heuristics, with the risk of sub-optimal solutions.

Ideally, the challenge could be tackled by integrating domain knowledge held by experts (time constraints, required solution quality, budget limits) with data-driven models that learn relationships between HW requirements and AI algorithm performances. Having an automated tool to match AI algorithms, user-defined constraints and HW resources would be a great advantage for AI users. In this work, we make a step in this direction by

proposing HADA, an automated tool for HArdware Dimensioning of (AI) Algorithms, framing it as an optimization problem. As shown in Fig. 1, our tool can be represented as a black-box that receives as input a set of features describing an AI algorithm and some user-defined constraints (e.g. budget limits, time constraints and required solution quality), and it produces as output the optimal HW resource dimensioning needed to run the algorithm.

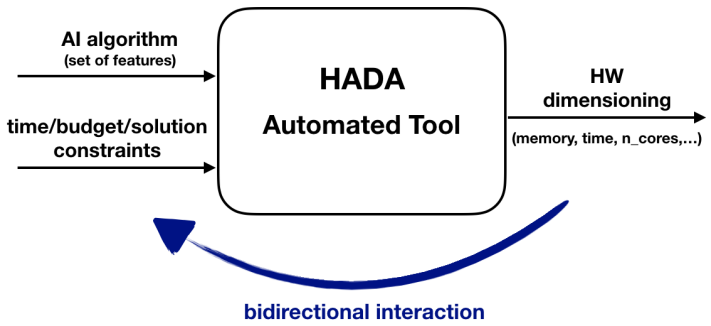

Figure 1: schema of HADA

As enabling technology for our approach, we use the *Empirical Model Learning* (EML) framework [1, 2], which allows the integration of ML models within an optimization problem to enable decision making over complex real-world systems. The idea is to *learn*, rather than directly express, the relationships between online algorithm performance and HW resources via ML models.

Our ML models learn the relations between the AI algorithm (defined based on the application domain), and HW dimension. These ML models are then embedded in the optimization problem, which provides the optimal HW resource dimensioning for a specific algorithm, given a set of user-specified constraints (e.g., bounding the run-time of the algorithm to obtain solutions with quality higher than a threshold).

The strengths of HADA are the following:

1. we *train the ML models only once* and reuse them in the optimization model on different data instances and different user-defined constraints, as they are posted on the backbone of the optimization model when needed.
2. EML surpasses standard ML as it enables *bidirectional interaction* between an AI algorithm, its performance, and HW dimensioning,

through the encoding of ML models via constraints and variables, upon which decisions can be made.

3. A byproduct of our approach is that, given a fixed HW architecture and configuration as input constraints, it is possible to obtain the most suitable algorithm together with its parameters.

As a case study, we focus on online optimization algorithms for an energy system domain, which have to take into account tight real-time constraints and uncertainty, e.g., renewable energy production and demand fluctuation. This is a typical real world problem with real time constraints. Selecting the optimal HW configuration for a given set of tight constraints on solution time and quality over multiple and diverse data instances (e.g., the specific details of the energy system) for a given online algorithm is a complex, and still open, problem [3]. In this work, we consider two stochastic algorithms for energy systems from the literature, CONTINGENCY and ANTICIPATE [4]. In particular, ANTICIPATE is an online (scenario-based) anticipatory algorithm, while CONTINGENCY is an integrated offline/online algorithm that tackles the uncertainty by building (offline) a pool of solutions to guide an efficient online method. Both these methods are characterized by a parameter that can be changed based on the required constraints in terms of time constraint and required solution quality (resp., the number of scenarios for ANTICIPATE, and the number of traces for CONTINGENCY); in our work we focus on the online phases of these methods.

The results of the experimental evaluation reveal that the proposed approach can correctly select the best algorithm and its configuration while respecting the user-defined constraints and optimizing the desired function. The results were thoroughly validated to assess the robustness of the method. The rest of the paper is organized as follows: Sec. 2 overviews the related literature, Sec. 3 introduces the proposed approach, Sec. 4 discusses the experimental results, and Sec. 5 concludes the paper.

2. Related Work

2.1. Hardware Dimensioning

In the last years, many computationally-intensive AI algorithms and new techniques became available, but often they are not mature enough to be used in the real-world, due to hardware constraints or availability. In this context, [5] proposes a systematic literature review on hardware implementation of

AI algorithms in order to explore the available hardware accelerators for the AI tools; and [6] studies how different state-of-the-art AI algorithms can be adapted to different hardware settings based on some restrictions or/and programming challenges. Many works [7, 8] propose general benchmarks and evaluation metrics for algorithm performances in order to fill the gap between the huge number of AI domains and the lack of a standard format or repository for this data. Also in the energy physics field, where applications require a serious computing challenge, many works [9, 10] tackle the issue of HW dimensioning by considering different and multiple aspects, such as how to run complex complex algorithms on a system with very constrained resources.Moreover, other works propose a characterization of a possible HW dimensioning for targeted algorithms [11, 12] or for specific task of computationally intensive AI applications [13]. There is an emerging need for a general and modular automated tool able to recommend an optimal HW configuration for a given AI algorithm, by knowing its performance on diverse architectures and based on user requests. To the best of our knowledge, our work represents a first attempt in this direction: we defined a general and automated decision support tool to match a given AI algorithm, user-defined constraints and HW resource dimensioning.

2.2. Black-box Optimization

Our approach can be comparable to a black-box optimization approach, since we are optimizing functions which are very costly to evaluate (e.g., Costa et al. developed RBFOpt [14], an open-source library for optimization with black-box functions. Another family of approaches in this field is *Bayesian Optimization* [15, 16, 17].The goal is to obtain a good approximation of the target function with the minimum number of iterations. All these methods are very competitive in finding optimal parameter configurations and good approximation of the target function for a given algorithm. The lack of constraints in the optimization part has been addressed in more recent works [18, 19]. A recent work by Eriksson and Poloczek[20] proposes a novel approach, called Scalable Constrained Bayesian Optimization (SCBO), to tackle constrained optimization problems with Bayesian optimization, aiming at increased scalability. SCBO targets constraint optimization problems where both the objective and constraints are black-box functions (e.g., no derivative information is given). Differently, we can train the ML models only once and reuse them in the optimization model on different data instances and

different user-defined constraints, as they are posted on the backbone of the optimization model when needed.

2.3. Algorithm Selection & Configuration

As mentioned already, the bidirectional interaction offered by our approach allow to perform algorithm selection and configuration as well, a vastly studied research area [21, 22, 23, 24]. For example, Hutter et al. [25, 26] propose a Sequential Model-based optimization for general Algorithm Configuration (SMAC), an automated procedure for algorithm configuration that explores the space of parameters settings. These approaches go in promising direction that we extend here by taking into account, beside the time-to-solution, other metrics such as memory requirements and solution quality. Recently, a series of methods were proposed to jointly tackle algorithm selection and configuration, as both phases can be cast as automated tuning problems, collectively called Combined Algorithm and Hyperparameter Optimization (CASH) [27, 28]. The main difference of our approach and CASH methods is the possibility to naturally express and enforce constraints as allowed by the flexible mathematical programming formulation, and to easily change the objective function; this means that the same model can be used for a series of different queries.

3. HADA

The core idea of our approach is to solve the problem finding the optimal hardware requirements for an AI algorithm, subject to user-defined constraints such as budget, run-time and solution quality. At the basis of our approach is the Empirical Model Learning (EML) paradigm, introduced by Lombardi et al.[1], a technique to embed ML models within combinatorial optimization problems; EML is especially well-suited to deal with complex functions hard to be analytically expressed (e.g., [29, 30, 31]).

Broadly speaking, EML deals with solving declarative optimization models with a complex component h, which represents the relation between variables which can be acted upon x (the decision variables) and the observables related to the system considered; the function $h(x) = y$ describes this relationships. As the $h(x)$ is complex, we cannot optimize directly over it; in general, we cannot even be sure to be capable of expressing it in an analytical fashion. Hence, we exploit *empirical* knowledge to build a surrogate model $h_\theta(x)$ learned from data, where θ is the parameter vector.

The general structure is the following:

$$
\begin{aligned}
\min \quad & f(x, y) && && (1)\\
& h_\theta(x) = y && && (2)\\
\text{s.t.} \quad & g_j(x, y) && \forall j \in J && (3)\\
& x_i \in D_i && \forall x_i \in x && (4)
\end{aligned}
$$

where x is the decision variables vector (each x_i has its own domain D_i) and y is the vector of observed variables. The goal is to minimize the objective function $f(x, y)$, which might depend on both decision and observed variables. Both decision and observed variables can be subject to a set of constraints $g_j(x, y)$, such as classical inequalities from Mathematical Programming and combinatorial predicates from Constraint Programming (e.g., global constraints) The function $h_\theta(x)$ represents the approximate behaviour of the complex relation between decision and observed variable, and it is, in practice, the encoding of a ML model.

The surrogate model $h_\theta(x)$ is trained in the typical ML fashion; we assume here to be in a supervised setting, but the reasoning holds in different contexts as well, with minor changes to the formulation. EML requires a training set $\mathcal{S} = (x_i, h(x_i))_{i=1}^m$ which is then used to find the parameter vector θ minimizing a loss function:

$$\operatorname*{argmin}_\theta \frac{1}{m} \sum_{i=1}^{m} L(h_\theta(x_i), y_i^*) \tag{5}$$

where y_i^* are the ground-truth labels (or targets, in case of regression) of each data point in the training set $\mathcal{S}$ and L is the loss function (e.g., L_1 or L_2 loss for scalar ys).

The key phases of the EML methodology are the following:

1. define the combinatorial structure of the optimization model, identifying decision and observed variables;
2. acquiring a data set describing the complex phenomenon – in our case, the relationships between HW resources and algorithm performance;
3. train one or multiple ML models (either regression or classification tasks) to learn these relations;
4. embed the trained ML models as a set of constraints forming the core of a combinatorial model;

5. extend the declarative model with additional constraints, encapsulating either domain knowledge which can be analytically expressed and user requirements;
6. add the objective function (depending on the desired task) and solve the optimization problem.

The overall architecture of HADA, cast in the EML framework, is described in Figure 2. At its core there is an optimization model composed by 1) a data-driven component, in the form of a set of ML models (encoded as constraints) which encapsulate empirical knowledge, 2) a knowledge-based component representing the domain knowledge (when available), and 3) an objective function and constraints specified by the user according to her needs. The first two components (domain-based and empirical knowledge) constitute the inner kernel of the methodology and once correctly encoded can be reused to answer different queries created by posting user goal and constraints; this flexibility is allowed by the modular approach. The empirical information is encoded with several, different ML regression models. These ML models are then embedded in a constrained optimization model, detailed in Sec. 3.3. To train the ML models we rely on a data set consisting of many runs of the algorithms with different parameters and data instances.

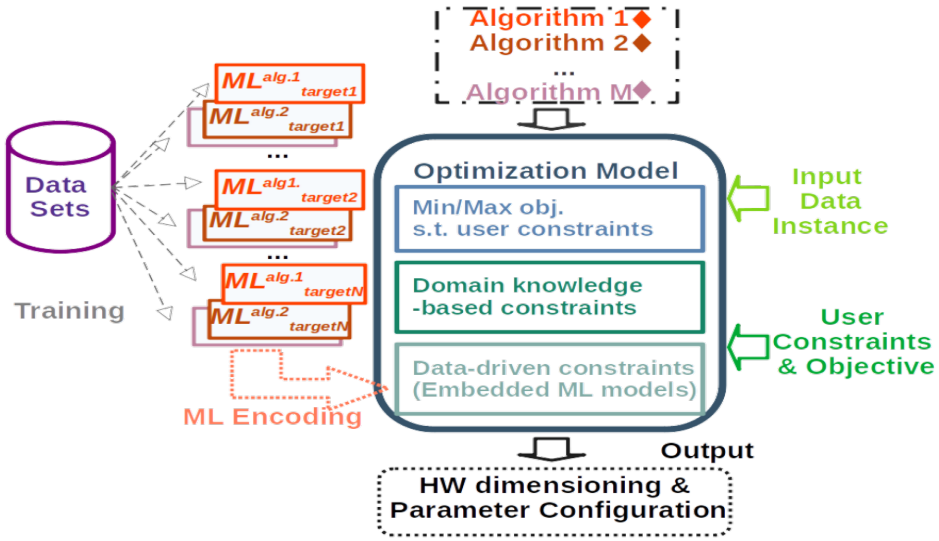

Figure 2: Proposed Approach Scheme

Albeit the proposed approach is general, for the sake of concreteness in the rest of this section we describe the method "grounded" on the specific

algorithms we selected as use case for the experimental evaluation, namely two online state-of-the-art stochastic algorithms for energy systems, ANTICIPATE and CONTINGENCY [4]. Both these algorithms can be configured by setting a particular parameter, respectively the number of scenarios for ANTICIPATE and the number of traces for CONTINGENCY. Additionally, as targets for the ML models we consider time, memory, and solution cost. Each ML model is learnt based on a specific algorithm and predicts a specific target: (1) the time required by the algorithm *alg* to find a solution ($\mathrm{ML}^{\mathrm{alg}}_{\mathrm{time}}$); (2) the amount of RAM memory used ($\mathrm{ML}^{\mathrm{alg}}_{\mathrm{mem}}$); (3) the solution quality, expressed in terms of its *cost* ($\mathrm{ML}^{\mathrm{alg}}_{\mathrm{cost}}$).

In the following subsections we will first describe the data set used to train the ML models (3.1), then, we provide details about the ML models used to learn the relationship between HW resources, algorithm configuration and performance (3.2). Then, we present the optimization model (3.3); finally, we describe how domain-knowledge is used (3.3.2).

3.1. Data Set Creation through Benchmarking

We built a training set based on grounding the two stochastic algorithms, i.e., ANTICIPATE and CONTINGENCY [32, 4] from the energy management system domain. The two algorithms calculate the amount of energy that must be produced by the energy system to meet the required load, minimizing the total energy cost over the daily time horizon and by taking into account the uncertainty. As shown by the authors in [4], ANTICIPATE and CONTINGENCY are general approaches for online stochastic optimization that can be applied to different real-world use cases. These algorithms are run in a multistage fashion, dividing the daily time horizon in 96 15-minutes time intervals. ANTICIPATE is a typical online scenario-based anticipatory algorithm: it replaces at each timestamp a greedy heuristic with a two-stage Linear Programming model, by applying the Sample Average Approximation [33]. This look-ahead approach makes it possible to take better decisions based on knowledge of future uncertainty. However, due to the limited time constraints in online settings, it could be useful to use the offline available information (e.g., assumption on probability distributions of the uncertain elements or historical samples) as a kind of preparation (contingency plans, tuning of probabilistic models) before the online execution starts [34]. In this perspective, the second algorithm, i.e. CONTINGENCY, is based on this idea: it is composed by an online fixing heuristic that uses traces (pre-computed solutions generated in an offline phase) to solve the online problem and guide the decisions [4].

Both these algorithms are characterized by a single configuration parameter, respectively the number of scenarios for ANTICIPATE and the number of traces for CONTINGENCY. Our ML models learn the relations between the optimal parameter configuration for the online algorithms (number of scenarios or number of traces), HW dimensioning, and characteristics of the problem (application domain). Learning such relations is fundamental for our goals, since the number of traces/scenarios has repercussions on the solution time and the memory consumption, beside affecting the solution quality. HADA treats these two algorithms as black-boxes exposing a configurable parameter (i.e., the number of scenarios or the number of traces).

The training data set is generated by executing the algorithms on different problem instances. The uncertainty model used to sample (instance) realizations is technically a Gaussian mixture. It is designed to ensure a realistic level of dependency between the random variables (see [4] for further details). For the energy system case study, we assume that both the RES generation and the load (L) at each stage t may exhibit normally distributed deviations from a number of different possible behaviors. Formally, each mean and standard deviation is controlled by a second "mode" random variable ψ. Based on this model, 100 different instance realizations were considered in order to create the data set. Then, the algorithms run on each instance 100 times, each time considering a different number of the configurable parameter (from 1 to 100 traces/scenarios). The time horizon is 96 (NT) timestamps of 15 minutes each, in order to obtain a daily time horizon for optimization. This value is set based on a preliminary analysis on different experimental settings of [4], which revealed that running the algorithms on each instance 100 times sufficiently explores the parameter space; each time a different parameter value is used (from 1 to 100 scenarios/traces). Therefore, the training set is composed by 10,000 records (100 runs x 100 instances). The attributes/features are related to the algorithm inputs and performance.

For each record, we have the following information: 1) *nParameter*, n^P: an integer $\in [1, 100]$ that represents the number of scenario/traces used by the algorithm. 2) *Load*, L, (kW): a vector of 96 values representing the load observations sampled in 96 stages (every 15 minutes over the course of a day). 3) RES, (kW): a vector of 96 values representing the observations of available renewable production. 4) solution cost (k€): a real number representing the value of the solution, in practice a measure of the daily total energy cost. It is obtained as the sum over the entire time horizon (over 96 stages) of all partial solutions. 5) time (sec): the time required for finding the solution

with the algorithm. It is obtained as sum over the entire time horizon of all 96 partial running times. 6) mem (MB): represents the RAM used by the algorithm on the machine where it was performed. It is an average of the memory used by the algorithm in 96 partial runs.

For the sake of completeness, we want to point out that we did not consider as inputs the technical parameters of the energy system (e.g. physical bounds of generators or storage systems), since these values do not change on varying of the instance. In the context of our ML task (explored in detail in Sec. 4.1), n^P, L and RES represent the input features for the learning task, while solution, time and memory (the algorithm performance metrics) are the three targets; the goal is to learn the relationship between the input features and the targets, using a separate ML model for each target. As the targets are continuous values, we are dealing with a regression task. All the runs for training set creation have been performed on Intel Core i5 (3,1 GHz) machines with 16 GB of RAM. The data set generated through multiple runs of the target algorithms has been published and made available with open access policy on a public repository[1].

3.2. ML Models

We use ML models to characterize the behaviour of each algorithm considered; this is data-driven knowledge which captures dynamics not easily expressible in analytical form. As already mentioned, this empirical knowledge is a core component of our approach, as it forms the backbone of the optimization model, once encoded in a suitable way (through Empirical Model Learning). We do not feed the entire vectors representing the data instance to the ML models, as using their raw representation would have resulted in a very high-dimensional feature space. Instead, we compute synthetic statistics describing the instances and we use the transformed features. In particular, we compute the mean and the variance of the instance, so each vector is reduced to a couple of values – RES_μ and RES_σ, and L_μ and L_σ. We want to learn the relationships between these input features (n^P, RES_μ, RES_σ, L_μ, and L_σ) and the three targets we are considering, time to obtain a solution (time), RAM memory required (mem), and solution quality (cost) expressed in terms of cost – lower costs represent solutions of higher quality. For each algorithm we use three different ML models, as each target represents a

[1] `https://zenodo.org/record/5838437`

distinct regression task; overall, we obtain six ML models.

These ML models have to be trained and validated using the data set we generated, then they are embedded in the optimization model. As embedding ML models has a cost, both in terms of variables and constraints introduced in the optimization model and of computational time required to solve it[2], we were not just interested in using the most accurate ML models, but we also took computational demands into account. As described in Sec. 4.1, we opted for the ML models that provides the best trade-off between accuracy and complexity, that is decision trees (DTs).

3.3. Optimization Model

In this section we will provide more details about the different components of the optimization model, starting from the empirical knowledge (Sec. 3.3.1) and the domain knowledge (Sec. 3.3.2). We then describe the user-defined constraints and objective (Sec. 3.3.3). We also provide more details about the encoding of the ML models in the optimization model in Sec. 3.3.4. As the ML models are accurate but represent an approximation, we show how to increase the robustness of the optimization model in Sec. 3.3.5. Finally, Sec. 3.3.6 shows a linearization of the quadratic constraints introduced in the previous sections, as quadratic constraints are not accepted in several solvers.

3.3.1. Empirical Model Constraints

Formally, we indicate the set of possible algorithms as A; in the case considered in this paper $A = \{\textsc{anticipate}, \textsc{contingency}\}$. The optimization model is composed by two groups of decision variables. First, there are x_p^a ($a \in A$ and $p \in P^a$) which represent the parameters P^a associated to the algorithm a; for instance, both ANTICIPATE and CONTINGENCY possess a single configurable parameter, respectively the number of scenarios and the number of traces[3] – hence, we have two parameter-related decision variables. x_p^a might be continuous or integer, depending of the corresponding parameter; in our case, both variables are integer. The domain of x_p^a depends on the specific algorithms and parameters, as well; in our case, both the number of scenarios and traces vary between 1 and 100 therefore $x_p^a \in [1, 100], \forall a \in A, \forall p \in P^a$.

[2]For instance, embedding deep DTs or large and complex NNs has a higher cost than using models with few layers and/or neurons.

[3]Thus, $P^{\textsc{anticipate}} = \{\text{N. Scenarios}\}$ and $P^{\textsc{contingency}} = \{\text{N. Traces}\}$

We then have a second group of binary variables $b^a \in [0,1]$ $(a \in A)$ that enable the algorithm selection through multiplication with x_p^a.

Then, there are the four continuous variables corresponding to the data instances di_f, where $f \in F$ indicate the features used to describe the instance; in our case we have four features: $F = \{\mathrm{RES}_\mu, \mathrm{RES}_\sigma, \mathrm{L}_\mu, \mathrm{L}_\sigma\}$. These do not behave as "real" decision variables as their value is directly set by four constraints which bound their values to the specific data instance I: $x_{RES_\mu} = RES_\mu^I$, $x_{RES_\sigma} = RES_\sigma^I$, $x_{L_\mu} = L_\mu^I$, and $x_{L_\sigma} = L_\sigma^I$. The union of the decision variables configuring the algorithm x_p^a and the instance descriptors di_f are the input of the ML models (in our case DTs). We then introduce a group of variables representing the outputs of the ML models (the targets) y_t^a, where $a \in A$ indicates the algorithm and $t \in T$ the target; in our case the possible outputs are the three described previously, thus $T = \{\mathrm{cost}, \mathrm{time}, \mathrm{mem}\}$. These are continuous variables and for each one the range is bounded by the minimum and maximum values encountered in the data set.

The data-driven core of the optimization model (without recapping the variables already described and the constraints used to fix the instance features) is the following:

$$\sum_{a \in A} b^a = 1 \tag{6}$$

$$DT_t^a(x_p^a \cup di_f, y_t^a) \qquad \forall t \in T, \forall a \in A, \forall p \in P^a, f \in F \tag{7}$$

Eq. 6 enforces that exactly one algorithm must be selected and Eq. 7 encapsulates the DTs described in the previous section, and it represents the surrogate model $h_\theta(x)$ in Eq. 2. There are different models for each algorithm a and target t; in our case this means that we have six different DTs (two algorithms multiplied by three targets). The ML models indicated as DT_t^a take as input the algorithm parameters and the instance description. The embedding of the DTs introduces a series of binary variables in the model linked together with the optimization model variables corresponding to the input $(x_p^a \cup di_f)$ and output of the DT (y_t^a), through logical constraints. As the number of variables and constraints introduced is quite large (hundreds or thousands, depending on the depth) we decided not to list them explicitly; rather, we summarize the relationships introduced by embedding each DTs with the notation $DT_t^a(x_p^a \cup di_f, y_t^a)$. The details on how the decision trees are encoded as a set of constraints are provided in Sec. 3.3.4, together with an example of a DT of limited depth.

3.3.2. Domain Knowledge Constraints

In addition to the empirical knowledge, it is also possible to directly use domain knowledge, when available; as shown in Figure 2. In this paper, we decided to show how domain knowledge can be used by considering an additional HW resource, that is the number of *cores* (and thus the number of *computational threads*), and its impact on the algorithms runtime. Albeit this relationship could have been learnt through ML models later embedded as constraints, we opted to treat it differently to demonstrate an alternative method which can be adopted when domain knowledge is available[4].

In the optimization community, an important research avenue aims at characterizing the performance of optimization solvers and algorithms according to the number of cores available and the degree of parallelism. In this context, the most exhaust and public data sets are the MIPLIB2010 and MIPLIB2017 (see Mittelmann et al. [35]), the results of a longstanding benchmarking effort with multiple solvers. We employed this public data set to estimate the behaviour of the algorithms considered in this paper under different degrees of parallelism, i.e. using different numbers of computing cores – we assume that the number of threads spawned would saturate the available cores (as it is the case of Gurobi, the optimization solver used to run ANTICIPATE and CONTINGENCY). The general rule states that increasing the number of concurrent computational threads leads to a decrease in runtime; clearly, the impact of the parallelism strongly depends on the nature of the parallelized application. However, in this work we are interested in demonstrating the potential of using this domain-based knowledge, rather than obtain the most accurate possible model of our algorithms behaviour when run in a parallel fashion[5]; thus, we used the average runtime speedup values obtained with Gurobi as a solver and reported in MIPLIB2017. We also assume that the memory consumption and the solution quality are not impacted by the number of available cores.

The domain knowledge is modelled in the following fashion. We have a set of degrees of parallelism, i.e. the number of threads (or computing cores)): $NT = \{1, 2, 4, 8, 12, 16, 32\}$ (i.e., the number of threads and for each

[4]Such knowledge has also a computational and temporal advantage: we do not have to generate a data set containing multiple runs with different number of threads to then train the ML models

[5]This will be the subject of future works

level $k \in NT$ there is an associated time reduction r_k, inferred from the MIPLIB2017 benchmark as previously described; for the single core case we have $r_1 = 1$. We introduce another set of binary variables $\tilde{b}_k^a \in [0, 1]$ with $a \in A$ (the set of algorithms) and $p \in P$; the $\tilde{b}_k^a$ selects the number of threads for the selected algorithm. We introduce as well another continuous variable y_{ptime}^a representing the algorithm runtime potentially reduced due to parallelism (we remind the reader of the existence of the variable expressing the estimated algorithm runtime without parallelism, y_{time}^a – in practice, $y_{\text{time}}^a = y_{\text{ptime}}^a$ when $k = 1$). We then enforce the following constraints:

$$\sum_{k \in NT} \tilde{b}_k^a = 1 \qquad \forall a \in A \tag{8}$$

$$\tilde{b}_k^a = 1 \Rightarrow y_{\text{time}}^a = y_{\text{ptime}}^a r_k \qquad \forall a \in A, \forall k \in NT \tag{9}$$

Eq. 8 enforces the assignment of a specific number of cores/threads (k) for each algorithm through the binary variable $\tilde{b}_k^a$; the algorithm is selected using instead the b^a mentioned earlier. Eq. 9 instead serves to connect the runtime with no parallelism with the reduced runtime y_{ptime}^a obtained by using multiple cores; this is a set of relations encoded as indicator constraints. We assume that the memory and the solution quality are not affected by the number of threads.

3.3.3. User-specified Constraints and Objective Function

As already mentioned, the objective of the optimization model and the constraints bounding runtime, memory consumption and solution quality are provided by the users according to their needs. These constraints and objective guide the choice of the number of threads, the selection of the algorithm and its configuration, using the binary variables described in the previous subsections and the y_t^a variables representing the targets of the ML models. In addition to the ML models target we also have the (reduced) runtime obtained with different degrees of parallelism, y_{ptime}^a, thus we have an extended set of possible targets $T' = T \cup \{\text{ptime}\}$. The user-defined part of the optimization model is the following:

$$\min_{a \in A} b^a y_t^a \qquad t \in T' \tag{10}$$

$$y_u^a b^a \gtreqless C_u \qquad \forall u \in U \subseteq T', \forall a \in A \tag{11}$$

Eq. 10 is the objective of the optimization problem and is expressed as a minimization function (alternatively, maximisation problems can be formulated by minimizing the inverse of the target that needs to be maximised). The objective function aims at minimizing one target t among the set of currently allowed ones T'; different targets can be used to tackle different problems and use-cases (e.g., minimizing the algorithm runtime or maximizing the solution quality).

Eq. 11 represents the user constraints, which can assume the form of inequality or equality ($\gtreqless$) with respect to the user-specified bound C_u, where $u \in U$ indicates the target on which to impose the constraint[6], e.g., the memory. These constraints need to be enforced for each algorithm, bounding all potential $y_u^a b^a$ products. The model is flexible and accepts multiple constraints respecting the specified format, with the goal of enabling multiple use cases. For instance, a user might want to run the algorithm on a device with a limited amount of memory and obtain a solution within a certain time; in this situation, a set of inequality constraints will be added to the optimization model. Alternatively, the user might not want to impose any bound on time and memory usage but would rather like to obtain a solution of quality better than a baseline, thus imposing different bounds.

3.3.4. Decision Trees Encoding

DTs are embedded in the optimization model as a set of linear and non-linear constraints. For the embedding (that is, translating the ML models in a set of constraints) we employed the open-source EML library, *EMLlib*[7]. The details were amply provided in previous works describing EML methodology (e.g., [1]), thus we do not describe them here but we offer a brief summary. The interested reader can find an illustrative example in Appendix A. The key observation for embedding DTs is that in a DT each path from root to leaf can be viewed as a logical implication, and that the set of leaves labeled with a certain target specifies all and only the input configurations that should be mapped to such target value. Hence, a DT can be encoded as a collection of logical expressions[8] constraining that the class/target variable y of the DT takes the value v if at least one of the implications associated to the paths

[6]Note that $U \subseteq T'$

[7]The library can be found here `https://github.com/emlopt/emllib`

[8]If the solver used for the implementation does not support logical constraints, a reformulation through reified constraints is still possible.

labeled with v is true.

3.3.5. Increased Robustness

As described in details in Sec. 4.3, it is unrealistic to pretend ML models with perfect accuracy (see Sec. 4.2 as well), hence it must be expected that the solution provided by the proposed approach could be unfeasible if the embedded DTs produce inaccurate estimates for the targets. In practice, this means that the solution found (selected algorithm and configuration) might not respect the user-defined constraints when actually executed. There are multiple ways to cope with this intrinsic and unavoidable inaccuracy, ranging from iterative methods to more sophisticate training of ML models.

The technique adopted in our approach consists in *incorporating the uncertainty in the optimization model and making it more robust* with the usage of chance constraints [36]; we consider only individual chance constraints, as they are more computationally tractable and sufficient for our purpose. In practice, we specify the desired confidence level for the user-defined constraints taking into account the uncertainty of the ML model, modeled as a random variable associated to each user constraints. If we assume that the uncertainties are normally distributed, we can measure the uncertainty of each DTs by computing the prediction errors made over a validation set (a subset of the data described in Sec. 3.1 and not used for training the ML models), storing (for each DT) its standard deviation σ_t^a and p-quantile q_t^a, where a and t indicate, respectively, algorithm and target (as before). These quantities can be used to provide solutions more robust towards the ML models uncertainty, with the potential drawback of cutting feasible solutions very close to the infeasibilty boundary (by making conservative choices).

We can then modify the constraints expressed in Eq. 11 as follows:

$$y_u^a b^a \leq C_u - \sigma_u^a q_u^a \tag{12}$$

$$y_u^a b^a \geq C_u + \sigma_u^a q_u^a \tag{13}$$

$$C_u - \sigma_u^a q_u^a < y_u^a b^a < C_u + \sigma_u^a q_u^a \tag{14}$$

$\forall u \in U \subseteq T, \forall a \in A$. Equations 12, 13, 14 correspond (respectively) to the three types of allowed constraints, less or equal, greater or equal, and equality. Varying the "robustness" coefficient $\sigma_u^a q_u^a$ allows to modulate the trade-off between tight constraint satisfaction and solution quality.

3.3.6. Linearization of Quadratic Constraints

Eq.11 (and the more robust versions as well) belongs to the class of quadratic constraints, which are not naturally handled by most MILP solvers. Luckily, these quadratic constraints are multiplications of a continuous and a binary variable and can be linearized fairly easily, for instance by exploiting big-M methods. We start by denoting with $\mathrm{UB}(x)$ and $\mathrm{LB}(x)$ the upper and lower bound of variable x. To apply the following linearization we have to make an assumption on the upper and lower bounds for the y_t^a variables, namely $\mathrm{UB}(y_t^a) \geq 0,\ \mathrm{LB}(y_t^a) \geq 0 \quad \forall a \in A, \forall t \in T$; this is not a limitation for the set of regression targets considered in this paper, but we plan to extend in future versions of the model.

We directly consider the robust version of the user-defined constraints. Eq. 12 after linearization becomes:

$$y_u^a - \mathrm{UB}(y_u^a)(1 - b^a) \leq C_u - \sigma_u^a q_u^a \qquad \forall u \in U \subseteq T, \forall a \in A \tag{15}$$
$$\mathrm{LB}(y_u^a) b^a \leq C_u - \sigma_u^a q_u^a \qquad \forall u \in U \subseteq T, \forall a \in A \tag{16}$$

Linearizing Eq. 13 we obtain:

$$y_u^a - \mathrm{LB}(y_u^a)(1 - b^a) \geq C_u + \sigma_u^a q_u^a \qquad \forall u \in U \subseteq T, \forall a \in A \tag{17}$$
$$\mathrm{UB}(y_u^a) b^a \geq C_u + \sigma_u^a q_u^a \qquad \forall u \in U \subseteq T, \forall a \in A \tag{18}$$

Finally, the equality constraint is linearized as well (Eq. 14):

$$\mathrm{LB}(y_u^a) b^a \leq C_u - \sigma_u^a q_u^a \qquad \forall u \in U \subseteq T, \forall a \in A \tag{19}$$
$$\mathrm{UB}(y_u^a) b^a \geq C_u + \sigma_u^a q_u^a \qquad \forall u \in U \subseteq T, \forall a \in A \tag{20}$$
$$y_u^a - \mathrm{UB}(y_u^a)(1 - b^a) \leq C_u - \sigma_u^a q_u^a \qquad \forall u \in U \subseteq T, \forall a \in A \tag{21}$$
$$y_u^a - \mathrm{LB}(y_u^a)(1 - b^a) \geq C_u + \sigma_u^a q_u^a \qquad \forall u \in U \subseteq T, \forall a \in A \tag{22}$$

4. Experimental Evaluation & Validation

This section describes the results of the empirical analysis that we performed to evaluate the proposed approach. In the following, we first discuss the performance of ML techniques, then we analyze the solutions of the constrained optimization model. With this experimental evaluation we plan to answer three key questions:

- **RQ1** - *Can the relationship between hardware resources and algorithms performance be modelled via ML models?* The accuracy of the ML

models need to be considered, while taking into consideration the possible trade-off due to current limitations of the EML library (e.g., only some types of ML models are currently supported).

- **RQ2** - *Is the optimization model capable of answering the desired queries while being flexible and reusable?* We will demonstrate how the optimization results directly answer different queries such as suggesting the best hardware configuration needed to reach user-desired solution quality and finding the optimal algorithm and configuration (for the given problem).

- **RQ3** - *Do the solutions actually satisfy the user-defined bounds?* The optimization model provides a solution which satisfies the given constraints according to the internal ML model, which is not a perfect oracle, and thus might produce inaccurate estimates, which in turns can lead to solutions not respecting the desired constraints – this means that the robustness of the method must be validated.

4.1. ML Models Exploration

As a first experiment, we evaluated the performance of different ML techniques in the task of learning the relationship between the number of traces (and data instances) and the targets (computation time, required memory, and solution quality). In particular, six different ML model types were taken into account: (1) decision trees (DT), (2) random forests (RF), (3) neural networks (NN), (4) support vector machines (SVM), (5) ridge regression (RR), (6) gradient boosting (GB). For the evaluation of the ML models we simply split the whole data set into two subsets, the training set (used to train the ML models) and the test set (used to evaluate the generalization capability of the trained ML model); this simple strategy is sufficient to provide an indication of the ML models accuracy, which will be corroborated as well by the validation results reported in Sec. 4.3. The data set was split in training and test sets by randomly selecting data samples, 80% of the data for training and the remaining 20% for the test set. To simplify the comparison among different ML models, we normalize the input features of the training data. In particular, we transform features by scaling each input feature to a given range. Each feature is scaled and translated individually, such that it is in the given range on the training set – in particular we specified a range between zero and one. For additional details on the scaling procedure

(and the equations underlying the adopted transformation) we refer to the scikit-learn implementation[9].

Models were run by using the default parameters indicated in their original implementations[10]. The NN was implemented in Keras and Tensorflow, and it is composed by the input layer and the output layer (one neuron) and four hidden layers, each with 100 neurons.

In Table 1 we report the results obtained with the different ML models[11]. The first column represents the regression target; the second column is the model type. As for DT, we explored different versions: DT indicates a decision tree without bounding the depth (and using all default hyperparameters, as for the other ML methods), while DT_j indicates a decision tree with a maximum depth equal to j. Then, the table presents two triple columns, reporting R^2 coefficient, Mean Squared Error (MSE), and Explained Variance (EV); for each metric, we distinguish between the algorithms considered (CONTINGENCY and ANTICIPATE).

The results clearly show that the ML models have indeed learned the relationship between input feature and target. For all targets the best models are RFs, closely followed by DTs; clearly, decreasing the depth reduces the model accuracy. It can be also noted that other ML techniques tend to perform worse, with GB and NN outperforming SVMs and RR. Interestingly, memory and time seems to be very dependent on the number of traces – while the instance features do have an impact (see higher precision with full feature set), removing them leads to relatively contained accuracy decrease. Conversely, the solution cost appears to be tightly linked to the particular instance, as without including instance-related features all models underperform. These observations lead to two conclusions:

1. ML methods currently supported by EML, i.e., DT and NN obtain comparable results w.r.t. other ML techniques – accordingly, we will embed such methods in our optimization model;
2. for memory and time ML models it is sufficient to consider just the

[9] `https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html`; we employed the default parameters, and we speficied the [0,1] range.

[10] scikit-learn, `https://scikit-learn.org/stable/index.html`

[11] Scatter plots with predicted versus true targets (not included due to lack of space and relatively marginal added insights) can be found here `https://github.com/AndywinXp/HW_Dimensioning_for_OnlineAlgs/tree/main/graphs`

Target	ML Model	CONTINGENCY			ANTICIPATE		
		R^2	MSE	EV	R^2	MSE	EV
Memory	DT	0.998	0.0	0.998	0.997	0.0	0.997
Memory	DT_5	0.994	0.0	0.994	0.998	0.0	0.998
Memory	DT_{10}	0.998	0.0	0.998	0.998	0.0	0.998
Memory	DT_{15}	0.998	0.0	0.998	0.997	0.0	0.997
Memory	RF	0.998	0.0	0.998	0.998	0.0	0.998
Memory	RR	0.759	0.016	0.759	0.993	0.001	0.993
Memory	GB	0.999	0.0	0.999	0.998	0.0	0.998
Memory	SVM	0.918	0.005	0.919	0.974	0.002	0.978
Memory	NN	0.994	0.0	0.994	0.998	0.0	0.998
Cost	DT	0.983	0.0	0.983	1.0	0.0	1.0
Cost	DT_5	0.685	0.007	0.685	0.936	0.003	0.936
Cost	DT_{10}	0.93	0.002	0.93	0.996	0.0	0.996
Cost	DT_{15}	0.982	0.0	0.982	1.0	0.0	1.0
Cost	RF	0.987	0.0	0.987	1.0	0.0	1.0
Cost	RR	0.379	0.013	0.379	0.988	0.001	0.988
Cost	GB	0.88	0.003	0.881	0.999	0.0	0.999
Cost	SVM	0.743	0.006	0.749	0.953	0.002	0.953
Cost	NN	0.878	0.003	0.879	0.999	0.0	0.999
Time	DT	0.995	0.0	0.995	0.929	0.003	0.929
Time	DT_5	0.997	0.0	0.997	0.945	0.002	0.945
Time	DT_{10}	0.996	0.0	0.996	0.955	0.002	0.955
Time	DT_{15}	0.995	0.0	0.995	0.932	0.003	0.932
Time	RF	0.997	0.0	0.997	0.952	0.002	0.952
Time	RR	0.995	0.0	0.995	0.869	0.005	0.869
Time	GB	0.997	0.0	0.997	0.961	0.002	0.961
Time	SVM	0.955	0.004	0.978	0.899	0.004	0.901
Time	NN	0.994	0.0	0.997	0.932	0.003	0.934

Table 1: Performance comparison between the different ML models (using normalized data). MSE: Mean Squared Error; EV: Explained Variance.

number of traces as input, greatly reducing the complexity of their embedding (reduced number of constraints and variables) – instead, the ML model for the cost includes the complete set of instance features.

However, simply looking at the quantitative results reported on Table 1 can be slightly misleading. In fact, after embedding the ML models in the optimization framework we soon realized that, albeit accurate, the prediction errors distributions suffer from long tails, e.g., are characterized by few but very large errors.

Fig. 3 shows exactly the distribution of the prediction error (not normalized); the distribution are plotted in logarithmic scale to highlight the tails. We show the errors committed while predicting time and memory, distinguishing between ANTICIPATE and CONTINGENCY with different colors[12]. Two things can be observed: 1) significant tails can be observed and 2) the problem is much more significant for ANTICIPATE. These plots refer to the DTs with depth equal to 10 but further analysis revealed a very similar phenomenon for all considered ML models (only marginally mitigated by more complex

[12]On the GitHub repository additional plots (e.g., the cost) are available

ML models). This has an impact on the optimization model and it will be discussed in the next section.

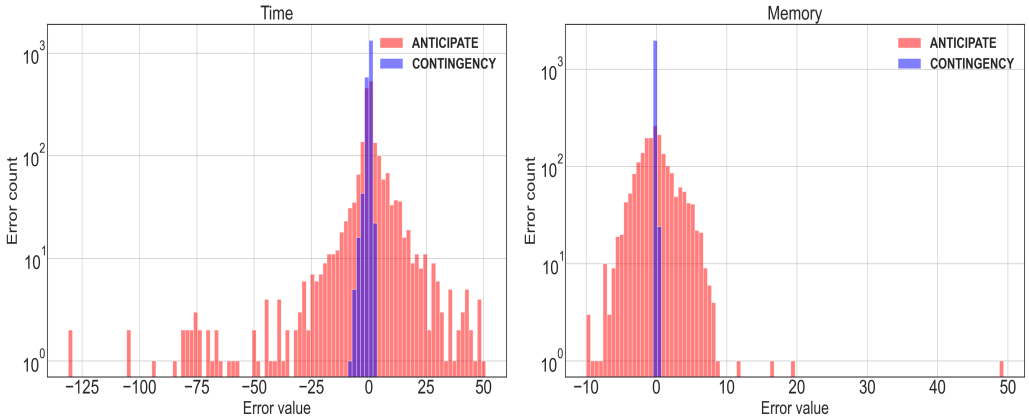

Figure 3: Error distribution for time (left) and memory (right) targets

Currently, only DTs and NNs are fully supported by EML (with RF implementation under development); the results show that DTs and NNs obtain comparable results w.r.t. other ML techniques. As DTs slightly outperform NNs (and tend to require considerable less computational effort to be encoded as a set of constraints) we opted to focus on DTs models for the rest of the paper; in particular, we consider DTs with depth equal to 10.

4.2. Optimization Results

In this section we consider the results of the optimization model with the embedded DTs; in this way we deal with **RQ2** and we demonstrate how the approach can be used to answer different types of queries. We first analyse the HW dimensioning task (Sec. 4.2.1) followed by the algorithm selection and configuration (Sec. 4.2.2). In Sec. 4.2.3 we study the impact of allowing the parallelism to be chosen by the optimization model; finally, Sec. 4.2.4 explore the possibility to improve the solution quality over a baseline provided by the user. We stress out again is that the different usage modes are obtained by posting different user-defined constraints and objective on top of the underlying optimization core constituted of data-driven and domain knowledge. We solved the optimization model using IBM ILOG CPLEX 12.8.0[37], via the Python API.

4.2.1. Hardware Dimensioning

In this section, we show the HW dimensioning experiment results. We answer the question: *Given an algorithm and constraints on its behaviour, what HW architecture should be used?* In this case we impose constraints on the run time and on the solution quality and we aim at minimizing the memory footprint of the algorithm. As we wanted to evaluate the different aspects of the model in a separate fashion (to better illustrate their characteristics) for this set of experiments we did not involve the domain knowledge and we set a number of threads equal to one; this means that no parallelism is considered (for the moment) – in this case $y^a_{\text{time}} = y^a_{\text{ptime}}$. We start by computing a *baseline* solution (cost) by exploiting a greedy heuristic. Such a heuristic is a state-of-the-art baseline for online optimization for efficient energy management [38]; we then constrain the solution found by HADA to be better (i.e., smaller) than the baseline by the desired amount – each input instance has its own baseline value.

Table 2 reports the experimental results for a selection of significant instances (whose ID is reported in the first column) and using the algorithm ANTICIPATE[13]. The second and third columns indicate the bound on the runtime (expressed in seconds) and on the solution quality; the fourth column is the optimal parameter chosen and the fifth column represents the actual HW dimensioning, that is the minimal amount of memory required to obtain the user-desired results. The constraint on the solution quality is expressed as the improvement over the baseline: a value equal to 0.1 indicates that the HADA has to find a solution with a solution cost 10% smaller than the baseline (lower costs imply higher quality solutions) and similarly 0.2 corresponds to a cost reduction of 20%.

From Table 2 we can clearly see that the required memory does not depend on the instance; the only factor impacting the minimal memory needed is the number of scenarios (the parameter governing the behaviour of ANTICIPATE). Without constraints on the runtime nor the solution cost the number of scenarios is kept at its lowest value (i.e., 1), as this guarantees the minimal amount of memory required. When the user requests higher quality solutions

[13]The full set of experiments (all instances and both algorithms) can be found here `https://github.com/AndywinXp/HW_Dimensioning_for_OnlineAlgs/tree/main/experimental_results`. This repository contains the full set of experimental results synthetically reported in the following sections.

InstanceID	Time Bound (s)	Sol. Improvement	nScenarios	memAvg(MB)
2	no	no	1.0	60.12
5	no	no	1.0	60.12
15	no	no	1.0	60.12
19	no	no	1.0	60.12
28	no	no	1.0	60.12
2	no	0.1	2.0	65.98
5	no	0.1	2.0	65.98
15	no	0.1	2.0	65.98
19	no	0.1	2.0	65.98
28	no	0.1	2.0	65.98
2	no	0.2	35.0	176.83
5	no	0.2	37.0	179.17
15	no	0.2	no_sol	no_sol
19	no	0.2	34.0	174.66
28	no	0.2	34.0	174.66
2	60	no	1.0	60.12
5	60	no	1.0	60.12
15	60	no	1.0	60.12
19	60	no	1.0	60.12
28	60	no	1.0	60.12
2	60	0.1	2.0	65.98
5	60	0.1	2.0	65.98
15	60	0.1	no_sol	no_sol
19	60	0.1	2.0	65.98
28	60	0.1	2.0	65.98
2	60	0.2	no_sol	no_sol
5	60	0.2	no_sol	no_sol
15	60	0.2	no_sol	no_sol
19	60	0.2	no_sol	no_sol
28	60	0.2	no_sol	no_sol
2	120	no	1.0	60.12
5	120	no	1.0	60.12
15	120	no	1.0	60.12
19	120	no	1.0	60.12
28	120	no	1.0	60.12
2	120	0.1	2.0	65.98
5	120	0.1	2.0	65.98
15	120	0.1	2.0	65.98
19	120	0.1	2.0	65.98
28	120	0.1	2.0	65.98
2	120	0.2	35.0	176.83
5	120	0.2	no_sol	no_sol
15	120	0.2	no_sol	no_sol
19	120	0.2	34.0	174.66
28	120	0.2	34.0	174.66
2	300	no	1.0	60.12
5	300	no	1.0	60.12
15	300	no	1.0	60.12
19	300	no	1.0	60.12
28	300	no	1.0	60.12
2	300	0.1	2.0	65.98
5	300	0.1	2.0	65.98
15	300	0.1	2.0	65.98
19	300	0.1	2.0	65.98
28	300	0.1	2.0	65.98
2	300	0.2	35.0	176.83
5	300	0.2	37.0	179.17
15	300	0.2	no_sol	no_sol
19	300	0.2	34.0	174.66
28	300	0.2	34.0	174.66

Table 2: HW dimensioning experiments on a subset of instances for algorithm ANTICIPATE

the number of scenarios must increase, and the memory required increases as well – this is a clear indication for guiding the HW dimensioning. For some instances, such as #15, it is simply impossible to obtain the desired level of improvement, even without any bound on the runtime – the solution cost is strictly dependent on the input instance. The constraint on the runtime does not directly impact the memory, as these two aspects are only indirectly correlated through the mediation of the number of scenarios: when the number of scenarios increases, both memory and runtime increase, albeit with different gradients. However, imposing tighter runtime bounds has an unavoidable impact on the results, as a runtime bound forces the number of scenarios not to be greater than a certain value. This "implicit" constraint on the algorithm parameters might prevent the "explicit" constraint on the solution cost to be satisfied; this is what happens in the case of runtime bounded at 60 seconds and 20% required improvement over the baseline: HADA cannot find any solution as there exist no HW and algorithm configuration capable of satisfying both constraints at the same time.

Approach Complexity. It is important to stress as well the computational effort required to create and use the overall approach (ML + Optimization). For this reason, in Table 3 we report the cumulative time needed to train and embed the six DTs used for the trials in Table 2; we also report the mean time spent by the optimizer to find a solution; finally, we report the number of variables and constraints present in the optimization model including those automatically generated by the embedding of the DTs, but discarding including user-defined constraints. The training of the ML models is very fast (as expected with quite shallow DTs); the solving time is very short as well, while the most time-consuming phase is the embedding of the ML models. Luckily, this is an operation that must be performed only once, after the training of the ML models themselves; afterwards, *the encoded models can be reused multiple times, on different input instances and user-defined constraints.* Table 3 provides a synthetic overview of the temporal complexity of the proposed approach, not considering the time needed to create the training set by benchmarking the target algorithms (see Sec. 3.1).

Additionally, we can provide some details on the algorithm complexity as well. First, we must separate three main phases constituting HADA:

1. data set collection - an initial phase to collect the data set by running multiple times the target algorithms, under different configurations;

Training Time	Embedding Time	Mean Solve Time	#Constraints	#Variables
0.115	98.49	1.09	17660	1928

Table 3: Statistics for the DTs used in the trials of Table 2; all times are measured in seconds

2. surrogate model creation - once a training set is available, a set of ML models is then trained on such data (in the paper Experimental section – Sec. 4 – we consider 6 DTs), and then these models are encoded as a set of variables and constraints following EML paradigm;
3. optimization – post the user-defined constraints and objective function on top of the combinatorial structure formed by the encoded ML models and the domain-knowledge constraints, and finally solve the optimization model (either until an optimal solution or a time limit is reached.

The first and second phases need to be performed only once, as the core combinatorial structure can be reused. The complexity of the first phase is entirely dependent on the algorithmic complexity of the target algorithm (ANTICIPATE and CONTINGENCY), as it merely consists of running said algorithm under multiple configurations. The complexity of the second phase depends on the choice of the ML model (we remind the reader that the current version of EML handles either DTs and NNs), and on the specific training algorithm. In the experimental evaluation we consider DTs and their implementation with the Scikit-learn Python module which employs the CART algorithm; as generally speaking, finding an optimal DT is a NP-hard problem[39], heuristic or greedy algorithms are typically used, such as CART – we refer to [40] for details. Finally, solving mixed-integer linear programming problems (as the optimization model considered in this paper) is notoriously NP-Hard, and the complexity strongly depends on the number of variables and constraints in the model and the specific "hardness" of the instance. In our experiments we solve the optimization problem using CPLEX, which is based on a heavily optimized algorithm belonging to the general class of Branch and Cut[41]; we refer to the CPLEX documentation page for further details[14].

[14] `https://www.ibm.com/docs/en/icos`

4.2.2. Algorithm Selection and Configuration under user-defined constraints

As noted, our approach can be used as well to select the best algorithm and configuration for the user-defined constraints. In this case, we specialize the model described in Sec. 3.3 by minimizing the solution cost while bounding the algorithm runtime under the user-defined limit C_{time} and not to exceed the allowed memory consumption C_{mem}. The core of the model is still defined by the constraints embedding the DTs and connecting time and memory with the algorithm and its configuration. Again, we fix the parallelism to one.

As described in the previous section (see Fig. 3), there is an unavoidable inaccuracy in the prediction made by the embedded ML models, hence the robust constraints (Eq.12, 13, 14). As σ_u^a is very large the optimization models become *too conservative* and fail to produce solutions except for the laxer constraints. We cope with this by computing σ_u^a and q_u^a after having excluded the long tail from the error distribution, i.e., we first remove errors with absolute value larger than 20 (threshold empirically obtained). In this way we trade some robustness for a decreased risk of pruning feasible solutions.

To evaluate the optimization model we created a set composed by 30 instances not seen during the training of the DTs; for each instance, we used our approach to select algorithm and configuration for 30 different combinations of user-defined bounds (on both memory and time)[15], while minimizing the solution cost; we thus created a 900 elements set. Table 4 reports the results obtained on the same representative subset of instances considered previously; the second and third column show whether the trial has been executed with a "$\leq$" type bound (and with which value) on memory and/or time; the fourth and fifth columns show which algorithm has been chosen for the optimization ("A" for ANTICIPATE and "C" for CONTINGENCY) and therefore which is the optimal value for the corresponding parameter ("nTraces" for CONTINGENCY, "nScenarios" for ANTICIPATE); finally, the last three columns on the right show the value of the regression targets, as estimated by the embedded ML models given the optimizer-selected algorithm and configuration.

Table 4 shows that in the unbounded case ANTICIPATE is preferred with the largest value of the configurable parameter (i.e., 100) for most of the instances. This does not hold for instance #2 which reaches a plateau in the solution quality with the configurable parameter lower than 100. In this case,

[15]The bound values were selected after an empirical analysis

InstanceID	Time Bound (s)	Memory Bound	Algorithm Chosen	Parameter Chosen	time(sec)	memAvg(MB)	sol(keuro)
2	no	no	A	98	199.37	331.47	267.01
5	no	no	A	100	206.34	333.93	280.61
15	no	no	A	100	206.34	333.93	275.63
19	no	no	A	100	206.34	333.93	259.69
28	no	no	A	100	206.34	333.93	259.69
2	60	100	C	30	23.27	86.18	310.11
5	60	100	C	11	11.85	83.95	310.75
15	60	100	A	6	8.47	89.70	366.38
19	60	100	C	30	23.27	86.18	310.11
28	60	100	C	11	11.85	83.95	321.49
2	60	200	C	30	23.27	86.18	310.11
5	60	200	C	11	11.85	83.95	310.75
15	60	200	A	9	13.03	101.99	366.38
19	60	200	C	13	13.08	84.17	310.11
28	60	200	C	11	11.85	83.95	321.49
2	60	300	C	30	23.27	86.18	310.11
5	60	300	C	11	11.85	83.95	310.75
15	60	300	A	9	13.03	101.99	366.38
19	60	300	C	13	13.08	84.17	310.11
28	60	300	C	11	11.85	83.95	321.49
2	120	100	C	100	65.05	88.94	307.33
5	120	100	C	100	65.05	88.94	300.27
15	120	100	C	100	65.05	88.94	361.21
19	120	100	C	30	23.27	86.18	310.11
28	120	100	C	30	23.27	86.18	321.49
2	120	200	C	100	65.05	88.94	307.33
5	120	200	C	90	59.47	88.36	300.27
15	120	200	A	35	64.83	176.83	333.53
19	120	200	C	28	22.08	85.97	310.11
28	120	200	C	11	11.85	83.95	321.49
2	120	300	C	100	65.05	88.94	307.33
5	120	300	C	100	65.05	88.94	300.27
15	120	300	A	35	64.83	176.83	333.53
19	120	300	C	28	22.08	85.97	310.11
28	120	300	C	29	22.67	86.09	321.49
2	300	100	C	100	65.05	88.94	307.33
5	300	100	C	100	65.05	88.94	300.27
15	300	100	C	100	65.05	88.94	361.21
19	300	100	C	30	23.27	86.18	310.11
28	300	100	C	30	23.27	86.18	321.49
2	300	200	C	100	65.05	88.94	307.33
5	300	200	C	100	65.05	88.94	300.27
15	300	200	A	40	80.34	187.96	329.14
19	300	200	C	13	13.08	84.17	310.11
28	300	200	A	40	80.34	187.96	317.27
2	300	300	A	79	158.54	288.77	285.33
5	300	300	A	79	158.54	288.77	298.59
15	300	300	A	77	168.78	282.42	293.84
19	300	300	A	79	158.54	288.77	279.52
28	300	300	A	79	158.54	288.77	279.52

Table 4: Algorithm selection and configuration experiments conducted on a subset of instances, with increased robustness (with q_u^a from Equations 12-14 set to the 90-th percentile of the prediction error for the DTs)

our model is able to correctly configure it to 98. Differently, when a memory bound ranging from 100 to 200 is set, CONTINGENCY is preferred: in general, it presents high quality solutions with a much lower value of the parameter configuration (w.r.t. ANTICIPATE in the same setting). Some instances, e.g.

#15, tend to always be solved with ANTICIPATE even in the most stringent cases of bounds. This is probably due to the need of exploring more the available HW resources to try to reach the maximum possible solution quality for the considered instance. Finally, in the case of larger constraints (i.e., memory and time with bound equal to 300) the model selects ANTICIPATE again for all the instances, as in the unbounded case.

4.2.3. Increasing the parallelism

In this section, we study the effect of the domain knowledge, and precisely we conduct experiments where we let the optimizer decide the optimal number of threads to run the algorithms. Also in this case we fix the algorithm but let the algorithms parameter to be configured by the optimization model, in addition to the number of cores. We consider again a series of runtime constraints; no other user constraints are imposed as in the current approach the number of threads may impact only the runtime. The objective function is the minimization of the solution cost and of the number of threads. In this case, the output provided by HADA is the number of cores and the memory (the HW dimensioning), and the algorithm parameter.

Figure 4 reports the experimental results for both ANTICIPATE and CONTINGENCY algorithms. For each algorithm there are three sub-figures, each portraying the runtime bound (expressed in seconds) in the x-axis and with, respectively, number of cores, algorithm parameter value (number of scenarios or number of traces), and memory on the y-axis. The runtime bound is tighter closer to the origin. The box plots are used to show the results distribution for all the considered data instances; the combinations of instance and runtime bound where no solution was returned are not included in the figure. In the case of ANTICIPATE we can notice how the variability among the data instances is very reduced, as highlighted by the very narrow boxes. With more stringent constraints on the runtime the number of scenarios and the required memory are forced to be as low as possible; when the runtime is allowed to grow, the memory usage rises as well, together with the number of scenarios, as this leads to better solutions (we are minimizing the solution cost). Conversely, the number of cores seems not to be directly correlated with the temporal constraint; this is due to the fact that ANTICIPATE with a low number of scenarios tend to be fast enough not to require additional parallelization via multiple cores. With middle values of runtime bounds (between 20 and 100 seconds) the situation is more nuanced and a higher number of threads brings benefits by allowing large values for the number of

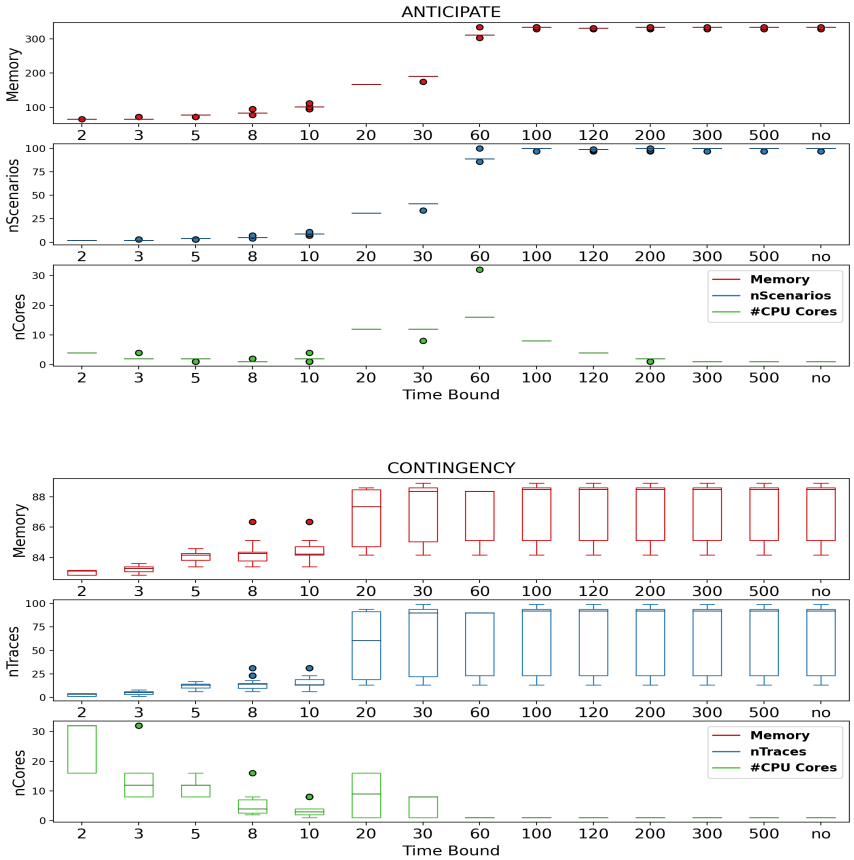

Figure 4: Parallelism results for ANTICIPATE (top) and CONTINGENCY (bottom). The three figures for each algorithm report the number of cores, the number of scenarios and the memory chosen by HADA under different runtime constraints (expressed in seconds).

scenarios, which would incur in too high runtime without parallelization.

The behaviour of CONTINGENCY is different. First, there is more variability among the instances, especially in terms of number of traces and parallelism; the large boxes for the memory are partially misleading as for the memory the overall variation is very narrow (much narrower than ANTICIPATE). Second, in this case there is a clear gain in using a higher number of threads with tighter constraints on the runtime. This is due to the comparatively minor influence of the number of traces on the memory consumption; in the case of CONTINGENCY, the relationship between algorithm runtime and configuration dominates the algorithm behaviour – this suggests HADA to increase the

parallelism with tight time constraints. Similarly to the ANTICIPATE case, also with CONTINGENCY the number of traces and memory consumption tend to rise when the time bound is relaxed. The different behaviour of the algorithms is due to the different impact of data instances on the runtime, and on the different relationships between the variables considered (runtime, memory, solution cost); thanks to the empirical knowledge encoded in the ML models HADA is capable to flexibly handle both (and potentially many more) algorithms.

4.2.4. Comparison with Baseline

Now we illustrate a different type of query: *given a bound on the solution quality, provide indications about the required hardware resources and expected runtime*. We again fix the number of threads to one. We employed the same solution baseline described in Sec.4.2.1. Next, we introduce a user-defined constraint regarding the desired improvement w.r.t. the baseline solution; the objective function is the minimization of the runtime. In other terms, we impose the optimization model to provide (if it is feasible) the best algorithm and its configuration to obtain the required solution improvements over the baseline. This allows the user to know in advance which hardware is required to obtain the desired solution, together with an estimated run-time. We stress out that these information are provided *for new and unseen instances, without executing the online algorithm itself*, as the knowledge about the cost-memory-time-configuration relationships is encoded in the ML models.

In order to show the capability of this formulation, we impose the desired improvement expressed as a percentage w.r.t. the baseline solution. In particular, we set 2% as minimum improvement, and then we impose larger improvements (increasing by 2-3% each time: 2%, 5%, 7%, 10% and so on), until the model finds no solution. The results of this experiment are reported in Figure 5. For the sake of clarity, we only report the results for five different instances, selected as they represented broader instance types. For each plot, the x-axis reports the normalized requested improvement, e.g., 2% corresponds to 0.02, 5% to 0.05, etc. The y-axis reports the memory increase (left) and time increase (right). In this case, memory increase is calculated as the ratio between the memory employed by the algorithm selected and configured by the optimizer (either CONTINGENCY or ANTICIPATE) and the memory employed by the baseline. The same calculation is applied to obtain time increase.

The analysis of Figure 5 shows a partially heterogeneous trend, with

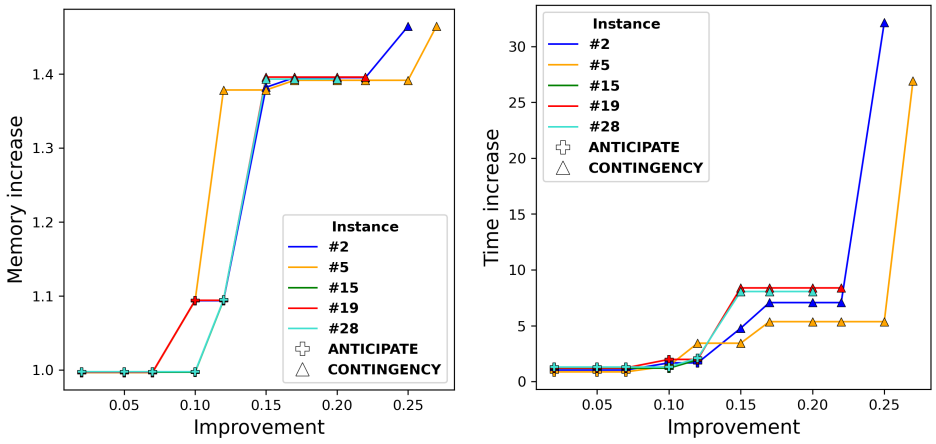

Figure 5: Baseline comparison results: on the left the memory increases as the optimization model is forced to improve the algorithm solution cost with respect to the baseline, on the right the increase in runtime is reported; memory and time increase are normalized over the baseline values. Five representative instances were chosen.

slightly different behaviour for the different instances. However, a common trend is very clear, that is improving the solution of the baseline implies larger memory consumption and longer runtime. In addition, it can be noted how the magnitude of time increases significantly surpasses the memory ones; this suggests that the considered algorithms are dominated by sequential phases. As shown by our empirical evaluation, the relation between quality of the solution and computational requirements and configurations is often hard to formalize, and this finally confirms the intuition behind this work. Thus, approaches for vertical matching can support users for automatic hardware dimensioning and algorithm configuration in real-world applications given a set of constraints (solution time, available hardware, solution cost, etc.).

4.3. Validation Results

After having analysed the ML models and the results of the optimization model, we need to assess the robustness of the approach. This will answer **RQ3**: does the solution provided (e.g., the chosen algorithm and configuration) actually respect the user-defined constraints? As described earlier, our method takes into account the implicit inaccuracy of ML models (see Sec. 3.3.5 and, in particular, Equations 12-14) and now we plan to examine its robustness.

For each element of the set described in the previous section we ran the selected algorithm with the suggested configuration, recording the actual

memory consumption, run time, and solution cost. This operation was repeated using different robustness levels, the $\sigma_u^a q_u^a$ from Eq. 12 (we are considering only $\leq$ constraints). We will now report the aggregated results, but all validation sets can be found on the online repository already mentioned in Sec. 4.1[16].

Robustness Coefficient	Avg. Time Bound Compliance	Avg. Mem. Bound Compliance	% Satisfied Instances
No robustness	-0.37	-0.27	96.57
$q_u^a = 90$	-0.48	-0.29	100
$q_u^a = 95$	-0.49	-0.30	100

Table 5: Validation results computed on the entire validation set, for the algorithm selection & configuration use case. The results are aggregated over 30 validation instances and on 30 different combinations of bounds applied on time and memory (hence the entire set is composed by 900 elements). The avg. compliance measures the degree of bound satisfaction; the % of satisfied is computed over the 900 elements set.

Table 5 reports the validation results; each row corresponds to a different robustness level, starting from no robustness coefficient $\sigma_u^a q_u^a$ applied, to more and more robust approaches, with q_u^a equal to the 90-th and 95-th percentiles (σ_u^a is the standard deviation and is kept fixed). The second and third columns report the level of user-defined constraint compliance for the time and memory constraint, respectively. The compliance is measured as $\frac{\text{actual}-\text{desired}}{\text{desired}}$, where actual is the actual value obtained by running the chosen algorithm with the selected configuration and desired is the user-defined bound. The number reported is averaged over all instances and bound combinations. Positive values would indicate that, on average, the corresponding bound is not respected, while negative values indicate otherwise (hence they are preferable). The fourth column reports the percentage of instances and combinations where the bounds were respected (out of the 900 possible elements).

It can be easily seen that all approaches tend to amply satisfy the constraints, as demonstrated by the negative values of the bound compliance, and in particular for the more robust cases. However, the non-robust approach fails to satisfy the constraints in slightly more than the 3% of the cases; this problem is entirely solved by the more robust approaches. We then decided to

[16]In particular, here: `https://github.com/AndywinXp/HW_Dimensioning_for_OnlineAlgs/tree/main/validation_sets`

investigate in more detail the combinations where the bounds are not satisfied, and it turns out that the key culprit are the 30 validation instances which were subjected to a run time bound of 200 seconds and no constraint on the memory consumption. Figure 6 reports the results with that temporal bound for 5 representative instances (the same of Tab. 4); for each instance we plot a bar with a different color, which indicates the robustness level (same values as Tab. 5), while the selected algorithm is represented with the hatch mark. On the left there are the actual times obtained while running the selected algorithm. The horizontal line indicates the user-defined bound. When the bar is above the horizontal line, it means that the solution does not meet the user-defined bound. On the right the normalized solution quality is reported, measured as the solution cost obtained without any bound on the runtime divided by solution cost with the bound. The optimal ratio is equal to 1 (unbound cost equal bounded cost) as smaller solution costs indicate higher quality; by bounding the runtime the solution cost might increase (worse solution), hence the ratio might become smaller than one (the denominator becomes larger than the numerator).

This figure serves also as an indication of the trade-off between enforcing constraints in a more robust manner and the solution quality. Indeed, more conservative approaches tend to produce solutions with higher cost. We can clearly see that the time bound of 200 seconds is not respected with the non-robust approach, since all the instances exceed the bound to achieve maximum possible solution quality. This problem is completely solved with the two robust approaches. For example, by analyzing instance #2, we can observe that the robust approach with $q_u^a = 90$ maintains the same choice (i.e. ANTICIPATE) of the non-robust approach by correctly adjusting the parameter configuration, while the robust approach with $q_u^a = 95$ also changes the selected algorithm to be more conservative in terms of run time. It is important to stress that this robust approach is able to explore the possibilities of achieving solution quality results very close to ANTICIPATE (in the same setting) but with a dramatically smaller online computational cost. We recall that CONTINGENCY is an integrated offline/online algorithm that builds (offline) a pool of solutions, called traces, that can then guide an efficient (fast) online method. This result confirms that, often, a tighter offline/online integration can lead to substantial improvements in terms of solution/time trade-off. Accordingly, our approach can be useful to define a set of guidelines in order to choose the most suitable algorithm for the considered setting.

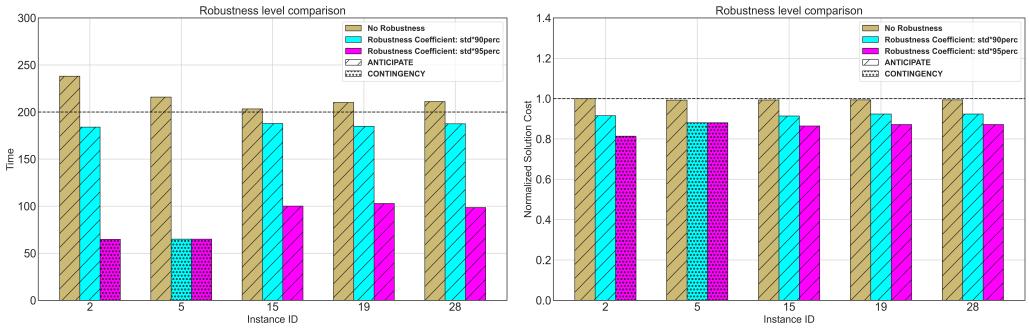

Figure 6: Trade-off between robustness and solution quality on 5 problematic validation instances. On the left there is actual solution time with the different level of robustness (the horizontal line indicates the user-defined bound); on the right, the normalized solution quality is reported (larger values indicate solutions closer to the baseline, i.e., without enforcing temporal bounds)

Broadly speaking, these validation results are extremely promising, as they demonstrate that the approach is very robust even without taking in consideration explicit corrective measures. If we take corrective actions (Eq. 12, 13, and 14) the constraints are satisfied in every case, albeit with the unavoidable risk of producing lower quality solutions. In future works, we plan to better explore the trade-off between robustness and solution quality, via improvement to the ML models and more sophisticated objective functions for the optimization model.

Finally, in Table 6 we report the validation results for the improvement over baseline experiment. Differently from the algorithm selection & configuration use case where the bounds are imposed on run time and memory consumption while minimizing solution cost, here we are constraining the solution cost – a harder challenge from the robustness point of view as the ML models with solution cost as target are inherently more uncertain, due to the more complex relationship between algorithms hyperparameter and data instance and solution quality. Table 6 is analogous to Tab. 5: the first column indicates the level of robustness obtained withe the chance constraints (see Sec. 3.3.5 for the details), where q_u^a is the p-quantile and σ_u^a the standard deviation; the second column reports the level of compliance with the constraint on the solution cost (is the solution cost found by EML actually smaller than the desired baseline?), defined as for the previous use case; the third column is the percentage of instances where the bounds were respected (out of the 900 possible elements).

We remind the reader that negative values for the bound compliance are to

be favoured (positive value indicate that the bound is not satisfied). Looking a the table is it easy to see that, broadly speaking, the EML approach does indeed satisfy the imposed constraint – as highlighted by the negative values for the compliance column; however, the EML approach is still not perfect, as in few instances . This happens since even with increased robustness is very difficult to impose correct bounds when the prediction of the ML model is significantly off. This is especially a problem in the solution cost case as it is a harder-to-evaluate target thane the other ones; the solution cost strongly depends on the particular instance according to a complex relationship, and the ML models struggle to fully capture it. This issue is exacerbated by the fact that we cannot use arbitrarily complicated ML models for the prediction task, as we must embed them in the optimization model, hence we have a cap on their accuracy – in future works we plan to extend the EML library in order to accommodate more complicated ML models, thus limiting this issue. However, we must note that in the vast majority of instances and bound combinations our approach is capable to obtain the desired solution improvement.

Robustness Coefficient	Avg. Baseline Solution Bound Compliance	% Satisfied Instances (Baseline)
No robustness	-0.034	93.30
$\sigma_u^a = 0.5, q_u^a = 90$	-0.035	93.60
$\sigma_u^a = 0.5, q_u^a = 95$	-0.035	94.31
$\sigma_u^a = 1, q_u^a = 90$	-0.035	94.73
$\sigma_u^a = 1, q_u^a = 95$	-0.042	94.77

Table 6: Validation results computed on the entire validation set, for the improvement over the baseline use case . The results are aggregated over 30 validation instances and on 30 different combinations of bounds applied on time and memory (hence the entire set is composed by 900 elements). The avg. compliance measures the degree of bound satisfaction; the % of satisfied is computed over the 900 elements set.

4.4. Discussion

Our empirical evaluation demonstrates the feasibility and flexibility of the proposed approach. In particular, HADA has been thoroughly tested on two different algorithms (ANTICIPATE and CONTINGENCY) from the energy domain, where they represent the state-of-the-art (see [4]). This test phase was devised to answer crucial questions to assess the usability of HADA in a real application domain; we elected energy systems as it is well-known that

finding optimal configurations for the algorithms used in this field is very complex [42].

In order to properly evaluate the experimental results, we need first to clarify some points about comparison between HADA and the state of the art. As mentioned in 2, since the related techniques have a different focus w.r.t. HADA, a direct comparison with them is not totally fair. Indeed, to the best of our knowledge, HADA represents a first attempt in the direction of defining a general and automated decision support tool to match a given AI algorithm, user-defined constraints and HW resource dimensioning. With this in mind, we can summarize the following points:

1. We show in our results (see 4.2.1 and 4.2.2) that HADA is able to find the optimal configuration for a target algorithm treated as a black-box exposing a set of inputs and tunable parameters. We can further emphasize that: (1) w.r.t. black-box optimization approaches (see 2), HADA is able to handle constraints in the optimization part, and it can also tackle both continuous and mixed-variable problems. In addition, HADA can uncouple the learning part from the optimization one: once trained, the ML models can be reused on different data instances and different user-defined constraints, as they are posted on the backbone of the optimization model when needed. *This suggests a great potential for the application of integrated (learning and optimization) methods in this field.* (2) w.r.t. algorithm selection and configuration approaches (see 2), we consider methods that are competitive in finding optimal parameter configurations and good approximation of the target function for a given algorithm. However, they do not automatically consider constraint-based combinations of hardware dimensioning, computation time and solution quality requirements, as HADA. The main advantage of HADA is the possibility to naturally express and enforce constraints as allowed by the flexible mathematical programming formulation, and to easily change the objective function: this means that the same model can be used for a series of different queries.
2. We can also highlight the benefits brought by HADA in the specific energy domain, from which the target algorithms were selected. Considerable research effort has focused on improving the efficiency of online algorithms [3, 34]: many works focused on tuning the cost/quality trade-off by adjusting the number of scenarios and the look-ahead horizon. Currently, finding the right hardware and parameter configurations

for ANTICIPATE and CONTINGENCY is done manually by researchers, a decidedly non-trivial task. The results clearly show (see 4.3) that HADA is able to overcome this limitation thanks to its approach merging data-driven and optimization models.

5. Conclusion

In this paper, we propose an approach combining learning and optimization to automatically perform HW dimensioning and configuration for online algorithms in the energy system domain, under an heterogeneous set of constraints. We rely on the integration of ML models within an optimization problem, following the Empirical Model Learning paradigm. ML is used to predict the online algorithms performance on different HW configurations and optimization is used to find the optimal matching of computing resources and algorithm configuration, while respecting user-defined constraints (e.g., cost, time, solution quality). We evaluate the proposed approach on two different online algorithms grounded on an energy management system, and a single HW resource (RAM memory). As shown in the experiments, our approach is flexible and robust, as demonstrated by the validation on unseen instances.

In future works, we will extend the approach to deal with different target algorithms and domains, and we will consider heterogeneous HW resources. This will require to benchmark multiple algorithms on different architectures in order to obtain accurate ML models, which could then be embedded in the optimization problem for HW dimensioning and algorithm fine-tuning. In this direction, active and transfer learning strategies will be explored to reduce the size of the required training set (e.g., [31]), or domain knowledge could be injected in the ML models (e.g., [43]), for instance to improve their explainability and causability[44]. Finally, we plan to tackle more directly the unavoidable uncertainty implicit in the ML models, which we now address through chance constraints. In particular, we will train probabilistic ML models capable to produce distributions as output, rather than simple scalar; this will allow for a finer-grained control on the robustness trade-off, as we could associate an confidence interval to each prediction, while we are currently applying the same uncertainty measure, measured using a validation set, to all ML predictions.

Acknowledgements

This work has been partially supported by European ICT-48-2020 Project TAILOR (g.a. 952215) and EU Horizon 2020 Project StairwAI (g.a. 101017142).

References

[1] M. Lombardi, M. Milano, A. Bartolini, Empirical decision model learning, Artificial Intelligence 244 (2017) 343–367.

[2] M. Lombardi, M. Milano, Boosting combinatorial problem modeling with machine learning, in: Proceedings IJCAI, 2018, pp. 5472–5478.

[3] L. Mercier, P. Van Hentenryck, Performance analysis of online anticipatory algorithms for large multistage stochastic integer programs., in: IJCAI, 2007, pp. 1979–1984.

[4] A. De Filippo, M. Lombardi, M. Milano, How to tame your anticipatory algorithm, in: IJCAI, 2019, pp. 1071–1077.

[5] W. Van Ranst, Real world applications of artificial intelligence on constrained hardware, Ph.D. thesis, PhD thesis, Computer Science Technology TC, De Nayer (2019).

[6] M. A. Talib, S. Majzoub, Q. Nasir, D. Jamal, A systematic literature review on hardware implementation of artificial intelligence algorithms, The Journal of Supercomputing 77 (2021) 1897–1938.

[7] B. Bischl, P. Kerschke, L. Kotthoff, M. Lindauer, Y. Malitsky, A. Fréchette, H. Hoos, F. Hutter, K. Leyton-Brown, K. Tierney, et al., Aslib: A benchmark library for algorithm selection, Artificial Intelligence 237 (2016) 41–58.

[8] K. Eggensperger, F. Hutter, H. H. Hoos, K. Leyton-Brown, Efficient benchmarking of hyperparameter optimizers via surrogates, AAAI'15, AAAI Press, 2015, p. 1114–1120.

[9] P. Bouvry, Matching next-gen hpc with target applications, in: 2019 16th International Joint Conference on Computer Science and Software Engineering (JCSSE), IEEE, 2019, pp. XXV–XXV.

[10] O. W. Saastad, K. Kapanova, S. Markov, C. Morales, A. Shamakina, N. Johnson, E. Krishnasamy, S. Varrette, H. Shoukourian, Prace best practice guide 2020: Modern processors, Tech. rep., PRACE aisbl (2020).

[11] M. Gagliolo, V. Zhumatiy, J. Schmidhuber, Adaptive online time allocation to search algorithms, in: J.-F. Boulicaut, F. Esposito, F. Giannotti, D. Pedreschi (Eds.), Machine Learning: ECML 2004, Springer Berlin Heidelberg, Berlin, Heidelberg, 2004, pp. 134–143.

[12] A. E. Howe, E. Dahlman, C. Hansen, M. Scheetz, A. von Mayrhauser, Exploiting competitive planner performance, in: S. Biundo, M. Fox (Eds.), Recent Advances in AI Planning, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 62–72.

[13] M. de Prado, N. Pazos, L. Benini, Learning to infer: Rl-based search for dnn primitive selection on heterogeneous embedded systems, in: 2019 Design, Automation & Test in Europe Conference & Exhibition (DATE), IEEE, 2019, pp. 1409–1414.

[14] A. Costa, G. Nannicini, Rbfopt: an open-source library for black-box optimization with costly function evaluations, Mathematical Programming Computation 10 (4) (2018) 597–629.

[15] J. Snoek, H. Larochelle, R. P. Adams, Practical bayesian optimization of machine learning algorithms, Advances in neural information processing systems 25 (2012) 2951–2959.

[16] P. I. Frazier, A tutorial on bayesian optimization, preprint arXiv:1807.02811 (2018).

[17] T. T. Joy, S. Rana, S. Gupta, S. Venkatesh, Fast hyperparameter tuning using bayesian optimization with directional derivatives, Knowledge-Based Systems 205 (2020) 106247.

[18] J. R. Gardner, M. J. Kusner, Z. E. Xu, K. Q. Weinberger, J. P. Cunningham, Bayesian optimization with inequality constraints., in: ICML, Vol. 2014, 2014, pp. 937–945.

[19] J. M. Hernandez-Lobato, M. Gelbart, M. Hoffman, R. Adams, Z. Ghahramani, Predictive entropy search for bayesian optimization with unknown

constraints, in: F. Bach, D. Blei (Eds.), Proceedings of the 32nd International Conference on Machine Learning, Vol. 37 of Proceedings of Machine Learning Research, PMLR, Lille, France, 2015, pp. 1699–1707. URL https://proceedings.mlr.press/v37/hernandez-lobatob15.html

[20] D. Eriksson, M. Poloczek, Scalable constrained bayesian optimization, in: A. Banerjee, K. Fukumizu (Eds.), Proceedings of The 24th International Conference on Artificial Intelligence and Statistics, Vol. 130 of Proceedings of Machine Learning Research, PMLR, 2021, pp. 730–738. URL https://proceedings.mlr.press/v130/eriksson21a.html

[21] I. Araya, M.-C. Riff, A filtering method for algorithm configuration based on consistency techniques, Knowledge-Based Systems 60 (2014) 73–81. doi:https://doi.org/10.1016/j.knosys.2014.01.005. URL https://www.sciencedirect.com/science/article/pii/S0950705114000161

[22] L. Kotthoff, Algorithm selection for combinatorial search problems: A survey, in: Data Mining and Constraint Programming, Springer, 2016, pp. 149–190.

[23] P. Kerschke, H. H. Hoos, F. Neumann, H. Trautmann, Automated algorithm selection: Survey and perspectives, Evolutionary computation 27 (1) (2019) 3–45.

[24] C. Wang, H. Wang, C. Zhou, H. Chen, Experiencethinking: Constrained hyperparameter optimization based on knowledge and pruning, Knowledge-Based Systems 223 (2021) 106602.

[25] F. Hutter, H. H. Hoos, K. Leyton-Brown, Sequential model-based optimization for general algorithm configuration, in: International Conference on Learning and Intelligent Optimization, Springer, 2011, pp. 507–523.

[26] F. Hutter, L. Xu, H. H. Hoos, K. Leyton-Brown, Algorithm runtime prediction: Methods & evaluation, Artificial Intelligence 206 (2014) 79–111.

[27] L. Kotthoff, C. Thornton, H. H. Hoos, F. Hutter, K. Leyton-Brown, Auto-weka 2.0: Automatic model selection and hyperparameter optimization

in weka, Journal of Machine Learning Research 18 (25) (2017) 1–5.
URL http://jmlr.org/papers/v18/16-261.html

[28] M. Feurer, K. Eggensperger, S. Falkner, M. Lindauer, F. Hutter, Auto-sklearn 2.0: The next generation, arXiv preprint arXiv:2007.04074 (2020).

[29] A. Bonfietti, M. Lombardi, M. Milano, Embedding decision trees and random forests in constraint programming, in: L. Michel (Ed.), Integration of AI and OR Techniques in Constraint Programming, Springer International Publishing, Cham, 2015, pp. 74–90.

[30] M. Lombardi, S. Gualandi, A lagrangian propagator for artificial neural networks in constraint programming, Constraints 21 (4) (2016) 435–462.

[31] A. Borghesi, G. Tagliavini, M. Lombardi, L. Benini, M. Milano, Combining learning and optimization for transprecision computing, in: Proceedings of the 17th ACM International Conference on Computing Frontiers, 2020, pp. 10–18.

[32] A. De Filippo, M. Lombardi, M. Milano, Off-line and on-line optimization under uncertainty: A case study on energy management, in: International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Springer, 2018, pp. 100–116.

[33] A. Shapiro, A. Philpott, A tutorial on stochastic programming, Manuscript. Available at www2. isye. gatech. edu/ashapiro/publications. html 17 (2007).

[34] A. D. Filippo, M. Lombardi, M. Milano, The blind men and the elephant: Integrated offline/online optimization under uncertainty, in: C. Bessiere (Ed.), Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20, International Joint Conferences on Artificial Intelligence Organization, 2020, pp. 4840–4846, survey track. doi:10.24963/ijcai.2020/674.
URL https://doi.org/10.24963/ijcai.2020/674

[35] A. Gleixner, G. Hendel, G. Gamrath, T. Achterberg, M. Bastubbe, T. Berthold, P. Christophel, K. Jarck, T. Koch, J. Linderoth, et al., Miplib 2017: data-driven compilation of the 6th mixed-integer programming library, Mathematical Programming Computation (2021) 1–48.

[36] W. Van Ackooij, R. Zorgati, R. Henrion, A. Möller, G. De Gaulle, Chance constrained programming and its applications to energy management, Stochastic Optimization-Seeing the Optimal for the Uncertain (2011) 291–320.

[37] IBM, Cplex optimization studio v12.8 user manual, https://www.ibm.com/support/knowledgecenter/SSSA5P_12.8.0/ilog.odms.cplex.help/CPLEX/UsrMan/topics/preface/preface_title_synopsis.html, online; accessed 19 November 2018.

[38] W. Shi, N. Li, C.-C. Chu, R. Gadh, Real-time energy management in microgrids, IEEE Transactions on Smart Grid 8 (1) (2015) 228–238.

[39] L. Hyafil, R. L. Rivest, Constructing optimal binary decision trees is np-complete, Information Processing Letters 5 (1) (1976) 15–17. doi:https://doi.org/10.1016/0020-0190(76)90095-8. URL https://www.sciencedirect.com/science/article/pii/0020019076900958

[40] L. Breiman, J. H. Friedman, R. A. Olshen, C. J. Stone, Classification and regression trees, Routledge, 2017. doi:https://doi.org/10.1201/9781315139470.

[41] R. Lima, Ibm ilog cplex-what is inside of the box, in: Proc. 2010 EWO Seminar, 2010, pp. 1–72.

[42] A modeler's guide to handle complexity in energy systems optimization, Advances in Applied Energy 4 (2021) 100063. doi:https://doi.org/10.1016/j.adapen.2021.100063. URL https://www.sciencedirect.com/science/article/pii/S266679242100055X

[43] A. Borghesi, F. Baldo, M. Lombardi, M. Milano, Injective domain knowledge in neural networks for transprecision computing, in: Machine Learning, Optimization, and Data Science, Springer International Publishing, Cham, 2020, pp. 587–600.

[44] A. Holzinger, B. Malle, A. Saranti, B. Pfeifer, Towards multi-modal causability with graph neural networks enabling information fusion for explainable ai, Information Fusion 71 (2021) 28–37.

Appendix A. Decision Trees Encoding: An Illustrative Example

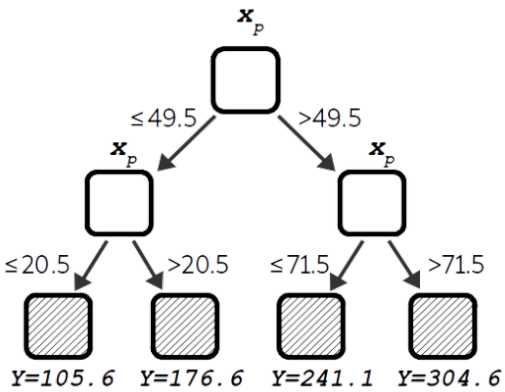

Figure A.7: DT of depth 2 for predicting the memory consumption (y) associated to ANTICIPATE algorithm, according to the hyperparameter x_p (the number of scenarios).

To provide an example of how DT is embedded exploiting the EML technique, we present here an example. In particular, we consider a decision trees of depth two, as its shallowness allows us to represent it with a limiter number of constraints. The DTs actually referred to in the rest of the paper are significantly deeper and require many more variables and constraints. As an example we consider one of the six DTs required by the proposed approach, namely the DT for ANTICIPATE algorithm and with mem as regression target. We can then simplify the notation (only for this example): x_p^a becomes x_p (as $a =$ ANTICIPATE, while y_t^a becomes y (as $t =$ mem; in practice, we will show how to embed a DT of depth equal to 2 which takes as input the algorithm configuration (x_p) and predicts its memory consumption. We this adopted another simplifying assumption, that is the DT makes its prediction based only on the algorithm hyperparameter, discarding the instance-descriptor features. The DT is depicted in Fig. A.7, where the white squares represent the tree nodes where the white squares are the decision nodes (for every node there is just one attribute for splitting, the algorithm hyperparameter x_p, as this is the only input to the tree) and the shaded nodes (the leaves) indicate the value for the regression target (y, i.e., mem) along the associated path. The regression target values are not normalized; clearly a DT of depth equal to two has a limited capacity, as the continuous range for the memory is approximated through just 4 different values.

If the solver used for the implementation supports logical expressions, the

key to encode a DT is to observe that each path pi from root to leaf can be viewed as a logical implication. From this observation, the DT presented as example can be described with the following constraints:

$$[x_p \leq 49.5] \wedge [x_p \leq 20.5] \Longrightarrow [y = 105.6] \tag{A.1}$$
$$[x_p \leq 49.5] \wedge [x_p > 20.5] \Longrightarrow [y = 105.6] \tag{A.2}$$
$$[x_p > 49.5] \wedge [x_p \leq 71.5] \Longrightarrow [y = 241.1] \tag{A.3}$$
$$[x_p > 49.5] \wedge [x_p > 71.5] \Longrightarrow [y = 304.6] \tag{A.4}$$

If we do not want to require the availability of logical constraints, as in the case of our approach, Equations A.1-A.4 can be expressed in a pure MILP formulation. We need to introduce four additional binary variables (this is automatically done by the EML library), denoted d_i with $i \in [0, 3]$; there is a binary variable for each leaf node in the tree – in practice, each binary variable indicates the path from root to leaf to follow. We also assume that $\mathrm{UB}(x_p)$ and $\mathrm{LB}(x_p)$ the upper and lower bound of variable x_p; in this example, x_p represents the number of scenarios used by ANTICIPATE and its domain is $[1, 100]$. The encoding constraints are the following:

$$y = \sum_{i \in [0,3]} k_i d_i \tag{A.5}$$
$$\sum_{i \in [0,3]} d_i = 1 \tag{A.6}$$
$$x_p + (\mathrm{UB}(x_p) - 49.5)(d_0 + d_1) \leq \mathrm{UB}(x_p) \tag{A.7}$$
$$x_p + (\mathrm{UB}(x_p) - 20.5)d_0 \leq \mathrm{UB}(x_p) \tag{A.8}$$
$$x_p + (\mathrm{LB}(x_p) - 20.5)(d_1 + d_2 + d_3) \geq \mathrm{LB}(x_p) \tag{A.9}$$
$$x_p + (\mathrm{LB}(x_p) - 49.5)(d_2 + d_3) \geq \mathrm{LB}(x_p) \tag{A.10}$$
$$x_p + (\mathrm{UB}(x_p) - 71.5)(d_0 + d_1 + d_2) \leq \mathrm{UB}(x_p) \tag{A.11}$$
$$x_p + (\mathrm{LB}(x_p) - 71.5)d_3 \geq \mathrm{LB}(x_p) \tag{A.12}$$

The coefficients k_i in Eq. A.5 correspond to the value of the target at each leaf node; in our case then $k_0 = 105.6, k_1 = 176.6$, etc. Eq. A.6 specifies that only one path of the tree can be active. Then, for each logical constraint in Equations A.1-A.4 we have two constraints: Eq. A.1 is transformed in Eq. A.7 and A.8 (corresponding to the path to the leaf node with $y = 105.6$), while

Eq. A.2 is converted to Eq. A.7 and A.9. Equation A.7 is used for both logical constraints as it represents the top-most left branch in the DT ($x_p \leq 49.5$, see Fig. A.7), thus being involved in two paths; for compactness we chose not to repeat twice the shared constraint. Similarly, Eq. A.3 becomes Eq. A.10 and A.11, while Eq. A.3 corresponds to Eq. A.10 and A.12 (again, the shared equation was not repeated). It is easy to see how even with such a small DT the number of constraints (and variables) is not negligible, especially considering that our approach require six different ML models. The number of constraints steeply increases with the depth of the tree, thus we have to strike the right balance between a DT model sufficiently deep to correctly learn the input-target relationship and whose encoding does not strain excessively the optimization solver.