ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

DOTTORATO DI RICERCA IN

COMPUTER SCIENCE AND ENGINEERING

Ciclo 36

Settore Concorsuale: 09/H1 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

Settore Scientifico Disciplinare: ING-INF/05 - SISTEMI DI ELABORAZIONE DELLE INFORMAZIONI

NOVEL TECHNIQUES FOR HARNESSING SYMBOLIC AND STRUCTURED INFORMATION INTO MACHINE LEARNING

Presentata da: Mattia Silvestri

Coordinatore Dottorato

Ilaria Bartolini

Supervisore

Michele Lombardi

Esame finale anno 2024

Abstract

In recent years, we have assisted to a new spring of Artificial Intelligence (AI). This transformation has been characterized by a shift from the symbolic methods prevalent in the last century to a focus on sub-symbolic techniques, driven by the remarkable achievements of deep learning in areas such as computer vision and natural language processing. The renewed interested in AI is attributed to two pivotal factors: the advent of powerful, dedicated hardware like graphical processing units and tensor processing units, providing the computational power necessary for training complex deep learning models, and the enormous data availability in the age of big data. The latter, especially augmented by the widespread adoption of Internet of Things (IoT) technologies, has led to an abundance of data from diverse sources, particularly sensor measurements, fueling the application of machine learning and deep learning in various industrial scenarios.

Despite the successes of sub-symbolic, data-driven methods, recent years have seen a growing inclination towards hybrid models that synergize symbolic and sub-symbolic approaches. This trend stems from several inherent limitations in purely data-driven systems. Firstly, these systems often redundantly learn concepts that are already part of common knowledge or are well-understood by domain experts. This redundancy raises the question of how to prevent machine learning algorithms from re-learning these established concepts. Secondly, data-driven methods may struggle to adhere to specific constraints, such as those dictated by natural laws or user-imposed rules, whereas symbolic methods can manage these constraints more easily. Lastly, the black-box nature of sub-symbolic methods poses challenges in terms of interpretability and explainability, in contrast to the more transparent symbolic approaches.

In the context machine learning and deep learning, these challenges have given rise to the emergent field of informed machine learning. This new domain aims to exploit the strengths of both symbolic and sub-symbolic methods by formalizing and incorporating existing task-specific knowledge into traditional machine learning workflows. The goal is to create systems that are not only more efficient and reliable but also more interpretable and adaptable to various constraints.

The core objective of this thesis is to explore and advance the field of informed machine learning. It presents innovative algorithms within this domain and conducts a thorough investigation of existing methodologies. The applications of these algorithms are explored in two significant areas of AI: predictive modeling and decision support systems. To validate the practical utility of these algorithms, the thesis undertakes a comprehensive empirical evaluation. This evaluation encompasses real-world application as well as abstract problems commonly used in the scientific community to investigate practical use cases. The findings from these studies provide concrete evidence of the effectiveness of informed machine learning solutions in addressing the highlighted challenges. Moreover, the thesis demonstrates how informed machine learning can significantly enhance the capabilities and applicability of sub-symbolic methods by effectively harnessing diverse forms of existing knowledge.

Table of contents

List of figures

List of tables

Research activities and publications

Research activities

- As a junior researcher, I was involved in the KINeMA project co-funded by the Bi-rex competence center (https://bi-rex.it/). KINeMA stands for "Knowledge Integration in Neural Networks for e-Maintenance". The goal of the project was to apply cutting-edge informed machine learning methods in the realm of predictive maintenance. My role in this project was multifaceted: I not only contributed to developing the core research idea but also played a pivotal role in the development of a working prototype, bridging the gap between theoretical research and practical application.

- During the mandatory period abroad, I visited the research group of the professor Tias Guns at KU Leuven. This experience was crucial for enhancing my expertise in methods that integrate learning with optimization. It also provided a fertile ground for developing new ideas about incorporating prior knowledge of combinatorial optimization problems into machine learning frameworks. The fruitful outcome of this collaboration is elaborated in section 6.4 of my thesis.

List of pubblications In the following, I will list the publications resulted from my research activity:

- Silvestri, Mattia, Michele Lombardi, and Michela Milano. *"Injecting domain knowledge in neural networks: a controlled experiment on a constrained problem."* Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18. Springer International Publishing, 2021.

- Silvestri, Mattia, et al. *"Supervised Anomaly Detection in Crude Oil Stabilization."* PAIS 2022. IOS Press, 2022. 114-127.

- Silvestri, M., De Filippo, A., Ruggeri, F., and Lombardi, M. (2022, June). *"Hybrid Offline/Online Optimization for Energy Management via Reinforcement Learning"*. In International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research (pp. 358-373). Cham: Springer International Publishing.

- Silvestri, M., Baldo, F., Misino, E., Lombardi, M. (2023). *"An Analysis of Universal Differential Equations for Data-Driven Discovery of Ordinary Differential Equations"*. In: Mikyška, J., de Mulatier, C., Paszynski, M., Krzhizhanovskaya, V.V., Dongarra, J.J., Sloot, P.M. (eds) Computational Science – ICCS 2023. ICCS 2023. Lecture Notes in Computer Science, vol 10476. Springer, Cham. https://doi.org/10.1007/978-3-031-36027-5_27

- Mattia Silvestri and Senne Berden and Jayanta Mandi and Ali İrfan Mahmutoğulları and Maxime Mulamba and Allegra De Filippo and Tias Guns and Michele Lombardi. *"Score Function Gradient Estimation to Widen the Applicability of Decision-Focused Learning"*. ICML 2023 Workshop on Differentiable Almost Everything: Differentiable Relaxations, Algorithms, Operators, and Simulators, 2023, https://openreview.net/forum?id=ty046JUllZ.

- Silvestri, M., De Filippo, A., Lombardi, M., and Milano, M. (2022). *"UNIFY: a Unified Policy Designing Framework for Solving Constrained Optimization Problems with Machine Learning"*. arXiv preprint arXiv:2210.14030. (Under review).

Chapter 1

Introduction

In a broad sense, AI aims to build machines capable of solving problems typically requiring human intelligence, such as playing games, solving equations, image recognition, and understanding natural language. Humans learn to solve new problems through experience and by reasoning about previously solved challenges. However, learning is a multifaceted task that necessitates both sensory and reasoning capabilities. For instance, when solving mathematical equations, we first recognize the written digits and symbols, then we rely on prior algebraic knowledge to solve the problem.

Informed machine learning emerges as a response to the limitations of traditional data-driven AI. While conventional machine learning (ML) algorithms excel at pattern recognition and predictive modeling, they often lack the depth and context that come from established *domain knowledge*. Informed ML seeks to bridge this gap by integrating external, validated knowledge into the ML process. Consider the challenge of solving algebraic equations from images: an informed ML solution might employ a deep learning model to recognize digits and symbols from images, followed by a symbolic engine to solve the resulting equations. This example underscores the promise of informed ML but also highlights the complexity in its implementation.

The first challenge in implementing informed ML lies in defining and formalizing knowledge. Within AI and computer science, knowledge transcends mere data or information; it embodies a set of validated entities, concepts, and relations, rigorously established and authenticated. Techniques for formalizing this knowledge vary, ranging from logical constructs to algorithmic representations, each tailored to the unique requirements of the task at hand.

Furthermore, knowledge is *heterogeneous* and originates from diverse sources. Algebraic equations and logic rules, for instance, might represent world and commonsense knowledge, while differential equations describe complex systems and natural laws. In industrial facilities, expertise may be encoded as causal relationships between different components. This diversity poses a significant challenge: devising a unified approach to integrate these varied forms of knowledge into sub-symbolic methods.

Another key challenge is determining the optimal way to inject knowledge into an ML algorithm. ML pipelines are articulated in four main steps: 1) training data; 2) hypothesis set; 3) learning algorithm; and 4) final hypothesis. The integration of knowledge can vary depending on its source and form, and may affect different steps of the pipeline.

This thesis investigates the adoption of different forms of knowledge, injected at various stages of an ML workflow, with the aim of enhancing *predictive models* and *decision support systems*. In the realm of predictive models, we empirically validate innovative informed ML methodologies, with a particular

emphasis on physics-informed ML, balancing empirical accuracy with scientific interpretability. Practical applications are explored through challenges in industrial predictive maintenance, such as in oil and gas facilities, where leveraging expert knowledge is crucial to effectively manage complex machinery.

We also highlight the declarative formulation of optimization problems as a valuable source of knowledge. A tighter integration between learning and optimization emerges as an effective solution to enhance decision support systems. The thesis introduces a novel framework that marks a significant leap in this area, demonstrating how the fusion of ML with combinatorial optimization techniques can lead to more sophisticated, effective decision support tools. Practical applications in Energy Management Systems are explored, revealing the intricate balance required in decision-making under uncertainty. These real-world scenarios, along with abstract but challenging problems like the knapsack and weighted set multi-cover problems, serve as grounds for demonstrating the efficacy of informed ML in the context of decision support systems.

The thesis is structured as follows. Chapter 2 provides the background necessary for understanding informed ML techniques, encompassing ML, constrained optimization, and optimization under uncertainty. Chapter 3 introduces informed ML and the state-of-the-art approaches, focusing on knowledge represented as algebraic equations, differential equations, logic rules, simulation results, Bayesian networks, and the declarative formulation of an optimization problem. Chapter 4 presents the use cases that will be investigated, namely an Energy Management System, an oil and gas facility, an RC-circuit for thermal modeling, and some abstract but challenging combinatorial optimization problems. The main contributions of the thesis begin with Chapter 5, where we depict a novel methodology for integrating prior knowledge in the form of external black-box models. We then deeply investigate the physics-informed ML framework in terms of accuracy and interpretability. Chapter 6 presents novel informed ML techniques for decision support systems. The first approach distills knowledge into the weights of a neural network to speed up the decision process during inference. We then describe UNIFY, a general framework that incorporates knowledge of optimization problems into the learning step of an ML algorithm. Finally, we present a specific instance of the UNIFY framework. Chapter 7 provides final conclusions and discussion.

Chapter 2

Background

In this chapter, we present an overview of three foundational research areas central to the innovative work discussed in this thesis: optimization with constraints, optimization under uncertainty, and machine learning.

Optimization with constraints addresses the challenge of identifying the optimal solution within a set of choices while adhering to specific limitations or requirements. *Optimization under uncertainty* extends the concept by acknowledging the inherent unpredictability present in many practical scenarios. These methods enable robust decision-making in domains such as finance, logistics, and energy management, where variables can exhibit high volatility and unpredictability. *Machine learning* represents a paradigm shift from traditional problem-solving approaches. By leveraging historical data, machine learning empowers systems to learn from experience, discover patterns, and excel at predictive tasks that often elude human analysis.

In this thesis, we harness the power of optimization with constraints and optimization under uncertainty as decision support systems to tackle complex challenges, including energy management and production scheduling. Simultaneously, machine learning serves as a predictive tool, enhancing human decision-making by providing valuable insights based on historical data.

2.1 Optimization under constraints

From a high-level perspective, constrained optimization involves identifying the optimal values for a set of variables, referred to as the solution, that minimize or maximize a cost or profit function. However, not all values are allowed; in other words, not all solutions are feasible. The feasible solutions must satisfy a set of constraints that restrict the space of values we explore, known as the *feasible region*.

In real-world scenarios, various problems emerge. For instance, a manufacturing company seeks to minimize production costs while meeting customer demands. An Energy Management System must allocate the minimum-cost power flows from different sources while satisfying user loads. A logistics company needs to schedule a fleet of vehicles to complete its delivery tasks while minimizing total travel times. These examples illustrate the breadth of constrained optimization problems, sparking significant interest in the research community to develop methods that efficiently explore the feasible region to find optimal solutions.

Formally, constrained optimization involves solving the following problem:

$$x^\star = \arg\min_{x \in \mathcal{F}(x)} f(x) \tag{2.1}$$

Here, $z \in Z$ with Z denoting the domain of the variables, $f : Z \to \mathbb{R}$ represents the objective function (cost or profit), and $\mathcal{F}(z) \subseteq Z$ is the feasible region.

Over time, optimization problems under constraints have been classified into different categories based on the form of the function f and the feasible region $\mathcal{F}$. This classification aims to develop dedicated methods for each category. In the following sections, we provide a non-exhaustive review of these different classes of problems and methods, directing interested readers to additional references.

2.1.1 Constrained Satisfaction Problems

Constrained satisfaction problems (CSPs) represent a specific instance of the optimization problem outlined in eq. (2.1). In CSPs, the primary objective is to identify solutions, if they exist, that satisfy a given set of constraints. Notable examples of CSPs include the scheduling problem, which involves optimizing the utilization of limited resources for task completion (e.g., job scheduling on processors), and the maximum cut problem. The latter entails partitioning an undirected graph into two sets of vertices to maximize the number of edges between them—a problem with applications in network design and production scheduling.

Formally, CSPs are characterized by the triplet $\langle X, D, C \rangle$, where X is the set of variables, $X_i \in D_i, \forall i \in |X|$, and C is the set of constraints defining feasible solutions. CSPs are usually solved via search techniques, with backtracking and local search being widely adopted. Backtracking maintains a list of feasible candidates while incrementally constructing a solution. In case of failure (e.g., when the domains of all variables are empty), it restarts the search with a new candidate until a solution is found or the search space is exhausted. Local search explores the solution space through variable assignment changes.

CSP techniques often leverage methods to intelligently reduce the space of feasible solutions for efficient search. Constraint propagators, for instance, prune variable domains while ensuring consistency in possible assignments. Propagators can achieve consistency at three levels: node consistency, where every unary constraint on a variable must be satisfied by all values in its domain; arc consistency, indicating that a variable is arc-consistent with another if every value in its domain is consistent with at least one value in the other variable's domain according to all constraints; and path consistency, a generalization of arc consistency involving the comparison of a pair of variables with a third one.

This sub-section provides a concise overview of CSPs, and for a more in-depth discussion, interested readers are referred to [4]. It's important to note that CSP methods focus solely on finding solutions that satisfy the given constraints. In contrast, in the more general constrained optimization setup (and in many practical scenarios), preferences may exist among feasible solutions based on an evaluation function. The subsequent sub-section describes various classes of such problems and the techniques that can effectively address them.

2.1.2 Constrained Optimization Problems

Conversely to CSPs, constrained optimization problems (COPs) require finding the set of feasible solutions that optimize an objective function. Due to their numerous applications and the computational

complexity challenges, substantial research activity has been devoted to developing efficient methods to solve these problems, particularly within the operations research (OR) community. Formally, a COP involves solving the following problem:

$$\min_z f(z) \tag{2.2}$$

$$s.t. \quad g_i(z) \leq 0 \quad \forall i = 1, \ldots, I \tag{2.3}$$

$$h_j(z) = 0 \quad \forall j = 1, \ldots, J \tag{2.4}$$

where z is the set of decision variables, $f : Z \to \mathbb{R}$ is the objective function, and I and J are, respectively, the number of inequality and equality constraints. Depending on the structure of the objective function and feasible region, dedicated methods have been developed to solve these problems more efficiently. Below, we provide an overview of the classes of COPs along with their properties and challenges.

Convex Optimization Convex optimization problems [5] constitute a special case of COPs where the objective function f is convex, the inequality constraints $g_i, \forall i = 1, \ldots, I$ are convex functions, and the equality constraints are affine transformations, i.e., $h_j = a_j \cdot z - b_j \ \forall j = 1, \ldots, J$, with a_j as a vector and b_j as a scalar. Convex optimization problems possess the following properties: 1) every local minimum is a global minimum, 2) the set of optimal solutions is convex, and 3) if the objective function is strictly convex, the problem admits at most one optimal solution. While convex optimization is generally NP-hard, many problems admit polynomial-time algorithms in practice, such as the Interior Point method [6].

Linear Programming Linear programming (LP) problems are a special case of convex optimization where the objective function, disequality, and equality constraints are all affine, and the feasible region is defined by a polyhedron. An LP problem can be formulated as:

$$\min_z c^T z \tag{2.5}$$

$$s.t. \quad Gz - h \leq 0 \quad \forall i = 1, \ldots, I \tag{2.6}$$

$$Az - b = 0 \quad \forall j = 1, \ldots, J \tag{2.7}$$

where $z \in \mathbb{R}^n$, and $c \in \mathbb{R}^n$ is the cost vector. Since LP is a special case of convex optimization, it retains the same properties. LP problems are combinatorial, with the set of feasible solutions being finite since, if an optimal solution exists, it will be attained at a vertex of the polyhedron. Moreover, LP problems are solvable in polynomial time, and various optimization algorithms, such as the Simplex algorithm, the ellipsoid method and Karmarkar's algorithm [7] exist.

Quadratic Programming Problems where the feasible region is a polyhedron, but the cost function is quadratic, are referred to as quadratic programming (QP) problems and can be formulated as follows:

$$\min_z \frac{1}{2} z^T Q z + q^T z \tag{2.8}$$

$$s.t. \quad Gz - h \leq 0 \quad \forall i = 1, \ldots, N \tag{2.9}$$

$$Az - b = 0 \quad \forall j = 1, \ldots, M \tag{2.10}$$

where $z \in \mathbb{R}^n$, $Q \in \mathbb{R}^{n \times n}$ is the quadratic cost matrix, and $q \in \mathbb{R}^n$ is the linear cost vector. QP problems are solved using interior point methods or extensions of the Simplex algorithm.

Integer Linear Programming and Mixed Integer Linear Programming So far, we have considered only continuous decision variables. Many real-world optimization problems involve integer decision variables to model resources that are discrete by nature. Problems where the cost function and the equality and inequality constraints are affine functions, and the decision variables are integer values, are referred to as integer linear programming (ILP) problems [8] and can be formulated as follows:

$$\min_z c^T z \tag{2.11}$$

$$s.t. \quad Gz - h \leq 0 \quad \forall i = 1, \ldots, I \tag{2.12}$$

$$Az - b = 0 \quad \forall j = 1, \ldots, J \tag{2.13}$$

$$z \in \mathbb{Z}^n \tag{2.14}$$

In general, we can have both integer and continuous variables, resulting in the more general mixed integer linear programming (MILP) problems. Due to integrality constraints, the feasible region is no longer convex, and thus, they do not retain the properties of convex optimization. The methods previously listed can not be applied to solve this class of problems, and dedicated branch-and-bound algorithms are instead used in this setup.

2.2 Optimization under Uncertainty

In the previous section, we focused on problems with perfect information, where all parameters are known at decision time. These problems are typically referred to as deterministic problems. However, many real-world optimization problems involve a degree of *uncertainty*. For instance, in an Energy Management System, the energy provided by renewable resources is highly uncertain and uncontrollable (e.g., photovoltaic panels affected by weather); in vehicle fleet routing, the travel times might be uncertain due to traffic conditions and new customers might appear during the travel; another notable example is finance where the asset prices are highly stochastic and uncontrollable. In this scenario, some optimization problem parameters are treated as random variables (e.g., photovoltaic production or customers' locations) with associated probability distributions, which may even be unknown.

Despite having numerous applications, optimization under uncertainty poses significant challenges. Ideally, one would optimize for every possible outcome, leading to large-scale or even intractable problems. One simplified approach to tackle such problems involves disregarding uncertainty and assuming that all parameters are deterministic. However, this is not feasible when uncertainty significantly impacts solution accuracy, necessitating dedicated methods.

There are two main classes of approaches to dealing with uncertainty in optimization problems [9] [10] [11]: robust and stochastic optimization. *Robust optimization* aims to optimize for the worst-case scenario without assuming knowledge of the distribution that models uncertainty. In contrast, *stochastic optimization* algorithms optimize the expected value of the objective function, assuming knowledge of the probability distribution. For example, in an Energy Management System we might opt for a stochastic optimization solution: uncertainty stems from uncontrollable deviations of the renewable energy resources productions which provide low price energy and the goal is to minimize the expected

cost. At the same time, due to the availability of flexible energy sources (e.g. combined heat and power generator) we are less concerned about the worst-case scenario. Conversely, in the engineering field, for example when designing a mechanical structure, we aim at dealing with severe variations in the material properties and operating conditions. Robust optimization is computationally tractable for many uncertainty sets and problem types, making it appealing. However, when uncertainty can be adequately modeled and reformulation is computationally feasible, stochastic optimization is generally preferred as it typically provides better solution quality.

Stochastic optimization often involves multiple stages, where at each stage, a subset of uncertain elements is revealed, and decisions must be made. Two-stage stochastic programming requires determining two sets of optimal decisions—one for the first stage and another for the second stage (often referred to as *recourse actions*). The first stage decisions are made before uncertainty is revealed, and recourse actions are applied afterward to recover from feasibility issues due to partial information from the first stage. For instance, in a vehicle routing problem, if travel times exceed expectations, rescheduling the route may be required. The overall objective is to minimize the first stage cost and the expected value of the second one.

A notable approach widely used for stochastic programming is the Sample Average Approximation (SAA) [12] [13]. In SAA, probability distributions of random variables are approximated by drawing a finite set of samples, yielding a set of realizations called *scenarios*. The approach involves solving a deterministic version of the problem with a copy of the decision variables for each scenario, allowing estimation of the expected cost by averaging the contributions from each copy. SAA provides high-quality solutions with a sufficiently large number of sampled scenarios but comes with a significant computational cost, posing scalability issues.

2.3 Machine Learning

Machine Learning (ML) [14] [15] [16], a pivotal subbranch of AI, eschews explicit programming to solve complex tasks. Instead, it learns from experience, building statistical models to predict outcomes from given inputs. These models are trained on historical data, which may be structured (like tabular data and time series) or unstructured (such as images, text, and audio). ML also encompasses learning from interactions within an environment that provides feedback.

In recent years, neural networks have become a popular tool for solving problems typically addressed by ML algorithms. The perceptron [17], a fundamental unit of neural networks, was conceived in the mid-20th century. However, neural networks have gained prominence only recently, thanks to technological advancements in the hardware and the avilability of a vast amount of data. These technological breakthroughs have revived interest in neural networks, stimulating new research and discoveries in the field.

Deep learning [18], a subset of ML, has gained attention for tackling complex tasks. “Deep learning” refers to complex neural architectures, with multiple layers and interconnections, that simulate how the brain works. This approach is effective in complex perceptual tasks like vision, natural language understanding, and speech recognition.

The extensive literature on ML and deep learning is beyond this the scope of this section. For a comprehensive overview, additional references are recommended [14] [15] [16] [18].

ML algorithms are broadly classified into three main approaches: 1) supervised learning, 2) unsupervised learning, and 3) reinforcement learning. In this section, we will provide a brief overview of these

approaches since some of the concepts are useful for a better understanding of the work realized within the scope of this thesis.

Supervised learning In *supervised learning*, the objective is to learn a mapping function $f : X \to Y$, where $X \in \mathbb{R}^n$ represents an input vector (information), and $Y \in \mathbb{R}^m$ is a target vector (desired outcome). Supervised learning assumes access to a dataset $\mathcal{D} = \{(x_k, y_k)\}_{k=1}^{K}$, where each (x_k, y_k) is an *example*, and K is the number of available examples—a representative sample of the true underlying distribution $P(X, Y)$.

Supervised learning is further categorized into *classification*, when the output belongs to a finite set of categories, or *regression*, when the output is numerical.

Given a candidate set $h \in \mathcal{H}$, where $h : X \to Y$, finding the optimal hypothesis h^* for the mapping f is known as *training*. Formally, this is expressed as:

$$h^* = \arg\min_{h \in \mathcal{H}} \left\{ \sum_{k=1}^{K} \mathcal{L}(y_k, \hat{y}_k) \right\} \tag{2.15}$$

Here, $\hat{y} = h(x_k; \theta)$, and $\mathcal{L}$ is a *loss function* measuring the dissimilarity between model predictions and desired outcomes. Common loss functions include categorical cross-entropy for classification and mean squared error (MSE) for regression.

When $\mathcal{H}$ consists of parameterized models $h(x; \theta) \in \mathcal{H}$ with parameters θ, eq. (2.15) becomes:

$$\theta^* = \arg\min_{\theta} \left\{ \sum_{k=1}^{K} \mathcal{L}(y_k, h(x_k; \theta)) \right\} \tag{2.16}$$

Supervised learning finds practical applications in various domains, including computer vision (e.g., object detection in images), natural language processing (e.g., sentiment analysis, argumentation mining, machine translation), and time series forecasting.

Unsupervised learning While supervised learning assumes knowledge of the desired outcome, *unsupervised learning* [19] operates with input features alone, lacking any associated labels. From a probabilistic standpoint, instead of estimating the conditional distribution $p(y|x; \theta)$, as commonly done in supervised learning, unsupervised learning seeks to estimate $p(x; \theta)$ conditioned solely on the parameters θ.

Unsupervised learning is widely employed for discovering patterns in data. One prominent application is *clustering*, where similar data points are grouped into clusters sharing similar properties. It finds utility across diverse fields, including computer vision (e.g., grouping pixels for image segmentation), natural language processing (e.g., text categorization, topic modeling), and market analysis (customers segmentation, targeted advertising), among others.

Another prevalent application of unsupervised learning is *dimensionality reduction*. This involves projecting high-dimensional input features into a smaller space that retains the main properties of the data. This approach facilitates data interpretation and is often used as a pre-processing step for supervised models, aiming to enhance their accuracy.

Practical applications include genomics, where dimensionality reduction is employed to analyze gene expression data, given the simultaneous measurement of thousands of genes; signal processing, to separate signals into distinct sources; and in natural language processing, where dimensionality reduction

techniques capture the contextual meaning of words in a lower-dimensional space, enabling more efficient processing of textual data.

Reinforcement Learning The goal of reinforcement learning (RL) consists of training an agent to maximize a reward signal received from an environment with which the agent interacts. RL differs from supervised learning since the agent is not explicitly instructed on how to act. Instead the behavior is inferred from the interactions with the environment and by observing the actions outcome. Even if labels are not provided, it is not a subset of unsupervised learning: finding hidden patterns may be useful to increase the reward but it is not a key part of the paradigm.

An RL system consists of two main entities: an *agent* and an *environment*. The environment is a structured framework that simulates the dynamics and constraints of the real-world scenario relevant to the task, providing the agent with the necessary context and challenges for learning and decision-making. Its internal state is accessible through observations. The observations capture only salient features of the environment state and might not reflect the state in its entirety, i.e. some features might be hidden from the outside.

The agent is the entity interacting with the environment through actions and capable of processing the observations. Some examples are: a robot capable of moving, grabbing and leaving objects; a self-driving cars, with sensors and cameras, which takes the driving decisions; an Energy Management System that schedules the optimal power flows while minimizing the cost; a recomendation system that provides the user with personalized content and aims at maximizing the attractibility of a product; a non-player characters in videogames.

The actions model the outcome of the agent and how it affects the environment's internal state. RL does not pose strict restriction on what actions are but they are usually classified between discrete and continuous: discrete actions are represented with a set of finite choices, while continuous ones are associated with numeric values. For example, a robot movement might be modeled as a discrete set of four possible actions (up, down, right, left) or as two-dimensional continuous vector (speed and direction).

Beside the agent and the environment, other elements of an RL system are the *policy*, the *value-function* and, optionally, a *model* of the environment. While the reward represents short-term evaluation of the agent's behavior, the value-function represents the long-term desirability of a given state. Value-functions take into account not only the current reward but also the possible future states and rewards. The policy represents the agent's behavior: given the current state, it establishes which action should be performed, and it is formally defined as a function $\pi_\theta : S \to A$, with parameters θ, where S and A are respectively the states and actions spaces. In general, it's stochastic, meaning that can be represented as probability distribution over states and actions, but it can also be deterministic. Finally, the model of the environments allows the RL agent to infer additional information about it; for example it may be able to predict next state before acting.

RL algorithms are mainly adopted to solve sequential decision-making problems by relying on the Markov decision process (MDP) mathematical formulation. Formally, a fully-observable MDP is defined by a tuple (S, A, p, r, γ), where S is the set of states, A is the set of actions, $p(\cdot|s, a)$ is the probability distribution of next states, $r(\cdot|s, a)$ is the probability distribution of the reward and $\gamma \in [0, 1]$, called discount factor, controls the impact of future rewards.

The sequential decision making problem is then cast down to a recurrent process where the RL agent interacts with the environment by performing actions according to the policy π. Consequently,

these actions provoke the agent's state transitions. The learning process is then formulated as the maximization problem of cumulative rewards along state-action trajectories τ, dictated by π.

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=1}^{T} \gamma^t r(s_t, a_t) \right] \tag{2.17}$$

$$p_\theta(\tau) = p_\theta(s_1, a_1, \ldots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t) \tag{2.18}$$

where T is the trajectory time horizon.

RL algorithms can be classified into two main categories: model-free and model-based RL. Model-free algorithms try to find the optimal policy π^* such that the expected cumulative discounted reward from the initial state $s_{t=1}$ is maximized. The idea of model-based RL is to learn the model of the environment, i.e. the transition probabilities $p(\cdot|s, a)$, rather than the optimal policy and then use the learned model to choose the optimal actions.

Within the model-free family, policy gradient algorithms are widely used when the actions space is continuous. One such an example is REINFORCE [20]: given a parametric policy π_θ, the parameters θ are optimized by gradient ascent to directly maximize $J(\theta)$:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right] \tag{2.19}$$

Policy gradient algorithms are known to suffer from high variance. Several non-mutually exclusive solutions can be employed to mitigate this issue, such as baseline subtraction to correctly isolate positive actions. Among the possible baselines, actor-critic (AC) methods are particularly effective in reducing variance. Instead of using a state-dependent baseline, one can reduce the variance by computing the advantage of taking an action a_t in state s_t. The advantage is defined as $A_\pi(s_t, a_t) = r + \gamma V_\pi(s_{t+1}) - V_\pi(s_t)$, where $V_\pi(s_t) = \mathbb{E}_\pi [J(\tau)|s = s_t]$ is the value function, r is the reward and s_{t+1} is the next state. Thus, the actor is represented by the policy, whereas the value function acts as the critic.

Conversely, value-function methods aim at estimating the optimal value-function or the advantage-function which is then used to choose the optimal action at each state. Notable examples of this class of RL algorithms are Q-learning and SARSA.

Modern RL approaches take advantage of deep learning models as powerful tools for representation learning [21]. More precisely, neural networks are employed to approximate the policy π_θ and $V_\theta(\cdot)$. The scientific community usually refers to this research field as deep reinforcement learning (DRL).

Conclusions

This chapter provided a foundational overview of three research fields: optimization with constraints, optimization under uncertainty, and machine learning. Each of these domains offers unique insights and tools for tackling complex problems in various fields.

In optimization with constraints, we explored how optimal solutions can be identified within predefined boundaries, emphasizing the importance of efficient methods to navigate and exploit the feasible region. This category encompasses constrained satisfaction problems, addressed through constrained

programming techniques, and constrained optimization problems, typically tackled using methods from operations research.

Optimization under uncertainty highlighted the challenges and methodologies for dealing with unpredictable elements in decision-making processes. The concepts of robust and stochastic optimization underscore the need for strategies that can adapt and perform well under various conditions. We described the Sample Average Approximation method as a notable stochastic programming approach.

Finally, ML showed us the power of data-driven approaches in understanding and solving complex tasks. The rise of neural networks, deep learning, and RL techniques underlines a significant shift towards systems capable of learning and evolving, pushing the boundaries of what can be achieved through computational intelligence.

Chapter 3

Informed Machine Learning

In our daily life, we continuously cite and rely on the concept of "knowledge". But *what* is knowledge? *How* can we assess knowledge of something? What are the *origin* and *nature* of knowledge? *Epistemology* [22] [23] is the science that addresses these questions. However, there are still no unique and universally endorsed answers, and research and debate are ongoing.

In the scope of this thesis, we will focus on knowledge from a computer science perspective. We define knowledge as a set of entities, concepts, and relations between them that have been validated by trusted authorities or via an empirical protocol. In the context of ML, we refer to *prior knowledge* as a source of knowledge that is separated from the learning algorithm and the training data. Despite being external to the usual ML pipeline, it is clear that prior knowledge is a valuable resource for learning solutions from experience.

The discipline that studies ML algorithms by explicitly integrating prior knowledge is referred to as *informed machine learning*. A formal and clear definition of informed ML is provided in [24]: "*Informed machine learning describes learning from a hybrid information source that consists of data and prior knowledge. The prior knowledge comes from an independent source, is given in formal representations, and is explicitly integrated into the machine learning pipeline* "

Figure 3.1 illustrates a schematic representation of an informed ML pipeline, highlighting the process of deriving a mapping function f from an input domain X to a target domain Y. This mapping encapsulates various learning paradigms: in supervised learning, it addresses regression or classification

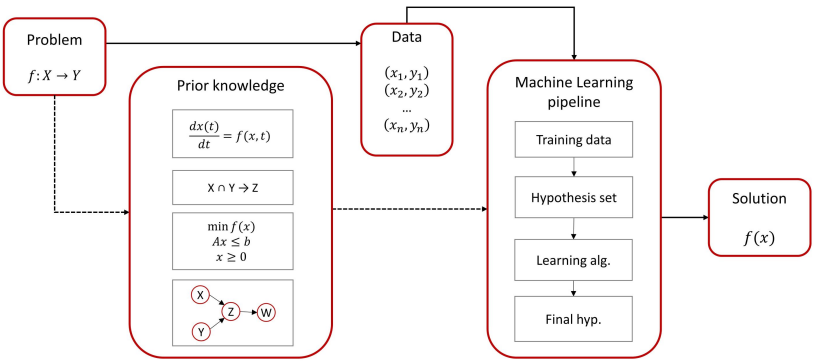


Figure 3.1: Informed machine learning pipeline.

tasks, mapping input features (domain X) to labels (domain Y). In the unsupervised paradigm, the same input domain X leads to either clusters (domain Y in clustering tasks) or a reduced input space (in dimensionality reduction). RL differs slightly, as X represents the observation space, while Y corresponds to the action space. The data, or "experience", feeding into this model encompasses diverse formats such as images, textual content, tabular structures, and time series. While supervised learning pairs each input feature x with a corresponding label y, unsupervised learning operates solely on the input x. RL adds another layer, combining observations with rewards, but it does not provide an optimal action as part of the target.

The core of the ML pipeline consists of the following steps:

1. *Training data.* The data are preprocessed to make them digestible by the algorithm.
2. *Hypothesis set.* As also previously described in section 2.3, a candidate set of hypotheses $\mathcal{H}$ is identified, from which the optimal model is chosen.
3. *Learning algorithm.* The learning algorithm aims to find the optimal hypothesis h^* from the candidate set $\mathcal{H}$.
4. *Final hypothesis.* The output of the learning algorithm is then selected as the solution for the problem.

Informed ML techniques are classified based on three dimensions: 1) the knowledge *source*, 2) the knowledge *representation*, and 3) the step of the ML pipeline where knowledge is *integrated*. In the following, we provide a short description of each dimension.

Knowledge source As previously mentioned, the knowledge source is validated by a trusted group of individuals from a specific domain. It can originate from science, world common sense, or human experts. The source of knowledge is broad and heterogeneous, and the validation process can be formal or informal, explicit or implicit.

The most common source of knowledge is *science*, encompassing many disciplines like physics, maths, chemistry, engineering, and technology. Scientific knowledge is usually well-established through empirical procedures and formalized in natural laws or algebraic equations that are easy to integrate into an ML pipeline.

Another readily available source of knowledge stems from the *world* surrounding us, i.e. facts and experiences from everyday life that we implicitly acquire. It usually refers to objects and concepts, and the relations among them. While world knowledge is an extensive and readily verifiable resource, its formalization into a format that is easily interpretable by ML algorithms can be challenging.

The final major category of knowledge is *expert* knowledge, which is acquired through years of specialized work in particular fields. This form of knowledge is especially prevalent in engineering, where practitioners develop a deep understanding of specific systems through extensive professional experience. While undeniably valuable, expert knowledge is relatively scarce, often residing with a limited number of individuals. Furthermore, it frequently takes an informal or implicit form, rooted in intuition rather than well posed laws or equations.

Knowledge representation The knowledge representation is the formalization used as an interface with the ML algorithm. In the following, we will briefly overview the most widespread knowledge representations employed in informed ML (some of them are depicted in fig. 3.1).

A *differential equation* is a mathematical equation that involves one or more derivatives of an unknown function. They are usually employed to describe how a function or a system of functions change over time or space. Differential equations are classified into various categories depending on the highest order of the derivatives involved or whether the functions are linear or not, but two of them are the most common: ordinary differential equations (ODEs) and partial differential equations (PDEs). ODEs involve derivatives with respect to a single independent variable, for example, time when describing a dynamical system. On the other hand, PDEs involve derivatives with respect to multiple independent variables, e.g., Navier-Stokes equations used to model fluid dynamics.

Logic rules [25] [26] are fundamental principles that allow the formalization of knowledge about entities and their relationships. Logic rules ensure consistency and validity in reasoning and inference. A rule consists of boolean expressions combined with operators such as OR, AND, and negation.

An *algebraic equation* is a mathematical statement that asserts the equality of two algebraic expressions. Algebraic expressions are combinations of variables, constants, and mathematical operations like addition, subtraction, multiplication, division, and exponentiation. Algebraic equations are used to describe relationships between quantities and to find values of variables that satisfy the given equation. Notable examples are the physics laws but also the declarative formulation of an optimization problem.

Probabilistic graphical models are frequently used to model cause-effect relations, especially for expert knowledge. One such an example are Bayesian networks [27]: a probabilistic graphical model capable of handling incomplete data and update beliefs with new evidence.

Simulators play a crucial role in replicating physical phenomena that are challenging to reproduce in real-world settings, like fluid dynamics. These tools serve as approximate yet precise representations of physical systems. They involve solving a system of complex equations via numerical solvers. The flexibility of simulators is further highlighted by their ability to be configured with various parameters, enabling detailed exploration and analysis of specific scenarios.

Knowledge integration Informed ML techniques inject prior knowledge into one of the four steps of the classical ML pipeline. Informed ML methods that act on the training data usually augment the available dataset via simulation or expert feedback. When acting on the hypothesis set, the goal is to find good candidates based on the available knowledge, e.g., when classifying images, employing neural architectures designed for grid-like input data (e.g., convolutional neural networks [28] [29]). Many approaches focus on the learning part of the ML pipeline, for example, by adding regularization terms to the loss function to encode a well-established rule. Finally, informed ML methods might alter the final hypothesis, forcing it to reflect the prior knowledge.

Symbolic knowledge integration In this thesis, our focus is on symbolic knowledge—knowledge formalized through symbols and rules, which is both comprehensible to humans and easy to manipulate by machines. For instance, an optimization problem can be formalized by a domain expert in a declarative manner. Likewise, algebraic and differential equations are effective tools for depicting complex systems in a format amenable to human understanding. Moreover, logic programming offers a suite of symbols and rules for encapsulating general concepts and commonsense knowledge.

The integration of symbolic knowledge in the inference phase of ML models is relatively straightforward, typically involving the evaluation of outputs from both the ML model and the symbolic engine. However, the true challenge lies in training ML models to effectively leverage symbolic knowledge. This endeavor faces numerous obstacles. For instance, neural networks are commonly trained via

backpropagation [30], necessitating differentiable operations. Yet, computing gradients for operations that handle symbolic knowledge is often complex. Furthermore, the training stability of data-driven methods may be compromised by the incorporation of prior knowledge, demanding cautious approach.

Given the diverse origins of symbolic knowledge, the research community has proposed specific integration methodologies. The subsequent sections of this chapter will summarize state-of-the-art informed ML techniques that utilize symbolic knowledge, categorizing them based on the type of representation, namely: 1) *algebraic equations*, 2) *differential equations*, 3) *simulation results*, 4) *Bayesian networks* and 5) declarative formulation of an *optimization problem*.

3.1 Algebraic equations

One effective method to inject knowledge in the form of algebraic equations into an ML algorithm involves incorporating a regularization term in the loss function. This modification impacts the learning stage of the pipeline. This solution is frequently adopted in physics to inject some well-known laws. In supervised learning scenarios, eq. (2.15) is adapted as shown below:

$$h^* = \arg\min_{h \in \mathcal{H}} \left\{ \lambda_s \sum_{k=1}^{K} \mathcal{L}_s(y_k, \hat{y}_k) + \lambda_p \mathcal{L}_p(f(x_k)) \right\} \tag{3.1}$$

In this equation, $\mathcal{L}_s$ denotes the standard supervised learning component (such as MSE or categorical cross-entropy), while $\mathcal{L}_p$ represents the regularization term, designed to align predictions with physical laws. The weights λ_s and λ_p correspond to the supervised and physics-informed components, respectively.

For instance, [31] introduced Physics-guided Neural Networks (PGNNs), which incorporate physics knowledge in the form of algebraic equations as a regularization term in the loss function, specifically for lake temperature modeling. This method leverages the principle that water density increases monotonically with depth. Consequently, for any two depths d_1 and d_2 at a given timestep t, where $d_1 < d_2$, the following inequality must hold:

$$\rho[d_1, t] - \rho[d_2, t] \leq 0$$

Considering a grid with n_d depth levels and n_t timesteps, the density difference for consecutive depths d_i and d_{i+1} (where $d_i < d_{i+1}$) is calculated as:

$$\Delta[i, t] = \hat{\rho}[d_i, t] - \hat{\rho}[d_{i+1}, t]$$

A positive value in $\Delta[i, t]$ indicates a violation of the physical law. Thus, the regularization term is defined as:

$$\mathcal{L}_p(\hat{\rho}) = \frac{1}{n_t(n_d - 1)} \sum_{t=1}^{n_t} \sum_{i=1}^{n_d - 1} \max(0, \Delta[i, t])$$

This approach exemplifies how algebraic equations can model simple physical properties such as monotonicity and feasible ranges [32]. Notably, physics-informed regularization provides a form of weak supervision, making it particularly valuable for datasets where human annotations are scarce or challenging to obtain. For example, [33] demonstrates the feasibility of training CNNs for computer vision tasks without explicit supervision. They utilized the parabolic equation of free fall to detect

objects, applying the equation $y_i = y_0 + v_0(i\Delta t) + a(i\Delta t)^2$, where y_i is the object's height, y_0 and v_0 are its initial height and speed, $a = -9.8m/s^2$ is the acceleration due to gravity, i is the frame index, and Δt is the duration between frames.

However, adding a regularization term is not the sole approach to embed physics knowledge in the form of algebraic equations during learning. An alternative strategy involves formulating these equations as constraints. For instance, [34] [35] [36] describe various methodologies. [35], specifically, augments a linear support vector machine (SVM) classifier with prior knowledge encapsulated in polyhedral sets corresponding to each class. These sets, defined by a series of equality and inequality constraints, represent regions within the input space. Similarly, [36] translates nonlinear known implications into linear inequalities, subsequently solving the learning problem as a LP problem.

An alternative approach to integrate algebraic equations into ML algorithms involves encoding these equations directly into the ML model's architecture. This method effectively constrains the hypothesis set $\mathcal{H}$. For instance, Lu et al. [37] incorporated physics laws into the intermediate layers of a neural network, targeting applications in electrochemical micro-machining. Similarly, Ramamurthy et al. [38] utilized physics-based knowledge combined with the concept of the mean of multiple computations [39] to enhance the sample efficiency of a RL agent designed for robotic tasks.

The data preparation stage in ML pipelines extends beyond merely considering the dataset size; it crucially involves the selection and engineering of input features. Algebraic equations offer a method to enrich the dataset with new features that encapsulate physical properties or constraints. A pertinent example of this is found in the work of Ladičký et al. [40]. By relying on the Navier-Stokes equations, they expanded the initial feature vector to include additional features such as viscosity, pressure, surface tension, and incompressibility. This enriched feature set enabled them to efficiently train a regression forest [41], significantly reducing the computational resources required to predict subsequent frames in a fluid simulation process.

Algebraic equations are also used to incorporate symbolic knowledge provided by an expert [32, 36]. For instance, constraints like monotonicity or feasible ranges are typically represented algebraically. In [32], constraints are added to the loss function as:

$$\arg\min_{h\in\mathcal{H}} \mathcal{L}(y, h(x)) + \lambda_D \mathcal{L}_D(h(x)), \tag{3.2}$$

where x and y denote input and target features, $\mathcal{L}$ is the standard supervised loss, and $\mathcal{L}_D$ represents the domain knowledge loss, weighted by λ_D.

In [36], the authors present a distinct approach, employing polynomial regression where the hypothesis set is defined by:

$$\hat{y}_\theta(x) = \sum_{|\alpha|\leq m} \theta_\alpha x^\alpha, \tag{3.3}$$

with $\alpha = (\alpha_1, \ldots, \alpha_D)$ and total degree $m \in \mathbb{N}$. Monotonicity constraints are enforced by controlling the sign of partial derivatives:

$$\sigma_j \frac{\partial \hat{y}_\theta(x)}{\partial x_j} \geq 0, \quad \forall j \in J, x \in X, \tag{3.4}$$

where $\sigma_j \in \{-1, 0, 1\}$ indicates the desired monotonicity for coordinate j.

This section explores integrating algebraic equations into ML algorithms, primarily through regularization in the loss function and constraints formulation. Examples include Physics-guided Neural Networks for lake temperature modeling and polynomial regression with monotonicity constraints. These

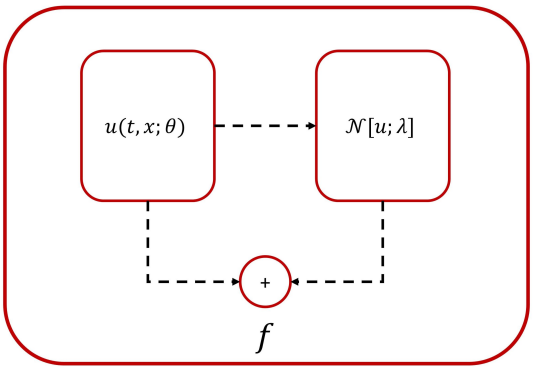


Figure 3.2: Schematic overview of the PINNs computation.

methods enable adherence to physical laws and expert knowledge, enhancing prediction accuracy and interpretability. Additionally, the use of algebraic equations in data preparation and feature engineering is highlighted, illustrating their broad applicability in ML pipelines.

3.2 Differential equations

Differential equations play a pivotal role in describing the dynamics of systems evolving over time and space and are extensively used in physics. These equations are primarily utilized in two steps of ML pipelines: to enhance the learning process and to define custom architectures that incorporate prior physics knowledge.

An early attempt to solve differential equations with neural networks is provided in [42]. The solution is computed as the sum of two terms: the first term satisfies the initial and boundary conditions whereas the second term is provided by a neural network which is trained to satisfy the differential equation. The method has been applied to both ordinary and partial differential equations.

A seminal contribution to the fusion of differential equations and neural networks are Physics-Informed Neural Networks (PINNs), as introduced by Raissi et al. [43]. In this framework, a neural network is trained in a supervised manner to predict the state of a dynamical system. The authors consider nonlinear differential equations of the general form:

$$u(t, x) + \mathcal{N}[u, \lambda] = 0, \quad \text{s.t } x \in \Omega, t \in [0, T] \tag{3.5}$$

where $u(t, x)$ represents the system state at time t and x is the observable vector within domain $\Omega \subset \mathbb{R}^d$. The nonlinear operator $\mathcal{N}[\cdot, \lambda]$ is parameterized by λ, and $t \in [0, T]$ denotes the time interval. An illustrative example is the resistor-capacitor circuit with a constant voltage generator which is described by a simple ODE: $\mathcal{N}[u, \lambda] = \lambda(V_0 - u(t))$, with $\lambda = \frac{1}{\tau}$ and $\tau = R \cdot C$ representing the circuit's time constant, V_0 is the voltage of the source and $u(t)$ is the voltage value at time t.

PINNs primarily serve two functions: 1) computing data-driven solutions of partial differential equations, and 2) discovering partial differential equations in a data-driven manner.

For continuous time models with known parameters λ, the first task is addressed by using PINNs as follows. The PINN is defined by:

$$f(t,x) = u(t,x;\theta) + \mathcal{N}[u] \tag{3.6}$$

Here, $u(t,x;\theta)$, approximating the system state, is represented by a neural network with parameters θ. The loss function is then given by:

$$\mathcal{L}(\theta) = \frac{1}{N_u}|u(t_u^{(i)}, x_u^{(i)};\theta) - u^{(i)}|^2 + \frac{1}{N_f}|f(t_u^{(i)}, x_u^{(i)};\theta)|^2 \tag{3.7}$$

where $\left\{t_u^{(i)}, x_u^{(i)}, u^{(i)}\right\}_{i=1}^{N_u}$ are the initial and boundary training data for $u(t,x)$, and $\left\{t_f^{(i)}, x_f^{(i)}\right\}_{i=1}^{N_f}$ are collocation points for $f(t,x)$.

In discrete time models, Runge-Kutta numerical solvers [44] can be applied to Eq. 3.5 to obtain the system state at the next timestep, u^{n+1}, as:

$$u_i^n = u^{n+c_i} + \Delta t \sum_{j=1}^{q} a_{ij}\mathcal{N}[u^{n+c_j}], \quad i = 1,\ldots,q, \tag{3.8}$$

$$u_{q+1}^n = u^{n+1} + \Delta t \sum_{j=1}^{q} b_j \mathcal{N}[u^{n+c_j}] \tag{3.9}$$

where $u^{n+c_j}(x) = u(t^n + c_j\Delta t, x)$ for $j = 1,\ldots,q$. The specific numerical solver employed is dependent on the chosen parameters a_{ij}, b_j, and c_j, which define the time stepping scheme. The neural network computes $\left[u^{n+c_1}(x),\ldots,u^{n+c_q}(x), u^{n+1}\right]$, and the output of the PINN is $\left[u_1^n(x),\ldots,u_q^n(x), u_{q+1}^n(x)\right]$.

The second task that PINNs can achieve is the data-driven discovery of partial differential equations, specifically identifying the parameters λ that optimally align with the observed data. In this context, the PINN formulates the function:

$$f(t,x;\theta,\lambda) = u(t,x;\theta) + \mathcal{N}[u;\lambda] \tag{3.10}$$

Here, the state $u(t,x;\theta)$ is approximated using a neural network, and λ denotes the set of parameters within the PINN. The optimization process involves training not only the neural network parameters θ but also the PINN parameters λ, by minimizing the loss function 3.7. Considering the RC circuit example, the PINN is expressed as:

$$f(t,x;\theta,\lambda) = u(t,x;\theta) + \frac{1}{\tau}\left(V_0 - u(t,x;\theta)\right) \tag{3.11}$$

where $\lambda = \{\tau, V_0\}$ are the trainable parameters of the PINN.

In the study conducted by Yang et al. [45], physics knowledge is integrated into generative models in the form of differential equations. Specifically, the conditional probability distribution of the model, given the latent variable, is constrained by established physical laws. This relationship is represented as follows:

$$p(u|x,t,z),\ z \sim p(z),\ \text{s.t. } u(t) + \mathcal{N}[u] = 0 \tag{3.12}$$

In this formulation, $p(u|x,t,z)$ denotes the conditional probability distribution of the state variable u given the observable features x, time t, and latent variable z. The latent variable z follows the distribution $p(z)$, and the constraint $u(t) + \mathcal{N}[u] = 0$ embeds the physics law into the generative model.

In the preceding discussion, the focus has been on deterministic models. However, quantifying the model's confidence in the solutions of differential equations is often a critical concern. Addressing this, Zhu et al. [46] propose a novel approach that involves the physics-constrained training of a probabilistic model. This methodology aims to estimate distributions of possible solutions rather than a single solution, thereby providing a measure of uncertainty and confidence in the model's predictions.

So far, our focus has been on physics-informed ML methods that leverage differential equations to augment the learning phase in ML pipelines. Differential equations can be seamlessly integrated into the hypothesis set, for instance, through the development of specialized neural architectures. This concept is extensively discussed in literature [47] [48] [49] [50] [51].

A particularly noteworthy instance is Thermal Neural Networks (TNNs) [51]. In TNNs, complex thermal models are substituted with more manageable lumped-parameter thermal networks (LPTNs). LPTNs, while defined by a set of simple ODEs, incorporate prior knowledge about geometry and material properties. Remarkably, certain unknown terms in TNNs are estimated using a universal approximator, like a neural network. Due to their simplified structure, TNNs facilitate real-time estimations, finding practical applications in temperature prediction for electric power systems [52] [53] and electric machines [54] [55] in automotive and automation industries. A TNN is mathematically represented as:

$$C_i(\theta(t))\frac{d\theta_i}{dt} = P_i(\theta(t)) + \sum_{j \in M_i} \frac{\theta_j - \theta_i}{R_{i,j}(x(t))} + \sum_{j=1}^{n} \frac{\tilde{\theta}_j - \theta_i}{R_{i,j}(x(t))} \tag{3.13}$$

Here, C symbolizes the thermal capacitance, P the power loss, and $R_{i,j}$ the bidirectional thermal resistance between nodes i and j of the thermal model, with $x(t)$ being the observable features at time t and θ the temperature at time t, where $M = \{1, 2, \ldots, m\}$. Sources for which the temperature is measurable during operations (such as ambient temperature) are incorporated as ancillary nodes whose temperature is $\tilde{\theta}$. In TNNs, the thermal resistances $R_{i,j}$ and capacitances C_i are estimated using neural networks. As evidenced in [51], TNNs not only enhance predictive accuracy relative to purely black-box models but also offer superior interpretability.

A more direct approach involves employing a neural architecture that reflects the differential equation modeling the system under study. In this spirit, [47] employs a convolution-deconvolution neural network to address the task of forecasting sea surface temperature, which is modeled using advection-diffusion equations. The design of this approach is versatile, enabling its application to a diverse broad range of transportation problems governed by advection-diffusion principles.

Similarly, [48] introduces Deep Lagrangian Networks (DeLaN), a neural network architecture that integrates Lagrangian mechanics directly through the Euler-Lagrange equation, which is a second-order ODE. Although DeLaN does not achieve state-of-the-art results, it demonstrates increased robustness and sample efficiency. Furthermore, it facilitates real-time robot tracking control tasks.

In a similar spirit, [50] integrates rigid body dynamics into a neural model through a linear complementarity problem (LCP) [56], providing an analytical method for differentiating through the optimal solution of the LCP. This integration leads to the creation of a completely differentiable end-to-end simulator.

This section explores the integration of differential equations into ML algorithms, highlighting their use in enhancing learning processes and defining custom architectures. Early examples include Physics-Informed Neural Networks for solving differential equations and predicting system states, and Physics-guided Neural Networks that incorporate physical laws as regularization terms in supervised

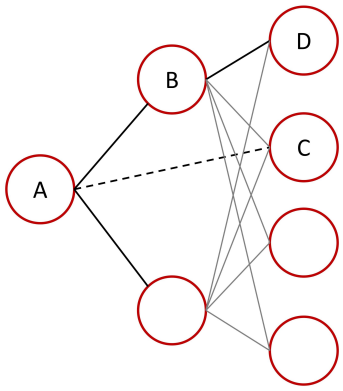


Figure 3.3: An example of KBANN. Gray lines represent learnt propositions, absent from the initial ruleset.

learning. The section also discusses how differential equations facilitate the creation of specialized neural architectures, like Thermal Neural Networks, which estimate complex thermal models using simplified ODEs. Additionally, it covers approaches like Deep Lagrangian Networks (DeLaN) that embed mechanical principles directly into neural network architectures for tasks such as real-time robot control.

3.3 Logic rules

Logic rules serve as pivotal tools in representing worldly concepts and commonsense knowledge. They articulate properties of objects (e.g., `SQUARE(x)` $\implies$ `POLYGON(x)` $\wedge$ `SIDES(x,4)`) and relationships (e.g., `isDaughterOf(Anna, Bob)` $\iff$ `isFatherOf(Bob, Anna)`). In ML, these rules are predominantly integrated into the hypothesis set or the learning algorithm of an ML pipeline.

Two principal strategies exist for incorporating logic rules into the model architecture: *neuro-symbolic methods* [57–62] and *statistical relational learning* [63–71].

A pioneering neuro-symbolic method, Knowledge-based Artificial Neural Network (KBANN), translates a set of rules into a neural network structure. It uses positively/negatively weighted connections to represent existing/negated relations, respectively. Hidden units in this network are relations absent from the initial rule set. These weights are further refined through the learning process. Figure 3.3 shown an example of KNANNs where the given ruleset is $\{A \implies B \wedge \neg C,\ B \implies C\}$, and gray lines are additional connections that do not reflect the initial set of rules. While KBANN focuses on propositional logic, subsequent advancements like CILP [58] and CILP++ [59] extend support to first-order logic.

Statistical relational learning [68], probabilistically integrates logic rules into the hypothesis set. A notable method within this domain is Markov logic networks (MLNs) [66], which employ Markov networks—undirected probabilistic graphical models with nodes representing random variables and edges denoting relations. MLNs blend probabilities into knowledge, articulated through first-order logic rules, with each predicate assigned a weight indicating its likelihood.

Closely aligned with MLNs is Probabilistic Soft Logic [67], distinguished by two key features. Firstly, it relaxes boolean truth values to the continuous interval $[0, 1]$. Secondly, it permits first-order formulas in conjunctive forms. These modifications render the inference process a convex optimization, thereby enhancing computational efficiency.

A recent innovation in statistical relational learning is DeepProbLog [69]. Building upon the probabilistic logic language ProbLog [72], DeepProbLog incorporates neural models to represent some grounded facts. It enables joint training of neural networks and probabilistic fact parameters through gradient descent, leveraging examples for learning.

Another prevalent application of logic rules in informed ML occurs during the learning phase, often through a regularization term [73–78]. Semantic Based Regularization (SBR) enables an ML model to concurrently learn from perceptual data and prior knowledge. This prior knowledge is derived from first-order logic rules, which are transformed into a continuous format using fuzzy logic [79, 80] and t-norm functions. Such a transformation allows training with gradient-based algorithms and backpropagation. Differing from traditional logic where predicates are binary ($\{0,1\}$), fuzzy logic permits continuous values within the interval $[0,1]$. A t-norm is defined as a function $t : [0,1] \times [0,1] \to [0,1]$, characterized by continuity, commutativity, associativity, monotonicity, and having the neutral element 1. In continuous representation, these logic rules are incorporated as a regularization term in the loss function, yielding the loss:

$$\mathcal{L}(f) = \|f\|_{\mathcal{H}}^2 + \sum_{h=1}^{H} \lambda_h (1 - \phi_h(f)), \tag{3.14}$$

where $\phi_h(f)$, with $0 \leq \phi_h(f) \leq 1$ and $h = 1, \ldots, H$, represents the t-norm form of the logic rules, and λ_h is the weight for the h-th constraint. Equation (3.14) is applicable even in the absence of labeled data (unsupervised setting), simply by enforcing the fuzzy logic rules.

Hu et al. [77] proposed a model to learn knowledge confidence jointly with the neural network. In their later work [78], they introduced an iterative student-teacher learning framework. In this framework, the teacher neural network is derived from projecting the student network into a rule-regularized subspace. The student network is trained to balance between emulating the teacher's output and accurately predicting the true label.

This section explores the integration of logic rules, a key representation of word concepts and commonsense knowledge, into ML pipelines. We examine two primary integration strategies: neuro-symbolic methods, such as KBANN and its advancements, which translate logic rules into neural network structures; and statistical relational learning, notably Markov logic networks and Probabilistic Soft Logic, which probabilistically blend logic rules with hypotheses. More recently, DeepProbLog integrates neural models into probabilistic logic for joint training. Additionally, we discuss the use of Semantic Based Regularization for concurrent learning from data and knowledge, employing fuzzy logic and t-norm functions for continuous logic rule representation. This approach highlights the sophisticated application of rule-regularized ML models in both supervised and unsupervised learning settings, advancing the field of informed ML.

3.4 Simulation results

Simulation software and tools are extensively used in physics to run experiments that are challenging to replicate in real-world settings. They are usually modeled by an human expert by relaying on a mix of algebraic and differential equations. The output from these simulations serves as a valuable source of knowledge, which can be effectively integrated into ML pipelines, often enhancing the training data [1, 81–85].

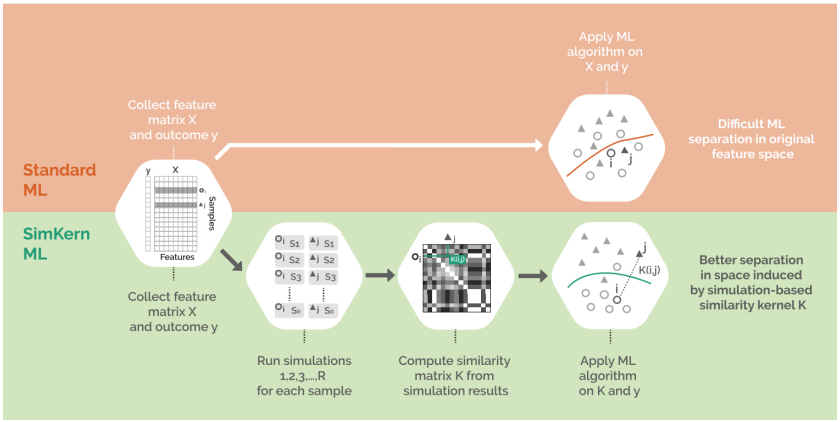


Figure 3.4: Comparison between the SimKern and a traditional ML workflow, as described in [1]

.

Simulation results can be used to generate additional input features. In [1], instead of relying solely on raw training samples, the authors execute multiple simulations for each sample, subsequently calculating a similarity score based on the outcomes. According to the experimental results with various ML algorithms, this simulation-based similarity matrix yields a more distinct separation of data samples in the input space compared to using raw features alone.

When real data are scarce or expensive to collect, simulation outputs can serve as a more consistent and reliable target. For instance, [31] employs simulations in lake temperature modeling to supplement missing historical data.

However, the simulation process can be computationally demanding. In such cases, the simulation's output may become the target feature, enabling the training of an ML algorithm to mimic the simulation process [82]. The resultant surrogate model can match the accuracy of the original simulator while operating at a significantly faster rate.

In robotics, acquiring real-world labeled data is often expensive or impractical thus preventing deep neural networks from learning and generalizing effectively. In a scenario where one can rely entirely on synthetic data for training, [83] explores how CNNs can learn the trajectories of toy wooden blocks using data generated through a 3D game engine. Their experimental analysis reveals that these networks are capable of generalizing to real-world images and previously unseen conditions, such as varying the number of blocks.

In many real-world scenarios, a non-negligible discrepancy may exist between true and simulated data distributions. To bridge this gap, [85] introduces SimGAN, an approach inspired by generative adversarial networks (GANs) [86]. This method aims to enhance the realism of simulator-generated images. SimGAN consists of two networks: a refiner, which strives to improve the simulated images, and a discriminator that classifies images as real or simulated having access to real but unlabeled images. SPIGAN [81] is similar to SimGAN, but it additionally leverages physics knowledge by accessing the simulator's internal information about the world.

Besides generating additional data, simulator can be employed as a part of the neural architecture [87] [88] [89], as shown in fig. 3.5. This approach is particular helpful when the simulator is not very accurate and only a small amount of data are available.

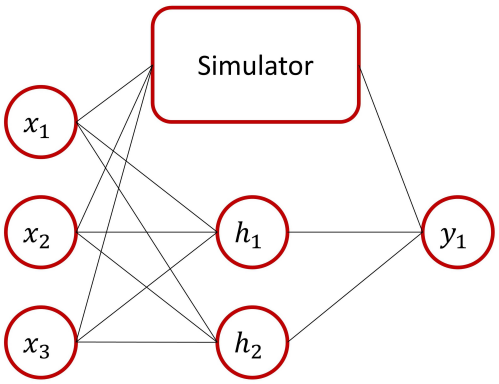


Figure 3.5: A low fidelity simulator can be incorporated in the architecture of a neural network.

.

Simulation tools, integral in physics for replicating complex experiments, are increasingly utilized in ML pipelines, often enhancing training data with their outputs modeled through algebraic and differential equations. These simulations not only provide additional input features and reliable targets for scenarios with limited real data but also aid in creating efficient surrogate ML models. Techniques like SimGAN and SPIGAN have been developed to reconcile differences between simulated and real-world data, improving realism and accuracy. Furthermore, simulations are sometimes integrated directly into neural network architectures, particularly beneficial when data is scarce or simulations lack precision.

3.5 Bayesian networks

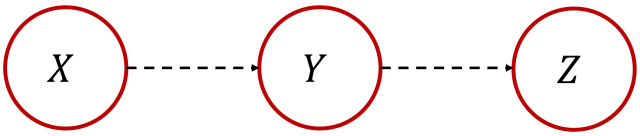
Bayesian networks are probabilistic graphical models that allow to formalize causal relations. A Bayesian network is a directed acyclic graph (DAG) where each node represents a random variable and the edges encode conditional dependencies. Formally, for a set of random variables $\{X_1, X_2, \ldots, X_n\}$, a Bayesian network represents the joint probability distribution as:

$$P(X_1, X_2, \ldots, X_n) = \prod_{i=1}^{n} P(X_i | Parents(X_i))$$

where $Parents(X_i)$ are the nodes with edges directed towards X_i. A simple illustrative example is depicted in fig. 3.6.

The literature is rich with works that employ Bayesian networks in informed ML [2] [90] [91] [92] [93] [94]. In the field of clinical medicine, where data can be scarce and the development of a full data-driven approach is not always viable, Bayesian networks have shown notable applicability. For instance, in [2], the authors design a decision support system for clinical settings that integrates expert knowledge with empirical data, demonstrating the practical value of Bayesian networks in complex decision-making scenarios. As depicted in fig. 3.7, expert knowledge and meta-analysis are used to define the structure of a Bayesian network, whose parameters are then learned leveraging the available data.

Besides clinical medicine, incorporating prior knowledge in Bayesian networks finds application in industrial problems, where expert knowledge is usually gathered thanks to many years spent working on a specific system. For example, [93] explores the combination of principal knowledge and empirical



$$P(X,Y,Z) = P(Z|Y) \cdot P(Y|X) \cdot P(X)$$

Figure 3.6: A simple Bayesian network and the resulting decomposition of the joint probability distribution.

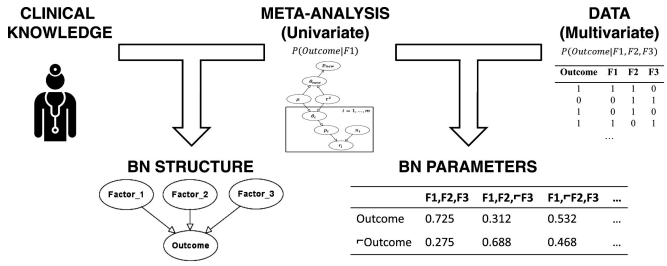


Figure 3.7: Overview of the Bayesian network-based decision support system developed in [2] (image from [2]).

data into a Bayesian network to identify the quality characteristics of mechanical products. Similarly, [94] investigates the fusion of expert knowledge and Bayesian networks for risk and safety analysis in the chemical and process industry, while [95] demonstrates how Bayesian networks can improve the efficiency and accuracy of production processes in semiconductor manufacturing.

Due to the lack of statistical expertise, the expert knowledge belief's might be biased [96]. To compensate for this limitation, [90] proposes designing Bayesian networks while preserving the expected value, independent of the expert judgment, of the random variables in the available dataset. This approach not only mitigates the bias inherent in expert opinions but also enhances the explanatory capabilities of models in complex domains like medicine and industry, providing a more robust framework for decision-making under uncertainty.

In summary, Bayesian networks offer a flexible and robust framework for incorporating expert knowledge in various fields, effectively handling the inherent uncertainty in complex systems. Their applications range from clinical decision support to industrial process optimization, demonstrating their versatility and importance in data-driven decision-making.

3.6 Declarative formulation of an optimization problem

Optimization problems are typically defined by human experts in a declarative manner, specifying decision variables, constraints, and an objective function to be minimized or maximized. This formulation provides structured source knowledge and delineates the properties of an optimal solution.

OR methods efficiently explore the solution space of these problems. However, the computational complexity of many optimization problems, particularly NP-hard ones, exponentially increase with the size of the input (number of decision variables). Despite this, state-of-the-art OR methods can often find optimal solutions within reasonable, albeit sometimes really long, time.

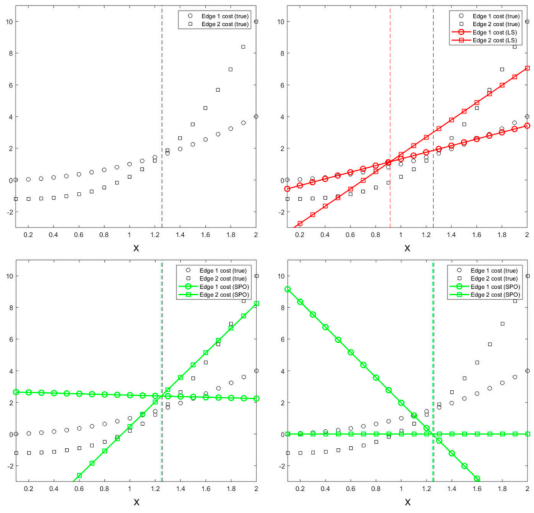


Figure 3.8: Illustrative example on how the prediction error might affect the dowstream task loss (image from [3]).

ML techniques have emerged as a promising tool to develop efficient heuristics [97, 98]. These methods, often employing deep neural networks, are trained to directly output optimal solutions in a supervised fashion or to optimize the objective function via RL [98–103]. However, they require extensive data, struggle with combinatorial structures, and have limited generalizability to larger instances [104]. Moreover, the inherent combinatorial nature of problems is often obscured in the learned function, posing interpretability challenges.

An alternative approach involves using neural networks to enhance heuristic algorithms [105–107] or to aid the search process [108–112]. While the former aligns with end-to-end methodologies, the latter more effectively utilizes problem formulations by integrating with existing solvers. Notably, these methods assume complete knowledge of the problem at solution time.

In real-world constrained optimization problems, certain parameters of the model might be *unknown at solution time*. For instance, in package delivery problems, travel times might be uncertain due to traffic and weather conditions. Similarly, in production scheduling, customer demands are often unpredictable and influenced by factors like the time of year or inflation rates. In such scenarios, ML models can estimate these unknown parameters based on observable features.

In a typical ML workflow, models are trained to maximize accuracy, which involves minimizing the MSE in regression problems or the categorical cross-entropy in classification problems, thereby aligning predictions closely with ground-truth values. However, this approach overlooks the influence of prediction errors on the loss incurred in the subsequent optimization task, i.e., the cost associated with the optimization problem.

Figure 3.8 illustrates this concept. Consider a problem where the objective is to choose between two solutions (edges) to minimize cost, formulated as the following binary LP:

$$\begin{aligned} \min \quad & c_1 z_1 + c_2 z_2 \\ \text{s.t.} \quad & z_1 + z_2 = 1 \end{aligned}$$

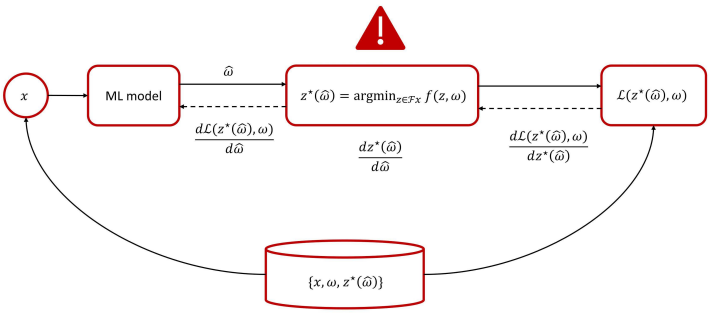


Figure 3.9: Schematic overview of a DFL pipeline. $\mathcal{F}$ denotes the feasible region and f the problem objecitve function.

Here, the edge costs are functions of an observable feature x. The relation between the input feature and the cost of the first edge is represented by circles, and for the second edge by squares in the figure. The optimal decision policy, as deduced from the figure, involves selecting the second edge if $x \leq \approx 1.2$, and the first edge otherwise. Employing a simple linear regression model as a predictor would result in errors due to the polynomial nature of the relationship between x and edge costs. Such errors, depicted in the top right chart of fig. 3.8, hinder the identification of the optimal policy.

The lower part of fig. 3.8 demonstrates two different predictive models. Although these models exhibit non-zero prediction errors, possibly higher than the previous model, they successfully identify the optimal decision policy. This is because these models were trained by incorporating knowledge of the optimization problem, focusing on *minimizing the task loss* (i.e., $c_1x_1 + c_2x_2$) rather than maximizing accuracy [3]. Training an ML model to minimize the task loss introduces challenges, which have been extensively explored in the field of decision-focused learning (DFL) [113]. DFL contrasts with traditional prediction-focused learning (PFL), which aims at maximizing model accuracy.

A primary challenge in DFL is computing the derivative of the task loss to train the predictive model using gradient descent. The derivative of the task loss can be expressed using the chain rule of differentiation as follows:

$$\frac{d\mathcal{L}(z^\star(\hat{\omega}), \omega)}{d\theta} = \frac{d\mathcal{L}(z^\star(\hat{\omega}), \omega)}{dz^\star(\hat{\omega})} \cdot \frac{dz^\star(\hat{\omega})}{d\hat{\omega}} \cdot \frac{d\hat{\omega}}{d\theta} \tag{3.15}$$

where $\hat{\omega}$ and ω represent the predicted and ground-truth optimization problem parameters, respectively, and θ denotes the ML model's trainable parameters. Calculating the first and last terms of the chain is straightforward: $\mathcal{L}$ is usually differentiable with respect to the solution $z^\star(\hat{\omega})$, and the term $\frac{d\hat{\omega}}{d\theta}$ can be computed using automatic differentiation tools like TensorFlow [114] or PyTorch [115]. However, computing the second term $\frac{dz^\star(\hat{\omega})}{d\hat{\omega}}$ is challenging for two reasons: 1) the mapping $\hat{\omega} \rightarrow z^\star(\hat{\omega})$ (i.e., computing the optimal solution) may lack a directly differentiable closed-form expression; 2) for many practical optimization problems (e.g., LP, ILP, MILP), this mapping is either non-differentiable or its gradient is zero, hindering gradient-based optimization.

The subsequent sections provide an overview of DFL methods addressing these challenges, categorized into two main approaches (schematically depicted in fig. 3.10: 1) treating the optimization solver as a layer within a neural network architecture; 2) employing surrogate losses that compute $\frac{d\mathcal{L}(z^\star(\hat{\omega}), \omega)}{d\hat{\omega}}$ without explicitly requiring the derivative of the solution process. For a comprehensive survey, readers are referred to [113].

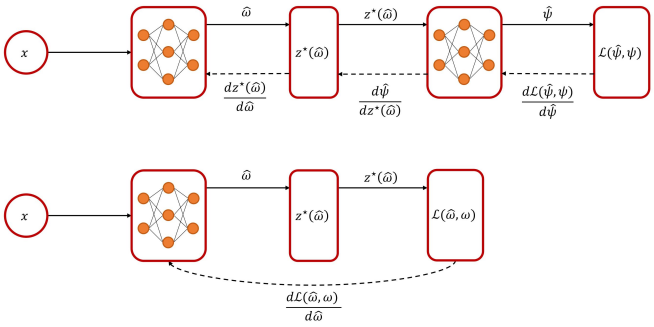


Figure 3.10: Solver-as-a-layer (upper) and surrogate loss (lower) DFL approaches.

3.6.1 Solver-as-a-layer

Amos et al. [116] pioneered the integration of an optimization solver as-a-layer of a deep neural network. Their methodology, named OptNet, embeds QP problems, as defined in eq. (2.8). The core concept involves employing implicit differentiation of the Karush-Kuhn-Tucker (KKT) optimality conditions [5]. This approach enables the computation of exact gradients of the optimal solution $z^\star$ with respect to the parameters of the optimization problem. The gradients are obtained by solving the following system of linear equations:

$$\begin{bmatrix} Q & G^T & A^T \\ D(\lambda^\star)G & D(Gz^\star - h) & 0 \\ A & 0 & 0 \end{bmatrix} \begin{bmatrix} dz \\ d\lambda \\ dv \end{bmatrix} = \begin{bmatrix} -dQz^\star - dq - dG^T\lambda^* - dA^T v^* \\ -D(\lambda^*)dGz^\star + D(\lambda^*)dh \\ -dAz^\star + db \end{bmatrix}$$

Here, λ and v represent the Lagrangian multipliers, while D denotes a diagonal matrix. Additionally, the authors developed a batch QP solver optimized for GPUs, demonstrating reduced computation time compared to traditional commercial solvers like Gurobi [117] and Cplex [118] used in this setup. As a practical illustration of its utility, OptNet was successfully applied to solving Sudoku puzzles without prior knowledge of the game's rules, showcasing generalization capabilities in comparison to a conventional end-to-end CNN architecture.

Following the seminal work of Amos et al. [116], several notable extensions have emerged. Konishi et al. [119] employ gradient boosting algorithm [120] as predictive model. This model, trained in an end-to-end manner, requires the computation of the second-order derivative of the solution, adding a layer of complexity to the approach. Concurrently, Agrawal et al. [121] developed a methodology for differentiating through conic programs, leveraging similar principles to those of [116]. Building upon this, in their subsequent work [122], they proposed a more generalized solver for convex optimization problems, further broadening the applicability and scope of optimization within deep learning architectures.

The methodologies previously outlined address the issue of differentiating through the arg min mapping. However, these approaches encounter limitations when applied to combinatorial problems, such as LP, ILP or MILP. In these cases, the gradient of the task loss is typically undefined or zero. To

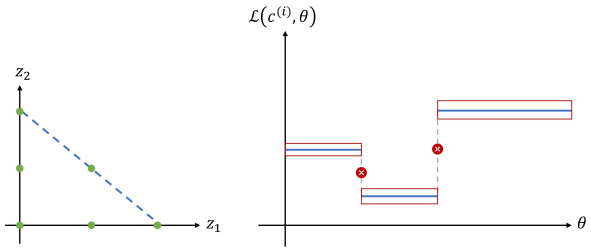


Figure 3.11: Feasible region (left) and task loss (right) for the illustrative ILP. θ are the parameters of the ML model.

illustrate this issue, consider the following ILP:

$$\begin{aligned} \min \quad & c_1 z_1 + c_2 z_2 \\ \text{s.t.} \quad & z_1 + z_2 \leq 2 \\ & z_1 \geq 0, \ z_2 \geq 0 \\ & z \in \mathbb{Z} \end{aligned}$$

In this scenario, the cost vector $c = (c_1, c_2)$ is not known a priori and must be estimated. The feasible region of this problem, as shown on the left side of fig. 3.11, is finite due to the integer nature of the decision variables. The right side of fig. 3.11 depicts how the task loss $\mathcal{L}(z, \hat{c}) = \hat{c}^T z$ may vary for a single sample. A minor alteration in the predictions can lead to a discrete change in the optimal solution within the feasible set. These changes are graphically represented at the steps of $\mathcal{L}$, where the function becomes discontinuous and, consequently, non-differentiable. In regions where the prediction modifications do not change the solution, $\mathcal{L}$ remains constant, resulting in its gradient being zero.

In LP problems, the optimal solution often lies in the vertices of the feasible region [123], presenting a major challenge in DFL. To address this, Wilder et al. [124] augment the linear objective function with the Euclidean norm of the decision variables, formulating the objective as:

$$z^\star(c) = \arg\max_z c^T z - \mu \|z\|_2^2,$$

creating a continuous mapping $\hat{c} \to z^\star(\hat{c})$. The ML model is trained to predict the cost vector c using backpropagation in the framework of [116]. Rather than the objective function, the task loss is the regret, defined for minimization problems as:

$$Regret(z^\star(\hat{c}, c)) = f(z^\star(\hat{c}), c) - f(z^\star(c), c), \tag{3.16}$$

a concept first introduced in [3].

Mandi et al. [125] introduce a differentiable LP solver based on log-barrier regularization. Assuming an unknown cost vector c, the parametrized LP problem is:

$$z^\star(\hat{c}) = \arg\min_z \hat{c}^T z + \lambda \sum_i z_i \tag{3.17}$$

$$\text{s.t. } Az = b, \tag{3.18}$$

where the solver approximates the LP's homogeneous self-dual (HSD) embedding using an interior point method. Since it solves an LP, this method is computationally more efficient than [124] (which involves solving a QP).

Similar approaches, employing regularization to make solvers differentiable, are used in other works for sorting, ranking [126], and multi-label classification problems [127].

For ILP problems, [124] suggests dropping the integrality constraints, treating the resulting LP as described earlier. Extending this, Ferber et al. [128] introduce Mipaal (MILP as a Layer) for general MILP problems, using cutting plane methods to align the solutions of the LP and corresponding MILP. While Mipaal improves performance in terms of regret, it introduces scalability issues due to the cut generation process.

Besides regularization techniques, an alternative approach to enabling differentiation through solvers in DFL involves smoothing the objective function by randomly perturbing the predictions [129–131].

Pogančić et al. [129] use a linear interpolation as a perturbation for the mapping $\hat{c} \to z^\star(\hat{c})$. This is achieved by interpolating between the points $\hat{c}$ and $\hat{c} + \delta \frac{d\mathcal{L}(z^\star(\hat{c}))}{dz}|_{z=z^\star(\hat{c})}$. Given the derivative of the loss with respect to the solution, $\frac{d\mathcal{L}(z^\star(\hat{c}))}{dz^\star(\hat{c})}$, the derivative of the loss with respect to the cost vector is:

$$\frac{d\mathcal{L}(z^\star(\hat{c}))}{d\hat{c}} \approx \left(z^\star \left(\hat{c} + \delta \frac{d\mathcal{L}(z^\star(\hat{c})}{dz^\star(\hat{c})} \right) - z^\star(\hat{c}) \right). \tag{3.19}$$

This method has been applied to various problems where the input is an image rather than the problem formulation, such as the shortest path problem in Warcraft II images [132], a traveling salesman problem with flag images, and a min-cost perfect matching problem on MNIST images [133].

Sahoo et al. [130] propose using the negative identity matrix for the gradient $\frac{dz^\star(\hat{c})}{d\hat{c}}$ during backpropagation. They address the issue of instability in scale-invariant optimization problems by projecting the cost vector $\hat{c}$ onto the unit sphere.

Berthet et al. [131] present a different approach by modeling the conditional distribution $p(z|c)$, employing the reparametrization trick [134, 135] for generating samples. They perturb c with a random vector $\epsilon\tilde{\eta}$, where $\tilde{\eta}$ is sampled from a probability distribution η and ϵ is a temperature value. The perturbed solution $z^\star_\epsilon$ is considered a sample from $p(z|c)$ for a given ϵ. The derivative of $z^\star_\epsilon$ can be estimated via Monte Carlo sampling:

$$\frac{dz^\star(c)}{dc} = -\frac{1}{\epsilon M} \sum_{m=1}^{M} z^\star(c + \epsilon\eta^{(m)})\nu'(\eta^{(m)})^T. \tag{3.20}$$

Similarly, Niepert et al. [136] also model the conditional distribution $p(z^\star|c)$ but use the Sum-of-Gamma distribution for noise generation.

In conventional approaches, predictive models are constrained to estimating the cost vector and do not allow training in a DFL manner to predict unknown parameters within constraints. To address this limitation, Paulus et al. [137] introduced CombOptNet, a novel method that integrates an ILP solver into the architecture of deep neural networks. This integration allows for the simultaneous prediction of

both cost vectors and constraint parameters. The authors propose the following proxy function:

$$P_{\Delta_k}(A,b) = \begin{cases} \min\limits_j \text{dist}(a_j, b_j; y) & \text{if } y'_k \text{ is feasible and } y'_k \neq y, \\ \sum\limits_j [\![a_j \cdot y'_k > b_j]\!] \text{dist}(a_j, b_j; y'_k) & \text{if } y'_k \text{ is infeasible,} \\ 0 & \text{if } y'_k = y \text{ or } y'_k \notin Y, \end{cases} \tag{3.21}$$

where dist represents the Euclidean distance between a point and a hyperplane and $[\![\cdot]\!]$ are the Iverson brackets. Here, $y'_k = y + \Delta_k$ is defined as an integer point adjacent to y, oriented in the direction of $dy = \sum_{k=1}^{n} \lambda_k + \Delta_k$, with $\Delta_k \in \{-1, 0, 1\}^n$, and $\lambda_k \geq 0$ are scalar values.

3.6.2 Surrogate loss functions

The methods illustrated in the previous section enable the computation of exact or approximated values of $\frac{dz^\star(\hat{c})}{d\hat{c}}$, which can be integrated as a layer in a deep neural architecture. In contrast, the methods described in this section approximate the value of $\frac{d\mathcal{L}(z^\star(\hat{c}),c)}{d\hat{c}}$ where $\mathcal{L}$ is the regret as defined in Equation 3.16. These approaches require the availability of ground-truth cost vectors in the training data.

One seminal DFL work is the Smart "Predict, Then Optimize" (SPO) approach by Elmachtoub et al. [3]. They propose the SPO+ loss as a convex upper bound on the regret:

$$\mathcal{L}_{\text{SPO+}}(z^\star(\hat{c})) = 2\hat{c}^T z^\star(c) - c^T z^\star(c) + \max_{z \in \mathcal{F}} \{c^T z - 2\hat{c}^T z\} \tag{3.22}$$

The SPO+ loss has a useful subgradient given by:

$$z^\star(c) - z^\star(2\hat{c} - c) \in \partial \mathcal{L}_{\text{SPO+}} \tag{3.23}$$

which can be utilized to train the ML model via gradient descent.

Mulamba et al. [138] adopt the noise contrastive estimation (NCE) concept [139], which involves discriminating true data from noise data. In the DFL context, they train a ML model to predict $\hat{c}$ that leads the optimal solution and avoid the non-optimal solutions. Their loss function is defined as:

$$\mathcal{L}_{\text{NCE}}(\hat{c}, c) = \sum_{z' \in S} f(z^\star(c), \hat{c}) - f(z', \hat{c}) \tag{3.24}$$

where $z' \in S$ represents the set of non-optimal solutions. Notably, $\mathcal{L}_{\text{NCE}}$ does not require the computation of $z^\star(\hat{c})$ or its gradient.

The same authors propose an alternative based on self-contrastive estimation (SCE) [140], where the model's most likely output is contrasted against the ground-truth solution:

$$\mathcal{L}_{\text{SCE}}(\hat{c}, c) = \sum_{z' \in S} f(z^\star(c), \hat{c}) - f(z', \hat{c}) \tag{3.25}$$

where

$$z' = \arg\min_{z \in S} f(z, \hat{c}) \tag{3.26}$$

Mandi et al. [141] develop a surrogate loss function based on pairwise learning to rank [142]. The model is trained to predict $\hat{c}$ so that for every pair $(z^\star(c), z')$, where $z' \in S$, the rankings are consistent for both $\hat{c}$ and c. The loss function, which includes a margin $\Theta > 0$, is:

$$\mathcal{L}_{\text{Pairwise}}(\hat{c}, c) = \sum_{z' \in S} \max\left(0, \Theta + (f(z^\star(c), \hat{c}) - f(z', \hat{c}))\right) \tag{3.27}$$

They further extend this approach to the listwise learning to rank framework. The loss is the cross-entropy between the probability distributions induced by $\hat{c}$ and the ground-truth c:

$$\mathcal{L}_{\text{Listwise}}(\hat{c}, c) = -\frac{1}{|S|} \sum_{x' \in S} p_r(x'|c) \log p_r(x'|\hat{c}) \tag{3.28}$$

Recent research has shifted focus to surrogate loss functions in the context of problems with unknown parameters in constraints [143]. In [143], the authors introduce the concept of post-hoc regret. Given the uncertainty of the true feasible region at the time of solution, a correction function, $z^\star(\hat{\omega}) \to z^\star_{\text{corr}}(\hat{\omega}, \omega)$, is required. This function projects the estimated solution, $z^\star(\hat{\omega})$, into the true feasible region. Additionally, a non-negative penalty is imposed for this correction. The post-hoc regret, $P\text{Reg}(\hat{\omega}, \omega)$, is then formulated as follows:

$$P\text{Reg}(\hat{\omega}, \omega) = \text{obj}(x^\star_{\text{corr}}(\hat{\omega}, \omega)) - \text{obj}(x^\star(\omega), \omega) + \text{Pen}(x^\star(\hat{\omega}) \to x^\star_{\text{corr}}(\hat{\omega}, \omega)) \tag{3.29}$$

Hu et al. develop correction and penalty functions specifically for linear packing and covering problems. Furthermore, they extend the method proposed by Mandi et al. [125] by deriving the implicit differentiation of $\frac{\partial P\text{Reg}(\hat{\omega}, \omega)}{\partial \theta}$, where θ denotes the parameters of the neural network.

In a subsequent work, [144] they generalize the Branch and Learn framework [145] to train linear predictive model, via coordinate descent [146], to estimate unknown parameters in the constraints. However, the approach is applicable only for recursively solvable problems.

Since some of these methods are solver-free, they do not only bypass gradient-related issues but also offer computational efficiency as computing the optimal solution can be expensive.

This section delves into the integration of optimization problems with ML, emphasizing how declarative formulations (e.g. decision variables, constraints, and objective functions) can augment data-driven methods. A pivotal advancement in this field is DFL, which redirects the training emphasis of ML models from maximizing accuracy to minimizing task loss, an essential consideration for real-world scenarios. This shift necessitates sophisticated techniques such as incorporating optimization solvers into neural network structures and employing surrogate losses for approximating task loss derivatives. These methods enable more effective and interpretable ML solutions for intricate optimization tasks. While scalability remains a challenge in DFL, recent developments in solver-free approaches have shown promising outcomes. Additionally, there is a growing interest in extending DFL scope beyond predicting cost vectors to include estimating parameters within constraints. Section 6.4 will introduce a promising method that significantly broadens DFL applicability.

Chapter 4

Use cases

Before delving into the details of the designed methods, we will first provide a formal description of the use cases involved.

The first one is an Energy Management System that can be framed as a sequential decision-making problem under uncertainty. Finding the optimal power flows is difficult due to uncertainty in energy production and user demands.

The second use case is a predictive maintenance task which requires to monitor an oil and gas facility to anticipate faults or anomalies. The main challenge here is the presence of a complex equipment with several components. On the other hand, a lot of knowledge is available in the form *expertise* of the plant operator and the components' vendor.

We then investigate a synthetic resistor-capacitor circuit which is often used to model thermal systems.

The last set of problems that we consider are the knapsack and weighted set multi-cover problems. Despite being abstract, they have been studied for a long time in the OR community and thus they provide solid bechmarks. Moreover, they find grounding in many practical use cases, as further detailed in the dedicated section.

For the Energy Management System and the abstract combinatorial optimization problems, the source of knowledge is the *declarative formulation of the problems* themselves that we will leverage for a tighter integration with the ML models.

4.1 Energy Management System

In recent years, power distribution networks have undergone a significant transformation, shifting away from traditional centralized power stations toward the adoption of distributed energy resources (DERs). Unlike conventional power stations, DERs offer modularity and proximity to the locations they serve. DERs draw power from different sources, namely flexible and reliable power sources, e.g. thermal loads, but also highly uncertain ones, such as photovoltaic or wind plants (or more generally renewable energy sources). The inherent variability and unpredictability of these renewable systems necessitate intricate coordination and decision-making processes, typically managed through the implementation of smart grids.

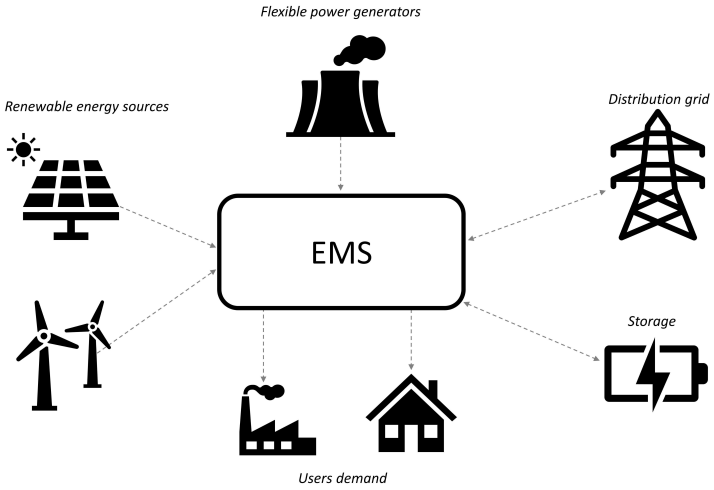


Figure 4.1: Schematic example of an EMS.

Within the context of this thesis, we employ the term "Energy Management System" (EMS) to denote the framework responsible for optimizing the allocation of the most cost-effective power flows from various DERs and, if applicable, an energy storage unit, which can accumulate surplus energy for future use. Our EMS framework aligns with the one proposed in earlier works such as [147, 148], accommodating exogenous uncertainty. This uncertainty arises from uncontrollable deviations in planned consumption loads and the presence of renewable energy sources (RES).

Based on actual energy prices and on the availability of DERs, the EMS needs to decide: 1) *how much energy* should be produced; 2) *which generators* should be used for the required energy; 3) whether the surplus energy should be *stored or sold* to the energy market. Decisions need to be taken and implemented at fixed intervals, the stages, (e.g. every 15 minutes), so that *power balance* can be maintained to prevent grid failure. This results in tight restrictions on the response time for any decision-making policy. Furthermore, power flows to and from individual generators and the storage system are subject to capacity constraints that restrict their utilization. Provided that power balance is maintained, the EMS goal is to minimize the cost over one day of operation since several key pieces of information (e.g. grid energy prices) are provided with a daily frequency. Typically, historical data in the form of past energy prices, forecasted and actual power generation, and user load demands are available for analysis.

The EMS framework presents an ideal opportunity to explore the integration of knowledge into ML pipelines: some elements are *known explicitly* (e.g., constraints on power flows and their balance, cost of buying/selling energy) and can be formulated in a declarative fashion, whereas others are only in implicit form (e.g., renewable energy generation) but they can be inferred from data via training an ML model. At the same time, the task is challenging due to the uncertainty that affects some of the power sources and user demands. As an additional restriction, the uncertainty is assumed to be non-anticipative and exogenous, meaning that only past load and demands may affect the future values and that decisions have no impact on the uncertain elements.

For the EMS, it is possible to account for the explicit problem elements by relying on declarative optimization. In particular, we can model the cost function and the constraints via the following LP, similarly to what was done by [147]. For stage k we have:

$$\arg\min_{z^{(k)}} \sum_{i=2}^{m} c_i^{(k)} z_i^{(k)} \tag{4.1}$$

$$\text{s.t.} \sum_{i=1}^{m} z_i^{(k)} = x_{load}^{(k)} \tag{4.2}$$

$$l_i \leq z_i^{(k)} \leq u_i \qquad \forall i = 1..m \tag{4.3}$$

$$0 \leq x_{storage}^{(k)} - \eta z_1^{(k)} \leq q \tag{4.4}$$

$$z_i^{(k)} \in \mathbb{R} \qquad \forall i = 1..m \tag{4.5}$$

The decision variables $z_i^{(k)}$ correspond to the power flows from (for a positive sign) or to (for a negative sign) each power generator, the grid, and the storage system. There is a linear cost associated with every power flow, except for the storage system, which is associated with index 1. The costs change over each interval, but they are known at planning time (since they are communicated one day ahead). The net energy produced must match the observed demands, i.e. $x_{load}^{(k)}$. All flows must satisfy lower and upper physical bounds, i.e. l_i and u_i. We have $l_i \geq 0$ for every unit except for the grid, since selling energy is always an option. The energy level in the storage system cannot be negative and cannot exceed its capacity q. The charging rate η depends on the time interval length and the storage system efficiency.

The LP we have presented can provide feasibility guarantees (assuming the physical limits from/to the grid are large enough) and optimize the cost for a single stage. It can also be solved quickly enough that latency constraints are not an issue. However, the model is *totally myopic*: it lacks any mechanism to anticipate future load, energy production, and costs. In particular, the model will never store energy in preparation for price spikes, since, within a single stage, using energy to satisfy demand or selling to the grid is always more efficient than moving it to the storage system.

4.2 Predictive Maintenance

In an industrial facility, the necessity for maintenance operations arises both to prevent potential faults and to address faults that have already occurred. The task of determining the optimal schedule for these maintenance operations is inherently challenging, as both the maintenance operations themselves and the subsequent repairing actions can entail significant costs. Three primary strategies are typically employed in this context:

- *Prescriptive maintenance.* In this approach, maintenance operations are scheduled at a frequency that minimizes the probability of faults occurring. While this strategy effectively reduces the incidence of faults, it comes at a high cost due to the frequency of operations.
- *Reactive maintenance.* This strategy foregoes scheduled maintenance operations entirely, with repairing actions only being initiated in response to a detected fault. While this approach saves time and reduces the cost associated with scheduled maintenance, it may result in a higher number of unexpected faults.
- *Predictive maintenance.* Under this strategy, a minimal maintenance schedule is established based on the estimated condition of the facility's components. The objective is to minimize maintenance operations without incurring in any faults.

The field of predictive maintenance has been under study for numerous years, but the emergence of advanced technologies, such as ML, has significantly enhanced its efficacy. ML algorithms can serve as predictive models for various tasks, including:

- *Early failure detection.* The output is 1 if the model predicts that a failure will occur within a certain amount of time; 0 otherwise.
- *Anomalous behaviour detection.* Even when the model lacks specific knowledge about the nature of an impending failure, it can identify unusual behavior patterns that might indicate an anomaly.
- *Remaining Useful Life (RUL).* This task involves regression, where the model's goal is to predict how much longer a machinery component can be used before it needs replacement.

In the scope of this thesis, we focused on two use cases: a real-world oil and gas facility and an abstract RC circuit system that can be adopted to model thermal components.

4.2.1 Oil and gas facility

Oil and gas extraction is a complex process involving extensive facilities comprising numerous interconnected components. Any faults or malfunctions in this intricate system not only result in economic losses but can also pose environmental and safety risks to nearby communities. Consequently, the prevention of anomalies assumes paramount importance in this industry.

During the course of my Ph.D. research, I was involved in a national project where I applied informed ML algorithms to enhance the operational efficiency of an oil and gas facility. The facility in question was operated by a prominent multinational corporation headquartered in Italy. To respect confidentiality and protect sensitive information, the name of the corporation will remain anonymous throughout this discussion, and it will be simply referred to as "company".

The goal of the aforementioned facility is the extraction of hydrocarbons from underground reservoirs. In this operational context, oil and gas retrieved from the reservoirs are channeled from the wells through a gathering network to a stabilization facility. The role of this facility is to separate the oil, gas, and water phases, bringing them to stable conditions and subsequently directing them toward downstream processing facilities. These downstream processes can include refining for the oil component and integration into the gas distribution network for the gas components, among others.

While the stabilization process itself may be relatively straightforward, its successful execution is challenged by the frequent fluctuations in process conditions. These variations arise from several factors:

1. *Well drilling and expansion.* During the initial production phase of the oil and gas field, new wells are drilled, which leads to changes in the quantity of oil and gas that must be processed.
2. *Evolution over the field's lifecycle.* Over the field's operational lifespan, individual wells tend to produce an increasing amount of water alongside the oil. This shift alters the composition of the hydrocarbon stream and subsequently impacts the process conditions.
3. *Intermittent well interventions.* Periodically, wells are taken out of the production stream for various reasons, such as conducting well production tests or implementing production optimization measures like solvent jobs. These actions further contribute to the dynamic nature of the operating conditions of the plant equipment.

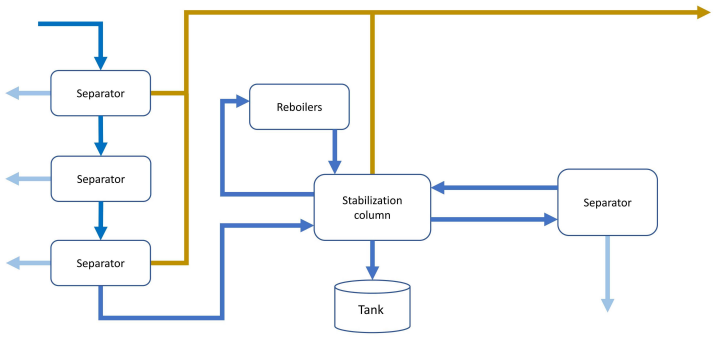


Figure 4.2: Simplified schema of the oil and gas facility. Dark blue, light blue and yellow lines are the flows of respectively the oil, water and gas.

The process for handling the oil extracted from the wells involves a sequence of three vessels, functioning as gravity separators. Due to the distinct densities of oil, water, and gas, specific separation actions occur within each vessel. A portion of the gas rises to the top part of each separator (depicted as yellow streams in fig. 4.2), while the water in the oil descends to the bottom part of the separators (represented as light blue streams). Meanwhile, the oil proceeds from one separator to the next one (indicated by dark blue streams). Importantly, these separators operate at progressively decreasing pressures, enabling the removal of additional gas with each successive stage.

Following the third separator, the oil is directed to an oil stabilization column. The oil enters the upper section of the column, traverses multiple plates, proceeds to a further separator for the removal of residual water, and then passes through additional plates at the column's base. Here, it encounters two reboilers that heat the oil. The heating process at the column's base induces an upward flow of vapor, which interacts with the descending oil on the plates, aiding in the removal of any remaining gas. Finally, the oil is cooled and stored in tanks.

A critical parameter in this process is the residual vapor pressure of the oil stored in the tanks, known as the reid vapor pressure (RVP). It is imperative that the RVP remains lower than the ambient pressure (1 bar) to prevent gas release within the tanks. A value exceeding 1 bar indicates a failure to meet the stabilization requirements, while a very low RVP (e.g., 0.2 – 0.6 bar) suggests waste of energy, resulting in over-stabilization of the oil.

In this context, precise control of the temperature at the column's base is of paramount importance. The temperature must be sufficiently high to achieve an RVP below 1 bar while being low enough to avoid excessive energy consumption and over-stabilization. The temperature at the base of the column is controlled through the use of reboilers, which heat the oil via thermal exchange between hot vapor and oil through a tubing system. Both the quantity and temperature of the vapor can be regulated to maintain the desired temperature at the column's base.

4.3 Resistor Capacitor circuit for Thermal Modeling

Ordinary differential equations (ODEs) are mathematical equations that involve derivatives of a single independent variable. They are a fundamental tool used to model dynamic systems. One practical example of such systems includes electric circuits with resistors and capacitors. In particular, resistor-

capacitor (RC) circuits with constant voltage sources find applications beyond electromechanics, such as thermal modeling [52, 53, 55, 149].

Thermal modeling plays a crucial role in predictive maintenance scenarios where real-time temperature measurements are essential to prevent irreparable damages (e.g. electric power systems). However, developing highly accurate thermal models can be computationally expensive. RC circuit models offer a cheaper yet accurate alternative, requiring only knowledge of the shape and material of the components involved. These considerations motivated us to incorporate experiments of informed ML techniques in the context of RC circuit systems.

In a RC circuit, the state evolution is described by the following ODE:

$$\frac{dV_C(t)}{dt} = \frac{1}{\tau}(V_s - V_C(t)) \tag{4.6}$$

where $V_C(t)$ is the capacitor voltage at time t, V_s is the voltage provided by the generator, and τ is the time constant which defines the circuit response. Despite being relatively simple, this ODE allows to investigate physics-informed ML techniques, as we will see in section 5.2.

4.4 Combinatorial Optimization

Within the scope of this thesis, we focused on two combinatorial optimization problems: the knapsack (KP) and the weighted set multi-cover (WSMC). Although these problems are abstract, they have a rich history in OR and find widespread real-world applications.

Knapsack The KP [150] is a well-known NP-complete ILP problem with numerous practical applications such as portfolio optimization, cargo loading, and cutting stock, among others. The primary objective in the KP is to select a subset of items from a given set that maximizes their total value while adhering to a specified budget constraint. The formulation is:

$$\max \sum_{j \in J} v_j z_j \tag{4.7}$$

$$\sum_{j \in J} w_j z_j \leq c \tag{4.8}$$

$$z \in \{0, 1\} \tag{4.9}$$

where v_j and w_j are respectively the value and weight for the j-th item, c is the budget/capacity constraint, z_j is 1 if the j-th item is selected and 0 otherwise.

Beside the classical formulation, we also considered a stochastic version where either the capacity or the item weights are stochastic and unknown at solution time thus requiring recourse actions. For example, in a production scheduling problem, the items weights might follow a random probability distribution, such as a Poisson distribution.

Formally, this scenario can be framed as a two-stage stochastic optimization problem with recourse actions. It involves solving the first stage problem by incorporating an estimate of the unknown parameters in the model formulation. Subsequently, after uncertainty is revealed, we solve the second stage by employing a recourse action if the first stage solution was not feasible. The recourse action allow to add/remove items although at the price of respectively reduced value and higher cost. This

penalty approach is actually employed in real-world scenarios; for instance, in a cargo loading problem, if the capacity is exceeded, we might incur additional costs for disposing of the exceeded items.

The problem is formulated as follows:

$$\max_{z,u^+,u^-} \quad \sum_{j\in J}\left(c_j z_j + \frac{1}{\rho}v_j u_j^+ - \rho v_j u_j^-\right) \tag{4.10}$$

$$\text{s.t.} \quad \sum_{j\in J} w_j(z_j + u_j^+ - u_j^-) \le C \tag{4.11}$$

$$z \ge u^- \tag{4.12}$$

$$z + u^+ \le 1 \tag{4.13}$$

$$u^+, u^- \in \{0,1\} \tag{4.14}$$

where u^+ and u^- are respectively the selected/removed items during the second stage, w is the realization of the items weights and $\rho > 1$ is the penalty coefficient.

Weighted Set Multi-cover The WSMC is a simplified version of a production scheduling problem. Formally, we assume a factory can manufacture products out of a universe I. Products can be built only in specific combinations, each associated with a different construction cost and represented as sets over I. Its formulation is:

$$\min \sum_{j\in J} c_j z_j \tag{4.15}$$

$$\sum_{j\in J} a_{i,j} z_j \ge y_i \qquad \forall i \in I \tag{4.16}$$

$$z_j \ge 0, z_j \in \mathbb{Z} \qquad \forall j \in J \tag{4.17}$$

$$\tag{4.18}$$

Similarly as for the KP, we considered a two-stage stochastic version of the problem where the demands are uncertain. Unmet demands can still be satisfied by buying additional products but at a higher cost. Our goal is to meet customer demands d (by either manufacturing or buying) while minimizing the total cost. Formally, this is a two-stage stochastic optimization problem with recourse actions. The problem can be modeled via the following MILP:

$$\min \sum_{j\in J} c_j z_j + \sum_{i\in I} \rho s_i \tag{4.19}$$

$$\sum_{j\in J} a_{i,j} z_j \ge y_i(1 - w_i) \qquad \forall i \in I \tag{4.20}$$

$$(w_i = 1) \implies s_i \ge d_i - \sum_{j\in J} a_{i,j} z_j \qquad \forall i \in I \tag{4.21}$$

$$z_j \ge 0, z_j \in \mathbb{Z} \qquad \forall j \in J \tag{4.22}$$

$$s_i \ge 0, w_i \in \{0,1\} \qquad \forall i \in I \tag{4.23}$$

While the vector of decision variables z specifies how many units of each set should be *manufactured*, the additional vector of variables s represents how many units should be *bought*. Equation (4.20) specifies

that all estimated demands should be met unless the "flag variable" w_i for the respective item is raised, i.e. $w_i = 1$. In this case, the indicator constraint in Equation (4.21) forces the s_i variable to be larger than the unmet demand. The objective combines the cost of manufacturing each set (built using the c_j coefficients) with that of buying items (with the ρ coefficient).

Chapter 5

Knowledge injection methods to improve predictive models

In this chapter, we explore knowledge integration approaches developed or investigated during my research activity, specifically tailored to enhance predictive models. The first part focuses on informed ML methods designed to improve the capabilities and accuracy of predictive models. In the second part, we delve into the balance between interpretability and accuracy by focusing on specific informed ML technique that leverages physics knowledge.

The rest of the chapter is organized as follows. In the first section, we describe a general methodology for integrating a blackbox model into a custom ML pipeline. Before delving into the results for the aforementioned methodology, we present a preliminary analysis and an anomaly detection system that we developed for the facility under investigation. In the remaining part of the chapter, we introduce and provide experimental insights into the physics-informed ML algorithm that was the subject of our analysis.

5.1 External model integration

In various practical applications, leveraging a blackbox model, whose internal details are inaccessible yet valuable, is common practice. Consider a large facility with numerous components, often provided from vendors rather than being internally manufactured. These *external models* are a valuable form of *prior knowledge* and might be characteristic curves describing component behavior, simulators or even ML models trained on undisclosed data (due to sensitive information concerns, for instance). Integrating such external models becomes essential when their encapsulated knowledge is otherwise unavailable, or obtaining similar knowledge is prohibitively expensive.

This situation raises crucial questions: *How can we effectively utilize the knowledge from these external models as they are? How can we seamlessly integrate them into a custom facility's operational model?* To address these challenges, there is a need for a methodology that enables an easy integration of external models into the operational framework of a specific facility.

General idea The idea behind the methodology is relatively simple: regardeless its nature, the external model E is described by the equation $y = f(x)$, where x and y are respectively the input and

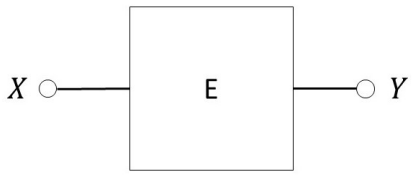


Figure 5.1: First integration schema of the external model.

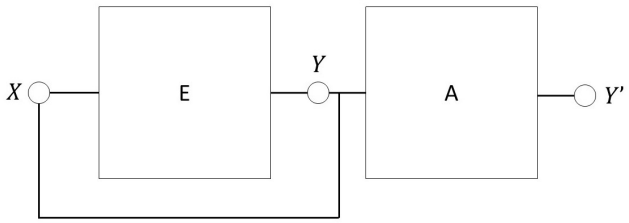


Figure 5.2: Second integration schema of the methodology.

output variables, and f is a function that maps each $x \in X$ to the corresponding $y \in Y$. This approach is very flexible since it allows to integrate either white or black boxes with custom data-driven models.

In the subsequent part of this section, we present a variety of schemas that exemplify the application of this methodology and enable to address the aforementioned challenges. Although the source of knowledge remains constant, each schema may impact a different stage of the ML pipeline, highlighting the flexibility and generality of the methodologies.

Schema n.1 In industrial plants, particularly with the widespread adoption of IoT solutions, data collection is mainly realized through sensor measurements. However, this approach is not always feasible, as certain measurements may require human intervention. This scenario presents several challenges and limitations: data collected can be noisy and scarce, and the collection operation itself might be costly. When a specific measurement is valuable and cannot be ignored, it becomes imperative to devise an economical and efficient method to obtain it.

To address this need, we developed an integration schema that impacts the *final hypothesis* step of the ML pipeline and enables fast and cost-effective estimation of hard-to-measure quantities. The concept, as illustrated in fig. 5.1, is simple yet highly effective: an external model E serves as a proxy to estimate the value of interest. If E is not overly complex, querying it is more economical than conducting laborious hand measurements. Moreover, if the external model accurately captures the component's behavior, its estimates can even surpass the accuracy of field-collected values, which are susceptible to noise.

Schema n.2 Ideally, the external model exhaustively encapsulates all necessary knowledge. For instance, a vendor might train a data-driven model to represent a component across a wide range of operational conditions. When trained to optimize an accuracy metric (such as minimizing the MSE), the model inherently considers the entire spectrum of input working conditions as equally relevant. However, in practical integration within a specific facility, the component often operates within a narrower range of conditions.

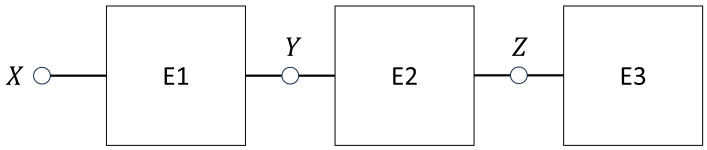


Figure 5.3: Third integration schema.

In such cases, the emphasis might be on achieving higher accuracy within this plant-specific operational range, with less concern about performance in the broader spectrum. To address this, we developed the integration methodology outlined in fig. 5.2. Here, we employ an in-house model A to enhance the outcomes of the external model by generating predictions Y' aligned with the plant-specific working conditions. As this approach modifies the model architecture, it constrains the initial *set of hypotheses* we draw before learning. This integration approach preserves flexibility as it neither necessitates updating E nor requires knowledge of its internal representation.

Schema n.3 In anomaly detection tasks, the objective is not only to detect ongoing anomalies but also to identify the source of the anomalous behavior. Returning to the example of a large facility, it is often essential to understand which components are malfunctioning. We differentiate between global anomalies, which simply indicate unexpected behaviors in the facility, and localized anomalies, which provide a more detailed identification of the involved components. To design such a sophisticated anomaly detection system, we leverage not only the external model but also a valuable source of *human knowledge*: the interconnections between the entities within the system, provided by an expert. As depicted, in fig. 5.3, by leveraging these interconnections, we can design a Bayesian network that reflects the causal dependencies, where each node is a component.

Similar to the previous schema, we depict an architecture (and thus an *hypothesis set*) that reflects these causal relations. Since the external model describe the ideal behaviors, we can use them to estimate the probability $P(X)$ of an observed variable or to compute the conditioned probability $P(Y|X)$ of the outcome Y given X. Low probabilities are associated with anomalous or very unlikely working conditions and by combining these probability values with the Bayes theorem, we can effectively localize faulty components.

In the context of the Bayesian network depicted in Figure 5.3, the joint probability distribution of variables X, Y, and Z can be expressed as $P(X, Y, Z) = P(X) \cdot P(Y|X) \cdot P(Z|Y)$. When this joint probability is low, examining the values of the conditional probabilities helps identify the nature of the anomaly. Common scenarios include:

- When $P(X, Y, Z)$ is low, and both $P(X)$ and $P(Z|Y)$ are high, but $P(Y|X)$ is low, the likely source of anomaly is $E2$. Generally, if all conditional probabilities are high except one, this suggests a localized anomaly in the system.

- If $P(X, Y, Z)$ is low and $P(X)$, $P(Z|Y)$, and $P(Y|X)$ are all low, it indicates a more widespread issue across the network, pointing to a global anomaly.

These patterns assist in pinpointing the source of anomalies within the system associated with the Bayesian network.

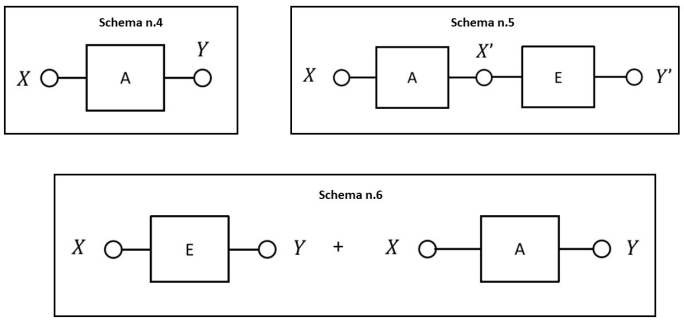


Figure 5.4: Other viable integration schemas derived from the proposed methodology.

Other integration schemas With the sole goal of demonstrating the generality and flexibility of the integration methodologies, we present an additional set of schemas that were not directly implemented in this thesis. A schematic overview is provided in fig. 5.4.

In Schema 4, the external model is not utilized to make predictions but to determine the input X that results in a desired outcome Y. This approach is useful for computing the value of a controllable variable that produces a specific outcome.

Schema 5 is closely related to Schema 2: the objective is to achieve more accurate predictions Y' for the specific system where the component is installed. The in-house model E is trained to rescale the input features X to a new range X' that aligns with the facility-specific operational conditions. This approach is beneficial when the operational conditions lie at the boundaries of the benchmark.

If the in-house model is data-driven, the availability of sufficient data is critical for its development. Unfortunately, in many practical scenarios, data collection is challenging, resulting in scarce facility data. As demonstrated in Schema 6, we outline an integration schema where the external model E is utilized to generate new data synthetically. This enables us to augment the available *training data* with new ones and enhances the efficiency of the training process.

This section addresses integrating blackbox models, which encapsulate valuable external knowledge, into ML pipelines for practical applications. Key methodologies include treating any external model E as a function $y = f(x)$, which allows its seamless integration into various stages of the ML pipeline. Demonstrated through several schemas, this approach enhances operational models in settings like industrial facilities, where external models are used for tasks such as estimating hard-to-measure quantities, enhancing model outcomes for specific operational conditions, and sophisticated anomaly detection using Bayesian networks. These integration techniques underscore the flexibility in utilizing external knowledge across diverse scenarios, from proxy estimations and model adaptation to anomaly localization and synthetic data generation for training enhancement.

5.1.1 Preliminary analysis

The methodology outlined in the previous section was applied to the oil and gas facility introduced in chapter 4. Before applying the methodology, we performed a preliminary analysis of the data and we designed an anomaly detection system capable of alerting the plant operator before the value of the

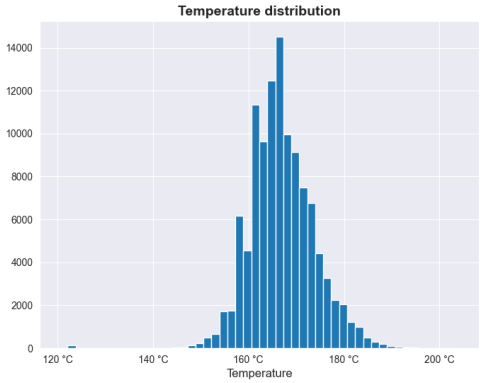


Figure 5.5: The histogram of the stabilization column temperature values.

column bottom temperature leaves an acceptable range. These preliminary data analyses and the design of an anomaly detection system played a crucial role in acquiring valuable insights about the system and served as the foundation for the subsequent development of informed ML techniques for this specific application.

In particular, we implemented a data-driven solution based on a predictive model as its core, plus preprocessing and postprocessing modules. Preprocessing transforms the input data so that it is digestible by the predictive models. During the postprocessing step, raw predictions are aggregated to obtain a more robust alarm signal, and a multi-objective quality criterion allows the operator to balance the sensitivity and specificity of the system. For the predictive core, we consider a variety of solutions including regression and classification approaches, operating with both sequence and aggregated data. Moreover, models are retrained over time to account for concept drift and evolving operating conditions.

In the investigated oil and gas facility, the RVP is an indicator of stabilization quality and thus of eventual anomalies. Unfortunately, real-time measurements of the RVP are not available since it is measured twice a day by human operators. However, according to domain experts, the temperature at the bottom of the column can be used as a proxy, since the two are highly correlated. Low temperatures are associated with high RVP values, whereas high temperatures lead to over-stabilization of the oil. In the considered scenario, a temperature within the $[140, 180]$°C range is considered acceptable. Its value can be controlled through the reboilers, which heat the oil through the thermal exchange of hot vapor and the oil across a tubing system.

As per the previous considerations, the temperature of the bottom of the column provides a natural target for an alarm system. It is therefore important to characterize its behavior in the context of the facility under investigation.

We start by inspecting the distribution of the temperature values: the histogram from fig. 5.5 shows this is not too far from being Normal, with a mean and a standard deviation of respectively 167°C and 6.8°C and a few outlier values at around 120°C (likely related to plan maintenance events). It can also be seen that the lower bound of the safe interval (140°C) is almost never passed, whereas the upper bound (180°C) is exceeded a non-negligible number of times. The temperature rarely reaches values close to 200°C, due to the control systems operating on the plant.

Finally, we attempt a characterization of anomalous events (i.e. out of bounds temperature). In fig. 5.6, we see an example that is representative of the issues that may affect the facility:

- Multiple anomalies may occur on the same day.

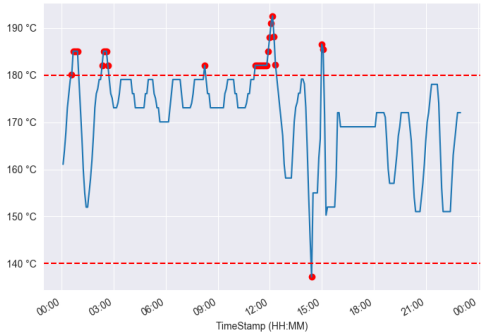


Figure 5.6: A representative example of anomalous events that may occur during a day.

- Anomalies may have different duration.
- Instances of multiple anomaly events are more probable when the temperature hovers around 180°C, indicating an going state of alarm.
- As shown in the timeframe 00:00 AM-03:00 AM, the temperature may overcome the desired bounds in different intervals which are relatively close in time. It is important to decide when they are classified as different anomalies and when they are assigned to the same one.

Overall, this is a practical scenario that is complex to handle, thus making the design of a well-behaving alarm system far from trivial.

The goal of this preliminary task was to developed a data-driven system designed to alert the user when the oil temperature at the bottom of the stabilization column is expected to be outside a specified $[t_{min}, t_{max}]$ interval over the next τ minutes. The idea is that, if we are able to notify the plant operators with a proper advance, they may adopt suitable containment measures.

In principle, an ML model could be used to obtain predictions about the temperature behavior in the considered time frame and then to directly warn the operator. More realistically, the predictions would need to be postprocessed to improve the robustness and sensitivity of the alarm signal. Since mistakes (both false alarms and missed anomalies) are unavoidable, such a postprocessing step should be flexible enough to accommodate priorities determined by the plant operator.

The general architecture of our alarm detection method is depicted in fig. 5.7. Data (either historical or real-time) are collected from remote sensors installed on the plant and stored via an architecture that records the measurements from the field. The collected data undergo a preprocessing step to make them digestible for a data-driven approach. Finally, raw predictions require a postprocessing step to generate an alarm signal that is easily interpretable by and manageable for the human operator.

In the remainder of the section, we will provide additional details about the building blocks of our method, namely data preprocessing, predictive model, and alarm signal generation, and after that the results of the experimental analysis.

Preprocessing During the preprocessing step, raw data from the plant are cleaned and transformed so as to make them digestible by the predictive modules. While certain preprocessing steps are applicable to both regression and classification models, and are independent of the specific predictive model, others are tailored to the nature of the task or the characteristics of the model being employed.

Figure 5.7: Architecture of the designed alarm detection system.

We start by enumerating a list of common practices and then we proceed by describing specific preprocessing steps. As a first step, we perform missing value imputation, so that we are not later constrained to choose predictive models that can handle them. Additionally, certain ML algorithms, such as neural networks, necessitate the standardization of input features to operate optimally within canonical ranges. To achieve this standardization, we apply a uniform scaling transformation to all input variables. As a result, all the input features are centered on zero and have unit variance.

We then proceed by describing a task-specific preprocessing step. We previously highlighted that plant dynamics are relatively slow, and that oil takes a non-negligible time to move between components. For this reason, feeding synchronous input measurements runs the risk of missing any delayed effect due to this mechanism. Luckily, a preprocessing step can be used to re-align the input data so as to keep into account the delayed responses of the stabilization column temperature. Specifically, if we refer to the m-dimensional input as x, where each input feature x_j has a delayed response with lag $\Delta t'_j$ and we want to predict the stabilization column temperature with an advance τ then the result of the alignment operation will be:

$$x_j^{(t)} := x_j^{(t-\max(\tau,\Delta t'_j))} \quad \forall j \in \{1,\dots,m\} \tag{5.1}$$

This procedure temporally aligns each input feature based on the delayed response without exceeding the necessary prediction advance.

The alignment operation holds particular significance in the context of regression models. In contrast, in a classification scenario, our predictions typically pertain to a relatively extended forward interval ($[t, t+\tau]$), which inherently accommodates the consideration of delayed responses. The delayed values are identified via a time-lagged correlation analysis between the input features and the stabilization column temperature; for simplicity, we use the (lagged) Pearson correlation coefficient:

$$\rho_{XY_{\Delta t}} = \frac{\sigma_{XY_{\Delta t}}}{\sigma_X \sigma_{Y_{\Delta t}}} \tag{5.2}$$

where Δt is the time lag. As we are working with time-series data, a crucial aspect of the design involves making thoughtful choices about how to manage the input for the predictive model. Two natural options are:

- Feeding the predictive model with a sequence of input data, corresponding to a reasonably long time window.

- Using aggregation function (e.g. common moment statistics such as mean and standard deviation) to extract meaningful features in the preprocessing step.

Both these solutions can be effective but they require a predictive model that handles the specific data format.

Predictive models From an ML perspective, the core predictive task within our system can be categorized into two distinct approaches:

- *Regression.* This involves predicting future temperature values τ time units ahead. This task is more challenging, requiring the estimation of specific temperature values well in advance to enable timely actions by the operator.

- *Classification.* This entails determining if the temperature will be within (class 0) or outside (class 1) a predefined safe range in the next τ time units. While less granular than regression, it aligns more closely with our primary task of ensuring temperatures remain within safe boundaries.

Given the operational context and initial experiments, our focus is primarily on developing classification models. These models are designed to indicate whether temperatures are likely to exceed safe limits, rather than providing detailed temperature predictions. We adopted neural architectures for their robust performance, but our methodology is not confined to this type of algorithm.

For the regression task, we formulate the problem as learning a function $f : X^{(t)} \to Y^{(t+\tau)}$, where $X^{(t)} \in \mathbb{R}^m$ represents the input features at time t, and $Y^{(t+\tau)}$ is the target temperature value at time $t+\tau$. The training of neural networks for this task is supervised, requiring a dataset $\mathcal{D} = \{(x^{(i)}, y^{(i)})\}_{i=0}^{N}$, with each $x^{(i)}$ being the input features and each $y^{(i)}$ being the corresponding target temperature value.

In the classification scenario, the task is binary, aiming to predict whether temperature values will remain within a safe range. The dataset for this task is labeled as anomalous or non-anomalous based on whether the temperature exceeds safe limits at any point from $[t, t+\tau]$. Here, the function to learn is $f : X \to Y$, where $X \in \mathbb{R}^m$ consists of input features, and Y is a binary class indicating anomaly. The dataset for this task is similarly structured to the regression scenario, but with binary targets indicating anomaly status.

The choice of input format is crucial and depends on the model's design. If the sequence input is selected, the data will be in the form of $m \times w$ matrices, where w is the window length, and m is the number of input features. This format is particularly suited for models like 1-dimensional CNNs, which can exploit the sequential nature of the data. Alternatively, for models designed for aggregated data, the input is a vector $X \in \mathbb{R}^q$, where $q = m \cdot u$ and u is the number of computed statistics (e.g., mean, standard deviation) for each feature. This aggregated format can be more suitable for models that do not inherently process sequential data.

Alarm signal generation The primary objective of our system is to generate an alarm signal that is both easily interpretable by field operators and aligns with business needs. Defining an effective alarm requires considering several factors:

- Both missed detections and false alarms are detrimental from a business perspective. Operators should be alerted only when necessary, as investigating a potential danger is costly. Additionally, frequent false alarms can lead to distrust in the system.

- Technicians do not continuously monitor the alarm system, so multiple closely-timed predictions can be aggregated into a single alarm signal.

- The lead time provided by anomaly detection should be sufficient for effective intervention and repairs.

To meet these requirements, we developed a method to transform raw predictions into interpretable alarms. While our approach primarily utilizes binary classifier predictions, it can be adapted to regression scenarios, such as by applying a threshold to the model's predictions. This method incorporates several parameters and operations:

- **Validation Undershoot:** Groups individual predictions to form a coarser-grained *alarm* signal. This parameter determines the number of predictions considered when validating an alarm.
- **Threshold:** The required proportion of anomalous predictions within a group (determined by the *validation undershoot*) to trigger an *alarm*.
- **Anomaly Undershoot:** Defines the interval within which two anomalous temperature readings are considered part of the same anomaly. Alternatively, it can specify the required separation between anomalies for them to be classified as distinct.
- **Alarm Undershoot:** Recognizes that operators are not constantly monitoring the system. Anomalies separated by sufficient time are treated as different alarms.
- **Minimum Advance:** Ensures that alarms provide enough lead time for operators to effectively respond to potential issues.
- **Blurred Area:** While strict safe boundaries are used for classifying data, evaluation can be more flexible. Ground truth values within ΔT°C of the boundaries are excluded during evaluation to avoid overemphasizing minor anomalies and affecting the evaluation score.

Experimental analysis Training and evaluation were conducted using real data acquired from sensor measurements at the facility. We used historical data from the period 2018/10/01 to 2019/10/31, sampled every 5 minutes. The dataset was segmented into four periods, each spanning 10 months, with a one-month overlap created by shifting the start date for each segment. This segmentation was designed to evaluate model robustness under changing operational conditions over time, anticipating potential distributional shifts in the data. Periodic retraining of the model might be necessary, and based on input from domain experts, it is believed that the plant's operational conditions evolve gradually. Therefore, a monthly retraining schedule is considered sufficient to address these changes.

For each of the four segments, the initial 8 months of data were used for training, while the subsequent 2 months were evenly divided between validation and testing. In the data aggregation phase of preprocessing, and in agreement with domain expert insights, we calculated the mean, standard deviation, and the average difference between consecutive values over a 20-minute window, reflecting the plant's dynamics.

The evaluation is based on the alarm signal methodology described earlier. For this, the validation, anomaly, and alarm undershoot parameters are set at 5, 10, and 5 minutes, respectively. We implemented a threshold of 1 and required a minimum advance of 5 minutes. Temperature values in the range $[175, 180[$°C are excluded from the evaluation. Throughout this section, we will refer to true alarms, false alarms, and missed anomalies as True Positives (TP), False Positives (FP), and False Negatives (FN), respectively.

Considering the business perspective, both FP and FN are significant, and our aim is to minimize them jointly. Defining an a priori balance between FP and FN can be challenging; hence, we adopted a *Pareto frontier analysis* approach during model selection. This method allows us to present a set of

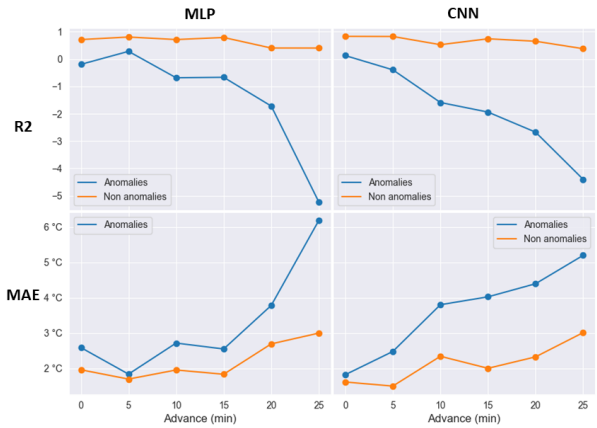


Figure 5.8: Average R2 score and the Mean Absolute Error for the MLP and CNN architectures on the 4 datasets.

solutions, enabling the user to choose based on specific business requirements. The threshold for binary classification notably influences the Pareto analysis, as increasing it reduces FPs but raises FNs. Finding an optimal threshold balance is crucial.

Our training and evaluation procedure includes the following steps:

1. Select a ML algorithm, identify hyperparameters for tuning, and determine their candidate values.
2. Randomly split the dataset into training, validation, and test sets.
3. Train model instances for each candidate hyperparameter set on the training set, and compute FP and FN on the validation set. Discard non-Pareto optimal candidates.
4. For each remaining hyperparameter set, train a new model instance on a dataset formed by appending the validation set to the training set.
5. Perform final evaluation on the test set, assessing both the quality and robustness of Pareto optimal solutions identified during validation. Model robustness is confirmed if test set results align closely with those from the validation set.

The hyperparameter search resulted in relatively simple architectures. The MLP comprises three hidden layers with 24, 12, and 6 units each, ReLU activation function, and L2-regularization. The CNN consists of a single convolutional layer with 8 filters, a kernel size of 3, and ReLU activation, followed by two fully-connected layers with 12 and 6 units, ReLU activation, and L2-regularization. In binary classification tasks, both architectures utilize a sigmoid function at the output layer. Neural networks were trained for up to 100 epochs, with a batch size of 512. Training stops if no improvement in the validation set loss is observed after 3 epochs. Network parameters are optimized using the Adam optimizer with a learning rate of 0.001.

We first present preliminary results for a regression model, which guide our decision to focus on the classification approach. The models are evaluated for prediction advances up to 25 minutes. Due to the dataset's imbalance, we separately compute the mean absolute error (MAE) and R2-score for anomalous and non-anomalous examples. The results, as shown in fig. 5.8, indicate that the MLP

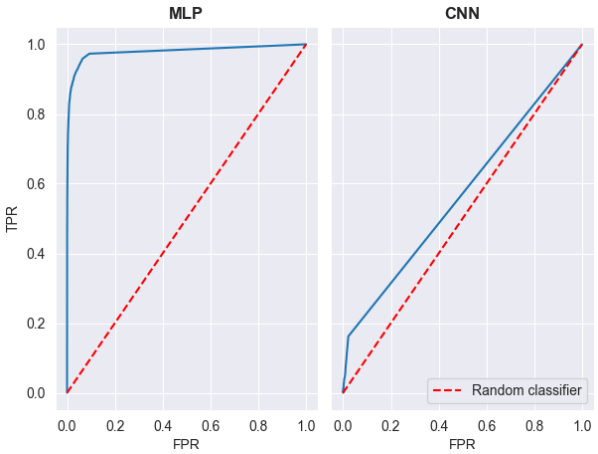


Figure 5.9: ROC curve for the binary classification problem.

and CNN accuracy deteriorates rapidly for anomaly ground truth examples as the prediction advance increases.

Given the regression model's unreliability in estimating anomalous temperature values, we shifted our efforts to the classification formulation. In binary classification, the receiver operating characteristic (ROC) curve is a classic metric for evaluating models, plotting the false positive rate (FPR) against the true positive rate (TPR). Figure 5.9 demonstrates that the MLP nearly achieves ideal performance, whereas the CNN slightly surpasses a random classifier. However, relying solely on the ROC curve for evaluation might lead to disregarding the CNN, despite it being inferior to the MLP in this metric. Yet, this single metric does not suffice for our specific use case, where the objective is to balance minimizing both FPs and FNs while meeting business requirements.

To cater to this need, we introduce a business-oriented validation framework utilizing Pareto frontier analysis. The binary classification threshold significantly influences FP and FN rates, hence our analysis concentrates on this hyperparameter. However, this evaluation can be extended to include all hyperparameters. We present results for the neural architectures with threshold values ranging from 0.1 to 0.9, incremented by 0.1. A singular evaluation metric is inadequate for this task; therefore, we assess not only FP and FN values but also the robustness of the models. Robust models should yield consistent results across both validation and test sets if the Pareto optimal solutions are valid.

For a fair comparison between the validation and test sets, we normalize FN and FP values. Specifically, FN is normalized by the total number of anomalies, and FP by the total number of days, based on the respective set. This normalization results in two distinct metrics: the "FN ratio" and "FP per day."

Results for the MLP and CNN across all datasets are depicted in fig. 5.10. The Pareto optimal solutions computed on the validation set are marked with red points, while the corresponding test set values are indicated in gold. Solutions on both sets are connected by red lines, with shorter lines denoting greater robustness in results. Contrary to the ROC curve analysis, the results here are more consistent, underscoring the limitations of traditional metrics for our specific problem. Notably, for both models, by tolerating as few as 4 FP every 10 days, we can achieve almost 0 FN.

However, it is important to highlight the MLP's marginally superior performance over the CNN. In most cases, the MLP's solutions on the test set closely mirror those on the validation set, indicating robust

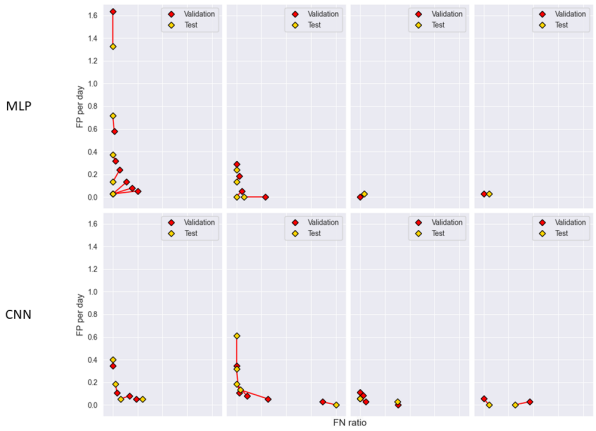


Figure 5.10: Pareto frontier results for the neural architectures.

performance. On the other hand, the CNN demonstrates some instances of performance deterioration from validation to test set, suggesting less consistency in results.

5.1.2 Experimental results on the integration methodology

To evaluate the effectiveness of our methodology, we implemented the integration schemas using real-world data obtained from the oil and gas facility described earlier. Due to the unavailability of a vendor-provided model, we emulated its behavior by training a ML predictor on simulation data. The simulation process allowed us to gather data across a wide spectrum of operational conditions, including scenarios not encountered in the actual facility. This simulation-based approach serves as a surrogate benchmarking process comparable to what a vendor might conduct.

In our experimental analysis, we considered two instances of the external model, each associated with distinct tasks:

- *Stabilization Process.* This includes separators, the stabilization column, and the reboiler. For this instance, we required a predictive model capable of estimating the RVP, a key measure of oil stabilization effectiveness.

- *Reboiler.* Focusing specifically on the reboiler, we needed a separate model to predict the outlet temperature since its role is crucial in heating the oil properly.

In the remainder of this section, we present experimental evidence of the successful application of the integration methodology to: 1) the estimation of the RVP, 2) in-house adaptation of the external model, and 3) the detection of localized anomalies.

Estimate of the RVP As an application of the first integration approach outlined in section 5.1, we successfully achieved the objective of estimating the RVP. The goal is to gather cost-effective RVP measurements, a task often assigned to human operators and known to be time-consuming. For the surrogate external model, we trained a feedforward fully-connected neural network on 75% of the simulation data (with 20% reserved for validation). Following hyperparameter tuning, the best performance was achieved with a relatively simple architecture, consisting of a single hidden layer with 8

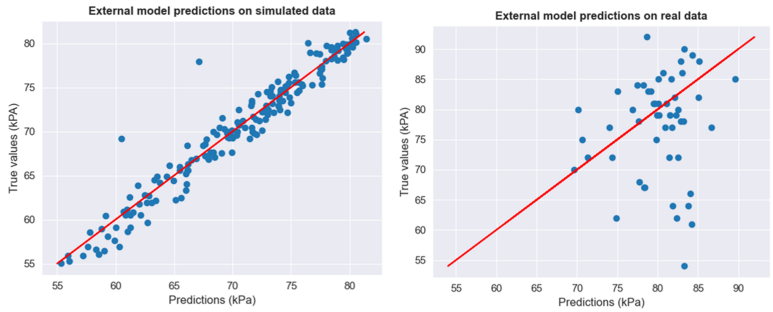


Figure 5.11: The external model is used to collect surrogate measurements of the RVP.

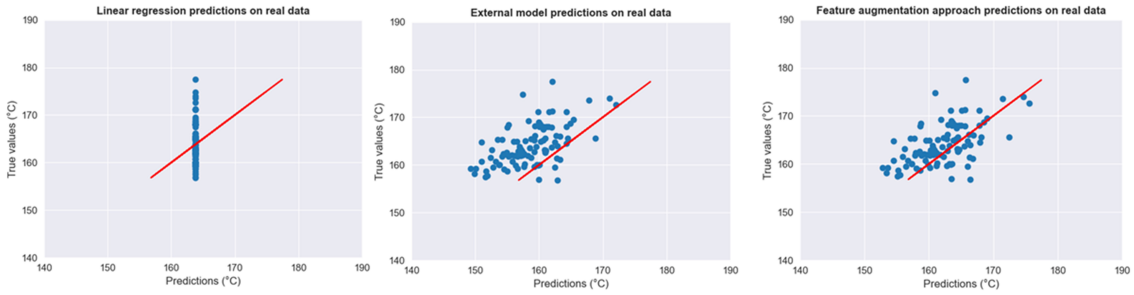


Figure 5.12: Prediction errors on the real data. From left to right: linear regression model, external model and the external model plus the adapter.

units and a ReLU activation function. The prediction errors of this model are depicted in the left-hand side of fig. 5.11.

As illustrated in the right-hand side of fig. 5.11, while the model captures the overall trend of the RVP, it exhibits a non-negligible estimation error. Nevertheless, our primary concern is not the precise value of the RVP but rather whether it falls within the interval of 60-100 kPa, indicating effective oil stabilization. In this context, our integration schema remains valuable, enabling us to *leverage the external model for a rough yet cost-effective estimate.*

In-house adaptation of the external model The aim of the second integration schema described in section 5.1 is to train an in-house model capable of adapting the external model to the facility-specific operating conditions. To empirically demonstrate the effectiveness of this approach, we focused on the reboiler as it was provided by a vendor. The external model employed here is a single-layer neural network with 8 units and a ReLU activation function, trained on simulation data, as detailed in the previous paragraph.

As depicted in the middle plot of fig. 5.12, the external model accurately captures the trend but consistently underestimates the outlet temperature values, implying a translation of its predictions relative to the true values. To address this, we utilized an offset as the adapter, defined as $Y' = Y + b$, where b represents the offset and is trained on the available data. We also considered a scenario with extremely limited data, using only 10 contiguous data points as the training set. To evaluate the method's effectiveness, we compared it with a linear regression model trained exclusively on the 10 data points. Both models were then assessed on the remaining data.

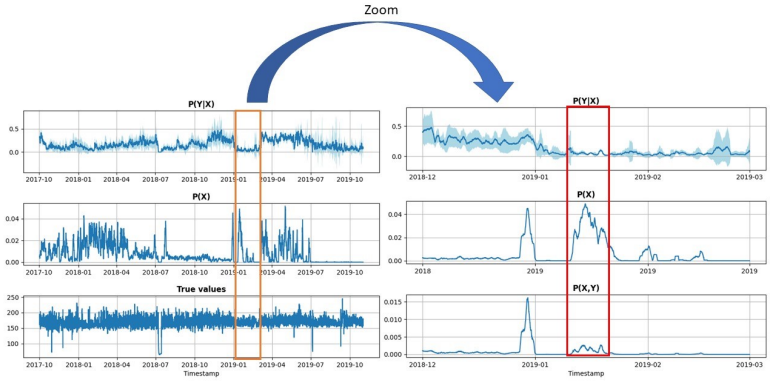


Figure 5.13: Example of an anomaly affecting the only reboiler.

Results are reported in fig. 5.12, highlighting that the dataset is not sufficiently large and variable for learning a robust linear regression model. Conversely, *the adapter demonstrates its effectiveness by appropriately translating the external model predictions* to align with the correct values. While in this case the adapter function is simple, the experimental results provide a proof of concept for more complex situations where the adapter function could, in principle, be nonlinear.

Detecting Localized Anomalies In the last integration methodology, we propose to identify localized anomalies by relying on the Bayesian networks. Referring with X and Y respectively to the input and output of the reboiler, we thus need probabilistic models that are capable of predicting: 1) the probabilities $P(X)$ of observing a specific value of X, 2) the joint probability $P(X, Y)$ of observing a specific value of X and Y and 3) the conditioned probabilities $P(Y|X)$ of observing a specific value of Y given the value of X.

For $P(X)$ and $P(X, Y)$, we employ the kernel density estimation (KDE) algorithm. For $P(Y|X)$, we assume that data are distributed according to a Normal distribution $\mathcal{N}(\mu, \sigma)$ where the mean μ and the standard deviation σ are outputs of a neural network. We trained both the KDE and the neural network (with the same architecture as for the previous methods) on the simulation data and treated them as vendor-provided models. We then used these models to compute the probabilities values $P(X)$, $P(X, Y)$ and $P(Y|X)$. Given these values, there are three possible scenarios:

1. If $P(X, Y)$, $P(X)$ and $P(Y|X)$ are low then an anomaly is occurring and it involves the whole facility.

2. If only $P(X, Y)$ and $P(X)$ are low, then the anomaly pertains to the upstream part of the plant.

3. If only $P(X, Y)$ and $P(Y|X)$ are low then an anomaly is occurring in the only reboiler.

In fig. 5.13, we show an example of a possible non-ideal behavior affecting the reboiler. The joint probability $P(X, Y)$ is low, indicating something anomalous is affecting the plant. At the same time, $P(X)$ is relatively high, whereas $P(Y|X)$ is low. This aligns with the third scenario previously described: we can thus hypothesize that the reboiler does not have the ideal behavior. Due to the lack of ground-truth data, we cannot conclude whether our hypothesis is correct. However, it demonstrates how this *methodology allows fine-grained anomaly detection.*

Discussion Large industrial facilities typically comprise extensive equipment with numerous interconnected components, presenting significant challenges in developing predictive maintenance solutions based on ML algorithms. However, a wealth of knowledge is often available in the form of expert insights, sourced from human operators in the field or the expertise of component vendors.

In this section, we outline a mathematical framework that enables the incorporation of a general blackbox component into a ML pipeline. The versatility of the framework is demonstrated by allowing various integration schemas with different purposes, impacting different steps of the pipeline. Moreover, experimental analysis reveals that the method enhances predictive performances and introduces capabilities not easily achieved via standard approaches.

The methodology has been successfully demonstrated on simple static equipment, but future works could address complex pieces of equipment, such as large turbine-driven multiple-stage compressors. As part of future research directions, it would be interesting to extend the analysis to a larger number of components or, owing to its generality, to investigate the methodology in different industrial facilities.

5.2 Universal Differential Equation for data-driven discovery of ODEs

Universal Differential Equations (UDEs) were first proposed in [151]. The formulation relies on embedded universal approximators to model forced stochastic delay PDEs in the form:

$$\mathcal{N}\left[u(t), u(\alpha(t)), \mathrm{W}(t), U_\theta(u, \beta(t))\right] = 0 \tag{5.3}$$

where $u(t)$ is the system state at time t, $\alpha(t)$ is a delay function, and $\mathrm{W}(t)$ is the Wiener process. $\mathcal{N}[\cdot]$ is a nonlinear operator and $U_\theta(\cdot)$ is a universal approximator parameterized by θ. The UDE framework incorporates *physical knowledge* in the *hypothesis set*, and it is general enough to express other frameworks that combine physics and ML models. For example, by considering a one-dimensional UDE defined by a neural network, namely $u' = U_\theta(u(t), t)$, we retrieve the Neural Ordinary Differential Equation framework [106, 152, 153].

UDEs are trained by minimizing a cost function C_θ defined on the current solution $u_\theta(t)$ with respect to the parameters θ. The cost function is usually computed on discrete data points (t_i, y_i) which represent a set of measurements of the system state, and the optimization can be achieved via gradient-based methods like ADAM or stochastic gradient descent (SGD).

In the scope of this thesis, we propose to employ *UDEs for data-driven discovery of ODEs and we investigate their interpretability.* More specifically, we restrict our analysis to dynamical systems described by ODEs with no stochasticity or time delay. The corresponding UDE formulation is:

$$u' = f(u(t), t, U_\theta(u(t), t)) \tag{5.4}$$

where $f(\cdot)$ is the known dynamics of the system, and $U_\theta(\cdot, \cdot)$ is the universal approximator for the unknown parameters. As cost function, we adopt the MSE between the current approximate solution $u_\theta(t)$ and the true measurement $y(t)$, formally:

$$C_\theta = \sum_i \|u_\theta(t_i) - y(t_i)\|_2^2 . \tag{5.5}$$

We consider discrete time models, where the differential equation in (5.4) can be solved via numerical techniques. Among the available solvers, we rely on the Euler method, which is fully differentiable and allows for gradient-based optimization. Moreover, the limited accuracy of this first-order method enlightens the effects of the integration technique on the unknown parameter approximation.

Our analysis starts from a simplified setting, in which we assume that the unknown parameters are fixed. Therefore, the universal approximator in Equation (5.4) reduces to a set of learnable variables, leading to:

$$u' = f(u(t), t, \theta) \tag{5.6}$$

We consider two approaches to learn Equation (5.6). Given a set of state measurements y in the discrete interval $[t_0, t_n]$, the first approach, mentioned by [43] and named here FULL-BATCH, involves 1) applying the Euler method on the whole temporal series with $y(t_0)$ as the initial condition, 2) computing the cost function C_θ, and 3) optimizing the parameters θ via full-batch gradient-based methods. An alternative approach, named MINI-BATCH, consists of splitting the dataset into pairs of consecutive measurements $(y(t_i), y(t_{i+1}))$, and considering each pair as a single initial value problem. Then, by applying the Euler method on the single pair, we can perform a mini-batch training procedure, which helps in mitigating the gradient vanishing problem [154]. Conversely to the FULL-BATCH approach, which requires data to be ordered and uniform in observations, the MINI-BATCH method has less strict requirements and can be applied also to partially ordered datasets.

5.2.1 Experimental analysis

The UDE framework is extensively employed in applied sciences for estimating the evolution of dynamical systems and for data-driven discovery of differential equations. Despite numerous proposed variants and investigations into various applications, a detailed analysis of the framework's capabilities and limitations is absent in the literature.

Using the RC circuit system as a use case, we performed a preliminary analysis, focused on ODE, by addressing 4 questions of scientific interests:

1. *How does the training procedure affect the accuracy of predictions and the efficiency of the training itself?* To address this question, we conducted a comparison between the FULL-BATCH and MINI-BATCH. The MINI-BATCH offers distinct advantages over the FULL-BATCH. Firstly, it can be applied to partially ordered time series. Secondly, it does not need to consider the whole input data as a single initial value problem, thereby preventing the development of very deep neural architectures, which are challenging to train.

2. *How does the solver accuracy impact the approximation of unknown parameters?* In the UDE framework, the model is trained to predict the system's evolution accurately by learning an approximation of the unknown parameters, minimizing the cost function C_θ. The formulation relies on the integration method to approximate the system state $u(t)$. However, the numerical solver may introduce approximation errors affecting the entire learning procedure. Since the Euler method is a first-order method, its error depends on the number of iterations per time step used to estimate the value of the integral. Thus, our analysis allows for direct control over the trade-off between execution time and solver accuracy.

3. *How accurately can UDE approximate an unknown functional dependence?* Leveraging the universal approximator in Equation (5.4), the UDE framework can learn not only fixed values for unknown

Table 5.1: Comparison between MINI-BATCH and FULL-BATCH methods.

	V_s	τ	$V_c(t)$	Time
MINI-BATCH	0.027 ± 0.013	0.163 ± 0.101	0.021 ± 0.010	9.21 ± 39.49
FULL-BATCH	0.018 ± 0.021	0.200 ± 0.081	0.014 ± 0.020	26.19 ± 5.69

parameters but also functional relationships between them and observable variables. Consequently, we express system parameters as functions of observable variables and investigate the capabilities of UDE in function reconstruction.

4. *Can we leverage the known dynamics of the system under analysis to design the data collection process and enhance approximation accuracy?* Since UDE is as a data-driven approach, it is crucial to explore its effectiveness under diverse data samplings. Simultaneously, we can utilize the knowledge of the differential equation to assess whether the collected samples are adequate for accurately learning the system parameters.

Evaluation and experimental setup. We evaluate the model accuracy by relying on two metrics: the *absolute error* (AE), to evaluate the estimation of the parameters, and the *root mean squared error* (RMSE), to study the approximation of the dynamic system state. For each experiment, we perform 100 trials, normalize the results, and report mean and standard deviation.

Training Procedure We conduct a comparative analysis between the FULL-BATCH and MINI-BATCH methods to determine their accuracy and efficiency. High-precision simulation is employed to generate RC circuit data, with an initialization of $V_c(0) = 0$. We sample 100 values for V_s and τ within the ranges $[5, 10]$ and $[2, 6]$, respectively. Data is generated using the analytical solution of Equation 4.6. From each resulting curve, we sample 10 data points $(V_c(t), t)$, equally spaced in the temporal interval $[0, 5\tau]$.

We assess the accuracy of UDE in approximating unknown parameters and the system state, tracking the total computation time required for convergence. Notably, the MINI-BATCH has an advantage over the FULL-BATCH: the latter predicts the entire state evolution given only the initial state u_0, while the former reconstructs the state evolution with intermediate values. To ensure a fair comparison, predictions of the MINI-BATCH are fed back to the model to forecast the entire temporal series with only u_0.

As illustrated in Table 5.1, both FULL-BATCH and MINI-BATCH accurately approximate V_s and $V_c(t)$ whereas the approximation of τ exhibits a non-negligible error. Notably, FULL-BATCH requires almost three times the computational time to converge. Given the similar estimation accuracy of both methods, we conclude that *MINI-BATCH is a more efficient method for training UDE compared to FULL-BATCH*. Consequently, we employ the MINI-BATCH in the remaining experiments.

Solver Accuracy In the context of ODE discovery, our focus is on approximating unknown system parameters. Despite an overall accurate estimation of the system state, the previous analysis results indicate that the UDE framework does not achieve high accuracy in approximating system parameters. This model inaccuracy may stem from the approximation error introduced by the integration method. To investigate the *impact of solver accuracy on the approximation of unknown parameters*, we conduct tests with different levels of solver accuracy by increasing the number of iterations between time steps in

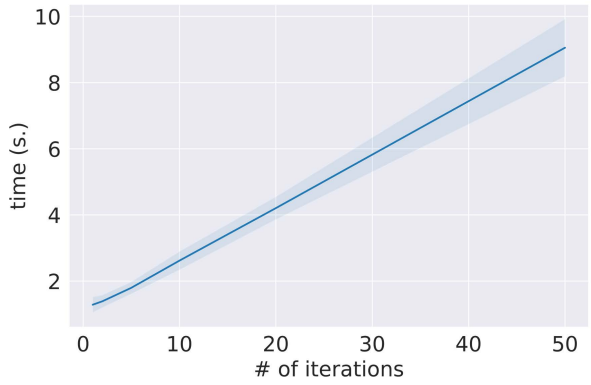


Figure 5.14: UDE training time as a function of the number of iterations per time step of the Euler method.

the integration process. A higher number of iterations per time step of the Euler method is expected to yield more accurate solutions of the ODE; however, this comes at the cost of increased computational time, as shown in Figure 5.14.

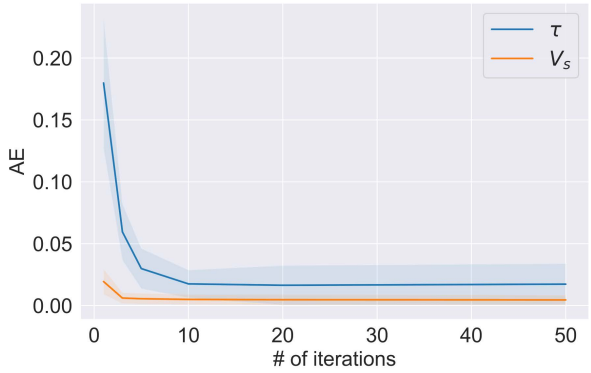


Figure 5.15: Average and standard deviation of the AE as a function of the number iterations per time step of the Euler method.

For this experiment, we use the same data generated for the *Training procedure* experiment. In Figure 5.15, we present the approximation error of the UDE framework when applying the Euler method with an increasing number of steps. As anticipated, in both use cases, increasing the precision of the Euler method leads to more accurate estimation of ODE parameters until reaching a plateau after 10 iterations per time step.

Functional Dependence and Data Sampling In real-world scenarios, the dynamical systems often depend on a set of external variables, or *observables*, influencing the system's behavior. These elements can include environmental conditions or control variables that affect the evolution of the system state. For example, since the RC circuit is often used to model thermal heating of mechanical components, such as rotating equipment, the temperature value might depend on some observables such as the external forces or the ambient temperature. As a preliminary analysis, we thus investigate the UDE framework

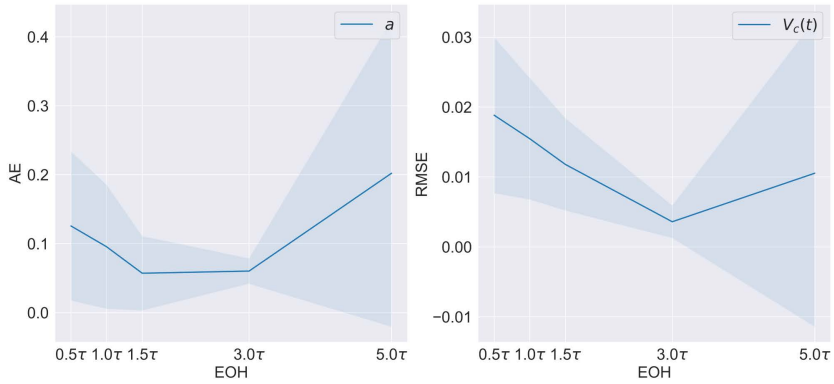


Figure 5.16: Linear coefficients and predictions error as a function of the EOH.

in the presence of observables, assuming a linear dependency between the independent and dependent variables.

In particular, we consider a controlled setup where τ is a linear function of a continuous input variable x changing over time, defined as $\tau(x) = ax$, where a and x are scalar values. Unlike previous experiments, we assume V_s to be known and equal to 1 to focus our analysis on the accuracy of approximating the linear relationship. Since the value of τ changes over time, we cannot rely on the analytic solution of Equation (4.6) to generate data. Therefore, we generate samples from one time step to the next by using a high-resolution integration method, namely the Euler method with $10,000$ iterations per time step. In the generation process, the linear coefficient a is randomly sampled from a uniform probability distribution in the interval $[2, 6]$, and the observable x is initialized to 1 and updated at each time step according to the equation:

$$x(t) = x(t-1) + \epsilon, \quad \text{with} \quad \epsilon \sim \mathcal{U}_{[0,1]}.$$

This procedure allows reasonable variations of τ to prevent physically implausible data. During the learning process, considering the results from the *Solver Accuracy* experiment, we use 10 iterations per time step in the Euler method as a trade-off between numerical error and computational efficiency.

In this set of experiments, our focus is on *evaluating the UDE accuracy in approximating the unknown linear dependence.* The resulting absolute error in the approximation of the linear coefficient a is 0.24 ± 0.27, indicating that the model is not correctly approximating the functional dependence.

Given that UDE is a data-driven approach, we hypothesize that the estimation errors may arise from data quality. Since we simulate the RC circuit using a highly accurate integration method resolution, we can assume that data points are not affected by noise. However, the sampling procedure may have a relevant impact on the learning process. The time constant τ determines how quickly $V_c(t)$ reaches the generator voltage V_s, and its impact is less evident in the later stages of the charging curve. Thus, sampling data at different time intervals may affect the functional dependence approximation.

To investigate *how data sampling affects the linear coefficient estimation*, we generate 10 data points in different temporal regions of the charging curve, considering intervals of the form $[0, EOH]$, where $EOH \in (0, 5\tau]$ refers to the end-of-horizon of the measurements. Since τ changes over time, we consider the maximum as a reference value to compute the EOH. As shown in Figure 5.16, the linear model approximation is more accurate when data points are sampled in an interval with $EOH \in [1.5\tau, 3\tau]$, where $V_c(t)$ reaches approximately 77% and 95% of V_s, respectively. With higher values of EOH, the

sampled data points are closer to the regime value V_s, and the impact of τ is less relevant in the system state evolution. Thus, the learning model can achieve high prediction accuracy of $V_c(t)$ without correctly learning the functional dependence.

Discussion In this section, we provide an in-depth analysis of the UDEs framework for solving the data-driven discovery of ODEs. Our experimental findings demonstrate that the MINI-BATCH gradient descent is faster than the FULL-BATCH version without compromising final performance. We highlight challenges that arise when combining data-driven approaches and numerical integration methods, such as discrepancies in accuracy between state evolution prediction and system parameter approximations. We investigate integration method precision as a potential source of error and discuss the trade-off between approximation accuracy and computational time. Additionally, we explore the significance of the data collection process in achieving higher parameter approximation accuracy. In summary, our analysis reveals that interpretability is not straightforward when using UDE. The blackbox model may have a negative impact, and therefore, the training data collection and learning approach should be carefully designed.

Several future works are viable. To name a few: i) testing different numerical integration solvers (e.g., higher-order Runge-Kutta), ii) considering the unknown parameters to be stochastic, rather than deterministic, iii) extending the analysis to PDEs or stiff equations.

Chapter 6

Knowledge Injection Methods to Enhance Decision Support Systems

Decision support systems aim to facilitate or improve strategic decisions made by human operators. Consider the example of a logistics company tasked with scheduling a fleet of vehicles for delivering a large quantity of products. Finding a cost-effective route is a complex task and is challenging for a human alone due to several reasons: 1) the number of potential solutions is prohibitively large, 2) travel times might be unknown and need estimation, 3) new customers might emerge. An ML model can assist the logistics company by addressing some of these challenges: a predictive model can estimate travel times and handle uncertainties arising from new customers. However, ML struggles with combinatorial decision spaces, while combinatorial optimization algorithms provides more effective techniques to address this challenge.

If we can find a way to effectively integrate learning methods with combinatorial optimization, we would be able to tackle all the aforementioned challenges. In this chapter, we describe a methodology to exploit the declarative formulation of an optimization problem into ML workflows. We start by describing a first integration attempt where only a part of the solution process (i.e. constraints propagators) is distilled in the neural network weights, showing advantages and drawbacks. We then present a new unification framework named, UNIFY, for methods that combine learning and combinatorial optimization. We demonstrate that our method is a generalization of approaches widely used for decision support systems, such as DFL, constrained RL, hybrid offline/online optimization, and stochastic optimization. In the last part of the chapter, we introduce a specific instance of UNIFY developed during my research activity, which widens the applicability of DFL.

6.1 Injecting Constraints Propagators in Neural Networks

Given enough data, DNNs are capable of learning complex input-output relations with high accuracy. Recent work has shown how this applies also to the solution process of CSPs, at least to some degree: examples include the approach from [155], relying on a pool of solutions, or RL approaches inspired by [102], relying on solution checkers/evaluators. This class of approaches, while still not close to the state of the art in combinatorial decision making, may have advantages in terms of robustness and when implicit soft or hard constraints are present. For example, course timetables often need to take into

account both explicit constraints (e.g. preferences, capacities) and informal agreements or manually enforced rules.

For CSPs, informed ML allows to leverage well-defined sources of symbolic knowledge at training, which cannot however be easily exploited at search time, e.g. particularly expensive (e.g. NP-hard) propagators [156]. In this context, a deep learning approach may learn to satisfy such constraints without the need for a propagator at search time.

In this section, we will describe a method that trains a network for identifying variable-value assignments that are likely to be feasible. We will assume the availability of both *implicit knowledge* (from data), and *explicit symbolic knowledge* that can be accessed prior to the search process.

Rather than tackling a real-world problem directly, we perform experiments in a controlled setting, with the aim to gauge the potential of the approach and identify the key challenges. The idea, in the spirit of [157], is to test the ground before starting the complex and time-consuming endeavor of applying such methods in a real-world use case.

In detail, we use as a benchmark the Partial Latin Square (PLS) completion problem, which requires to complete a partially filled $n \times n$ square with values in $\{1..n\}$, such that no value appears twice on any row or column. Despite its simplicity, the PLS is NP-hard, unless we start from an empty square, it has practical applications (e.g. in optical fiber routing), and serves as the basis for more complex problems (e.g. timetabling). We focus on the only PLS due to its clear structure, availability of multiple solutions that can be easily generated, and its single defining parameter (size).

Using a classical constrained problem as a case study grants access to symbolic domain knowledge (the declarative formulation), and facilitates the generation of empirical data (problem solutions). This combination enables controlled experiments that are impossible to perform on real-world datasets.

As a baseline, we train on a pool of solutions a problem-agnostic, data-driven, approach. We then devise a simple method to extract multiple training examples from a finite set of solutions, and we define a technique, building over Semantic Based Regularization, [158] to inject at training time domain knowledge coming from constraint propagators. We then adjust the amount of initial data (empirical knowledge) and of injected constraints (domain knowledge) and assess the ability of the approach to identify feasible assignments.

Baseline method The analysis that we aim to perform requires a data-driven technique that can solve a constrained problem, with no access to its structure. In the approach from [159], a neural network is used to learn how to extend a partial variable assignment so as to retain feasibility. Despite its limited practical effectiveness, this method shares the best properties of constraint acquisition (no explicit problem information), without being restricted to constraints expressed in a classical declarative language.

This last approach was chosen as our baseline, since it represents (to the best of our knowledge) the data driven method for constraint problems that requires the least amount of problem knowledge. In particular, it requires neither information about the problem constraints (like e.g. [155]), nor a fully known (or at least evaluable) problem model like all RL approaches.

The baseline approach is based on training a neural network to extend a partial assignment (also called a *partial solution*) by making one additional assignment, so as to preserve feasibility. Formally, the network is a function:

$$f : \{0,1\}^m \to [0,1]^m \tag{6.1}$$

Algorithm 1 DECONSTRUCT(x)

```
D = ∅
while ‖x‖_1 > 0 do
    Let y = 0    # zero vector
    Select a random index i such that x_i = 1
    Set x_i = 0, set y_i = 1
    Add the pair (x, y) to D
return D
```

whose input and output are m dimensional vectors. Each element in the vectors is associated to a variable-value pair $\langle z_j, v_j \rangle$, where z_j is the associated variable and v_j is the associated value. We refer to the network input as x, assuming that $x_j = 1$ iff $z_j = v_j$. Each component $f_j(x)$ of the output is proportional to the probability that pair $\langle z_j, v_j \rangle$ is chosen for the next assignment. This is achieved in practice by using an output layer with m neurons with a sigmoid activation function. The setup makes no assumptions on the constraint structure but requires a fixed problem size and variables with finite domains.

Dataset Generation Process. The input of each training example corresponds to a partial solution x, and the output to a single variable value assignment (represented as a vector y using a one-hot encoding). The training set is constructed by repeatedly calling the randomized deconstruction procedure of Algorithm 1 on an initial set of full solutions (referred to as *solution pool*). Each call generates a number of examples that are used to populate a dataset. At the end of the process, we discard multiple copies of identical examples. Two examples may have the same input, but different output, since a single partial assignment may have multiple viable completions.

Unlike [159], here we sometimes perform *multiple calls to Algorithm 1 for the same starting solution*. This simple approach enables to investigate independently the effect of the training set size and of the actual amount of empirical knowledge (the size of the solution pool).

Training and Knowledge Injection. The basic training for the NN is the same as for neural classifiers. Since the network output can be assimilated to a class, we process the network output through a softmax operator, and then we use as a loss function the categorical cross-entropy H. Additionally, we inject domain knowledge at training time via an approach that combines ideas of SBR and constraint programming.

Without loss of generality, we assimilate *domain knowledge to a constraint propagator*, in the sense that it can be used to flag specific variable-value pairs as either feasible or infeasible. In our experimentation, we indeed use a classical propagator (forward checking) as the source of symbolic knowledge.

Formally, given a constraint (or a collection of constraints) C, here we will treat its associated propagator as a multivariate function such that $C_j(x) = 1$ iff assignment $z_j = v_j$ has not been marked as infeasible by the propagator, while $C_j(x) = 0$ otherwise. Given that, we formulate three different approaches to augment the loss function with an SBR inspired term.

The first one relies on the usual assumption that pruned values are supposed to be provably infeasible. Given an example $\langle x, y \rangle$, we have:

$$L_{sbr}^{negative}(x) = \sum_{j=0}^{m-1} ((1 - C_j(x)) \cdot f_j(x)) \tag{6.2}$$

Algorithm 2 FEASTEST(X, C, h)

$J^* = \arg\max_{j|C_j(x)=1}\{h_j(x)\}$ {Most likely assignments}
Select j^* uniformly at random from J^*
Set $x_{j^*} = 1$
if SOLVE(x, C_{pls}, h_{rnd}) $\neq \perp$ **then**
    return 1 {Globally feasible}
else
    return 0 {Globally infeasible}

i.e. increasing the output of a neuron corresponding to a pair flagged as infeasible incurs in a penalty that grows with $f_j(x)$.

For the other two methods, we just acknowledge that the domain knowledge may be incomplete, discouraging provably infeasible pairs, and encouraging the remaining ones. The only difference is in the cost function. In one instance the cost function is the binary cross-entropy, since for each partial solution there may exist many global viable completions, and the SBR inspired term is:

$$L_{sbr}^{bce}(x) = \sum_{j=0}^{m-1} (C_j(x) \cdot log(f_j(x)) + (1 - C_j(x)) \cdot log(1 - f_j(x)) \tag{6.3}$$

In the other case instead we employ the MSE as cost function for the SBR-inspired regularization:

$$L_{sbr}^{mse}(x) = \sum_{j=0}^{m-1} (C_j(x) - f_j(x))^2 \tag{6.4}$$

Our full loss is hence given by:

$$L(x, y) = H\left(\frac{1}{Z} f(x), y\right) + \lambda L_{sbr}(x) \tag{6.5}$$

where Z is the partition function and the scalar λ controls the balance between the cross-entropy term H and the SBR term, i.e. the amount of trust we put in the incomplete domain knowledge. Since we assume the domain knowledge/propagator to be incomplete, there is a risk of injecting incorrect information into the model. In practice, this is balanced by the presence of the categorical cross-entropy term in the loss: only the single pair that comes from the deconstruction of a full solution will be associated with a non-null component, and this pair is guaranteed to be *globally feasible*.

The method can be applied for all known propagators with discrete, finite domain, variables. By adapting the structure of the SBR term, it can be made to work for important classes of numerical propagators (e.g. those that enforce Bound Consistency).

Evaluation and Knowledge Injection. We evaluate the approach via a constraint solver, a classical PLS model, and a randomized search strategy. Formally, we assume access to a function SOLVE(x, C, h), where x is the starting partial assignment, C is the considered (sub)set of problem constraints, and h is a probability estimator for variable-value pairs (e.g. our trained NN). The function runs a depth-first search using the Google or-tools [160] constraint solver: the variable-value pair for the left branch is chosen at random with probabilities proportional to $h(x')$, where x' is the current state of assignments. The SOLVE function returns either a solution, or $\perp$ in case of infeasibility.

Our main evaluation method tests the ability of the NN to identify individual assignments that are globally feasible, i.e. that can be extended into full solutions. This is done via Algorithm 2, which 1) starts from a given partial solution; 2) relies on a constraint propagator C (if supplied) to discard some of the provably infeasible assignments; 3) uses the NN to make a (deterministic) single assignment; 4) attempts to complete it into a full solution (taking into account all problem constraints, i.e. C_{pls}). Replacing the NN with a uniform probability estimator provides an uninformed search strategy. We repeat the process on all partial solutions from a test set and collect statistics. This approach is identical to one of those in [159], with one major difference, i.e. the ability to use a constraint propagator for "correcting" the output of the probability estimator. This enables us to assess the impact of using the offline knowledge directly during the search, something that is allowed in our controlled setting, but that would be impossible (e.g.) with an actual simulator.

Unlike in typical ML evaluations, accuracy is not a meaningful metric in our case, as it is tied to the (practically irrelevant) ability to replicate the same sequence of assignments observed at training time. Incidentally, accuracy is very low when measured in the traditional way in all our experiments.

6.2 Empirical Analysis

In this section we discuss our experimental analysis, which is designed around three key questions:

Q1: *Does injecting knowledge at training time improve the network's ability to identify feasible assignments?*

Q2: *What is the effect of adjusting the amount of available empirical knowledge?*

Q3: *Can knowledge injection improve the ability to satisfy constraints in a soft fashion, i.e. in terms of the number of violations?*

While Q1 and Q2 focus on the feasibility of individual assignments, Q3 assumes that some degree of infeasibility can be tolerated. We present a series of experiments in our controlled use case that investigate such research directions. Details about the rationale and the setup of each experiment are reported in dedicated sections, but some common configurations can be immediately described.

We perform different experiments on 7×7, 10×10 and 12×12 PLS instances, resulting respectively in input and output vectors with 343, 1000 and 1728 elements. For all the experiments, we use a feed-forward, fully-connected neural network with three hidden layers, each with 512 units having ReLU activation function. This setup is considerably simpler than the one we used in [159], but manages to reach very similar results. We employ the Adam optimizer from Keras-TensorFlow 2.0, with default parameters. We use a batch size of 2048 for experiments on the PLS-7, whereas we adopt a batch size of 50,000 for the ones on PLS-10 and PLS-12.

Regularization methods comparison and λ-tuning As a first step to evaluate the impact of knowledge injection at training time, we compare the regularization methods and evaluate how the λ value affects the performance of each of them. We focus on the PLS-12, which is the greatest dimension among the ones examined in this work so that advantages and limitations for each method can easily emerge. We refer as NEGATIVE, BCE and MSE to the methods which respectively employ the SBR-inspired loss functions described in eq. (6.2), eq. (6.3) and eq. (6.4).

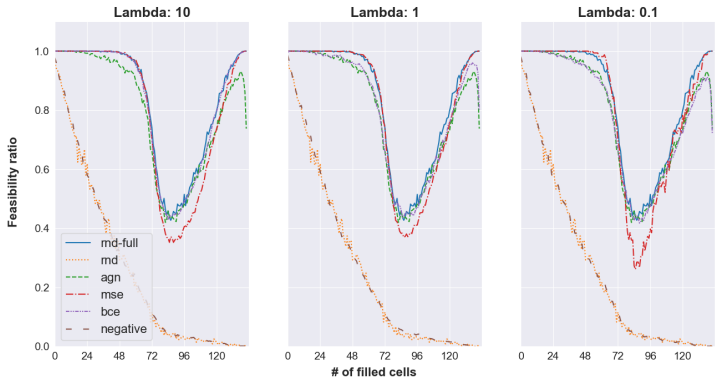


Figure 6.1: Effect of the injection of the all constraints at training time comparing the regularization methods for different λ values, on the PLS-12.

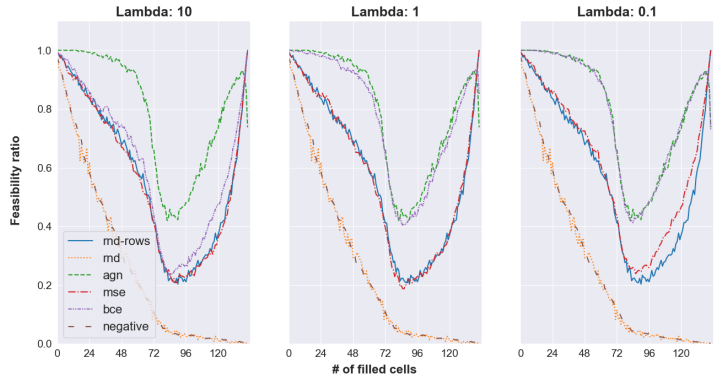


Figure 6.2: Effect of the injection of the only rows constraints at training time comparing the regularization methods for different λ values, on the PLS-12.

The evaluation concerns whether injecting domain knowledge *at training time* may help the neural network in the identification of feasible assignments, *assuming the same knowledge is not available at search time*. We also assume in this instance that a large number of historical solutions is available.

This experimentation is motivated by practical situations in which: 1) a domain expert has only partial information about the problem structure, but a pool of historical solutions is available; 2) some constraints (e.g. from differential equations or discrete event simulation) cannot be enforced at search time. In detail, the training set is generated using the deconstruction approach from Algorithm 1, starting from a set of 10,000 PLS solutions, 75% of which are used for training and the remaining ones for testing. Each solution is then deconstructed exactly once, yielding a training set of 1,000,000 examples. An additional validation set of 5,000 partial solutions is adopted to assess the improvements during training via the FEASTEST procedure, using the network as the heuristic h and an empty set of constraints as C (no propagation when choosing the assignment to be checked). Since this computation is really expensive, we perform the assessment every 10 epochs. If for 10 successive checks the best global feasibility ratio found so far is not improved then we stop the training.

For each regularization approach, we train two neural networks: one trained with knowledge about row constraints and another trained with knowledge about row and column constraints. For the first network, we use the SBR-inspired methods (and a forward checking propagator) to inject knowledge

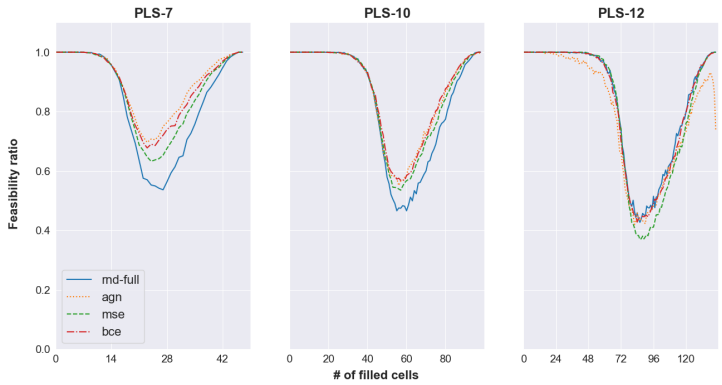


Figure 6.3: Full constraints injection at training time on different problem dimensions.

that both assigning a variable twice and assigning a value twice on the same row is forbidden. For the second one, we do the same, applying the forward checking propagator also to column constraints (i.e. no value can appear twice on the same column). Due to the use of an incomplete propagator, both the networks make use of incomplete knowledge.

In addition, we train a model-agnostic neural network that lacks even the basic knowledge that a variable cannot be assigned twice, since this is not enforced by our input/output encoding, and must infer that from data.

We evaluate the resulting approaches via the FEASTEST procedure, using the separated test set as X, the trained networks as h, and an empty set of constraints (i.e. no propagation at test time). We compare them with methods that randomly choose an assignment with an uniform probability distribution *but that can rely on a set of constraints C during the evaluation*. We consider the two scenarios in which C is the set of the row constraints (RND-ROWS) and the one in which C is the set of column and row constraints (RND-FULL). These methods are representative of the behavior (at each search node) of a constraint programming solver having access to either only row constraints or the full problem definition. It allows us to gauge the ideal effect of the offline symbolic knowledge.

Finally, we consider a very pessimistic baseline, referred to as RND, which again randomly chooses an assignment with an uniform probability distribution but does not rely on the propagation of any constraints (i.e. C is the empty set). We then produce "feasibility plots" that report on the x-axis the number of assigned variables (filled cells) in the considered partial solutions and on the y-axis the ratio of suggested assignments that are globally feasible. Since RND-ROWS and RND-FULL methods are the only ones that can rely on *online* constraints propagation, *we have highlighted them using solid lines*.

In fig. 6.1, we show results when all the constraints are employed by the forward checking constraints propagator, whereas in fig. 6.2 we do not propagate the columns constraints. The balance between learning the constraints from empirical data and the forward checking propagator is tuned by λ: reducing its value means giving more emphasis on the global feasible assignments obtained by deconstruction of the complete solutions rather than on the incomplete knowledge. We report results for λ equal to 10, 1 and 0.1. For all the λ values, the NEGATIVE approach's behavior is hardly distinguishable from RND. A reasonable explanation is that it encourages the network to keep the output the lowest as possible instead of discouraging the network to make provably infeasible assignments. Since this approach is not effective at all, we do not consider it for further analysis.

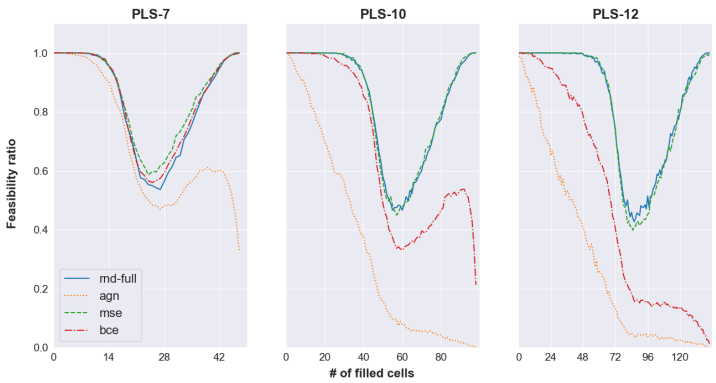


Figure 6.4: Full constraints injection at training time when the dataset is reduced to the 10% of its initial size.

We choose the best λ parameters for the BCE and MSE regularization methods with the aim of distilling the constraints propagator in the neural network's weights, finding a tradeoff between learning from correct knowledge and the incomplete one. Considering the overall performance, the MSE regularization method provides better results with $\lambda = 1$, so this value is chosen for the successive analysis. The BCE approach provides the best performance with $\lambda = 10$.

Despite in fig. 6.2 lower values of λ provide better feasibility ratios, these results are not preferable since they make the regularization not effective, i.e. the methods collapse to AGN. The BCE method provides a little improvement over the MSE one but, as we will see when answering question 2, it is not robust when only a limited amount of empirical knowledge is available.

Domain Knowledge at Training Time for different problem dimensions Unlike the previous section, here we extend the analysis to the PLS of dimensions 7 and 10, considering the only MSE and BCE regularization methods together with their best λ values. The datasets are generated as described in the previous section, yielding training sets of size $350,000$ and $700,000$ for respectively the PLS-7 and PLS-10.

In fig. 6.3, we show results when all the constraints are employed by the forward checking constraints propagator. As long as the problem size is small enough, AGN performs considerably better than RND-FULL, even if no propagation is employed at evaluation time: this is symptomatic of the network actually managing to learn the problem constraints from the available data, which (unlike the propagator output) is guaranteed feasible. As the problem size grows, the gap decreases, until it almost disappears for PLS-12.

For PLS-7, injecting incomplete symbolic knowledge appears to have an *adverse* effect, as it biases the network toward trusting too much the incomplete propagator. With a large problem dimension (i.e. PLS-12) the benefits introduced by knowledge injection become more visible, especially when using the BCE regularization method. The decreasing performance of the data driven methods is likely a consequence of the training set size staying constant, in the face of a search space that becomes increasingly large. In all cases, the feasibility ratio is high for almost empty and almost full squares, with a noticeable drop when $\sim$60% of the square is filled. The trend may be connected to a known phase transition in the complexity of this problem [161].

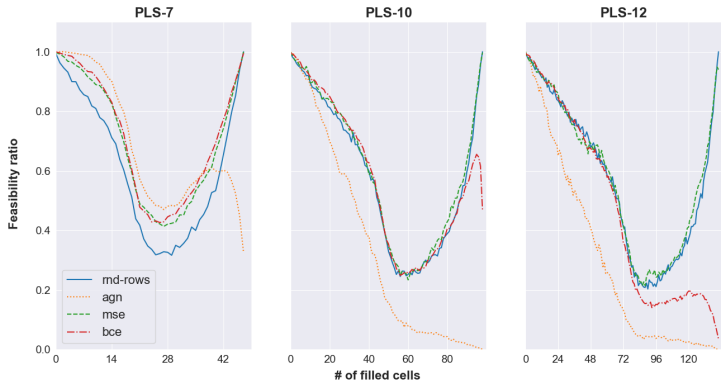


Figure 6.5: Rows constraints injection at training time when the dataset is reduced to the 10% of its initial size.

6.2.1 Training Set Size and Empirical Information

Next, we proceed to tackle Question 2, by acting on the training set generation process. In classical ML approaches, the amount of available information is usually measured via the training set size: this is a reasonable approach since the number of training examples has a strong impact on the ability of an ML method to learn and generalize. We performed experiments to probe the effect of the training set size on the performance of the data-driven approaches: the training sets are reduced to the 10% of the initial size, i.e. $35,000$, $70,000$ and $100,000$ for respectively PLS of size 7, 10 and 12.

In fig. 6.4 and fig. 6.5, we show results when respectively all the constraints and the only rows constraints are injected via the regularization methods. In this case, *knowledge injection at training time has a dramatic effect*: the AGN approach is very sensitive to the available number of examples and it has a great drop in performance. Despite being less pronounced, the BCE method has a major drop in performance too. Instead, the MSE approach provides much more robust results.

In our setup, we have also the possibility to apply the deconstruction process multiple times, so that the number of different examples that can be obtained from a single solution grows with the number of possible permutations of the variable indices (i.e. $O(n^2!)$ for the PLS). The approach opens up the possibility to *generate large training sets from very few starting solutions*. This is scientifically interesting since the "actual" empirical information depends on how many solutions are available; it is also very useful in practice since in many practical applications only a relatively small number of historical solutions exists.

The results of this evaluation are shown in Figure 6.6 for a solution pool of 100 elements, rather than the original 10,000. Due to the bad results provided with the reduced datasets, we do not further investigate the BCE regularization approach but we examine the only MSE method. For this analysis, we collapse the feasibility results of the neural network trained with full knowledge injection (referred to as MSE-FULL) and of the network trained without the columns constraints knowledge injection (MSE-ROWS) in a single plot. The size of the generated training set is comparable to the original. Despite the dramatically reduced number of training solutions, the MSE-ROWS and MSE-FULL methods perform really close to respectively RND-ROWS and RND-FULL, i.e. they behave similarly to what the propagator would if employed at search time. Instead, the performance of the AGN drops dramatically, stressing again its sensitivity to the available empirical information.

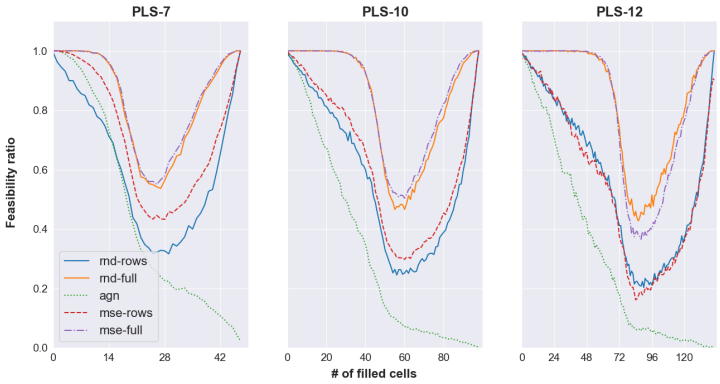


Figure 6.6: Effect of reducing the solution pool size from 10,000 to 100.

From a practical point of view, it seems that injecting constraints during training can be a *very effective strategy when only a small number of training solutions is available*. Constraint injection tends to be redundant if the same type of propagation can be performed at search time, but can be very useful in cases when this is not possible.

Constraint Violation Assessment In the last set of our experiments, we investigate the effectiveness of the trained neural networks at guiding a search process toward solutions that are close to being feasible, but not necessarily so. This is equivalent to treating constraints as soft and may be of practical relevance on overconstrained problems (e.g. many real-world timetabling applications). This setup tends to be more challenging for the ML models, since chains of variable-value assignments may lead to partial solutions that are remarkably different from those observed at training time.

In detail, we used each trained neural network as a value selection heuristic in depth-first search, once again for PLS of sizes 7, 10 and 12; we used for this experiment a fixed variable ordering. As a baseline for the comparison, we consider (uniformly) random value selection referred to as RND, while for the neural networks we select a random value with probability proportional to the network output. We generate a fixed number of solutions (500) from an empty square, rather than starting from partially filled ones. When generating the solutions, we never propagate the entirety of the PLS constraints: this setup serves as a controlled experiment for use cases where some constraints are either unknown or cannot be enforced at search time. We measure the degree of feasibility of the generated solutions by quantifying the violations for the constraints that were not propagated at search time. For this purpose, we measure violations by counting how many times a value is not appearing exactly once in the same row or column, depending on which constraint is being considered.

We train two model-agnostic neural networks: one on the dataset obtained by random deconstruction of 10,000 solutions (referred to as AGN-10K) and the other one on the dataset obtained by multiple random deconstructions of 100 solutions (referred to as AGN-100). Similarly, we train two neural networks with knowledge injection at *training time* of all the constraints by means of the mean squared error version of the SBR-inspired method and the forward checking propagator (referred to as SBR-10K and SBR-100). Neither row nor column constraints are propagated during the search, and therefore we count the violations of both in the final solutions.

Table 6.1: Number of soft constraints violations per generated solution.

	rnd		**agn-10k**		**sbr-10k**		**agn-100**		**sbr-100**	
	rows	cols	rows	cols	rows	cols	rows	cols	rows	cols
PLS-7	29	29	11	9	4	3	20	20	4	4
PLS-10	61	61	28	25	8	7	52	53	7	7
PLS-12	88	88	56	53	22	30	70	76	17	20

Results are shown in table 6.1: *the SBR-inspired approach allows to significantly reduce the number of violations, and it achieves very similar results even when only a small amount of empirical knowledge is available.* The AGN approach performs considerably better than RND, as long as a large pool of solutions is available, but the gap narrows when trained on examples generated from 100 solutions. It is interesting to see how, when constraints are interpreted in a soft fashion, injecting full problem knowledge at training time has a much more robust effect compared to the analysis in section 6.2.

6.3 UNIFY: a Unified Policy Designing Framework for Solving Integrated CO and ML Problems

Throughout my research, I played a key role in developing a unification framework for a family of existing ML and CO approaches, referred to as UNIFY. This approach assumes access to *problem knowledge in the form of both a declarative formulation (an objective function and a set of constraints) and data (either historical or from a simulator).* The framework is built upon the decomposition of the policy into two components: an unconstrained ML model and a CO problem. This decomposition facilitates a more effective *training step.* The interface between these components involves a set of *"virtual" parameters* in the CO problem model, serving as an additional (potentially useful) design handle.

Our method can be intuitively understood using a motorsport analogy. In such a setting, the vehicle control unit can adapt the input received from the rider (e.g. accelerating, breaking, stirring) to account for conditions such as weather, asphalt grip, and reservoir capacity. This simplifies the task faced by the driver, who can then focus on planning trajectories, dealing with other racers, and long-term strategy. Similarly, in our decomposition the CO problem (the vehicle control unit) exploits the explicit information to generate actionable decisions at each stage and to guarantee constraints satisfaction; the ML model (the driver) "pilots" the CO problem by adjusting some of its modeling parameters (e.g. costs or constraint thresholds), with the goal of optimizing robustness and long-term behavior.

Since the approach is based on a decomposition, *multiple learning and optimization methods* can be used for its implementation. In our presentation, we emphasize the use of RL for the learning task, due to its ability to handle both non-differentiable loss functions and sequential decision problems. Any CO technique can be used to tackle the optimization problem, such as mathematical programming (including mixed-integer approaches) or constraint programming. In addition to introducing UNIFY, we demonstrate how the method can address problems typically tackled with DFL, constrained RL, algorithm configuration, and stochastic optimization, and incorporates additional properties and functionality. While UNIFY cannot be expected to exceed the performance of all the methods it can replicate, it represents a strict improvement in terms of flexibility and applicability.

6.3.1 Key Problem Elements and Notation

In the most general case, our method targets problems with the following properties:

1) *Multiple decision stages*, referred to as a sequence of stage indices $\{k\}_{k=1}^{T}$. The notation T represents the end of the planning horizon, and it might be infinite for decision processes that need to run indefinitely. A decision stage may represent one working day in a production scheduling context or a 15-minute time interval in an EMS.

2) *Observables available at each stage*, referred to as vector of values $x^{(k)}$. These might represent information that is useful for making decisions (e.g., power generation and demand forecasts for the next state), for evaluating the impact of past decisions (e.g., actual power generation and forecast), for making predictions (e.g., weather at the current stage), or for describing the system state (e.g. current level of a storage device).

3) *Uncertainty*, affecting the values of the observable and represented via a probability distribution P, i.e. $x \sim P$. While we make no specific assumption on the distribution, its properties (e.g., non-anticipative or exogenous) can affect which solution methods can be employed to deploy our formulation.

4) *Decisions to be taken for each stage*, referred to as vectors of variables $z^{(k)}$. The variables might represent how many items to produce in a day, the power-flows for a 15 minute interval in an EMS, which donor-patient pairs to select for surgery in an organ transplant program, etc. We make no assumption on the domain of $z^{(k)}$, meaning that in the general case, even the vector size may not be fixed.

5) *Hard constraints for each stage*, which define the feasible values for the decision variables. We represent such constraints as a set, whose definition depends on the values of the observables: formally, we have that $z^{(k)} \in C(x^{(k)})$, with $C(\cdot)$ being a set-valued function. We assume without loss of generality[1] that the feasible set can be defined based on the observables for the current stage (e.g. the constraints may require a power balance).

6) *An immediate cost function*, referred to as f, which specifies the cost incurred in stage $k+1$ depending on the decisions at stage k, and on the observable at stages k and $k+1$ (i.e. the previous state and how uncertainty unfolds). Formally, we have that the cost incurred at stage $k+1$ is given by $f(x^{(k)}, x^{(k+1)}, z^{(k)})$. For example, the function may measure the total profit we get at stage $k+1$ depending on the observed demand (in $x^{(k+1)}$), by selling items produced ($z^{(k)}$) and stored (in $x^{(k)}$) at stage k. We assume the goal is to *minimize the expected cost* over all the decision stages.

Knowledge about the constraints and the cost function can typically be obtained *in explicit form* by talking to domain experts, while information about the uncertainty distribution P is typically available *in implicit form*, through collections of historical data.

As stated, the list is meant to define the most general conditions for the application of our method. A practical use case can (and typically will) introduce additional restrictions, e.g., a limited number of stages, a focus on exogenous uncertainty, or a deterministic cost function f. Similarly, not all terms in

[1] If this is not true, the observables can be redefined so as to capture all relevant information (typically with adverse effects on scalability).

Problem	**Stages**	**Observables**	**Uncertainty**	**Decisions**	**Constraints**	**Cost**
EMS	Time intervals	Past demands and production	True demands and production	Power flows	Power balance and limits	Power flows cost
Production scheduling	Production days	Customer information, time of the year	True product demands	Molds to use	Molds availability	Manifacture cost

Table 6.2: Examples of real-world problems that can be tackled with UNIFY and their components.

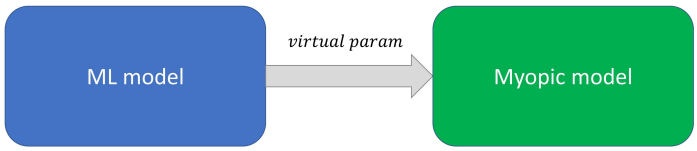


Figure 6.7: High-level overview of the approach, in the case of the EMS example.

$x^{(k)}$ might be relevant for defining the constraint set $C(x^{(k)})$ or the cost $f(x^{(k)}, x^{(k+1)}, z^{(k)})$. Table 6.2 gives an overview of the key elements of the use cases we illustrated in the previous section, from the perspective of our analysis.

The main idea in UNIFY is to extend an optimization approach by *allowing an external component to guide its behavior*. Doing this requires introducing a mechanism for enabling communication between such a component and the optimization solver. This is achieved in UNIFY by *adjusting the value of certain parameters* in the optimization problem. Such parameters can be either 1) chosen from those naturally present in the formulation or 2) introduced ad-hoc for this purpose. In both cases, the selected parameters remain interpretable (since they are employed in a symbolic model), but only in a loose sense (since they are determined by a problem-agnostic component): for this reason, we refer to them as *virtual parameters*.

So far we have introduced an abstract external component that tunes the virtual parameters. In UNIFY, such a component consists of a ML model, as shown in Figure 6.7. In particular, we introduce a ML model h, whose role is to predict the optimal virtual parameters $y^{(k)}$ at stage k given the current observation $x^{(k)}$, i.e. $y^{(k)} = h(x^{(k)})$.

Our design choice is motivated by a few observations: 1) ML is naturally well-suited to deal with uncertainty; 2) predictions made by ML models are contextual, meaning that in our case they can change depending on the observed $x^{(k)}$ values; 3) once trained, a ML model can very efficiently perform inference on unseen examples. Alternative options for the external component, such as blackbox optimization, do not provide the same advantages.

Unlike in classical ML tasks such as supervised learning, the ML model should not be trained for maximum accuracy. On the one hand, the model output consists of *virtual* parameters, which may lack a real-world counterpart, and therefore any ground truth value. However, our overall goal is not to make accurate predictions but rather to lead to optimal decisions. Therefore, our ML model should in principle be trained to minimize the decision cost of the overall policy.

6.3.2 UNIFY formalization

We now revisit from a formal perspective the concepts we have just introduced, with the goal to define a general and technically sound method.

Solution Approach as a Policy Using terminology borrowed from sequential decision-making, the process of choosing a decision vector $z^{(k)}$ based on the observables $x^{(k)}$ can be viewed as the application of a *policy*. Formally, this is defined as a function:

$$\pi : (x; \theta) \mapsto z \in C(x) \tag{6.6}$$

where we have that x is the vector of observables, z the vector of decisions, and $C(x)$ the feasible set. The requirement $z \in C(x)$ implies that any viable policy is expected to *consistently satisfy all the constraints* defined for a single decision stage.

The term θ represents a set of *training parameters* that can be used to adjust the function behavior. These should not be confused with the virtual parameters in our decomposition, which will be formally discussed later in this section. The θ parameters should be chosen to minimize the long-term cost of the decisions, based on the available information about uncertainty. We can therefore formulate the training problem as:

$$\begin{aligned} &\arg\min_{\theta \in \Theta} \mathbb{E}_{\tau \sim P} \left[\sum_{k=1}^{T} \gamma^k f(x^{(k)}, x^{(k+1)}, z^{(k)}) \right] \\ &\text{with: } z^{(k)} = \pi(x^{(k)}; \theta) \end{aligned} \tag{6.7}$$

where f is the cost function for a single stage and τ refers to a trajectory, i.e. to a sequence of observables and (feasible) decisions, i.e. $\tau = \{x^1, z^1, x^2, z^2, \dots x^T, z^T, x^{T+1}\}$.

The probability of a trajectory is given by the distribution P, which (in practical applications) will likely be approximated via a collection of historical data or via a simulator. The term T is the end of horizon, and it can be infinite if the sequence is not upper-bounded. In this case, the discount factor γ should be strictly lower than 1, while γ should be equal to 1 for decision problems over a finite horizon.

Once training has been performed and an optimal parameter vector θ^* has been found, the decision-making problem can be solved by repeatedly observing $x^{(k)}$, querying $\pi(x^{(k)}; \theta^*)$ to obtain a decision vector $z^{(k)}$, and deploying the decisions to move to the next step.

Decomposition to Simplify the Training Problem Solving the training problem from Equation (6.7) directly is very challenging without making additional assumptions, since the distribution P will typically be approximated via a *large dataset*, the decisions are expected to be *always feasible*, and the decision space can be *complex or combinatorial*. Formally, in UNIFY the monolithic policy is reformulated as:

$$\pi(x; \theta) = g(x, h(x; \theta)) \tag{6.8}$$

where $h(x; \theta)$ is a ML model and $g(x, y)$ corresponds to the solution of a constrained optimization problem. Both components take as input the observation x. The ML model is the only component in the decomposition whose behavior is affected directly by the training parameters θ; its output corresponds to the virtual parameters previously discussed. The virtual parameters serve as an additional input to the CO problem function, thus allowing the ML model to adjust its behavior. The formal definition for

the g function is as follows:

$$g(x, y) \equiv \underset{z \in \tilde{C}(x,y)}{\arg\min}\, \tilde{f}(x, y, z) \tag{6.9}$$

where the cost and constraint functions $\tilde{f}(x, y, z)$ and $\tilde{C}(x, y)$ need to be defined when formulating the decomposition. Both terms are related to the original cost and constraint function, but they are also different in some important aspects.

First, $\tilde{f}$ and $\tilde{C}$ depend on the virtual parameter vector y, which allows the ML model to alter the optimal solution of Equation (6.9). In practice, it is enough to make only one of the two terms explicitly dependent on y.

Second, the cost function $\tilde{f}$ does not use the next-stage observables as input. This has the effect of making the g function myopic, but it is also necessary to perform inference in practice, i.e. to evaluate Equation (6.8) and obtain a decision vector for the current stage. One way to define the $\tilde{f}$ function consists in: 1) starting from the original cost function f; 2) neglecting any term linked to future information (e.g. actual demand values in production scheduling); and 3) introducing terms linked to the virtual parameters (e.g. the cost for power flows from/to the storage system in our EMS example).

Third, the set $\tilde{C}(x, y)$ should imply the feasibility of the decision vector (i.e. the output of g) according to the original constraints. This is necessary for the policy to satisfy Equation (6.6) and can be formalized as:

$$z, y \in \tilde{C}(x) \Rightarrow z \in C(x) \tag{6.10}$$

In practice, the property can be enforced either 1) by retaining the same constraints as the original problem, i.e. $\tilde{C}(x, y) = C(x)$, as in the case of our EMS example, or 2) by incorporating the virtual parameters in a conservative fashion (e.g. temporal buffers over deadline constraints, or reduction factors over capacity constraints).

Defining the virtual parameters y, the cost function $\tilde{f}(x, y, z)$, and the constraint function $\tilde{C}(x, y)$ are the major design decisions when grounding our method on a practical use case.

Reformulated Training Problem With the decomposed policy reformulation, the training problem becomes:

$$\begin{gathered} \underset{\theta \in \Theta}{\arg\min}\, \mathbb{E}_{\tau \sim P} \left[\sum_{k=1}^{T} \gamma^k f(x^{(k)}, x^{(k+1)}, z^{(k)}) \right] \\ \text{with: } z^{(k)} = g(x^{(k)}, y^{(k)}) \quad \text{and: } y^{(k)} = h(x^{(k)}; \theta) \end{gathered} \tag{6.11}$$

Equation (6.11) is considerably easier to solve than Equation (6.7), since the decomposition allows one to partition the elements of complexity in the original problem and handle them via distinct, more appropriate techniques: the CO problem (which can be solved via mathematical programming or similar techniques) is in charge of ensuring feasibility and exploring a complex decision space; the ML model handles uncertainty and long-term feasibility; both components contribute to cost optimization.

The main challenge when solving Equation (6.11) is the fact that $g(x^{(k)}, y^{(k)})$ is defined through an arg min operator. In many practical cases, such as LPs or combinatorial problems, the decision vector may change in discrete steps in response to arbitrarily small changes in the virtual parameter vector, thus making $g(x^{(k)}, y^{(k)})$ piecewise constant and non-differentiable. Thankfully, optimizing over

functions with these properties is a much better-understood topic, thanks to recent developments in DFL, and decades of research in both black box optimization and RL.

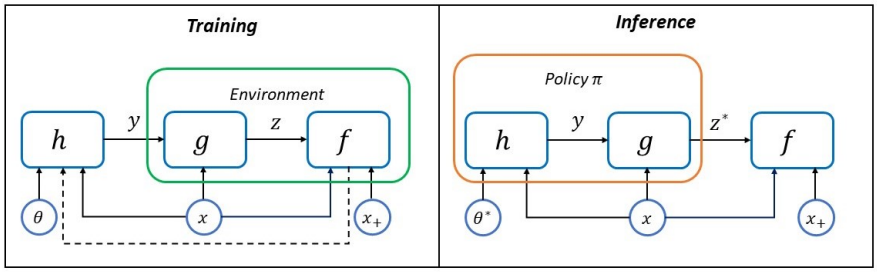


Figure 6.8: UNIFY decomposition for the training and inference problems.

In fact, Equation (6.11) can be mapped to a traditional RL problem by a simple change in perspective. At *training time*, the solution of the CO problem can be seen as *part of the environment* and the virtual parameter vector $y^{(k)}$ is viewed as the RL agent "action" for the k-th stage. Conversely, at *inference time* the ML model and the CO problem are components of a single policy (as already discussed). The two different viewpoints are depicted in Figure 6.8.

Formally, we can perform training as for a MDP $\langle X, Y, P^+, R, P^1, \gamma \rangle$ where: the set of possible states corresponds to the set X of possible observables; the set of possible decisions corresponds to the set Y of possible virtual parameters; the probability of the initial state P^1 is simply given by $P(x^1)$; and γ is the discount factor. Finally, the probability of the next state and the reward are defined as:

$$P^+(x^{(k+1)} \mid x^{(k)}, y^{(k)}) = P(x^{(k+1)} \mid x^{(k)}, g(x^{(k)}, y^{(k)})) \tag{6.12}$$

$$R(x^{(k+1)}, x^{(k)}, y^{(k)}) = f(x^{(k)}, x^{(k+1)}, g(x^{(k)}, y^{(k)})) \tag{6.13}$$

In other words, they are respectively the probability of the next observable given the current decision vector $z^{(k)} = g(x^{(k)}, y^{(k)})$, and the corresponding cost. As a major benefit, this mapping enables one to use *any RL algorithm* for policy training in UNIFY.

6.3.3 Generalization

The main appeal of our method lies in its *ease of use* (since it can be implemented by relying on standard RL libraries) and most of all its *versatility*. In particular, UNIFY can be used to tackle problems that are typically addressed via ad-hoc approaches. Additionally, in some cases UNIFY actually subsumes the existing methods. In this section, we will discuss how the method can be configured for several such scenarios, the benefits it can bring, and under which conditions existing approaches should instead be preferred.

Decision Focused Learning In the "predict, then optimize" paradigm, the objective is to estimate a subset of the optimization problem parameters that are unknown at solution time. For instance, in vehicle routing problems, it might involve estimating travel times, or in a production scheduling problem, predicting item demands to define the optimal scheduling plan. Unlike the goal of maximizing accuracy, DFL aims to train an ML model to minimize (or maximize) the task loss, representing the cost of the solution instantiated by the prediction. Given that obtaining a highly accurate model is often

challenging, DFL allows for the exploitation of the limited representational capacity of the ML model. The idea gained attention after the seminal work by [162], and it is well covered in a recent survey [163]. In practice, DFL methods address the following problem:

$$\arg\min_{\theta \in \Theta} \mathbb{E}_{x,y^* \sim P} \left[f(x, y^*, z^*) \right] \tag{6.14}$$

$$\text{with: } z^* = \arg\min_{z} \{ f(x, y, z) \mid z \in C \} \tag{6.15}$$

$$\text{and: } y = h(x; \theta) \tag{6.16}$$

where y is the ML model estimate of the unknown optimization problem parameters, z^* is the optimal solution of the problem instantiated by the predictions y, and y^* are the ground truth problem parameters. While the original approach was limited to QP, subsequent works have tackled linear and combinatorial problems, either in an approximate fashion via continuous relaxations [124], or in an exact fashion by assuming a fixed feasible space and linear costs [3, 164]. Outer and inner relaxations have also been used to improve scalability [165, 166]. UNIFY can be employed to address the DFL problem, as formulated in eq. (6.16), with the following grounding: 1) the problem has a single stage, i.e., $T = 1$; 2) the virtual parameters correspond to y in eq. (6.16), representing the estimated optimization problem parameters; 3) the unfolded uncertain values are the ground-truth parameters, denoted as $x^{(k+1)} = y^*$; 4) The feasible set $\tilde{C}$ is fixed and equal to the feasible set C of the original problem; 5) the objective function $\tilde{f}(x, x^{(k+1)}, z^{(k)})$ is exactly $f(x, y^*, z^*)$; 6) the distribution P is approximated by using a training set.

This specific grounding of UNIFY preserves several key benefits provided by DFL techniques. Notably, since the ML model is trained to minimize the task loss similar to DFL, UNIFY enables improvements in solution quality compared to "predict, then optimize" approaches. This improvement is achieved by trading off computational cost at training time for faster inference. Analogously to DFL, UNIFY allows the adoption of simpler ML models that are faster to evaluate and easier to verify using formal methods. The approach has a few additional properties, thanks to its flexibility. In particular, support for non-linear cost functions and soft constraints is seldom found in the DFL literature, and the problem of including estimated parameters in the constraints is still largely open. Moreover UNIFY supports sequential decision-making problems whereas classical DFL approaches typically focus on single stage problems.

Not all the DFL approaches admit a UNIFY grounding, e.g. methods based on *surrogate losses* such as [167]. Techniques in this class have been shown to provide advantages in terms of convergence speed and solution quality, meaning that, in cases where they are applicable, relying on them might be preferable than using the RL-based UNIFY implementation that we will adopt later in this paper.

Constrained RL Constrained RL addresses the fundamental task of training a policy to maximize a reward function while adhering to a set of constraints. These constraints may involve safety considerations, natural laws, or limitations on resource availability. This setup is common in various real-world scenarios. For instance, a self-driving car aims to reach its destination quickly and cost-effectively without causing harm to other vehicles or pedestrians, and without violating traffic regulations. Similarly, a drone may strive to achieve maximum speed without causing damage to itself, and a robot might need to complete a task within a specified time frame and without depleting its battery.

The aforementioned examples represent just a few instances of constrained RL applications and numerous instances exist beyond robotics and automation. Thanks to its flexibility, UNIFY can solve the

constrained RL problem. Formally, the constrained RL problem can be formulated as follow:

$$\arg\min_{\theta\in\Theta} \mathbb{E}_{\tau\sim P}\left[\sum_{k=1}^{T}\gamma^k f(x^{(k)}, x^{(k+1)}, z^{(k)})\right] \tag{6.17}$$

$$\text{with: } z^{(k)} = \pi(x^{(k)};\theta) \tag{6.18}$$

$$\text{s.t.: } z^{(k)} \in C(x^{(k)}) \tag{6.19}$$

where τ is a trajectory sampled from a probability distribution P, f is the reward function, π is the RL agent, $z^{(k)}$ and $x^{(k)}$ are respectively the action and observation at timestep k, and $C(x^{(k)})$ is the set of constraints. The constrained RL problem is similar to eq. (6.11). However, while with UNIFY we can flexibly demand constraints satisfaction via the decomposition, in constrained RL *the agent directly outputs the decisions*. This connection allows us to easily ground constrained RL into our framework by setting $g = \pi$.

Many constrained RL approaches enforce constraint satisfaction via a projection step in the decision space. In other words, once a baseline policy has provided the decision vector, this is projected into the feasible space by minimizing the Euclidean distance. Such a step is often presented as a "safety layer" on top of a neural network policy [168]. This technique can guarantee constraint satisfaction in a scalable way.

The safety layer approach for constrained RL can be replicated in UNIFY by grounding the method as follows:

$$\arg\min_{\theta\in\Theta} \mathbb{E}_{\tau\sim P}\left[\sum_{k=1}^{T}\gamma^k f(x^{(k)}, x^{(k+1)}, z^{(k)})\right] \tag{6.20}$$

$$\text{with: } z^{(k)} = \arg\min_{z}\{\|z - y^{(k)}\|_2^2 \mid z \in C(x)\} \tag{6.21}$$

$$\text{and: } y^{(k)} = h(x^{(k)};\theta) \tag{6.22}$$

When tackling traditional DFL problems, the main design choices are: 1) the virtual parameter vector $y^{(k)}$ is in the same space as the decision vector; this is the case since in constrained RL the ML agent is still in charge of producing actual decisions. Then, 2) the cost function for the CO problem is a (squared) Euclidean distance or some other kind of metric; 3) the feasible set for the CO problem is the same as for the overall problem, i.e. $\tilde{C}(x) = C(x)$. Another strategy, investigated by [169], enforces constraints by adding a projection step after gradient updates; the projection adjusts the policy weights so that the decision vector becomes feasible, as in the projected gradient method [170]. This approach is more numerically stable, but also more computationally expensive, due to the large number of parameters typically present in many ML models. This class of approaches can be seen as approximately solving eq. (6.7).

UNIFY retains all the key properties of constrained RL: 1) decisions are feasible by construction w.r.t. constraints defined for single stages; 2) fast inference, since the training cost is payed only once; 3) there is no need for the agent to have access to detailed problem knowledge. At the same time, it provides additional benefits: when available, symbolic knowledge can be easily exploited and the projection can be task-specific rather than task-agnostic. Moreover, the ability to decouple virtual parameters from actual decisions makes it much easier to handle complex decision spaces (e.g. combinatorial ones): for example, the ML model might output expected preferences, which a CO problem might use to compute

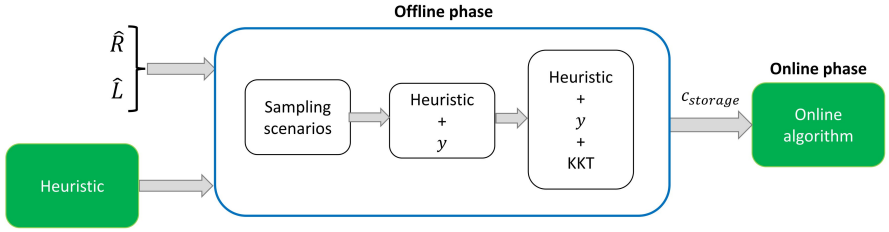


Figure 6.9: Schematic view of the TUNING algorithm.

a stable matching. Most techniques designed for RL can potentially be employed for UNIFY, either directly or with some adaptation. In our experiments we limit ourselves to the advantage actor-critic (A2C) algorithm, leaving this area open for investigation.

Integrated Offline/Online Optimization Many real-world problems consist of two distinct phases. During an offline phase, long-term "strategic" decisions are obtained via expensive but accurate approaches. Conversely, during a subsequent online phase, "operational" decisions are scheduled within strict time constraints and usually over multiple steps, requiring computational-efficient but often approximated methods. For example, in vehicle routing problems, we might plan the route in advance and then adjust it when new customers appear. In room allocation problems, we might revise an allocation solution whenever new rooms suddenly become available or unavailable.

While for many years these two phases have been addressed separately, recently there has been an increasing interest in a tighter integration between them [171]. In [172] the authors tackle the EMS problem described in Chapter 4. They use the LP from Equation (4.1) as a fast heuristic to handle online decisions, then compensate for the LP being myopic by: 1) introducing a virtual cost parameter for the storage system; and 2) adding an offline parameter tuning step, based on stochastic optimization.

The tuning process can be applied provided that uncertainty is exogenous and the online problem is convex. The method requires stating the KKT optimality conditions for the online model to obtain a set of constraints that characterize any solution that is compatible with the behavior of the online heuristic. Such constraints are incorporated in a mathematical program built using the SAA, which is then solved to obtain a schedule for the parameter over all online decision stages. The offline problem can be solved in an exact fashion: in this case, the parameter values are guaranteed to be optimal, within the limits of the sampling noise. The overall approach is referred to as TUNING and its behavior is schematically depicted in Figure 6.9.

The key behavior of approaches in this class can be replicated in UNIFY by grounding the framework as follows:

$$\arg\min_{y \in Y} \mathbb{E}_{\tau \sim P} \left[\sum_{k=1}^{T} \gamma^k f(x^{(k)}, x^{(k+1)}, z^{(k)}) \right] \tag{6.23}$$

$$\text{with: } z^{(k)} = \arg\min_{z} \{ \tilde{f}(x^{(k)}, y^{(k)}, z) \mid z \in \tilde{C}(x) \} \tag{6.24}$$

Through this formulation we are not able to fully exploit UNIFY potential. The CO problem here plays the role of the online heuristic. There is a single, major, restriction w.r.t. the general method: namely, there is no ML model and the optimization is performed directly over the virtual parameter vector y.

To fully exploit the UNIFY formulation, we can re-introduce the ML model. When configured in this fashion, UNIFY retains many of the key properties of the existing offline/online integration approaches. Moreover, we introduce additional benefits by making the method adaptable to characteristics of the current instance, or to information that unveils itself at online decision time (both captured via the observable vector $x^{(k)}$). Nonetheless, while TUNING assumes the online optimization problem is convex, UNIFY does not pose restriction.

Conversely to existing approaches like TUNING, the grounding of UNIFY for integrated offline/online optimization problems admits two alternative schemas depending on the output of the ML model h. The first solution frames the problem in the same way traditional hybrid offline/online optimization problems do: h simultaneously predicts the virtual parameters for all the stages, i.e., $y^{(1,\dots,k)} = h(x^{(1,\dots,k)})$. We refer to this approach as SINGLE-STEP. The second alternative involves predicting one virtual parameter at a time in a sequential fashion and leveraging the outcome of the uncertainty, i.e., $y^{(k)} = h(x^{(k)}, x^{+^{(k)}})$. This approach is referred to as SEQUENTIAL. While intuitively the SEQUENTIAL method should always be favored, in the experimental section, we will demonstrate an application on the EMS that shows this is not always the case.

Stochastic Optimization The area of stochastic optimization [9] also aims at *improving robustness in single- or multi-stage decision-making problems.* Most stochastic optimization algorithms rely on Monte Carlo methods to approximate expected values and assess constraint satisfaction. The SAA has long been a staple of the field; its convergence rate in the context of combinatorial problems was eventually provided in [13]. The SAA has been combined with Benders decomposition to address two-stage decision problems, leading to the family of L-shaped methods for stochastic optimization [173, 174]. Multi-stage decision problems have been often approximated as a sequence of 2-stage problems in so-called online anticipatory algorithms [175], and solved once again via L-shaped methods. The connection between multi-stage stochastic optimization and MDP is instead exploited directly by the method from [176].

The single-stage stochastic optimization problem can be formulated as:

$$\begin{aligned} &\arg\min_{y \in Y} \mathbb{E}_{x^+ \sim P}\left[f(x, x^+, z^*)\right] \\ &\text{with: } z^* = \arg\min_{z}\{f(x, y, z) \mid z \in C(x)\} \end{aligned} \tag{6.25}$$

We can thus ground single-stage stochastic optimization in UNIFY following these steps: 1) since there is a single decision stage, the summation from eq. (6.11) is absent and the notation x replaces $x^{(k)}$ and x^+ replaces $x^{(k+1)}$; 2) since the focus is on a single instance, there is no need to compute the expectation over x; 3) the virtual parameters can be assimilated to future values for the uncertain elements, so they can appear where x^+ is expected. 4) A ML model is absent and optimization is performed directly on y, i.e. we use a single, "virtual" scenario for decision-making; multiple samples are still used for evaluating the actual decision cost. Finally, 5) the cost function and constraints for the CO problem are the same as for the original problem, i.e. $\tilde{f}(x, y, z) = f(x, y, z)$ and $\tilde{C}(x, y) = C(x)$.

Two-stage stochastic problems expand on single-stage ones by introducing *recourse actions* that can be taken reactively once uncertainty is revealed. Unlike multi-stage problems, where all decisions are similar in nature (e.g., donor-patient matching, power flows, etc.) the recourse actions can be radically different from the first-stage decisions (e.g., buying products to satisfy unmet demand vs. deciding how many products to manufacture).

Two-stage stochastic optimization can be grounded in UNIFY by viewing the recourse actions as part of the decision variables, i.e. $z = (z_{\textit{first-stage}}, z_{\textit{recourse}})$. Then, z^* will specify values for both of the first-stage variables and the recourse ones; recourse variables in z^* will be ignored at deployment time since they have no immediate impact (as it is typically done in two-stage stochastic programming). The result is similar to the one already discussed: decision-making at deployment time is performed w.r.t. a single scenario, which is optimized at training time over a larger number of sampled scenarios. Using UNIFY rather than traditional stochastic optimization algorithms retains its key property, i.e. improved solution quality w.r.t. myopic approaches. Despite stochastic optimization approaches can provide very high-quality solutions for individual problem instances, provided that enough samples are used, as a major drawback, their scalability is limited, mostly as a result of sampling.

Conversely, UNIFY improves scalability by relying on a single "virtual" scenario and thus removing the main bottleneck that makes stochastic optimization more computationally expensive than its deterministic counterpart. Performing sampling only during cost evaluation also enables leveraging parallelization for a faster computation of the expected value. Scalability can be further improved by adding back the ML model so that there is no need to re-optimize y for every new problem instance. Moreover, UNIFY naturally supports endogenous uncertainty. As a side effect, using UNIFY may slightly overconstrain the decision process, since it can yield only solutions that can be defined based on a single scenario.

6.3.4 UNIFY: an Application to an EMS and a Production Scheduling Problem

In section 6.3, we formally introduced UNIFY from a theoretical standpoint. In this section, we will experimentally showcase the effectiveness of UNIFY in two use cases, extensively illustrated in chapter 4: a real-world EMS and a simplified production scheduling problem, namely the WSMC. The first part of the section is dedicated to illustrate the groundings for these two problem. Then, through our experimental evaluation, we will illustrate how to: 1) define the virtual parameters by either choosing existing problem parameters or introducing new ones and augmenting the solver with clairvoyant capabilities; 2) easily handle constraints in RL by delegating constraint satisfaction to a CO solver; 3) design an RL-based DFL approach; 4) improve scalability in stochastic optimization.

UNIFY grounding on the EMS In the EMS case, handling control of any of the existing parameters to the external component (e.g. l_i, u_i, η) might lead to the violation of a critical constraint. We can however introduce an ad-hoc parameter that allows the external component to alter the problem solution without affecting short-term feasibility. In particular, we will associate *a virtual cost* $y^{(k)}$ to the storage system, leading to the following modified LP:

$$\arg\min_{z^{(k)}} \; y^{(k)} z_1^{(k)} + \sum_{i=2}^{m} c_i^{(k)} z_i^{(k)} \tag{6.26}$$

$$\text{s.t.} \; \sum_{i=1}^{m} z_i^{(k)} = x_{\text{load}}^{(k)} \tag{6.27}$$

$$l_i \leq z_i^{(k)} \leq u_i \qquad \forall i = 1..m \tag{6.28}$$

$$0 \leq x_{\text{storage}}^{(k)} - \eta z_1^{(k)} \leq q \tag{6.29}$$

$$z_i^{(k)} \in \mathbb{R} \qquad \forall i = 1..m \tag{6.30}$$

Element	Notation	EMS Grounding
Decision stages	$\{k\}_{k=1}^{T}$	15-minute intervals over one day ($T = 96$)
Observables	$x^{(k)}$	energy level in the storage systems for stage k (i.e. $x^{(k)}_{storage}$), RES production and load for stage k (i.e. $x^{(k)}_{load}, x^{(k)}_{res}$), forecast for RES production and load for stage $k+1$ (i.e. $x^{(k)}_{loadf}, x^{(k)}_{resf}$)
Uncertainty	P	uncontrolled deviations from production and load forecast
Decisions	$z^{(k)}$	power flow from/to each generator and the storage system
Constraints	$C(x^{(k)})$	power balance of the energy system, upper/lower bounds for the power flows
Cost function	$f(x^{(k)}, x^{(k+1)}, z^{(k)})$	cost/profit of generating, buying, or selling energy for one stage

Table 6.3: EMS grounding for the main problem elements in the approach

Element	Notation	WSMC Grounding
Decisions stages	—	a single decision stage
Observables (current)	x	generic information correlated to future demands
Observables (future)	x^{+}	actual value of the customer demands
Uncertainty	P	the distribution for both x and x^{+}, in particular $P(x)$ and $P(x^{+} \mid x)$
Decisions	z	how many units to manufacture for each set of products (i.e. $z_{first-stage}$), estimated number of products to buy (i.e. $z_{recourse}$)
Constraints	$C(x)$	absent (demands can always be satisfied by buying products)
Cost function	$f(x^{+}, x, z)$	cost of both manufactured and bought items

Table 6.4: WSMC grounding for the main problem elements in the approach

Where we assume without loss of generality that $z_1^{(k)}$ refers to the flow to/from the storage unit. Now, by giving a negative value to $y^{(k)}$ it is possible to provide an incentive for the optimization model to fill the storage system, for example, to prepare for a forthcoming peak in the grid energy price. In other words, while the optimization problem is still largely unchanged (and still very easy to solve), it can now exhibit anticipatory behavior by adjusting the value of the virtual parameters.

We consider two versions of this problem. In both cases, the grounding of the UNIFY method is the one specified in Table 6.3. Moreover, the CO problem is defined via the LP in eqs. (6.26) and (6.30), i.e. the version where a virtual cost was associated with the storage system. The two versions differ in terms of how the ML model is used. In the first version, referred to as SEQUENTIAL, the ML model focuses on individual decision stages, so that it can take advantage of observed information as soon as it

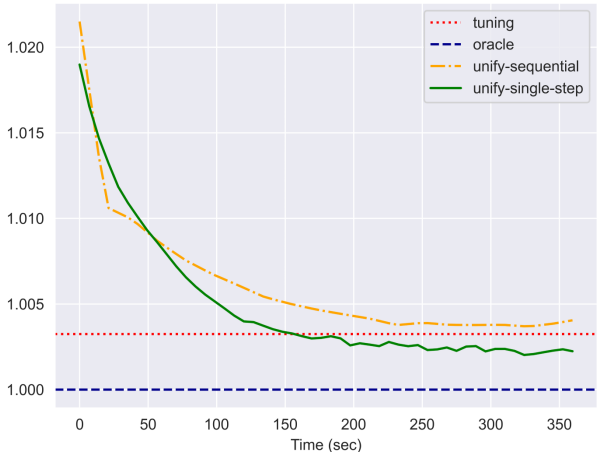


Figure 6.10: Optimality gap of the state-of-the-art TUNING approach and the UNIFY methods w.r.t. the computational time.

becomes available, including the storage energy level. The ML model output is the virtual cost for the storage system at the current stage, i.e. $y^{(k)}$.

In the second version, referred to as ALL-AT-ONCE, we assume a full schedule for the virtual costs needs to be provided one day in advance. In this case, the ML model input still includes forecasts (since they are also made available one day in advance), but it lacks information about the storage energy level. The ML model is only evaluated once, then power flow decisions are taken as usual in a sequential fashion. This setup is still practically meaningful, and it leads to a hybrid setting where the problem has a single decision stage when viewed from a RL perspective: in fact, according to fig. 6.8, once the ML model has provided the full virtual cost sequence $\{y^{(k)}\}_{k=1}^{T}$, the rest of the multi-stage process can be seen as part of the reward computation.

UNIFY **grounding on the WSMC** The grounding of UNIFY for the EMS is exhaustively described in table 6.4. For the virtual parameters we chose a simple yet effective design choice. We chose parameters that already appear in the optimization problem, i.e. the y vector specifies the value of the demand for the products that can be manufactured. However, it is crucial to consider them as "virtual" since they do not reflect ground-truth demands or any statistical value (e.g., mean or quantile).

Offline/Online Integration using UNIFY In this section, we demonstrate that UNIFY can replicate the *offline/online integration approach* based on the idea of tuning virtual parameters. Experiments are run on the EMS use case, where we employ UNIFY to tune the virtual cost $y^{(i)}$ of the optimization model described in eqs. (6.26) to (6.29). Intuitively, by associating a negative cost to storage we can provide an incentive for the CO problem to accumulate energy, in preparation for forthcoming spikes in the grid energy price.

We consider both the SEQUENTIAL and THE ALL-AT-ONCE versions of the problem described in the previous section. We use UNIFY to solve both versions of the problem and refer to the two approaches respectively as UNIFY-ALL-AT-ONCE and UNIFY-SEQUENTIAL. The two methods are equivalent to those we proposed in [177], which indeed can be considered an application-specific grounding of the UNIFY framework. As a baseline, we use the state-of-the-art TUNING approach from [172], which solves the

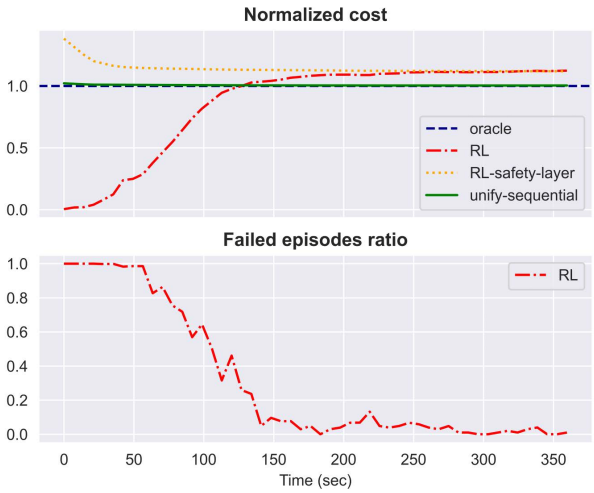


Figure 6.11: In this figure we show how demanding constraints satisfaction to the downstream solver greatly improves over a full end-to-end RL method and SAFETY-LAYER.

ALL-AT-ONCE version of the problem by relying on a mathematical program. Notably, this approach provably converges to the best possible non-clairvoyant solution, as the number of samples used to approximate uncertainty grows. However, scalability issues prevent its usage with a large number of samples. The method also requires the online decision problem to be convex, thus limiting its applicability.

We also compare the performance to that of a clairvoyant solution (referred to as ORACLE), to provide an optimistic reference for the solution quality. This approach is obtained by simply instantiating eqs. (6.26) to (6.29) for all decision stages, replacing all parameters with their actual realizations. Similarly to [177], we compare the methods ensuring they have access to the same computation time. Since the time execution of TUNING is constant, we chose this value as the time limit for training the other methods and plot its results as a horizontal line. In fig. 6.10, we show the optimality gap w.r.t. the computation time. As also highlighted in [177], exploiting the sequential nature of the problem does not provide clear benefits, possibly due to a suboptimal training solution, so that UNIFY-SEQUENTIAL yields slightly lower-quality solutions compared to UNIFY-ALL-AT-ONCE.

Both the UNIFY approaches performed remarkably well, with UNIFY-ALL-AT-ONCE beating the state-of-the-art TUNING method. Intuitively, the additional scalability provided by our framework enables collecting a much higher variety of samples, which was enough to compensate for the use of a suboptimal approach (RL) to tackle the training problem. Additionally, UNIFY *does not require convexity for the online problem*, making it more broadly applicable.

Constraints in RL In section 6.3.3 we showed how UNIFY can be used to deal with hard constraints and combinatorial decision spaces in RL. Similarly to safety-layer approaches, UNIFY handles constraints by applying a constrained optimization step on top of the output of a ML model. Unlike safety layer approaches, however, the ML model is not in charge of producing a decision vector, but rather of "piloting" the CO solver by adjusting the values of virtual parameters.

For this experiment, we rely again on the EMS benchmark. In this case, any feasible solution policy must satisfy hard constraints at each decision stage, i.e. the flow bounds and the power balance

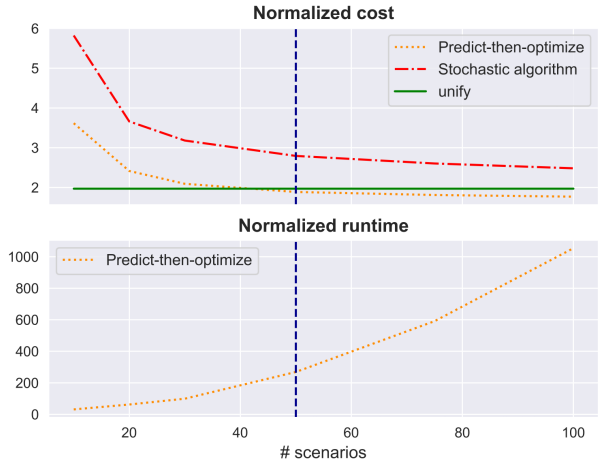


Figure 6.12: Optimality gap on the WSMC problem and the solution time of the predict-then-optimize approach w.r.t. the number of scenarios.

restrictions. We compare the UNIFY-SEQUENTIAL from Section 6.3.4 with two approaches from the literature, plus the clairvoyant oracle that serves as an optimistic reference. In particular, we train a full end-to-end DRL algorithm to provide a solution, by learning constraints satisfaction only from the reward signal (i.e. via reward shaping); we refer to the approach as RL. In this setup, designing the reward function is not trivial because it should provide a good trade-off between finding good solutions and exploring the feasible space. Projection-based DRL algorithms (e.g. safety layer) provide an alternative to full end-to-end methods when dealing with constraints: we have experimented with a safety layer implementation [168] for the EMS and we will refer to it as SAFETY-LAYER.

As shown in Section 6.3.3, both SAFETY-LAYER and UNIFY-SEQUENTIAL can be considered instances of our UNIFY framework, but they have a critical difference: in SAFETY-LAYER the projection step can fix infeasible decisions, but this is done in a cost-agnostic fashion and the RL agent still needs to output a meaningful decision vector. Conversely, in UNIFY-SEQUENTIAL the CO problem is capable of handling, at least partially, much of the problem elements, including the cost and constraints for a single stage. The ML model needs to guide such problem-specific solver by means of the virtual storage costs, which is arguably a simpler task.

In the upper and lower parts of fig. 6.11, we respectively show the optimality gap of all the methods and the number of failed episodes of RL due to constraints violations. In the early stages of training, RL never completes a full episode. It then progressively learns to satisfy constraints but, conversely, the cost of the solutions found increases. On the other hand, SAFETY-LAYER converges very quickly but the final solution cost is very close to the one provided by RL. UNIFY-SEQUENTIAL quickly converges as well, and it also *improves the previous methods by a significant margin.* These experimental results demonstrate that *RL can benefit from a policy decomposition that properly balances learning and optimization.*

Stochastic Optimization Solving stochastic optimization problems can be incredibly challenging. As mentioned in section 6.3.3, SAA methods are widely adopted in this field but they can be computationally expensive. In this section, we will show how UNIFY can be used to *improve the robustness of the downstream solver* by performing a set of experiments on the WSMC.

As a baseline approach to ensure robustness, we employ the SAA algorithm based on Monte Carlo sampling that relies on the following optimization model:

$$\min \sum_{j \in J} c_j z_j + \frac{1}{|\Omega|} \sum_{\omega \in \Omega} \sum_{i \in I} \rho s_{i,\omega} \tag{6.31}$$

$$\sum_{j \in J} a_{i,j} z_j \geq d_{i,\omega}(1 - w_{i,\omega}) \tag{6.32}$$

$$\forall i \in I, \omega \in \Omega$$

$$w_{i,\omega} = 1 \implies s_{i,\omega} \geq d_{i,\omega} - \sum_{j \in J} a_{i,j} x_j \tag{6.33}$$

$$\forall i \in I, \omega \in \Omega$$

$$z_j \geq 0 \tag{6.34}$$

$$w_{i,\omega} \in [0, 1] \tag{6.35}$$

$$s_{i,\omega} \geq 0 \tag{6.36}$$

$$z, w \in \mathbb{Z} \tag{6.37}$$

where $\omega \in \Omega$ are the sampled scenarios. If we increase Ω we also increase robustness but, at the same time, we drastically increase the computational complexity and thus reduce scalability.

More specifically, we compared three approaches: 1) *stochastic optimization*: SAA approach where scenarios are sampled directly from the training set. This method lacks contextual information and does not provide instance-dependent scenarios. 2) *predict-then-optimize*: SAA approach which relies on instance-specific samples by querying a Poisson probabilistic model trained for maximum likelihood estimation. This approach has an advantage over simple stochastic optimization as it utilizes the ML model, but its robustness improves with an increase in the number of sampled scenarios. 3) UNIFY *implementation*: the ML model is trained to minimize the cost and is inherently more robust without relying on the SAA. This intrinsic robustness allows it to achieve similar performance to predict-then-optimize with a large number of samples. It is worth highlighting that the comparison favors the predict-then-optimize method since it is designed by assuming exact knowledge of the type of probability distribution whereas the UNIFY implementation makes no such assumption.

Results are shown in fig. 6.12. In the upper part of the figure, we report the optimality gap of the three methods on a separate set of instances w.r.t. the number of sampled scenarios. Both the simple the stochastic algorithm and predict-then-optimize approaches benefit from increasing the number of scenarios. On the other side, the UNIFY implementation does not depend on it, because it directly predicts the demand values that are plugged into the optimization model. Despite the advantage previously discussed, the predict-then-optimize approach surpasses UNIFY only when at least ~ 50 scenarios are used in the downstream stochastic optimization model. In the lower part of the figure, we show the runtime required by predict-then-optimize as a multiple of the runtime of UNIFY, w.r.t. to the number of scenarios. As we can see, to obtain better results, predict-then-optimize requires more than 200 times the computation of UNIFY. We can thus conclude that a *smart implementation of* UNIFY *is a cheaper alternative to a SAA method for improving the robustness of the solver when tackling stochastic optimization problems.*

Discussion We designed the experimental analysis of UNIFY to clarify its versatility, despite its core idea is remarkably simple. More importantly, we would like to enlight how the framework *blurs the line between approaches that have been investigated mostly in isolation*, thus highlighting opportunities for cross-fertilization. For example, DFL and RL share a few key challenges (differentiating black-box or piecewise constant functions), suggesting that many ideas developed for one of the two fields could be adapted to the other. Similarly, drawing ties from ML to offline/online integration and stochastic optimization could open the way for more scalable approaches, while retaining the key advantages of such techniques (e.g. convergence and feasibility guarantees).

Finally, prior knowledge plays a crucial role in UNIFY when designing the virtual parameters. On the one hand, this is a non-trivial challenge for the method designer, since it requires an understanding of both ML and constrained optimization methods. On the other, however, choosing the semantics and size for y offers a rare opportunity to configure which aspects of the original problem are delegated to the ML model and which ones to the CO problem.

Making y more similar to a vector of decisions makes the approach closer to RL, and it might be better suited for use cases where declarative models are hard to craft. Using just a few key parameters in y (as we did in our EMS example) goes in the opposite direction, and allows one to capitalize on explicit problem knowledge in cases where this is available.

Overall, *choosing the virtual parameters can be thought of as a design handle that enables "partitioning" the complexity of the original problem* into either a ML or CO module. By managing this decision, it is possible to make sure that each module is used according to its strengths, at the same time compensating for known limitations.

6.4 Score Function Gradient Estimation to Widen the Applicability of DFL

The last methodological contribution of this thesis is a specific instance of the UNIFY in the context of DFL. The main challenge of DFL is the non-informative gradients of the task loss when the downstream optimization problem is combinatorial. State-of-the-art DFL methods overcome this, but are limited by the assumptions they make about the structure of the problem (e.g., that the problem is linear) or by the fact that they can only predict parameters that appear in the objective function.

During my research work, we addressed these limitations by predicting *distributions* over parameters in order to smooth the loss, and adopting score function gradient estimation (SFGE) to estimate the gradients and compute decision-focused updates to the predictive model. Our experiments show that by using SFGE we can deal with problems with linear or nonlinear objectives, with or without integrality constraints, and with unknown parameters that occur in the objective function, in the constraints, or in both. On the other hand, we demonstrated that DFL methods, when cleverly *leveraging the knowledge of the optimization problem*, typically yield superior results compared to more general approaches like SFGE.

Deriving the Score Function Gradient Estimator for DFL The central challenge in DFL is that when the task loss $\mathcal{L}$ depends on the outcome $z^\star$ of a combinatorial optimization procedure, it has zero-valued gradients with respect to the predictive model's parameters almost everywhere. Gradient-based

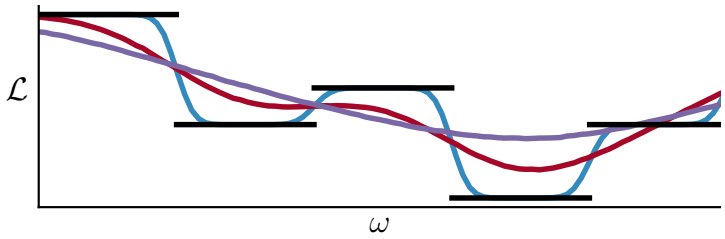


Figure 6.13: Illustration of a DFL loss with non-informative derivatives (■) smoothed by predicting a Gaussian over the parameters with increasing variances (■ ≤ ■ ≤ ■). The larger the variance, the more the loss gets smoothed, but the less it resembles the original piecewise-constant task loss.

learning is thus uneffective. This can be seen when applying the chain rule:

$$\frac{\partial \mathcal{L}(z^\star(\hat{y}), y)}{\partial \omega} = \frac{\partial \mathcal{L}(z^\star(\hat{y}), y)}{\partial z^\star(\hat{y})} \frac{\partial z^\star(\hat{y})}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial \omega} \tag{6.38}$$

The second factor, $\frac{\partial z^\star(\hat{y})}{\partial \hat{y}}$, measures the change in $z^\star(\hat{y})$ when $\hat{y}$ changes infinitesimally. However, since the problem is combinatorial, this change is zero almost everywhere. This in turn causes the entire gradient $\frac{\partial \mathcal{L}(z^\star(\hat{y}), y)}{\partial \omega}$ to be zero almost everywhere.

To tackle this issue, we shift from training a model that makes point predictions $\hat{y}$, to a model that predicts a vector θ that instantiates a distribution $p_\theta(y)$. In other words, instead of predicting parameter vectors, the model predicts *distributions* over parameter vectors. For instance, we may consider $\hat{y}$ to be sampled from a multivariate Gaussian distribution parameterized by its means μ and standard deviations σ, which the predictive model is trained to predict, i.e., $\theta = (\mu, \sigma)$.

By predicting the parameters θ of a distribution, the loss becomes an expectation:

$$L(\theta, y) = \mathbb{E}_{\hat{y} \sim p_\theta(y)}[\mathcal{L}(z^\star(\hat{y}), y)] \tag{6.39}$$

The motivation for this is that it removes the zero-gradient problem: by predicting distributions, the gradient of the loss with respect to the output of the predictive model is not zero anymore. This is illustrated in Figure 6.13, which shows how predicting a Gaussian distribution over parameters provides a smooth proxy for the original piecewise-constant loss. However, although the resulting gradient is not zero anymore, computing it is not trivial.

To train the predictive model, an estimate of $\frac{\partial L(\theta, y)}{\partial \theta}$ is needed. To compute such an estimate, we SFGE (also known as the REINFORCE algorithm in the context of RL) [178]. Consider the following derivation:

$$\begin{aligned}
\nabla_\theta L(\theta, y) &= \nabla_\theta \mathbb{E}_{\hat{y} \sim p_\theta(y)}[\mathcal{L}(z^\star(\hat{y}), y)] && \text{(6.40a)} \\
&= \nabla_\theta \int p_\theta(y) \mathcal{L}(z^\star(\hat{y}), y) d\hat{y} && \text{(6.40b)} \\
&= \int \mathcal{L}(z^\star(\hat{y}), y) \nabla_\theta p_\theta(\hat{y}) d\hat{y} && \text{(6.40c)} \\
&= \int \mathcal{L}(z^\star(\hat{y}), y) p_\theta(\hat{y}) \nabla_\theta \log p_\theta(\hat{y}) d\hat{y} && \text{(6.40d)} \\
&= \mathbb{E}_{\hat{y} \sim p_\theta(y)}[\mathcal{L}(z^\star(\hat{y}), y) \nabla_\theta \log p_\theta(\hat{y})] && \text{(6.40e)}
\end{aligned}$$

In (6.40d), the log derivative trick is used. The validity of the interchange between integral and differentiation is discussed in appendix A.

In the following, we prove the validity of bringing the gradient inward in (6.40c).

To get an estimation of this gradient that can effectively be used in training, the final gradient in (6.40e) can be estimated using a Monte Carlo method, giving

$$\nabla_\theta L(\theta, y) \approx \frac{1}{S} \sum_{i=1}^{S} \mathcal{L}(z^\star(\hat{y}^{(i)}), y) \nabla_\theta \log p_\theta(\hat{y}^{(i)}) \quad \text{with } \hat{y}^{(i)} \sim p_\theta(y) \tag{6.41}$$

with S as the total number of samples.

What sets this approach apart from existing DFL methods is its broad applicability within DFL, driven by the fact that Eq. (6.41) assumes nothing about the optimization problem's form, the location of the predicted parameters, or the choice of task loss $\mathcal{L}$. Furthermore, we consider problems where the *uncertain parameters appear only in the objective, or only in the constraints, or in both.* The prediction of distributions is introduced solely for the purpose of obtaining informative non-zero gradients. At inference time, we still want to obtain point predictions from the model to feed into the optimization problem. Therefore, at test time, we take the mean of the predicted distribution as parameters that we feed into the optimization problem.

6.4.1 SFGE applications to linear and non-linear optimization problems

To demonstrate the generality and wide applicability of our approach, we conducted the experimental analysis by focusing on three main research questions:

Q1) *How does* SFGE *compare with prediction-focused learning (PFL) and state-of-the-art DFL approaches when predicting parameters only appearing the objective function?*

Q2) *How does* SFGE *compare with PFL and state-of-the-art DFL approaches when predicting parameters that appear in the objective function and constraints?*

Q3) *How does* SFGE *fare against a PFL method that solves the problem as a two-stage stochastic optimization problem at inference time?*

In our experimental evaluations, when utilizing SFGE, we employ a linear regression model to predict the mean of a Gaussian distribution, from which we draw one sample of $\hat{y}$ per gradient estimation (i.e., $S = 1$), which we found to work best in practice. The standard deviation of the distribution is trainable but non-contextual (i.e., remains independent of the input features). Although we could train a regression model to predict the standard deviation as well, in practice this does not provide additional benefits.

Q1: Objective function parameters We start with the task of predicting parameters that appear linearly in the objective function, since this is the setting most commonly considered in existing works. More concretely, we use the 0-1 knapsack problem (KP) with 50 and 75 items, and the 0-1 quadratic KP with 8 and 10 items, both with all item values unknown.

We generate synthetic data by introducing a mapping between input features and targets in the same way as described in the shortest path experimental evaluation of Elmachtoub and Grigas [167],

Table 6.5: PFL, SFGE and SPO results on the linear and quadratic KP.

Method	*Rel. regret*	*MSE*	*Epochs*
	KP-50		
PFL	0.023 ± 0.006	$\mathbf{2.88 \cdot 10^4 \pm 1.84 \cdot 10^4}$	50.6 ± 42.0
SFGE	0.008 ± 0.001	$1.28 \cdot 10^5 \pm 5.04 \cdot 10^4$	80.7 ± 10.9
SFGE (contextual std.dev)	0.007 ± 0.001	$1.19 \cdot 10^5 \pm 3.34 \cdot 10^4$	91.4 ± 17.4
SPO	$\mathbf{0.004 \pm 0.001}$	$4.71 \cdot 10^4 \pm 2.24 \cdot 10^4$	$\mathbf{41.7 \pm 11.6}$
	KP-75		
PFL	0.024 ± 0.004	$\mathbf{2.87 \cdot 10^4 \pm 1.64 \cdot 10^4}$	$\mathbf{48.1 \pm 69.3}$
SFGE	0.008 ± 0.001	$1.26 \cdot 10^5 \pm 4.22 \cdot 10^4$	103.8 ± 14.7
SPO	$\mathbf{0.004 \pm 0.001}$	$5.15 \cdot 10^4 \pm 2.42 \cdot 10^4$	54.3 ± 15.4
	Quadratic KP-8		
PFL	0.034 ± 0.015	$\mathbf{2.35 \cdot 10^4 \pm 1.58 \cdot 10^4}$	$\mathbf{24.3 \pm 4.61}$
SFGE	0.006 ± 0.003	$1.06 \cdot 10^5 \pm 6.45 \cdot 10^4$	54.5 ± 14.9
SPO	$\mathbf{0.005 \pm 0.002}$	$6.95 \cdot 10^4 \pm 4.09 \cdot 10^4$	29.9 ± 10.4
	Quadratic KP-10		
PFL	0.041 ± 0.011	$\mathbf{2.37 \cdot 10^4 \pm 1.50 \cdot 10^4}$	45.8 ± 64.0
SFGE	0.008 ± 0.002	$8.46 \cdot 10^4 \pm 4.04 \cdot 10^4$	54.1 ± 13.1
SPO	$\mathbf{0.006 \pm 0.002}$	$6.47 \cdot 10^4 \pm 3.26 \cdot 10^4$	$\mathbf{30.7 \pm 8.6}$

with a degree of model misspecification $deg = 5$, number of input features $p = 5$, and a noise half-width $\bar{\epsilon} = 0.5$. In this setup, Smart Predict-then-optimize (SPO) [167] provides state-of-the-art results and will be used as reference DFL method for comparison. As a baseline, we employ a PFL model trained to minimize the MSE between the predictions and the ground-truth values.

We generate 5 different datasets and for each consider 3 different splits among training, validation and test sets with proportions of respectively 80%, 10% and 10%. All the methods are used to train a linear regression model with stochastic gradient descent, Adam as optimizer, a learning rate of 0.005 and a batch size of 32 samples. The training is stopped when the validation regret (for SPO and SFGE) or the validation MSE (for PFL) has not improved for 10 epochs.

The aggregated results are reported in table 6.5. In terms of MSE, the PFL method is the most accurate, followed by SPO. With respect to relative regret, although SPO performs best, SFGE *is able to outperform PFL*. In terms of convergence speed, SPO and PFL require a comparable number of epochs whereas SFGE is slower by a non-negligible gap. We conclude that when the uncertainty occurs solely in the objective, SFGE significantly outperforms the PFL method, but is still bested by the state-of-the-art SPO method.

Q2: Constraint parameters We now direct our attention to the task of predicting parameters in the constraints, a challenging problem that is not easily addressed by existing DFL methods.

To the best of our knowledge, Hu et al. [143] is the only method specifically designed to train a predictive model via gradient descent to predict constraint parameters of linear packing and covering problems within a DFL framework. Consequently, we compare it (which we refer to as P+O) with our SFGE method on the fractional KP with 10 items, which is also used in their experimental evaluation. In this benchmark, both the item values and the item weights are unknown and must be predicted. We choose the same correction and penalty functions as in Hu et al. [143]: when the solution instantiated

Table 6.6: PFL, SFGE and PO results on the fractional KP. We did not report the *Feas. rel. PRegret* for very high *Infeas. ratio*.

Method	*Rel. PRegret*	*Feas. rel. PRegret*	*Infeas. ratio*	*MSE*	*Epochs*
capacity=50, $\rho = 0$					
PFL	0.403 ± 0.015	$\mathbf{0.107 \pm 0.064}$	$\mathbf{0.72 \pm 0.15}$	$\mathbf{99.1 \pm 13.1}$	13.3 ± 2.9
P+O	$\mathbf{0.377 \pm 0.130}$	–	1.00 ± 0.0	$9.8 \cdot 10^5 \pm 1.2 \cdot 10^4$	$\mathbf{2.5 \pm 1.9}$
SFGE (ours)	0.385 ± 0.008	–	1.00 ± 0.0	$8.2 \cdot 10^5 \pm 6.8 \cdot 10^5$	13.4 ± 3.7
capacity=50, $\rho = 1$					
PFL	0.501 ± 0.033	$\mathbf{0.107 \pm 0.064}$	0.72 ± 0.15	$\mathbf{99.1 \pm 13.1}$	13.3 ± 2.9
P+O	$\mathbf{0.460 \pm 0.162}$	0.380 ± 0.018	0.61 ± 0.02	$3.8 \cdot 10^5 \pm 4.8 \cdot 10^3$	$\mathbf{2.1 \pm 1.9}$
SFGE (ours)	0.467 ± 0.016	0.177 ± 0.045	$\mathbf{0.55 \pm 0.10}$	$7.9 \cdot 10^5 \pm 5.1 \cdot 10^5$	14.9 ± 4.7
capacity=50, $\rho = 2$					
PFL	0.600 ± 0.077	$\mathbf{0.107 \pm 0.064}$	0.72 ± 0.15	$\mathbf{99.1 \pm 13.1}$	13.3 ± 2.9
P+O	$\mathbf{0.492 \pm 0.173}$	0.422 ± 0.009	$\mathbf{0.42 \pm 0.05}$	$3.5 \cdot 10^5 \pm 3.8 \cdot 10^3$	$\mathbf{1.6 \pm 0.6}$
SFGE (ours)	0.512 ± 0.036	0.237 ± 0.092	0.46 ± 0.18	$1.3 \cdot 10^6 \pm 9.3 \cdot 10^5$	16.8 ± 4.3
capacity=75, $\rho = 0$					
PFL	0.353 ± 0.014	$\mathbf{0.096 \pm 0.058}$	$\mathbf{0.72 \pm 0.15}$	$\mathbf{99.1 \pm 13.1}$	13.3 ± 2.9
SFGE	0.337 ± 0.009	–	0.99 ± 0.01	$6.20 \cdot 10^5 \pm 6.06 \cdot 10^5$	13.4 ± 4.1
P+O	0.332 ± 0.109	–	1.0 ± 0.0	$9.8 \cdot 10^5 \pm 1.1 \cdot 10^4$	$\mathbf{2.4 \pm 1.4}$
capacity=75, $\rho = 1$					
PFL	0.437 ± 0.023	$\mathbf{0.096 \pm 0.058}$	0.72 ± 0.15	$\mathbf{99.1 \pm 13.1}$	13.3 ± 2.87
SFGE	0.410 ± 0.010	0.172 ± 0.044	$\mathbf{0.52 \pm 0.09}$	$9.41 \cdot 10^5 \pm 5.40 \cdot 10^5$	16.3 ± 3.8
P+O	0.405 ± 0.145	0.332 ± 0.013	0.617 ± 0.035	$3.8 \cdot 10^5 \pm 4.5 \cdot 10^3$	$\mathbf{1.8 \pm 1.5}$
capacity=75, $\rho = 2$					
PFL	0.522 ± 0.057	$\mathbf{0.096 \pm 0.058}$	0.72 ± 0.15	$\mathbf{99.1 \pm 13.1}$	13.3 ± 2.87
SFGE	0.436 ± 0.010	0.230 ± 0.081	$\mathbf{0.40 \pm 0.16}$	$2.1 \cdot 10^6 \pm 1.53 \cdot 10^6$	20.8 ± 7.8
P+O	0.426 ± 0.149	0.378 ± 0.009	0.428 ± 0.025	$3.5 \cdot 10^5 \pm 4.0 \cdot 10^3$	$\mathbf{3.2 \pm 2.0}$

by the prediction exceeds the capacity, items are proportionally removed (i.e., the selected knapsack is scaled down) until the capacity constraint is satisfied. If the discarded amount of item i is Δ_i, then the penalty for removing it is $\rho v_i \Delta_i$, where v_i is the item's value. We consider problem configurations with a capacity of 50 and 75, and various penalty coefficients $\rho = 0, 1, 2$. To assess the performance of DFL methods with highly misspecified models, we train a linear model to predict both the item weights and costs of the fractional KP. We conduct experiments on 10 different training-validation-test splits, using the same proportions as described in the previous section, along with the same hyperparameters.

The results are presented in Table 6.6. For each method, we report the relative post-hoc regret (*Rel. PRegret*), the relative regret of solutions that do not require the correction action (*Feas. rel. regret*), the ratio of solutions that require a correction action (*Infeas. ratio*), the MSE, and the number of epochs before training is stopped. SFGE outperforms PFL in terms of relative post-hoc regret, while P+O provides the best average performance but with a higher standard deviation; *we can thus conclude that* SFGE *provides a better worst-case relative post-hoc regret* for this instance of the fractional KP. With increasing ρ, the DFL methods become more conservative: the infeasibility ratio decreases, but at the cost of a worse relative regret on the feasible solutions. Regarding convergence speed, P+O is the fastest, whereas PFL and SFGE are slower and require a comparable number of epochs. As expected, in terms

Table 6.7: MLE and SFGE results on the KP-50 with uncertain weights. We did not report the *Feas. rel. PRegret* for very high *Infeas. ratio*.

Method	*Rel. PRegret*	*Feas. rel. PRegret*	*Infeas. ratio*	*MSE*	*Epochs*
50-items, $\rho = 5$					
PFL	0.168 ± 0.036	0.001 ± 0.001	0.93 ± 0.03	$\mathbf{7.88 \cdot 10^4 \pm 4.04 \cdot 10^4}$	$\mathbf{34.0 \pm 20.3}$
CombOptNet	0.189 ± 0.068	0.008 ± 0.005	0.91 ± 0.02	$5.26 \cdot 10^7 \pm 2.26 \cdot 10^7$	41.0 ± 13.4
SFGE(ours)	$\mathbf{0.126 \pm 0.015}$	-	0.98 ± 0.02	$3.62 \cdot 10^5 \pm 5.34 \cdot 10^4$	136.0 ± 10.8
50-items, $\rho = 10$					
PFL	0.319 ± 0.081	0.001 ± 0.001	0.93 ± 0.03	$\mathbf{7.88 \cdot 10^4 \pm 4.04 \cdot 10^4}$	$\mathbf{34.0 \pm 20.3}$
CombOptNet	0.400 ± 0.146	0.008 ± 0.005	0.91 ± 0.02	$5.26 \cdot 10^7 \pm 2.26 \cdot 10^7$	41.0 ± 13.4
SFGE(ours)	$\mathbf{0.178 \pm 0.019}$	-	0.99 ± 0.01	$3.71 \cdot 10^5 \pm 6.74 \cdot 10^4$	128.8 ± 19.2
50-items, $\rho = 20$					
PFL	0.615 ± 0.174	0.001 ± 0.001	0.93 ± 0.03	$\mathbf{7.88 \cdot 10^4 \pm 4.04 \cdot 10^4}$	$\mathbf{34.0 \pm 20.3}$
CombOptNet	0.822 ± 0.302	0.008 ± 0.005	0.91 ± 0.02	$5.26 \cdot 10^7 \pm 2.26 \cdot 10^7$	41.0 ± 13.4
SFGE(ours)	$\mathbf{0.212 \pm 0.022}$	-	0.99 ± 0.01	$3.73 \cdot 10^5 \pm 6.53 \cdot 10^4$	119.3 ± 26.1

Table 6.8: PFL and SFGE results on the WSMC of different sizes and for different penalty coefficient values. We did not report the *Feas. rel. PRegret* for very high *Infeas. ratio*.

Method	*Rel. PRegret*	*Feas. rel. PRegret*	*Infeas. ratio*	*MSE*	*Epochs*
			10×50, $\rho = 1$		
PFL	2.35 ± 0.85	–	0.98 ± 0.01	$\mathbf{1.78 \cdot 10^5 \pm 3.03 \cdot 10^4}$	$\mathbf{35.3 \pm 44.2}$
CombOptNet	3.04 ± 1.20	–	1.0 ± 0.0	$8.09 \cdot 10^6 \pm 5.24 \cdot 10^6$	49.9 ± 29.5
SFGE (ours)	$\mathbf{1.93 \pm 0.50}$	–	$\mathbf{0.94 \pm 0.05}$	$3.60 \cdot 10^5 \pm 5.88 \cdot 10^4$	70.3 ± 13.1
			10×50, $\rho = 5$		
PFL	12.20 ± 4.73	$\mathbf{0.034 \pm 0.019}$	0.96 ± 0.01	$\mathbf{2.01 \cdot 10^5 \pm 3.55 \cdot 10^4}$	61.5 ± 82.4
CombOptNet	88.80 ± 34.3	–	1.0 ± 0.0	$6.34 \cdot 10^6 \pm 2.64 \cdot 10^6$	$\mathbf{45.8 \pm 13.9}$
SFGE (ours)	$\mathbf{4.86 \pm 1.15}$	0.665 ± 0.315	$\mathbf{0.65 \pm 0.08}$	$5.54 \cdot 10^5 \pm 1.25 \cdot 10^5$	81.4 ± 16.9
			10×50, $\rho = 10$		
PFL	22.40 ± 7.90	$\mathbf{0.027 \pm 0.041}$	0.98 ± 0.01	$\mathbf{2.18 \cdot 10^5 \pm 7.40 \cdot 10^4}$	72.3 ± 97.5
CombOptNet	374.17 ± 79.0	–	1.0 ± 0.0	$9.10 \cdot 10^6 \pm 3.95 \cdot 10^6$	$\mathbf{34.0 \pm 15.7}$
SFGE (ours)	$\mathbf{7.08 \pm 1.29}$	1.30 ± 0.53	$\mathbf{0.54 \pm 0.14}$	$7.39 \cdot 10^5 \pm 2.21 \cdot 10^5$	67.7 ± 14.4

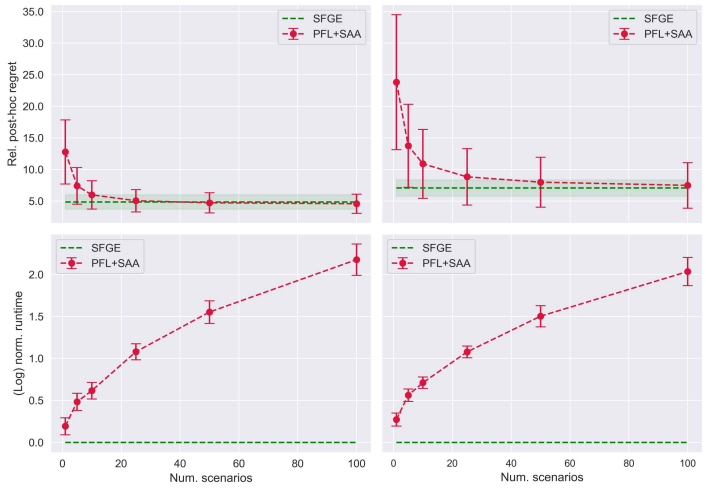


Figure 6.14: The relative post-hoc regret and normalized runtime at inference time of SFGE and PFL+SAA on the WSMC of size 10×50, for a $\rho = 5$ (left) and $\rho = 10$ (right).

of MSE, PFL delivers the best performance, whereas SFGE and P+O perform worse and show similar results. This makes sense, since SFGE and P+O train the model in a DFL fashion, rather than with the goal of maximizing accuracy.

While P+O is limited to linear packing and covering problems, many real-world combinatorial optimization problems involve integrality constraints and can be framed as ILP problems. Our SFGE method makes no assumptions about the optimization problem's structure, allowing it to be applied to ILP problems without modification. To the best of our knowledge, the only method that allows training a neural model in a DFL fashion to predict the constraint parameters of an ILP problem is CombOptNet [137].

To evaluate SFGE's performance when predicting parameters of constraints in an ILP problem, we considered two problem setups: the KP with unknown item weights and WSMC with unknown coverage requirements. To mimic a difficult-to-learn setting, we modeled the ground-truth relation between features and targets stochastically. More concretely, the ground-truth targets are sampled from a distribution whose parameters depend deterministically on the features. We assumed a Poisson distribution for both the item weights and coverage requirements. The number of input features and the degree of misspecification are the same as in the previous section. Since the ground-truth relation between features and problem parameters is now modeled stochastically, for the PFL method we employ the same probabilistic model used for SFGE: a Gaussian distribution whose mean is parameterized by a linear regression model and whose standard deviation is trainable but non-contextual (i.e., independent from the input features). This model is trained to maximize the likelihood (by minimizing the negative log-likelihood) of the ground-truth parameters and not to minimize the task loss. The optimization problem model and the recourse actions are the ones described in chapter 4.

We run experiments on the KP-50 and with $\rho \in \{5, 10, 20\}$. For the WSMC, the availability matrices were generated following a set of guidelines by Grossman and Wool [179] that lead to realistic instances. The set costs are generated uniformly at random from the range $[1, 100]$.

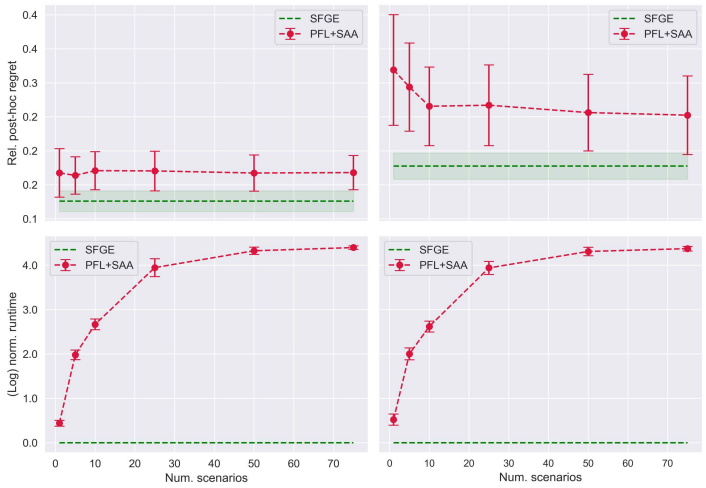


Figure 6.15: Comparison between SFGE and PFL+SAA on the KP-50 with stochastic item weights, for $\rho = 5$ (left) and $\rho = 10$ (right).

The results are presented in table 6.7 and table 6.8. SFGE *significantly outperforms the other methods in terms of relative post-hoc regret.* In WSMC, SFGE tends to be conservative, resulting in a higher relative regret for solutions that do not require the recourse action, especially with higher ρ values. Conversely, in the case of the KP, even though SFGE consistently achieves better post-hoc regret, it frequently resorts to recourse actions. One possible explanation for this phenomenon is that the option of adding items during the second stage always remains available, albeit at a reduced value. PFL is more accurate, though with a higher regret, and converges faster than the other methods, while CombOptNet is the least accurate. Similarly to the previous experiments, SFGE has a slower convergence speed, requiring a larger number of epochs.

Q3: PFL with stochastic optimization The ILP problems tackled in the last section involve stochasticity in the ground-truth relation from features to problem parameters. The PFL model is trained to map the true distribution and can be further leveraged by performing SAA *at inference time* [13]. This involves collecting a set of instance-specific samples, which are subsequently used as scenarios in the SAA algorithm to compute the optimal solution $z^\star$. This ideally improved solution is then employed to calculate the post-hoc regret. We refer to this pipeline as PFL+SAA. In contrast, as discussed in section 6.4, SFGE relies on stochasticity primarily to smooth the regret *but does not model the underlying distribution.* Simultaneously, it inherently minimizes the expected value of perfect information while only requiring a *single sample* during inference. Consequently, we compare these two methods to explore the considerable *scalability advantages* of SFGE.

In figs. 6.14 and 6.15, we present the relative post-hoc regret (top row) and the $\log_{10}$ normalized runtime (bottom row) during as functions of the number of sampled scenarios on the same ILP benchmarks as in the previous section. While for the WSMC we were able to solve the optimization problem (with scenarios) to optimality, in the case of the KP, we imposed a time limit of 30 seconds. This choice was necessitated by the higher computational demands of the KP, primarily stemming from the significant number of second-stage decision variables. The corresponding values for SFGE are drawn as horizontal lines since they do not require sampling.

As observed, with an increase in the number of scenarios, PFL+SAA generally improves on PFL in terms of relative post-hoc regret, but requires higher computation time. Because PFL learns to predict a distribution (a Gaussian) that is different from the true one (a Poisson), for high ρ values, *PFL+SAA struggles to catch up to* SFGE: even when 100 and 75 samples are collected for respectively the KP with unknown item weights and the WSMC, PFL+SAA does not surpass the performance of SFGE.

Discussion These experimental results showed that, when predicting parameters that appear in the objective function, SFGE is not able to outperform the DFL state-of-the-art that cleverly leverage the available knowledge, but still provides a major improvement over PFL approaches. On the other hand, on ILPs with uncertainty in the constraints, SFGE demonstrated superior performance in terms of both post-hoc regret and infeasibility ratio. Moreover, SFGE can be employed to dramatically reduce the computation time required to obtain robust solution for stochastic optimization problems. However, we did observe that it is slower in convergence speed in comparison. This issue was not unexpected since SFGE is known to suffer from the problem of high variance. Standardization plays a crucial role in addressing this issue. However, as part of future work, we plan to enhance our approach by incorporating variance reduction techniques from the existing literature, e.g. by learning critic for the post-hoc regret. Furthermore, we intend to expand our analysis to encompass problems where predictions appear non-linearly in the objective function, such as those involving trigonometric, polynomial, or exponential functions.

Chapter 7

Conclusions

This thesis presents novel informed ML methodologies, showcasing how various forms of knowledge can be integrated into different stages of the ML pipeline, with a focus on enhancing predictive models and decision support systems.

In the realm of predictive models, we propose a versatile mathematical framework to leverage prior knowledge in the form of external black-box models. This approach, characterized by its generality, allows for the integration of different types of black-boxes (e.g. general mathematical functions, other ML models) at various pipeline stages, including augmenting training data, designing custom neural architectures, and creating dedicated learning algorithms. The effectiveness of this methodology is demonstrated through a predictive maintenance task in an oil and gas facility. While this research currently focuses on a single application, future studies could explore its applicability to a broader range of real-world scenarios.

We also delve into the UDE for data-driven discovery of ordinary differential equations. While UDE aids in improving predictive accuracy, we identify its data-driven component as a potential barrier to achieving fully interpretable physics parameters approximation. Our analysis offers valuable insights for mitigating and controlling this undesirable behavior, exemplified by an experimental study on an RC circuit system often used to model thermal components.

The second part of the thesis shifts focus to informed ML techniques for enhancing decision support systems. We introduce a method that distills constraints propagators into the weights of a neural network, proving particularly beneficial when knowledge acquisition is costly and feasible only during training, not inference. This technique is validated using the PLS completion problem and controlled experiments assessing the impact of progressively integrating knowledge and data.

Furthermore, we propose a more structured approach to incorporate knowledge into ML models for decision support systems. The UNIFY framework is designed as a unified solution integrating ML with combinatorial optimization techniques, where knowledge is encapsulated in the declarative problem formulation. This framework cleverly decomposes the problem, partitioning the complexity and harnessing the strengths of both ML and combinatorial optimization. It effectively utilizes ML for predicting unknown parameters and managing uncertainty, while the optimization solver adeptly handles combinatorial input spaces and enforces constraints. An extensive evaluation on a real-world Energy Management System and abstract combinatorial optimization problems demonstrates UNIFY's versatility and effectiveness. While it outperforms existing methods in some cases, it also introduces additional capabilities in others.

Additionally, we develop a specific instance of UNIFY, namely score function gradient estimation to widen the applicability of decision-focused learning. This variant opens new avenues for applying decision-focused learning to challenging setups, such as predicting parameters appearing in constraints and requiring recourse actions for recovering from feasibility, and stochastic optimization.

In conclusion, this thesis contributes significantly to the integration of symbolic knowledge into ML algorithms for predictive models and decision support systems. Through extensive experimental analysis on both synthetic and real-world problems, the effectiveness of the proposed methodologies is substantiated. We highlight how informed ML offers versatile techniques for injecting knowledge in various forms and phases of an ML workflow. While enhancing accuracy, symbolic knowledge also improves interpretability, although sometimes requiring careful practices, especially in physics-informed ML. This work underscores the importance of interpretability, hoping to inspire future research in this field to place a stronger emphasis on it.

For decision support systems, we identify the declarative formulation of optimization problems as a crucial source of symbolic knowledge, with the learning process being the most suitable phase for its integration. The UNIFY framework is presented as a general approach for this integration. While this thesis provides a glimpse into UNIFY's potential, there are opportunities for further extensions, such as addressing long-term or chance constraints and applying it to robust optimization. In the realm of decision-focused learning, UNIFY could enhance applicability to more complex, nonlinear optimization problems, though scalability remains a challenge due to the requirement of solving multiple optimization problems.

Appendix A

Theoretical Results on Score Function Gradient Estimation

In this section, we prove the theoretical soundness of the method.

Recall the derivation of the gradient estimation:

$$\nabla_\theta L(\theta, y) = \nabla_\theta \mathbb{E}_{\hat{y}\sim p_\theta(y)}[\mathcal{L}(z^\star(\hat{y}), y)] \tag{A.1a}$$

$$= \nabla_\theta \int p_\theta(\hat{y})\mathcal{L}(z^\star(\hat{y}), y)d\hat{y} \tag{A.1b}$$

$$= \int \mathcal{L}(z^\star(\hat{y}), y)\nabla_\theta p_\theta(\hat{y})d\hat{y} \tag{A.1c}$$

$$= \int \mathcal{L}(z^\star(\hat{y}), y)p_\theta(\hat{y})\nabla_\theta \log p_\theta(\hat{y})d\hat{y} \tag{A.1d}$$

$$= \mathbb{E}_{\hat{y}\sim p_\theta(y)}[\mathcal{L}(z^\star(\hat{y}), y)\nabla_\theta \log p_\theta(\hat{y})] \tag{A.1e}$$

The validity of the equations is trivial except for the interchange of the integral and gradient in (6.40c). Mohamed et al. [178] states that the interchange is valid if:

(i) $p_\theta(y)$ is continuously differentiable in its parameters θ,

(ii) $p_\theta(\hat{y})\mathcal{L}(z^\star(\hat{y}), y)$ is both integrable and differentiable for all parameters θ, and

(iii) There exists an integrable function $g(\hat{y})$ such that $\sup_\theta ||\mathcal{L}(z^\star(\hat{y}), y)\nabla_\theta p_\theta(\hat{y})||_1 \leq g(\hat{y}), \forall \hat{y}$.

For an arbitrary choice of probability density p_θ and the loss function $\mathcal{L}$, it is very difficult to check that these three conditions hold (see, L'Ecuyer [180] or Glasserman [181] for a more detailed discussion). Therefore, we make nonrestrictive assumptions to prove that the above conditions hold for our case.

First, assume that p_θ is Gaussian, then (i) holds due to the smoothness of the density function. For the univariate case:

$$p_\theta(y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2}$$

where $\theta = (\mu, \sigma)$. Then, The derivatives of the Gaussian distribution with respect to μ and σ are

$$\frac{y-\mu}{\sigma^2} p_\theta(y) \text{ and } \left(\frac{(y-\mu)^2}{\sigma^3} - \frac{1}{\sigma}\right) p_\theta(y), \tag{A.2}$$

respectively. Both derivatives are continuous in their respective parameters, unless when $\sigma = 0$. This condition could be easily avoided by adding a small constant to the predicted σ. Thus (i) holds. The extension to multivariate cases is similar.

In order to show that (ii) holds, we can also assume that the post-hoc regret $\mathcal{L}(z^\star(\hat{y}), y)$ is bounded. The first part of the post-hoc regret, the regret, is bounded if the feasible region of the optimization problem is also bounded. We can also assume that the second part, the penalty, always gives a finite value, since in practice there is no infinitely infeasible decision to correct. Then, since $\mathcal{L}$ is bounded, and under the Gaussian assumption, the product p_θ and $\mathcal{L}$ is integrable and differentiable for all parameters θ. Hence (ii) holds under these assumptions.

To show (iii), first note that that $\nabla_\theta p_\theta(\hat{y})$ takes a finite value for all θ and it vanishes as $\theta \to \pm\infty$ for a Gaussiian random variable. The univariate case is clear in (A.2) and the extension to the multivariate case is similar. Therefore, $\nabla_\theta p_\theta(\hat{y})$ is bounded, i.e. there exists a (possibly large) positive real number $M(\hat{y})$ depending on $\hat{y}$ such that $\sup_\theta ||\nabla_\theta p_\theta(\hat{y})||_1 \leq M(\hat{y})$ for all $\hat{y}$.

Using the Cauchy–Schwarz inequality , we get:

$$||\mathcal{L}(z^\star(\hat{y}), y)\nabla_\theta p_\theta(\hat{y})||_1 \leq \mathcal{L}(z^\star(\hat{y}), y)||\nabla_\theta p_\theta(\hat{y})||_1$$

for all θ and $\hat{y}$ since $\mathcal{L}$ is a non-negative real-valued function. Taking the supremum of the both sides of the equation with respect to θ, we get:

$$\sup_\theta ||\mathcal{L}(z^\star(\hat{y}), y)\nabla_\theta p_\theta(\hat{y})||_1 \leq \mathcal{L}(z^\star(\hat{y}), y) \sup_\theta ||\nabla_\theta p_\theta(\hat{y})||_1$$

since the loss does not depend on θ. Then, we have:

$$\sup_\theta ||\mathcal{L}(z^\star(\hat{y}), y)\nabla_\theta p_\theta(\hat{y})||_1 \leq g(\hat{y}) := \mathcal{L}(z^\star(\hat{y}), y)M(\hat{y})$$

where g is constant and hence integrable. Hence (iii) holds.

Appendix B

Additional Results on SFGE for DFL

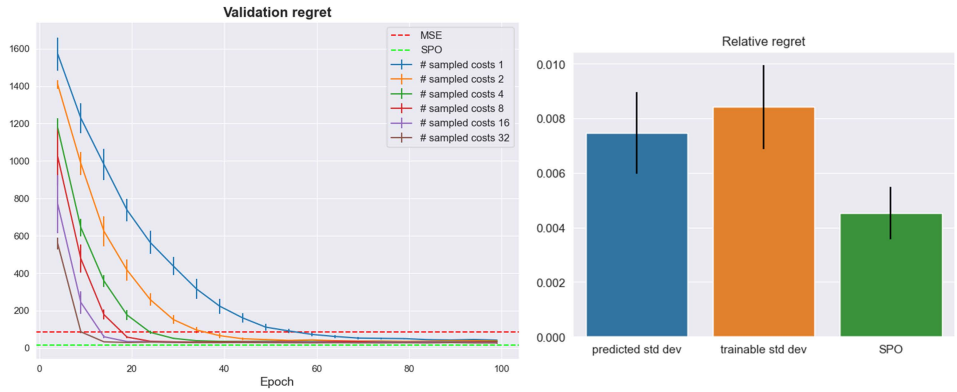


Figure B.1: Left: validation regret on the KP-50 w.r.t. the number of epochs when multiple predictions $\hat{y}$ are sampled for the same x. Right: test relative regret on the KP-50 when σ is contextual (*predicted std dev*) and a trainable parameter (*trainable*), compared with the state-of-the-art SPO.

Estimating the parameter distribution While we employ stochastic parameter estimates, it's important to note that they are a part of our smoothing approach and need not precisely reflect the actual distribution of y. This realization underscores a few key points: 1) We opt for a Gaussian distribution not because it perfectly represents the nature of y, but because it results in localized smoothing and more representative gradients. 2) Since the standard deviation primarily serves as a smoothing factor, our approach remains effective regardless of whether σ is trainable. Throughout our research, we conducted experiments with various σ settings, including a constant σ, a trainable non-contextual σ (the same for all examples), and a contextual σ (input-dependent). It was observed that using a trainable standard deviation tends to yield the best results, while introducing contextuality (i.e., $\sigma(x)$) did not yield significant advantages. In fig. B.1 (right), we present the relative regret of SFGE on KP-50 with a contextual σ. The results obtained with the best hyperparameter configuration closely resemble those achieved with a non-contextual σ setting.

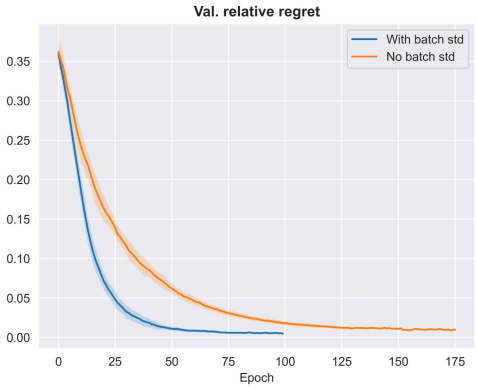


Figure B.2: Validation relative regret during training of SFGE with and without standardization on the KP-50

Improving the gradient estimate Our training problem can be understood as:

$$\underset{\theta}{\operatorname{argmin}}\, \mathbb{E}_{x,y\sim p(x,y),\hat{y}\sim p_\theta(\hat{y}|x)} \left[\mathcal{L}(z^\star(\hat{y}), y)\right] \tag{B.1}$$

where the expectation on x, y is approximated via mini-batches, and the expectation on $\hat{y}$ by sampling from the smoothing distribution. The motivation for using a higher number of samples is to obtain a good gradient for improved generalizability. First we want to highlight that we train using mini-batch gradient descent, which introduces stochasticity through different batches. This inherent randomness of mini-batch gradient descent contributes to generalizability even if only one sample is used for one instance. Nevertheless, using more samples still might lead to more reliable gradients, but it also requires solving more optimization problems per gradient descent step. In our research, we had indeed investigated this trade-off: as shown in fig. B.1 (left), using more samples results in fewer training epochs, not a faster training time since for each sample we need to solve an optimization problem.

As previously mentioned in the main body of the paper, we apply standardization to the regret within a single mini-batch to enhance convergence speed by reducing gradient variance. The standardization is computed as follows:

$$\tilde{R} = \frac{R - \mu}{\sigma^2 + \epsilon}$$

Here, R represents the regret, μ and σ denote the mean and variance of the regret within a mini-batch, and $\epsilon = 10^{-8}$ is a small constant introduced to prevent numerical instability. To empirically demonstrate the effectiveness of this standardization operation, we compare a model trained with SFGE with and without the standardization of the regret within mini-batches. We conducted experiments on the KP-50 dataset, following the same evaluation procedure as described section 6.4. In fig. B.2, we present a comparison of the validation relative regret between the two approaches, clearly illustrating that standardization significantly improves convergence speed.

Since the choice of mini-batch size affects the results of standardization and, consequently, the variance reduction, we conducted experiments with various batch sizes, specifically $\{2, 4, 8, 16, 32, 64, 128, 256, 512\}$, on the KP-50 dataset. We evaluated both the relative regret on the test set and the number of optimization problems solved before reaching convergence. The latter experiment provides insights into

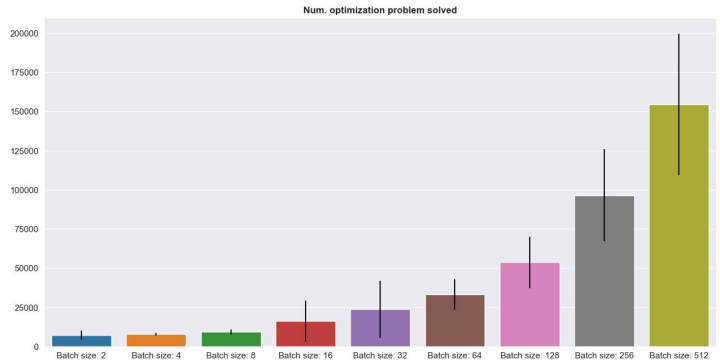


Figure B.3: Total number of optimization problems solved by SFGE during training w.r.t. the mini-batch size used in stochastic gradient descent.

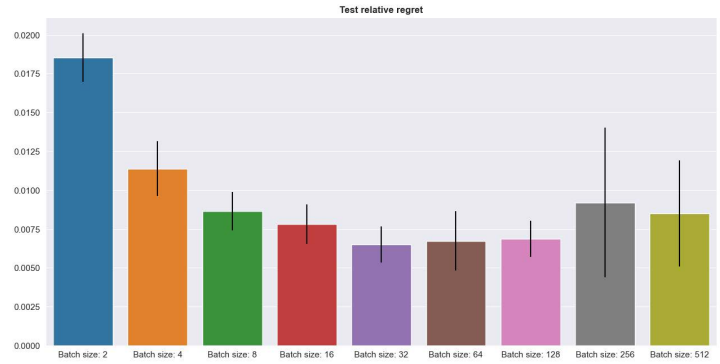


Figure B.4: Test relative regret w.r.t. the mini-batch size used in stochastic gradient descent.

the computational efficiency of different configurations, considering that solving a large optimization problem can be challenging. As depicted in fig. B.3, increasing the batch size results in a larger number of optimization problems required to reach convergence, as it necessitates more epochs. With a larger batch size, the number of mini-batches decreases, reducing the number of optimization steps per epoch. Overall, a batch size of 32 demonstrates the best trade-off in terms of computational cost and relative regret.

Bibliography

[1] Timo M Deist, Andrew Patti, Zhaoqi Wang, David Krane, Taylor Sorenson, and David Craft. Simulation-assisted machine learning. *Bioinformatics*, 35(20):4072–4080, 2019.

[2] Barbaros Yet, Zane B Perkins, Todd E Rasmussen, Nigel RM Tai, and D William R Marsh. Combining data and meta-analysis to build bayesian networks for clinical decision support. *Journal of biomedical informatics*, 52:373–385, 2014.

[3] Heyuan Liu and Paul Grigas. Online contextual decision-making with a smart predict-then-optimize method. *CoRR*, abs/2206.07316, 2022. doi: 10.48550/arXiv.2206.07316. URL https://doi.org/10.48550/arXiv.2206.07316.

[4] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006. ISBN 978-0-444-52726-4. URL https://www.sciencedirect.com/science/bookseries/15746526/2.

[5] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[6] Yurii Nesterov and Arkadii Nemirovskii. *Interior-point polynomial algorithms in convex programming*. SIAM, 1994.

[7] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, page 302–311, New York, NY, USA, 1984. Association for Computing Machinery. ISBN 0897911334. doi: 10.1145/800057.808695. URL https://doi.org/10.1145/800057.808695.

[8] Eva K. Lee and John E. Mitchell. *Integer programming: branch and bound methodsInteger Programming: Branch and Bound Methods*, pages 1634–1643. Springer US, Boston, MA, 2009. ISBN 978-0-387-74759-0. doi: 10.1007/978-0-387-74759-0_286. URL https://doi.org/10.1007/978-0-387-74759-0_286.

[9] Warren B Powell. A unified framework for stochastic optimization. *European Journal of Operational Research*, 275(3):795–821, 2019.

[10] Peter Kall, Stein W Wallace, and Peter Kall. *Stochastic programming*, volume 5. Springer, 1994.

[11] John R Birge and Francois Louveaux. *Introduction to stochastic programming*. Springer Science & Business Media, 2011.

[12] Sujin Kim, Raghu Pasupathy, and Shane G Henderson. A guide to sample average approximation. *Handbook of simulation optimization*, pages 207–243, 2015.

[13] Anton J Kleywegt, Alexander Shapiro, and Tito Homem-de Mello. The sample average approximation method for stochastic discrete optimization. *SIAM Journal on Optimization*, 12(2):479–502, 2002.

[14] T. Mitchell. *Machine Learning*. McGraw-Hill New York, 1997.

[15] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[16] Trevor Hastie, Robert Tibshirani, Jerome H Friedman, and Jerome H Friedman. *The elements of statistical learning: data mining, inference, and prediction*, volume 2. Springer, 2009.

[17] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

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

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

[20] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[21] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[22] Matthias Steup. Epistemology: Stanford encyclopedia of philosophy. *Stanford Encyclopedia of Philosophy*, 2005.

[23] Linda Zagzebski. What is knowledge? *The Blackwell guide to epistemology*, pages 92–116, 2017.

[24] Laura Von Rueden, Sebastian Mayer, Katharina Beckh, Bogdan Georgiev, Sven Giesselbach, Raoul Heese, Birgit Kirsch, Julius Pfrommer, Annika Pick, Rajkumar Ramamurthy, et al. Informed machine learning–a taxonomy and survey of integrating prior knowledge into learning systems. *IEEE Transactions on Knowledge and Data Engineering*, 35(1):614–633, 2021.

[25] George S Boolos, John P Burgess, and Richard C Jeffrey. *Computability and logic*. Cambridge university press, 2002.

[26] Robert S Boyer and J Strother Moore. *A computational logic*. Academic press, 2014.

[27] Judea Pearl. *Causality*. Cambridge university press, 2009.

[28] Laith Alzubaidi, Jinglan Zhang, Amjad J Humaidi, Ayad Al-Dujaili, Ye Duan, Omran Al-Shamma, José Santamaría, Mohammed A Fadhel, Muthana Al-Amidie, and Laith Farhan. Review of deep learning: Concepts, cnn architectures, challenges, applications, future directions. *Journal of big Data*, 8:1–74, 2021.

[29] Yann LeCun, Bernhard Boser, John Denker, Donnie Henderson, Richard Howard, Wayne Hubbard, and Lawrence Jackel. Handwritten digit recognition with a back-propagation network. *Advances in neural information processing systems*, 2, 1989.

[30] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[31] Anuj Karpatne, William Watkins, Jordan Read, and Vipin Kumar. Physics-guided neural networks (pgnn): An application in lake temperature modeling. *arXiv preprint arXiv:1710.11431*, 2, 2017.

[32] Nikhil Muralidhar, Mohammad Raihanul Islam, Manish Marwah, Anuj Karpatne, and Naren Ramakrishnan. Incorporating prior domain knowledge into deep neural networks. In *2018 IEEE international conference on big data (big data)*, pages 36–45. IEEE, 2018.

[33] Russell Stewart and Stefano Ermon. Label-free supervision of neural networks with physics and domain knowledge. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 31, 2017.

[34] Olvi L Mangasarian and Edward W Wild. Nonlinear knowledge-based classification. *IEEE Transactions on Neural Networks*, 19(10):1826–1832, 2008.

[35] Glenn Fung, Olvi Mangasarian, and Jude Shavlik. Knowledge-based support vector machine classifiers. *Advances in neural information processing systems*, 15, 2002.

[36] Martin von Kurnatowski, Jochen Schmid, Patrick Link, Rebekka Zache, Lukas Morand, Torsten Kraft, Ingo Schmidt, Jan Schwientek, and Anke Stoll. Compensating data shortages in manufacturing with monotonicity knowledge. *Algorithms*, 14(12):345, 2021.

[37] Yanfei Lu, Manik Rajora, Pan Zou, and Steven Y Liang. Physics-embedded machine learning: case study with electrochemical micro-machining. *Machines*, 5(1):4, 2017.

[38] Rajkumar Ramamurthy, Christian Bauckhage, Rafet Sifa, Jannis Schücker, and Stefan Wrobel. Leveraging domain knowledge for reinforcement learning using mmc architectures. In *Artificial Neural Networks and Machine Learning–ICANN 2019: Deep Learning: 28th International Conference on Artificial Neural Networks, Munich, Germany, September 17–19, 2019, Proceedings, Part II 28*, pages 595–607. Springer, 2019.

[39] Ulrich Steinkühler and Holk Cruse. A holistic model for an internal representation to control the movement of a manipulator with redundant degrees of freedom. *Biological Cybernetics*, 79(6): 457–466, 1998.

[40] L'ubor Ladickỳ, SoHyeon Jeong, Barbara Solenthaler, Marc Pollefeys, and Markus Gross. Data-driven fluid simulations using regression forests. *ACM Transactions on Graphics (TOG)*, 34(6): 1–9, 2015.

[41] Leo Breiman. Random forests. *Machine learning*, 45:5–32, 2001.

[42] Isaac E Lagaris, Aristidis Likas, and Dimitrios I Fotiadis. Artificial neural networks for solving ordinary and partial differential equations. *IEEE transactions on neural networks*, 9(5):987–1000, 1998.

[43] Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378:686–707, 2019.

[44] Arieh Iserles. *A First Course in the Numerical Analysis of Differential Equations*. Cambridge Texts in Applied Mathematics. Cambridge University Press, 2 edition, 2008.

[45] Yibo Yang and Paris Perdikaris. Physics-informed deep generative models. *arXiv preprint arXiv:1812.03511*, 2018.

[46] Yinhao Zhu, Nicholas Zabaras, Phaedon-Stelios Koutsourelakis, and Paris Perdikaris. Physics-constrained deep learning for high-dimensional surrogate modeling and uncertainty quantification without labeled data. *Journal of Computational Physics*, 394:56–81, 2019.

[47] Emmanuel De Bézenac, Arthur Pajot, and Patrick Gallinari. Deep learning for physical processes: Incorporating prior scientific knowledge. *Journal of Statistical Mechanics: Theory and Experiment*, 2019(12):124009, 2019.

[48] Michael Lutter, Christian Ritter, and Jan Peters. Deep lagrangian networks: Using physics as model prior for deep learning. *arXiv preprint arXiv:1907.04490*, 2019.

[49] Dimitris C Psichogios and Lyle H Ungar. A hybrid neural network-first principles approach to process modeling. *AIChE Journal*, 38(10):1499–1511, 1992.

[50] Filipe de Avila Belbute-Peres, Kevin Smith, Kelsey Allen, Josh Tenenbaum, and J Zico Kolter. End-to-end differentiable physics for learning and control. *Advances in neural information processing systems*, 31, 2018.

[51] Wilhelm Kirchgässner, Oliver Wallscheid, and Joachim Böcker. Thermal neural networks: lumped-parameter thermal modeling with state-space machine learning. *Engineering Applications of Artificial Intelligence*, 117:105537, 2023.

[52] Marco Iachello, Viviana De Luca, Giuseppe Petrone, Natale Testa, Luigi Fortuna, Giuliano Cammarata, Salvatore Graziani, and Mattia Frasca. Lumped parameter modeling for thermal characterization of high-power modules. *IEEE Transactions on Components, Packaging and Manufacturing Technology*, 4(10):1613–1623, 2014.

[53] Amir Sajjad Bahman, Ke Ma, Pramod Ghimire, Francesco Iannuzzo, and Frede Blaabjerg. A 3-d-lumped thermal network model for long-term load profiles analysis in high-power igbt modules. *IEEE Journal of Emerging and Selected Topics in Power Electronics*, 4(3):1050–1063, 2016.

[54] Nicolas Bracikowski, Michel Hecquet, Pascal Brochet, and Sergey V Shirinskii. Multiphysics modeling of a permanent magnet synchronous machine by using lumped models. *IEEE Transactions on Industrial Electronics*, 59(6):2426–2437, 2011.

[55] Oliver Wallscheid and Joachim Böcker. Global identification of a low-order lumped-parameter thermal network for permanent magnet synchronous motors. *IEEE Transactions on Energy Conversion*, 31(1):354–365, 2015.

[56] Richard W. Cottle. *Linear complementarity problemLinear Complementarity Problem*, pages 1267–1271. Springer US, Boston, MA, 2001. ISBN 978-0-306-48332-5. doi: 10.1007/0-306-48332-7_258. URL https://doi.org/10.1007/0-306-48332-7_258.

[57] Geoffrey G Towell and Jude W Shavlik. Knowledge-based artificial neural networks. *Artificial intelligence*, 70(1-2):119–165, 1994.

[58] Artur S Avila Garcez and Gerson Zaverucha. The connectionist inductive learning and logic programming system. *Applied Intelligence*, 11:59–77, 1999.

[59] Manoel VM França, Gerson Zaverucha, and Artur S d'Avila Garcez. Fast relational learning using bottom clause propositionalization with artificial neural networks. *Machine learning*, 94:81–104, 2014.

[60] Artur S d'Avila Garcez, Krysia Broda, and Dov M Gabbay. *Neural-symbolic learning systems: foundations and applications*. Springer Science & Business Media, 2002.

[61] Artur d'Avila Garcez, Tarek R Besold, Luc De Raedt, Peter Földiak, Pascal Hitzler, Thomas Icard, Kai-Uwe Kühnberger, Luis C Lamb, Risto Miikkulainen, and Daniel L Silver. Neural-symbolic learning and reasoning: contributions and challenges. In *2015 AAAI Spring Symposium Series*, 2015.

[62] Artur d'Avila Garcez, Marco Gori, Luis C Lamb, Luciano Serafini, Michael Spranger, and Son N Tran. Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. *arXiv preprint arXiv:1905.06088*, 2019.

[63] Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015.

[64] Martin Schiegg, Marion Neumann, and Kristian Kersting. Markov logic mixtures of gaussian processes: Towards machines reading regression data. In *Artificial Intelligence and Statistics*, pages 1002–1011. PMLR, 2012.

[65] Mrinmaya Sachan, Kumar Avinava Dubey, Tom M Mitchell, Dan Roth, and Eric P Xing. Learning pipelines with limited data and domain knowledge: A study in parsing physics problems. *Advances in Neural Information Processing Systems*, 31, 2018.

[66] Matthew Richardson and Pedro Domingos. Markov logic networks. *Machine learning*, 62:107–136, 2006.

[67] Angelika Kimmig, Stephen Bach, Matthias Broecheler, Bert Huang, and Lise Getoor. A short introduction to probabilistic soft logic. In *Proceedings of the NIPS workshop on probabilistic programming: foundations and applications*, pages 1–4, 2012.

[68] Luc De Raedt, Kristian Kersting, Sriraam Natarajan, and David Poole. Statistical relational artificial intelligence: Logic, probability, and computation. *Synthesis lectures on artificial intelligence and machine learning*, 10(2):1–189, 2016.

[69] Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018.

[70] Robin Manhaeve, Sebastijan Dumančić, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Neural probabilistic logic programming in deepproblog. *Artificial Intelligence*, 298:103504, 2021.

[71] Robin Manhaeve, Giuseppe Marra, and Luc De Raedt. Approximate inference for neural probabilistic logic programming. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pages 475–486. IJCAI Organization, 2021.

[72] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence*, pages 2462–2467. IJCAI-INT JOINT CONF ARTIF INTELL, 2007.

[73] Michelangelo Diligenti, Soumali Roychowdhury, and Marco Gori. Integrating prior knowledge into deep learning. In *2017 16th IEEE international conference on machine learning and applications (ICMLA)*, pages 920–923. IEEE, 2017.

[74] Jingyi Xu, Zilu Zhang, Tal Friedman, Yitao Liang, and Guy Broeck. A semantic loss function for deep learning with symbolic knowledge. In *International conference on machine learning*, pages 5502–5511. PMLR, 2018.

[75] Michelangelo Diligenti, Marco Gori, and Claudio Sacca. Semantic-based regularization for learning and inference. *Artificial Intelligence*, 244:143–165, 2017.

[76] Ming-Wei Chang, Lev Ratinov, and Dan Roth. Guiding semi-supervision with constraint-driven learning. In *Proceedings of the 45th annual meeting of the association of computational linguistics*, pages 280–287, 2007.

[77] Zhiting Hu, Zichao Yang, Ruslan Salakhutdinov, and Eric Xing. Deep neural networks with massive learned knowledge. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1670–1679, 2016.

[78] Zhiting Hu, Xuezhe Ma, Zhengzhong Liu, Eduard Hovy, and Eric Xing. Harnessing deep neural networks with logic rules. *arXiv preprint arXiv:1603.06318*, 2016.

[79] Vilém Novák. First-order fuzzy logic. *Studia logica*, 46:87–109, 1987.

[80] Petr Hájek. *Metamathematics of fuzzy logic*, volume 4. Springer Science & Business Media, 2013.

[81] Kuan-Hui Lee, German Ros, Jie Li, and Adrien Gaidon. Spigan: Privileged adversarial learning from simulation. *arXiv preprint arXiv:1810.03756*, 2018.

[82] Julius Pfrommer, Clemens Zimmerling, Jinzhao Liu, Luise Kärger, Frank Henning, and Jürgen Beyerer. Optimisation of manufacturing process parameters using deep neural networks as surrogate models. *Procedia CiRP*, 72:426–431, 2018.

[83] Adam Lerer, Sam Gross, and Rob Fergus. Learning physical intuition of block towers by example. In *International conference on machine learning*, pages 430–438. PMLR, 2016.

[84] Akshara Rai, Rika Antonova, Franziska Meier, and Christopher G Atkeson. Using simulation to improve sample-efficiency of bayesian optimization for bipedal robots. *The Journal of Machine Learning Research*, 20(1):1844–1867, 2019.

[85] Ashish Shrivastava, Tomas Pfister, Oncel Tuzel, Joshua Susskind, Wenda Wang, and Russell Webb. Learning from simulated and unsupervised images through adversarial training. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2107–2116, 2017.

[86] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

[87] Hong Seok Kim, Muammer Koc, and Jun Ni. A hybrid multi-fidelity approach to the optimal design of warm forming processes using a knowledge-based artificial neural network. *International Journal of Machine Tools and Manufacture*, 47(2):211–222, 2007.

[88] Fang Wang and Qi-Jun Zhang. Knowledge-based neural models for microwave design. *IEEE Transactions on Microwave Theory and Techniques*, 45(12):2333–2343, 1997.

[89] Stephen J Leary, Atul Bhaskar, and Andy J Keane. A knowledge-based approach to response surface modelling in multifidelity optimization. *Journal of Global Optimization*, 26:297–319, 2003.

[90] Anthony Costa Constantinou, Norman Fenton, and Martin Neil. Integrating expert knowledge with data in bayesian networks: Preserving data-driven expectations when the expert variables remain unobserved. *Expert systems with applications*, 56:197–208, 2016.

[91] David Heckerman, Dan Geiger, and David M Chickering. Learning bayesian networks: The combination of knowledge and statistical data. *Machine learning*, 20:197–243, 1995.

[92] Nico Piatkowski, Sangkyun Lee, and Katharina Morik. Spatio-temporal random fields: compressible representation and distributed estimation. *Machine learning*, 93:115–139, 2013.

[93] Tao tao Liu, Rui Liu, and Gui jiang Duan. A principle-empirical model based on bayesian network for quality improvement in mechanical products development. *Computers and Industrial Engineering*, 149:106807, 2020. ISSN 0360-8352. doi: https://doi.org/10.1016/j.cie.2020.106807. URL https://www.sciencedirect.com/science/article/pii/S036083522030512X.

[94] Lida Huang, Tao Chen, Qing Deng, and Yuli Zhou. Reasoning disaster chains with bayesian network estimated under expert prior knowledge. *International Journal of Disaster Risk Science*, pages 1–18, 2024.

[95] Xiaodong Fang, Chan Chang, and Genggeng Liu. Using bayesian network technology to predict the semiconductor manufacturing yield rate in iot. *The Journal of Supercomputing*, 77:9020–9045, 2021.

[96] Sindhu R Johnson, George A Tomlinson, Gillian A Hawker, John T Granton, and Brian M Feldman. Methods to elicit beliefs for bayesian priors: a systematic review. *Journal of clinical epidemiology*, 63(4):355–369, 2010.

[97] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[98] Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

[99] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.

[100] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2018.

[101] Iddo Drori, Anant Kharkar, William R Sickinger, Brandon Kates, Qiang Ma, Suwen Ge, Eden Dolev, Brenda Dietrich, David P Williamson, and Madeleine Udell. Learning to solve combinatorial optimization problems on real-world graphs in linear time. In *2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 19–24. IEEE, 2020.

[102] Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[103] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. Reinforcement learning for solving the vehicle routing problem. *Advances in neural information processing systems*, 31, 2018.

[104] Chaitanya K Joshi, Quentin Cappart, Louis-Martin Rousseau, and Thomas Laurent. Learning the travelling salesperson problem requires rethinking generalization. *Constraints*, 27(1-2):70–98, 2022.

[105] Hao Lu, Xingwen Zhang, and Shuang Yang. A learning-based iterative method for solving vehicle routing problems. In *International conference on learning representations*, 2019.

[106] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[107] Jean-Baptiste Mairy, Yves Deville, and Pascal Van Hentenryck. Reinforced adaptive large neighborhood search. In *The Seventeenth International Conference on Principles and Practice of Constraint Programming (CP 2011)*, page 55. Springer Berlin/Heidelberg, Germany, 2011.

[108] Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *Top*, 25:207–236, 2017.

[109] Quentin Cappart, Emmanuel Goutierre, David Bergman, and Louis-Martin Rousseau. Improving optimization bounds using machine learning: Decision diagrams meet deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1443–1451, 2019.

[110] Quentin Cappart, Thierry Moisan, Louis-Martin Rousseau, Isabeau Prémont-Schwarz, and Andre A Cire. Combining reinforcement learning and constraint programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 3677–3687, 2021.

[111] Yunhao Tang, Shipra Agrawal, and Yuri Faenza. Reinforcement learning for integer programming: Learning to cut. In *International conference on machine learning*, pages 9367–9376. PMLR, 2020.

[112] Félix Chalumeau, Ilan Coulon, Quentin Cappart, and Louis-Martin Rousseau. Seapearl: A constraint programming solver guided by reinforcement learning. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 18th International Conference, CPAIOR 2021, Vienna, Austria, July 5–8, 2021, Proceedings 18*, pages 392–409. Springer, 2021.

[113] Jayanta Mandi, James Kotary, Senne Berden, Maxime Mulamba, Victor Bucarey, Tias Guns, and Ferdinando Fioretto. Decision-focused learning: Foundations, state of the art, benchmark and future opportunities. *arXiv preprint arXiv:2307.13565*, 2023.

[114] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[115] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[116] Brandon Amos and J Zico Kolter. Optnet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*, pages 136–145. PMLR, 2017.

[117] Gurobi Optimization, LLC. *Gurobi Optimization, LLC*, 2023. URL https://www.gurobi.com. 31st December 2023.

[118] IBM Corp. *IBM ILOG CPLEX Optimization Studio*, 2023. URL https://www.ibm.com/products/ilog-cplex-optimization-studio. 31st December 2023.

[119] Takuya Konishi and Takuro Fukunaga. End-to-end learning for prediction and optimization with gradient boosting. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 191–207. Springer, 2020.

[120] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2): 337–407, 2000.

[121] Akshay Agrawal, Shane Barratt, Stephen Boyd, Enzo Busseti, and Walaa M Moursi. Differentiating through a cone program. *arXiv preprint arXiv:1904.09043*, 2019.

[122] Akshay Agrawal, Brandon Amos, Shane Barratt, Stephen Boyd, Steven Diamond, and J Zico Kolter. Differentiable convex optimization layers. *Advances in neural information processing systems*, 32, 2019.

[123] Mokhtar S Bazaraa, John J Jarvis, and Hanif D Sherali. *Linear programming and network flows*. John Wiley & Sons, 2011.

[124] Bryan Wilder, Bistra Dilkina, and Milind Tambe. Melding the data-decisions pipeline: Decision-focused learning for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 1658–1665, 2019.

[125] Jayanta Mandi and Tias Guns. Interior point solving for lp-based prediction+ optimisation. *Advances in Neural Information Processing Systems*, 33:7272–7282, 2020.

[126] Mathieu Blondel, Olivier Teboul, Quentin Berthet, and Josip Djolonga. Fast differentiable sorting and ranking. In *International Conference on Machine Learning*, pages 950–959. PMLR, 2020.

[127] Brandon Amos, Vladlen Koltun, and J Zico Kolter. The limited multi-label projection layer. *arXiv preprint arXiv:1906.08707*, 2019.

[128] Aaron Ferber, Bryan Wilder, Bistra Dilkina, and Milind Tambe. Mipaal: Mixed integer program as a layer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1504–1511, 2020.

[129] Marin Vlastelica Pogančić, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations*, 2019.

[130] Subham Sekhar Sahoo, Anselm Paulus, Marin Vlastelica, Vít Musil, Volodymyr Kuleshov, and Georg Martius. Backpropagation through combinatorial algorithms: Identity with projection works. *arXiv preprint arXiv:2205.15213*, 2022.

[131] Quentin Berthet, Mathieu Blondel, Olivier Teboul, Marco Cuturi, Jean-Philippe Vert, and Francis Bach. Learning with differentiable pertubed optimizers. *Advances in neural information processing systems*, 33:9508–9519, 2020.

[132] Jean Guyomarch. Warcraft II Open-Source Map Editor, 2017. URL http://github.com/war2/war2edit. 31st December 2023.

[133] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[134] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[135] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *International conference on machine learning*, pages 1278–1286. PMLR, 2014.

[136] Mathias Niepert, Pasquale Minervini, and Luca Franceschi. Implicit mle: backpropagating through discrete exponential family distributions. *Advances in Neural Information Processing Systems*, 34: 14567–14579, 2021.

[137] Anselm Paulus, Michal Rolínek, Vít Musil, Brandon Amos, and Georg Martius. Comboptnet: Fit the right np-hard problem by learning integer programming constraints. In *International Conference on Machine Learning*, pages 8443–8453. PMLR, 2021.

[138] Maxime Mulamba, Jayanta Mandi, Michelangelo Diligenti, Michele Lombardi, Victor Bucarey, and Tias Guns. Contrastive losses and solution caching for predict-and-optimize. *arXiv preprint arXiv:2011.05354*, 2020.

[139] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 297–304. JMLR Workshop and Conference Proceedings, 2010.

[140] Ian J Goodfellow. On distinguishability criteria for estimating generative models. *arXiv preprint arXiv:1412.6515*, 2014.

[141] Jayanta Mandi, Vıctor Bucarey, Maxime Mulamba Ke Tchomba, and Tias Guns. Decision-focused learning: through the lens of learning to rank. In *International Conference on Machine Learning*, pages 14935–14947. PMLR, 2022.

[142] Thorsten Joachims. Optimizing search engines using clickthrough data. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 133–142, 2002.

[143] Xinyi Hu, Jasper CH Lee, and Jimmy HM Lee. Predict+ optimize for packing and covering lps with unknown parameters in constraints. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 3987–3995, 2023.

[144] Xinyi Hu, Jasper CH Lee, and Jimmy HM Lee. Branch & learn with post-hoc correction for predict+ optimize with unknown parameters in constraints. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 264–280. Springer, 2023.

[145] Xinyi Hu, Jasper Lee, Jimmy Lee, and Allen Z Zhong. Branch & learn for recursively and iteratively solvable problems in predict+ optimize. *Advances in Neural Information Processing Systems*, 35: 25807–25817, 2022.

[146] Emir Demirovic, Peter J Stuckey, Tias Guns, James Bailey, Christopher Leckie, Kotagiri Ramamohanarao, Jeffrey Chan, et al. Dynamic programming for predict+ optimise. In *AAAI*, pages 1444–1451, 2020.

[147] D. Aloini, E. Crisostomi, M. Raugi, and R. Rizzo. Optimal power scheduling in a virtual power plant. In *2011 2nd IEEE PES International Conference and Exhibition on Innovative Smart Grid Technologies*, pages 1–7, Dec 2011.

[148] Allegra De Filippo, Michele Lombardi, and Michela Milano. How to tame your anticipatory algorithm. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence*, pages 1071–1077, 2019.

[149] Alfonso P Ramallo-González, Matthew E Eames, and David A Coley. Lumped parameter models for building thermal modelling: An analytic approach to simplifying complex multi-layered constructions. *Energy and Buildings*, 60:174–184, 2013.

[150] Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[151] Christopher Rackauckas, Yingbo Ma, Julius Martensen, Collin Warner, Kirill Zubov, Rohit Supekar, Dominic Skinner, Ali Ramadhan, and Alan Edelman. Universal differential equations for scientific machine learning. *arXiv preprint arXiv:2001.04385*, 2020.

[152] Yulia Rubanova, Ricky T. Q. Chen, and David K Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL https://proceedings.neurips.cc/paper/2019/file/42a6845a557bef704ad8ac9cb4461d43-Paper.pdf.

[153] Patrick Kidger, James Morrill, James Foster, and Terry Lyons. Neural controlled differential equations for irregular time series. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 6696–6707. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/4a5876b450b45371f6cfe5047ac8cd45-Paper.pdf.

[154] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256. JMLR Workshop and Conference Proceedings, 2010.

[155] Hong Xu, Sven Koenig, and TK Satish Kumar. Towards effective deep learning for constraint satisfaction problems. In *Proc. of CPAIOR*, pages 588–597. Springer, 2018.

[156] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[157] Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. Understanding the potential of propagators. In Laurent Michel, editor, *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*, volume 9075 of *Lecture Notes in Computer Science*, pages 427–436. Springer, 2015. doi: 10.1007/978-3-319-18008-3_29. URL https://doi.org/10.1007/978-3-319-18008-3_29.

[158] Michelangelo Diligenti, Marco Gori, and Claudio Saccà. Semantic-based regularization for learning and inference. *Artificial Intelligence*, 244:143 – 165, 2017. ISSN 0004-3702. doi: https://doi.org/10.1016/j.artint.2015.08.011. URL http://www.sciencedirect.com/science/article/pii/S0004370215001344. Combining Constraint Solving with Mining and Learning.

[159] Andrea Galassi, Michele Lombardi, Paola Mello, and Michela Milano. Model agnostic solution of csps via deep learning: A preliminary study. In Willem-Jan van Hoeve, editor, *Proc. of CPAIOR*, pages 254–262, Cham, 2018. Springer International Publishing.

[160] Google OR-Tools. https://developers.google.com/optimization. Accessed: [December 2023].

[161] Carla P Gomes, Bart Selman, et al. Problem structure in the presence of perturbations. *AAAI/IAAI*, 97:221–226, 1997.

[162] Priya Donti, Brandon Amos, and J Zico Kolter. Task-based end-to-end model learning in stochastic optimization. *Advances in neural information processing systems*, 30, 2017.

[163] James Kotary, Ferdinando Fioretto, Pascal Van Hentenryck, and Bryan Wilder. End-to-end constrained optimization learning: A survey. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 4475–4482. ijcai.org, 2021. doi: 10.24963/ijcai.2021/610. URL https://doi.org/10.24963/ijcai.2021/610.

[164] Marin Vlastelica Pogančić, Anselm Paulus, Vit Musil, Georg Martius, and Michal Rolinek. Differentiation of blackbox combinatorial solvers. In *ICLR 2020 : Eighth International Conference on Learning Representations*, 2020.

[165] Jayanta Mandi and Tias Guns. Interior point solving for lp-based prediction+optimisation. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/51311013e51adebc3c34d2cc591fefee-Abstract.html.

[166] Maxime Mulamba, Jayanta Mandi, Michelangelo Diligenti, Michele Lombardi, Victor Bucarey, and Tias Guns. Contrastive losses and solution caching for predict-and-optimize. In Zhi-Hua Zhou, editor, *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, pages 2833–2840. ijcai.org, 2021. doi: 10.24963/ijcai.2021/390. URL https://doi.org/10.24963/ijcai.2021/390.

[167] Adam N Elmachtoub and Paul Grigas. Smart "predict, then optimize". *Management Science*, 68 (1):9–26, 2022.

[168] Gal Dalal, Krishnamurthy Dvijotham, Matej Vecerík, Todd Hester, Cosmin Paduraru, and Yuval Tassa. Safe exploration in continuous action spaces. *CoRR*, abs/1801.08757, 2018. URL http://arxiv.org/abs/1801.08757.

[169] Tsung-Yen Yang, Justinian Rosca, Karthik Narasimhan, and Peter J Ramadge. Projection-based constrained policy optimization. In *International Conference on Learning Representations*, 2019.

[170] Neal Parikh, Stephen Boyd, et al. Proximal algorithms. *Foundations and trends® in Optimization*, 1(3):127–239, 2014.

[171] Allegra De Filippo, Michele Lombardi, and Michela Milano. The blind men and the elephant: Integrated offline/online optimization under uncertainty. In *IJCAI*, 2020.

[172] Allegra De Filippo, Michele Lombardi, and Michela Milano. Integrated offline and online decision making under uncertainty. *Journal of Artificial Intelligence Research*, 70:77–117, 2021.

[173] Richard M Van Slyke and Roger Wets. L-shaped linear programs with applications to optimal control and stochastic programming. *SIAM journal on applied mathematics*, 17(4):638–663, 1969.

[174] Gilbert Laporte and François V Louveaux. The integer l-shaped method for stochastic integer programs with complete recourse. *Operations research letters*, 13(3):133–142, 1993.

[175] Pascal Van Hentenryck and Russell Bent. *Online stochastic combinatorial optimization*. The MIT Press, 2006.

[176] Luc Mercier and Pascal Van Hentenryck. Amsaa: A multistep anticipatory algorithm for online stochastic combinatorial optimization. In *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, pages 173–187. Springer, 2008.

[177] Mattia Silvestri, Allegra De Filippo, Federico Ruggeri, and Michele Lombardi. Hybrid offline/online optimization for energy management via reinforcement learning. In *International Conference on Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 358–373. Springer, 2022.

[178] Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. Monte carlo gradient estimation in machine learning. *J. Mach. Learn. Res.*, 21(132):1–62, 2020.

[179] Tal Grossman and Avishai Wool. Computational experience with approximation algorithms for the set covering problem. *European journal of operational research*, 101(1):81–92, 1997.

[180] Pierre L'Ecuyer. Note: On the interchange of derivative and expectation for likelihood ratio derivative estimators. *Management Science*, 41(4):738–747, 1995.

[181] Paul Glasserman. *Gradient estimation via perturbation analysis*, volume 116. Springer Science & Business Media, 1990.