Università degli studi di Brescia
Dipartimento di Economia e Management

Dottorato di Ricerca in Modelli e Metodi per l'Economia e il Management - Analytics for Economics and Management (AEM)
Ricerca Operativa – MATH/09
Ciclo XXXVII

Neural Combinatorial Optimization for the Context-Aware Traveling Salesman Problem

Davide Angioni

Supervisor: Prof. M. Grazia Speranza

Abstract

Neural Combinatorial Optimization (NCO) has emerged in recent years as a useful paradigm for solving combinatorial optimization problems, especially when the problem size makes the use of exact methods impractical. The application of NCO in dynamic settings, particularly in routing problems, is still limited, despite the fact that Machine Learning (ML) — and especially Deep Reinforcement Learning (DRL) — offers predictive capabilities that are particularly valuable for problems evolving over time. This thesis presents a tutorial on NCO in general, with a particular focus on its application to the Traveling Salesman Problem (TSP). It then analyzes the use of ML for dynamic routing problems, highlighting that DRL is the most widely adopted paradigm. Finally, it introduces our proposed approach, which defines a new model for the Dynamic TSP, called the Context-Aware TSP. This model aims to better reflect real-world conditions, as the dynamics of edge travel times depend on external factors such as traffic, weather, day, and time of the week. A major contribution of this work is the implementation of the model using a traffic simulator, creating a realistic environment for the agent to interact with. The problem is then solved using a NCO approach based on DRL, specifically the Proximal Policy Optimization algorithm. The results show that NCO can learn a policy capable of generating high-quality solutions in very short computation times. However, in scenarios where traffic conditions remain relatively stable over time, a Mixed Integer Linear Programming based method using static but contextually accurate data can achieve comparable or even better performance.

Abstract (italiano)

La Neural Combinatorial Optimization (NCO) è emersa negli anni recenti come un paradigma utile per risolvere problemi di ottimizzazione combinatoria, soprattutto quando la dimensione del problema rende impraticabile l'uso di metodi esatti. L'applicazione della NCO in contesti dinamici, in particolare nei problemi di routing, è ancora limitata, nonostante il Machine Learning (ML) — e soprattutto il Deep Reinforcement Learning (DRL) — offra capacità predittive particolarmente preziose per i problemi che evolvono nel tempo. Questa tesi presenta un tutorial sulla NCO in generale, con un focus particolare sulla sua applicazione al Problema del Commesso Viaggiatore (TSP). Analizza poi l'uso dell'ML per problemi di routing dinamico, evidenziando che il DRL è il paradigma più ampiamente adottato. Infine, introduce l'approccio da noi proposto, che definisce un nuovo modello per il TSP Dinamico (Dynamic TSP), denominato Context-Aware TSP. Questo modello mira a riflettere meglio le condizioni reali, in cui l'evoluzione dei tempi di percorrenza degli archi dipende da fattori esterni come traffico, meteo, giorno e ora della settimana. Un contributo importante di questo lavoro è l'implementazione del modello usando un simulatore di traffico, che crea un ambiente realistico con cui l'agente interagisce. Il problema viene quindi risolto utilizzando un approccio NCO basato sul DRL, e in particolare utilizzando l'algoritmo Proximal Policy Optimization. I risultati mostrano che la NCO può apprendere una policy capace di generare soluzioni di qualità in tempi di calcolo molto ridotti. Tuttavia, in scenari in cui le condizioni del traffico rimangono relativamente stabili nel tempo, un metodo basato sulla MILP che utilizza dati statici ma contestualmente accurati può ottenere prestazioni comparabili o persino migliori.

Preface

This thesis is the result of my doctoral studies at the University of Brescia, within the Department of Analytics for Economics and Management (AEM), under the supervision of Professor M. Grazia Speranza. The work was carried out between November 2021 and October 2024.

The doctoral program was part of an Industrial PhD initiative in collaboration with Orobix Srl, a company based in Bergamo, Italy, specializing in data science and artificial intelligence.

The thesis is structured into seven chapters. Chapter 1 introduces the research topic and motivates the study on Neural Combinatorial Optimization (NCO) for the Dynamic Traveling Salesman Problem. Chapter 2 provides a tutorial overview of NCO, with particular emphasis on the Traveling Salesman Problem. Chapter 3 presents a survey of the use of Machine Learning methods for dynamic routing problems. Chapters 4, 5 and 6 describe the experimental work conducted during the PhD and discuss the corresponding results. Finally, Chapter 7 summarizes the main insights that can be inferred from this work.

Part of this work has already been published: Chapter 2 appeared as a journal article in Computers & Operations Research. Chapter 3 is currently being prepared for submission, and the material from Chapters 4 to 6 will be revised and condensed into a journal paper for future publication.

Acknowledgements

La prima cosa di cui sono grato è di aver avuto l'opportunità di lavorare con la professoressa Grazia Speranza come supervisor. Mentre le sue competenze sono già note a livello internazionale, che il suo lato umano eguagli quello tecnico è stato un aspetto che sono stato fortunato a poter scoprire. È stata una guida e un supporto durante tutto il percorso e mi ha insegnato tanto in questi anni di collaborazione.

Un enorme ringraziamento va anche al resto dei professori dell'Università di Brescia, che fin dal primo giorno mi hanno fatto sentire parte del gruppo. Ho apprezzato ogni momento condiviso con voi. Un grazie speciale lo riservo alla professoressa Claudia Archetti per l'incredibile lavoro svolto durante la stesura del primo articolo. Poter pubblicare col duo Archetti - Speranza è come poter giocare un doppio con Federer. È per me un onore aver avuto questo privilegio.

Un'attività fondamentale nella letteratura scientifica è quella della revisione dei manoscritti. Ci tengo perciò a ringraziare i revisori di questa tesi, i professori Daniele Vigo e Paolo Dell'Olmo, per la cura e l'attenzione dimostrati e il tempo dedicato a questo incarico.

Grazie ad Orobix per avermi dato l'opportunità di lavorare in un ambiente stimolante e ricco di competenze. Grazie soprattutto a Lisa e Luca, per i consigli e i confronti costruttivi avuti in questi anni.

Un grazie per il suo inestimabile aiuto va anche a Francesco. Poter scrivere un articolo insieme è stata una piacevole sorpresa. Insieme a lui ringrazio anche tutti i miei amici, che mi hanno supportato in questo percorso. Un grazie in particolare ad Andrea e il suo Gennaro, il cui supporto è arrivato nei momenti più difficili.

La mia famiglia potrebbe ormai essere stanca di venire ringraziata, essendo questa la terza volta. Ma ogni grazie non è mai abbastanza. Se sono chi sono lo devo prima di tutto a loro, un'infinita fonte di insegnamenti e ispirazione che tengo sempre vicino al cuore. Ad avermi dato calore e conforto in questi anni sono stati anche tanti gatti e un cane, capaci di un amore incondizionato. Non posso citarvi tutti, quindi ringrazio forte solo Blasco, Snow, Anselmo e Schizzo. Spero che il mio grazie vi raggiunga ovunque siate.

L'ingiustizia di questa tesi è che il nome di Flavia non compaia mai, se non nei ringraziamenti. Un grazie non è abbastanza: i momenti di brainstorming, le discussioni peripatetiche, il supporto nello sconforto e la gioia nei festeggiamenti, gli incoraggiamenti e tutto l'amore che mi hai dato hanno reso possibile e arricchito questo traguardo che, come dici tu, è in realtà un inizio. Questa tesi è tanto tua quanto mia, ed è un mattone del castello che costruiremo insieme.

All'inizio del mio percorso universitario mi hai lasciato tu, nonno. Lo termino dicendo addio a te, nonna. Grazie per le cure, l'amore e tutti gli insegnamenti che mi avete lasciato. Staranno e starete con me per sempre.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

The growing availability of data and the ongoing digitalization of business processes represent a phenomenon of remarkable and continuing expansion. The recent resurgence of interest in Artificial Intelligence (AI), particularly its widespread adoption in everyday applications, has further stimulated attention toward AI-based solutions within industrial settings. At the same time, the field of Operations Research (OR) has begun to integrate AI tools, primarily to exploit the increasing amount of data that were previously unavailable or difficult to analyze effectively.

Among the most challenging problems in OR is dynamic optimization, where the problem conditions evolve over time. In this context, the ability to exploit data to identify trends and patterns, and to predict the evolution of the system dynamics, is crucial for the development of high-quality solutions. One branch of AI that has demonstrated particularly promising results in addressing dynamic problems—though historically applied in fields such as robotics and game solving rather than in OR—is Reinforcement Learning (RL). This paradigm aims to design autonomous software agents capable of learning, through interaction with their environment, which actions are most effective in achieving a given objective.

In recent years, the RL framework has been increasingly recognized as a powerful approach for tackling combinatorial optimization (CO) problems. A growing body of research has explored its application to various CO domains, including routing, scheduling, graph theory, and packing, among others.

1.2 Motivation

This work is carried out within the framework of an industrial PhD program and, as such, aims not only to analyze and develop new AI techniques for solving CO problems, but also to assess their applicability in a real industrial context.

Many of the problems faced by companies are dynamic in nature—subject to unforeseen events yet characterized by underlying trends that are often recognized only by experienced workers and are difficult to formalize in a mathematical model. Moreover, it is frequently necessary to update problem solutions, either because new clients or requests arise, or because the conditions of the problem itself change. Such updates must be performed within very short time frames in order to remain practically useful.

In this scenario, having methods capable of providing rapid, though possibly suboptimal, solutions is of crucial importance. At present, many of these problems are addressed through the use of heuristic methods. However, heuristics are often difficult to design and typically tailored to a specific problem or context. When problem conditions change, the existing heuristic may no longer be valid and must be redesigned.

For this reason, Neural Combinatorial Optimization (NCO) has emerged as a promising direction, as it seeks to generate heuristics automatically by learning from data the underlying patterns and dynamics of the problem.

Nevertheless, the use of NCO for dynamic problems remains limited. In most existing studies, data are obtained by generating synthetic problem instances, which do not accurately reflect real-world conditions.

1.3 Contributions

This thesis aims to present NCO in a clear and accessible manner, so that it can be effectively understood and adopted by researchers and practitioners without a specialized background in AI. At the same time, it seeks to demonstrate how NCO can be employed to address dynamic problems, with particular focus on the Dynamic Traveling Salesman Problem (DTSP).

To this end, a new model for the DTSP is proposed, called the Context-Aware TSP (CA-TSP). This model aims to better capture real-world conditions, in which the travel times along edges depend on external factors such as traffic, weather, and the day and time of the week.

The main contributions of this thesis are as follows:

- The creation of the first tutorial on NCO in the literature, with a particular focus on the Traveling Salesman Problem (TSP) as a representative problem.
- The first survey, to the best of our knowledge, exploring the use of Machine Learning (ML) for dynamic routing problems. This survey highlights how Reinforcement Learning (RL) has become the most widely adopted paradigm for such problems, while also analyzing the lack of a consistent definition of the term "dynamic" within CO and the challenges this ambiguity introduces.
- The definition of a new model for the dynamic TSP, named CA-TSP. This model represents a step toward more realistic formulations of DTSP, generalizing the concept of problem dynamics by linking the problem evolution to a variety of contextual factors.
- The implementation of the model within a traffic simulator, thereby creating a realistic environment in which the agent can interact. This is an important contribution, as most works in the literature rely on synthetic datasets that do not reflect real-world conditions and often oversimplify the applicability of Reinforcement Learning, failing to expose the practical challenges encountered when working with real data.
- The solution of the CA-TSP using an NCO approach, specifically through the Proximal Policy Optimization (PPO) algorithm. The results show that the agent is capable of learning a policy that produces good solutions within very short computation times.

- The release of two open-source repositories on GitHub. One for didactic purposes, illustrating how to implement NCO solutions for the static TSP, and one for research purposes, enabling the testing of NCO algorithms on the proposed CA-TSP environment with integrated traffic simulation.

1.4 Thesis outline

The thesis is organized into seven chapters.

This chapter, Chapter 1, provides a general introduction to the thesis and summarizes the main results achieved.

Chapter 2 is a tutorial on NCO. It presents the key concepts necessary to understand NCO, starting from the fundamentals of RL and then describing the neural network architectures most commonly used in NCO, explaining why they are well suited to capture the structure of CO problems. To clarify the theoretical explanations, the chapter illustrates how the TSP can be solved using NCO, beginning with a simple approach based on Multilayer Perceptrons (MLPs) and progressing to more sophisticated methods that employ Graph Neural Networks (GNNs) and attention networks. The tutorial concludes by discussing current open challenges and potential future research directions for NCO.

Chapter 3 presents a survey on the use of ML for dynamic routing problems. It introduces the concept of a dynamic problem, particularly in the context of routing, and analyzes how it has been addressed in the literature. The survey then examines works that apply ML either as a supporting tool for traditional OR algorithms or as the main method for solving the problem. For each paper reviewed, the survey discusses the problem considered, the aspects that make it dynamic, the OR and ML methods employed, and—when relevant—how these two paradigms interact to obtain a solution. Since RL emerges as the most widely used paradigm, especially in studies that solve the problem without OR methods, the chapter also analyzes the neural architectures and RL algorithms adopted in the literature.

Chapter 4 introduces the CA-TSP model. It begins by providing a formal definition of the TSP in a static context and clarifies what is meant by dynamic TSP. In doing so, it emphasizes that the term "dynamic TSP" actually refers to a family of problems, and discusses the implications of this in the literature. The chapter then presents the CA-TSP model, which aims to offer a more realistic definition of a dynamic TSP, where the travel times on edges depend on external factors that define the context in which the problem is solved. It concludes by introducing the NCO approach used to solve the problem, along with the notation and theoretical formulation of the training algorithm, namely Proximal Policy Optimization (PPO).

The following chapter, Chapter 5, provides the implementation details, structured into three parts: the traffic simulator used to model the environment, the input data generation process, and the neural network architectures used to build the agent; whereas, Chapter 6 presents the results, compares the proposed solution method with existing benchmarks, and provides an analysis of the developed solution framework.

Finally, 7 briefly presents the main insight that can be drawn from this work.

1.5 Conclusions

In this thesis, we explored the field of NCO, focusing on how it can be applied to solve the static TSP. We also examined how, at present, NCO—and more generally RL—has become one of the most popular ML approaches to address routing problems with dynamic characteristics. Finally, we developed an NCO solution for a dynamic version of the TSP, called the CA-TSP, in which the problem evolution depends on a set of contextual factors describing the environment in which the salesman operates. To bring the study closer to a real-world setting, we embedded the RL agent within a traffic simulator forming the core of our experimental framework.

This thesis demonstrates that, although NCO is a powerful tool for solving dynamic optimization problems, its application to realistic, real-world scenarios remains far from trivial. We have shown how the high computational cost of data collection from traffic simulators significantly impacts the convergence speed of RL algorithms and restricts the choice of algorithms that can be effectively used. Moreover, we have addressed the challenges of building realistic traffic simulations that evolve over time in a way that allows the NCO agent to learn meaningful temporal patterns.

Our results show that when traffic variations are limited, exact optimization methods that do not rely on online traffic information can outperform heuristic or learning-based methods that do. However, in a proof-of-concept scenario with more pronounced traffic dynamics, exact methods fail to find optimal solutions, whereas NCO can successfully learn effective adaptive strategies.

Beyond the empirical findings, this work also lays the foundations for more comprehensive studies of dynamic routing under realistic conditions. The framework we developed—an RL environment integrated with a traffic simulator—represents a flexible and reproducible tool that other researchers can use to develop and evaluate algorithms for the CA-TSP and related problems.

Ultimately, we hope that this thesis inspires Operations Research practitioners to explore NCO and to view ML not as a separate domain but as a powerful ally. The synergy between OR and ML holds the potential to redefine how we approach complex decision-making problems, benefiting both research communities and practical applications.

Chapter 2

Neural Combinatorial Optimization: a tutorial

2.1 Introduction

The purpose of this tutorial is to introduce the field of Neural Combinatorial Optimization (NCO), a framework for solving Combinatorial Optimization (CO) problems using Deep Reinforcement Learning (DRL). Aim of this tutorial is to present the most common techniques used in the literature, highlighting the difficulties that arise when applying DRL to CO and showing how they have been handled so far. We assume that the reader is already familiar with the field of CO.

Deep Learning (DL), which uses non-linear function approximators known as Deep Neural Networks (DNNs), gained attention in CO with the work of Vinyals et al., 2015, where the authors developed a supervised learning model for CO problems using a particular DNN called Pointer Network. The expression *Neural Combinatorial Optimization* was later introduced in Bello et al., 2017, where the authors replaced the supervised learning approach with DRL. The latter gained popularity after notable successes in games such as Dota 2, Go, Chess and Atari (see, for example, Mnih et al., 2015; Berner et al., 2019 and Schrittwieser et al., 2020), as well as in real-world applications, like the chip design algorithm developed by Mirhoseini et al., 2021 and a faster matrix multiplication algorithm developed by Fawzi et al., 2022. This gave researchers the confidence that CO problems could also be solved using DRL.

While the term NCO lacks a universally accepted definition, in this tutorial we define it as the application of DRL techniques to learn a heuristic for solving a particular CO problem or, more commonly, a specific class of instances of the same CO problem (see Bengio et al., 2021). Apart from learning heuristics, DRL has also been applied in CO to enhance classical CO algorithms. For example, in Lodi et al., 2017 and in Gupta et al., 2020, the authors use a particular form of DRL called imitation learning to learn a branching strategy for a branch-and-bound algorithm after formulating the problem as a mixed integer linear programming model. In Zhao et al., 2021 and in Ye et al., 2023, DRL is used to learn a strategy for taking decisions in classical heuristic or metaheuristic methods.

An important breakthrough in the field of NCO was the introduction by Dai et al., 2017 of graph neural networks (GNNs), a class of DNNs that leverage the graph structure of data (see Scarselli et al., 2009). This class of DNNs are capable of taking into consideration the graph structure for CO problems defined over graphs, leading to better results

than Pointer Networks. Attention, a kind of DNN that extracts important relationships between pieces of data given some form of context, was made popular in the field of NCO by Bello et al., 2017; Deudon et al., 2018; Nazari et al., 2018; Kool et al., 2019, and demonstrated a good performance.

Since the work of Kool et al., 2019, the number of papers that can be classified as NCO has been growing rapidly, as CO problems pose significant challenges. Many of these papers focus on finding the DNN that best captures the structure of CO problems. We argue that there are other issues to consider when applying DRL methods to CO problems, and that a better understanding of the NCO field can only be achieved through the collaboration of researchers who are familiar with CO. Thus, in this paper, we aim to equip readers who are not familiar with DRL with all the tools necessary to comprehend recent advances in NCO, and to identify the issues that have been studied and those that are yet to be explored.

From the perspective of operations research, progress in NCO may offer better heuristics to already studied problems or serve as a heuristic for problems that currently lack one. The data-driven nature of DRL may also result in heuristics that are better suited for a specific use case. Additionally, since DRL is based on the theory of Markov Decision Processes (MDP), it can be extended to stochastic and dynamic settings.

From the perspective of DRL, NCO offers a new set of problems to be solved, that are already well studied, with a mathematically defined objective function, as opposed to other fields where the goal is not always expressed in mathematical terms (like in gaming or robotics, where the goal is not expressed as a function, for example "win a game" or "drive a car"). This allows researchers to better understand the behavior of DRL algorithms and to analyze them in a more rigorous way.

The structure of the paper is as follows. Section 2.2 provides a general overview of how Reinforcement Learning (RL) works and introduces key terminology. Section 2.3 discusses the most common DNNs used in NCO and what learning means in the context of DRL. Section 2.4 shows how DRL can be used to create a heuristic for a given CO problem. Throughout Sections 2.2, 2.3 and 2.4, we use the Knapsack Problem (KP) as a running example to clarify theoretical concepts. Section 2.5 presents with numerical examples how NCO has evolved over time to solve the Traveling Salesman Problem (TSP), which is the most studied problem in the field. We discuss open issues and possible future directions for NCO in Section 2.6. We provide an implementation of the first example of Section 2.5 in the GitHub repository https://github.com/DavideTr8/RL4TSP/. The most relevant pieces of code are presented in Appendix A.

2.2 Reinforcement learning

RL is a computational framework for decision-making and adaptive control that is concerned with how an *agent*, that is the entity interacting with an *environment*, ought to take actions in order to maximize a reward. The interaction between the agent and the environment happens in discrete time steps.

Environment is a generic term. From the point of view of an agent, the environment is a black box that, at each time step, receives the agent action and returns some data called *observation* and a scalar called *reward.* The agent is usually a piece of software that reads the information sent by the environment (the observation), processes it and responds with an action. By collecting these experiences in the form of $(observation, action, reward)$,

the agent learns how to interact with the environment in order to maximize the cumulative reward.

RL is a very broad field with techniques that can be applied to environments accepting both a discrete and a continuous set of available actions, with a finite or an infinite time horizon or in settings with multiple agents cooperating to reach a goal or playing against each other in a game.

Despite the general nature of RL, for the sake of clarity, we restrict the attention to environments with discrete actions, discrete time steps, with a finite horizon, bounded rewards and with a single agent. Within this particular setting, the environment is modeled as an MDP. Hence, the fundamental concepts underlying RL terminology and formalism are derived from the theory of MDPs. This section provides an introduction to MDPs and RL.

2.2.1 Markov decision process

We introduce the basic elements of an MDP. The notation used here is mostly taken from Sutton et al., 2018, one of the most influential books on RL.

An MDP serves as a mathematical framework for modeling sequential decision-making within a discrete stochastic environment. It encompasses the following elements:

- $\mathcal{S}$: the set of all possible environment states,
- $\mathcal{A}$: the set of available actions,
- $\mathcal{R} \subset \mathbb{R}$: the set of possible rewards,
- $p(s' \mid s, a)$: the transition probability function, representing the probability of transitioning from state $s \in \mathcal{S}$ to state $s' \in \mathcal{S}$ upon taking action $a \in \mathcal{A}$,
- $p(r \mid s, a)$: the reward probability function, that is the probability of obtaining reward r upon executing action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$.

The existence of two probability functions, one for state transitions and the other for rewards, acknowledges the fact that the agent lacks prior knowledge of the outcomes resulting from its actions.

Throughout the exposition, we employ the subscript $t = 0, 1, 2, ...$ to indicate the time step when encountering a state, action, or reward. For example, s_t denotes the state visited at time step t.

Notice that the transition probability function for a state s' only depends on the current state s and on the action a. This is called Markov property, and it means that the state s_{t+1} only depends on the state s_t and the action a_t, and not on how the MDP evolved in the past. The Markov property can be read also as the fact that looking at states in the past gives no additional information for choosing action a_t when visiting state s_t. For example, the game of chess has the Markov property, as one can choose an action only by looking at the current position of the pieces on the board and without knowing how that particular arrangement was reached.

We now introduce the KP example and formulate it as an MDP. Given a finite set of items $I = \{1, 2, 3 \dots N\}$, where each item i has an associated value u_i and a weight w_i, and given a capacity $W \in \mathbb{R}^+$ for the knapsack, the KP is the problem of finding the set $S \subseteq I$ such that we maximize the total value $\sum_{i \in S} u_i$ without exceeding the knapsack capacity, i.e. $\sum_{i \in S} w_i \leq W$.

We can formulate the KP as an MDP as follows. First, we need to describe it as a sequential decision problem. We introduce time steps t and, for each time step, the state s_t is represented by the set of items already inserted in the knapsack B_t, the set of all items I and the residual capacity of the knapsack W_t. The action a_t is the item to insert next. Since we have a constraint on the capacity, only items that do not exceed the capacity are available actions. The reward r_{t+1} for action a_t is the value of the chosen item u_{a_t}. Since we know exactly the state transition and the reward we obtain after an action, the probability functions $p(s' \mid s, a)$ and $p(r \mid s, a)$ are trivial. The transition probability function gives probability 1 to the state s_{t+1} that is obtained by adding item a_t to the knapsack and subtracting its weight from the residual capacity and probability 0 to all other states. The reward probability function gives probability 1 to the reward $r_{t+1} = u_{a_t}$. Finally, the process ends if all the items have been added to the knapsack or if no other item can be added due to the capacity constraint. We will use this formulation throughout the rest of the tutorial.

We point out that this is not the only possible MDP formulation for the KP. For example, we might decide that the state is defined similarly as above, but at each time step the environment also sends as observation one of the items not yet added that fits the capacity constraint and the action the agent performs is to decide whether that item should be added to the knapsack or not.

2.2.2 Elements of reinforcement learning

In RL, the state, the probability of transitioning from one state to another, and the reward probability function compose the environment. The agent is the entity that performs actions. The agent receives information about the current state of the environment in the form of an observation. In this tutorial we assume that the environment is fully observable, meaning that the observation of the agent is the state of the environment. Hence, we use the terms state and observation interchangeably. This does not always hold in RL.

The evolution of the environment from a starting state to a terminal state through the interaction with the agent is called an *episode.* T is the value that indicates the last step of an episode and is usually called episode length or horizon. This value is usually unknown in advance and might depend on how the episode evolves. The sequence of (s_t, a_t, r_{t+1}) triplets - *(observation, action, reward)* - collected during an episode is called a *trajectory.* This process is depicted in Figure 2.1. For example, an episode for the KP starts with an empty knapsack as state and ends when there are no more items to add due to the capacity constraint. Given a KP instance, the value of T is not known, and we only know that is finite, since $T \leq N$.

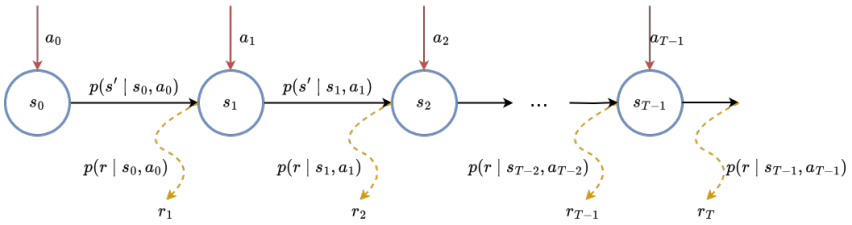


Figure 2.1: An episode. The environment starts from the initial state s_0, receives action a_0 and transitions to state s_1 emitting reward r_1. The process repeats until a terminal state is reached. The sequence $(s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_{T-1}, a_{T-1}, r_T)$ is the trajectory of the episode.

In RL the agent takes actions according to a policy, usually represented with π. $\pi(a \mid s)$ is a function that takes as input a state and an action and returns the probability of taking action a given state s. It is also common to write $\pi(\cdot \mid s)$ to represent the function that takes as input a state and returns the probability distribution over the set of available actions $\mathcal{A}$. This probability can be seen as the degree of confidence that the agent has in choosing a certain action. When the policy is defined, we say that the agent acts according to its policy π. For example, a policy for the KP could be the function that, at each step t, gives probability 1 to add the most valuable item that fits the residual capacity and probability 0 to all other items. If n items have the same value, each one gets probability $\frac{1}{n}$ (*greedy policy*). Another policy could be the function that gives probability $\frac{1}{k}$ to all the items, where k is the number of items that fit the residual capacity (*random policy*).

After receiving an action, the environment transitions to a new state according to its transition probability function and emits a reward. The cycle of interactions between the agent and the environment is depicted in Figure 2.2.

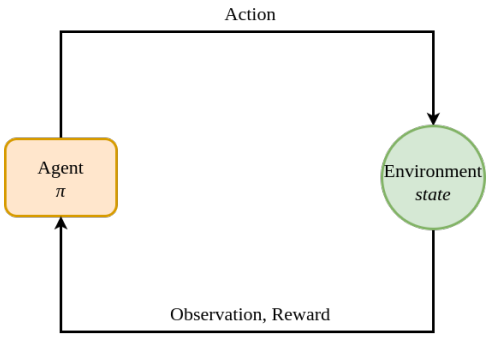


Figure 2.2: The agent-environment interaction. At each time step, the agent receives an observation from the environment, and based on that observation, takes an action according to its policy π. The environment then transitions to a new state and emits a reward. The cycle repeats until a terminal state is reached.

Let us introduce, for each step t, the discounted cumulative reward (usually also called *discounted return*) as

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-1} R_T \tag{2.1}$$

where $\gamma \in (0, 1]$ is called *discount factor* and R_t is a random variable such that $R_t \sim p(r \mid s_{t-1}, a_{t-1})$. Then the goal of a RL agent is to find a policy that maximizes the total

expected discounted cumulative reward:

$$\pi^* = \arg\max_{\pi} \mathbb{E}_\pi \left[G_0\right] = \arg\max_{\pi} \mathbb{E}_\pi \left[\sum_{t=0}^{T-1} \gamma^t R_{t+1}\right]. \tag{2.2}$$

Due to the Markov property, the optimal policy is also the policy that maximizes

$$\pi^* = \arg\max_{\pi} \mathbb{E}_\pi \left[G_t\right] = \arg\max_{\pi} \mathbb{E}_\pi \left[\sum_{t=\bar{t}}^{T-1} \gamma^t R_{t+1}\right] \quad \bar{t} = 0, 1, \ldots, T-1. \tag{2.3}$$

By looking at (2.3), we can understand the meaning of the discount factor γ. Indeed, the discount factor γ determines how much future rewards are devalued compared to immediate rewards. A value of γ close to 0 makes the optimal policy *greedy*, considering only immediate rewards, while a value close to 1 makes it *farsighted*, giving importance also to rewards obtained in the future. For instance, with $\gamma = 0$, $G_t = R_{t+1}$ and thus the optimal policy is the one that, at each step t, takes the action that maximizes the next expected reward. With $\gamma = 1$, instead, $G_t = R_{t+1} + R_{t+2} + \cdots + R_T$ meaning that the optimal policy takes into account all the future expected rewards (with equal weight) when taking an action. Finally, if we set $\gamma = 0.5$ we obtain $G_t = R_{t+1} + \frac{R_{t+2}}{2} + \frac{R_{t+3}}{4} + \ldots$, giving full importance to the immediate expected reward, half of the importance to the next one, and so on. Usually γ has a value close to one, in the range $[0.9, 1]$.

Once a policy is given, a *value* can be assigned to each state of the MDP. Given any time step t, the value $v_\pi(s)$ of a state s is the expected discounted return of the MDP starting from state s and following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s].$$

For the KP example, if we set $\gamma = 1$, the value $v_\pi(s)$ is the sum of the values of the items we add to the knapsack from state s until the end of the episode following policy π.

Similarly to what we have done for the value, we can define the Q-value for each state-action pair (s, a) at any time step t, that is the expected discounted return of the MDP starting from state s and following policy π after taking action a as

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a].$$

For the KP example, if we set $\gamma = 1$, the Q-value $q_\pi(s, a)$ is the sum of the values of the items we add to the knapsack from state s until the end of the episode, knowing that we have added item a after visiting state s.

A partial ordering can be defined over policies, where policy π is said to be greater than or equal to policy π' if the expected return using π is greater than or equal to the expected return using π' from every state $s \in \mathcal{S}$, i.e.

$$\pi \geq \pi' \iff v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S}.$$

The *optimal policy* π^* is a policy such that $\pi^* \geq \pi, \ \forall \pi$. The proof of the existence of such policy can be found in Puterman, 2014 and goes beyond the scope of this tutorial. Following the optimal policy, the optimal value and optimal Q-value functions can be defined as

$$v^*(s) = v_{\pi^*}(s) \text{ and } q^*(s, a) = q_{\pi^*}(s, a).$$

This means that finding the optimal policy is equivalent to finding the optimal Q-value function.

2.2.3 Solving an MDP in reinforcement learning

RL is a framework used to solve MDPs. Solving a problem framed as an MDP means finding the optimal policy π^*. There are two main issues that make this task challenging:

- The state space $\mathcal{S}$ and the action space $\mathcal{A}$ can be very large. To find the optimal policy, the agent should visit all possible states and take all possible actions. With a large state and action space, this is not feasible in a reasonable amount of time.
- The transition probability function $p(s' \mid s, a)$ and the reward probability function $p(r \mid s, a)$ are unknown to the agent. With unknown transition probabilities, the agent cannot compute the expected cumulative reward of a trajectory. This is the most common scenario in RL.

In both cases the agent needs to find a good approximation of π^* in a reasonable amount of time. Achieving this goal is not trivial and is done by balancing two conflicting strategies: *exploration* and *exploitation.* Exploration means visiting new states and taking novel or unpromising actions to gather information about the environment and gain a better estimate of the expected value in (2.2) Exploitation means using the knowledge acquired to avoid visiting unpromising states and save time. The problem of balancing these two conflicting strategies is known as the *exploration-exploitation dilemma.* For example, in the KP, let us suppose that an agent adds items $1, 2$ and 3 as its first three actions. Then it performs other actions until the end of the episode and the episode ends with a bad cumulative reward. Then, given the same instance, the agent adds items $4, 5$ and 6 as first actions. Then it plays other actions until the episode ends. This time the episode ends with a good cumulative reward. Given the same instance again, should the agent try again to start an episode by adding items $1, 2$ and 3 exploring new episodes with these starting actions, even if they led to a bad cumulative reward the first time, or always start by adding items $4, 5$ and 6, exploiting the fact that those actions led to a good cumulative reward the first time?

The experiences collected by interacting with the environment according to the current policy π are then used to refine the policy.

2.2.4 Reinforcement learning with parametric functions

Modern RL is *data-driven*, meaning that the agent *learns* by interacting with the environment and collecting data. The word *learns* is used here to indicate that the agent is provided a parametric function, with parameters θ, and is able to improve its performance over time by updating the parameters of such parametric function using data. There are also non-parametric methods in RL, but we do not cover them in this tutorial.

When the parametric function is the policy, meaning that the agent has to learn π_θ, the method used is called *policy-based*. When the optimal Q-value is learned instead (i.e. Q_θ), the method is called *value based* and the optimal policy can be obtained by choosing the action that maximizes the Q-value in each state. Learning the Q-value $Q_\theta(s, a)$ estimates the expected future rewards for each action in a state, deriving the policy by selecting actions that maximize these values. Learning a policy directly optimizes the action selection probabilities $\pi_\theta(a \mid s)$, without learning the Q-value of state-action pairs. The former indirectly optimizes the policy through value estimation, while the latter directly targets the policy. This has practical implications, for instance on how the data collected for learning are used. Nowadays, methods that learn both the policy and a value

function are the most common and are called actor-critic methods (see Mnih et al., 2016; Schulman et al., 2017). Another emerging class of methods is called *model-based* and aims to learn also the transition probability function $p(s' \mid s, a)$ and the reward probability function $p(r \mid s, a)$ (see Schrittwieser et al., 2020). In this tutorial, we focus solely on policy-based methods, since they are the most common in NCO. Moreover, we believe that a good understanding of policy-based methods can help the reader understand the other classes of methods as well.

In policy-based RL, the agent wants to find a good approximation of the optimal policy π^*. To do so, the agent interacts with the environment. The interaction happens through multiple episodes. With each episode, the environment provides a new instance of the problem to solve. These instances are used to collect the data necessary to update the policy. Once the policy is updated, the agent is ready to solve new instances of the problem. The process terminates when a termination condition is met (e.g. when the agent has solved a certain number of instances). Hence, the first step in order to define a RL framework, and to subsequently solve the corresponding problem, is to properly define the environment in which the agent operates.

The environment can take the form of a simulator, a real world system, or a combination of both.

Typically, the agent interacts with a simulator, as synthetic data are often easier to gather. Subsequently, the agent is deployed in the real world, where the policy is further refined to adapt to the actual system (see Kaufmann et al., 2023).

Regardless the case, the design of the MDP describing the environment plays a crucial role in defining a RL framework.

2.2.5 Solving an RL problem with parametric policy

As previously stated, we focus on policy-based RL, where the agent learns to approximate the policy using a parametric function π_θ. The parameters of such function are learned through the interaction with the environment. This means finding the parameters that minimize a certain *loss function.* Notice here that, even if the goal in RL is to maximize the expected discounted return, it is more common to re-frame the problem as a minimization problem by using a loss function that depends on the policy parameters and such that its minimization corresponds to the expected discounted return maximization. The convergence theory that underlies this approach is beyond the scope of this tutorial and can be found in Sutton et al., 2018 and Powell, 2022. The process of interacting with the environment and of subsequently using the collected data to update the parameters of the policy is called *training* of the agent.

Therefore, training can be subdivided into two phases: data collection and parameters update. During the data collection phase, the agent interacts with the environment until a certain condition, that we call *update condition*, is met. An example of such condition is the completion of an episode. During this phase, the environment state and the reward are computed after each agent action. After every episode, the environment provides a new instance of the problem to solve. Agent actions are sampled according to the current policy π_θ. In this phase, the parameters of the function approximator are fixed. Through interaction, the agent collects triplets $(state, action, reward)$. These data are usually stored in a *buffer*, a data structure that stores the data collected by the agent. When the update condition is reached, the data collection phase stops and the update phase starts. In this phase, the parameters θ of the policy π_θ are updated. To do so, collected data are

used to compute the value of the loss function. Once the loss function is computed, the agent uses a global optimization algorithm to update its policy parameters. Nowadays, the most common global optimization algorithms are first-order gradient-based methods like stochastic gradient descent. At this point, the updated policy is used to interact with the environment again, starting a new data collection phase. The process is then repeated until convergence (see Section 2.2.6).

Explicitly defining all the above steps, e.g. the loss function, the optimization algorithm, the training condition and the parametric functions to learn, is what differentiates one RL algorithm from another. A taxonomy of the most commonly used algorithms in the field of NCO is provided in Mazyavkina et al., 2021. Every RL algorithm can be broken down into two main sub-algorithms, each responsible for handling one of the aforementioned phases. The first one is responsible for the data collection, while the second one is responsible for the parameters update, where the parameters of the parametric policy are updated using an optimization algorithm (e.g. stochastic gradient descent).

The training pipeline is summarized in Figure 2.3. The agent is represented as a purple box that contains the policy and the optimization algorithm. The agent takes as input the observation directly from the environment, represented as a green box, and outputs an action using its policy. This action, the received observation together with the reward that the environment emits are stored in the buffer (the gray cylinder). The solid arrows indicate the data communication. Whenever the update condition is reached, the agent updates the policy parameters by using the data stored in the buffer and the update algorithm. The update process is represented with dotted arrows.

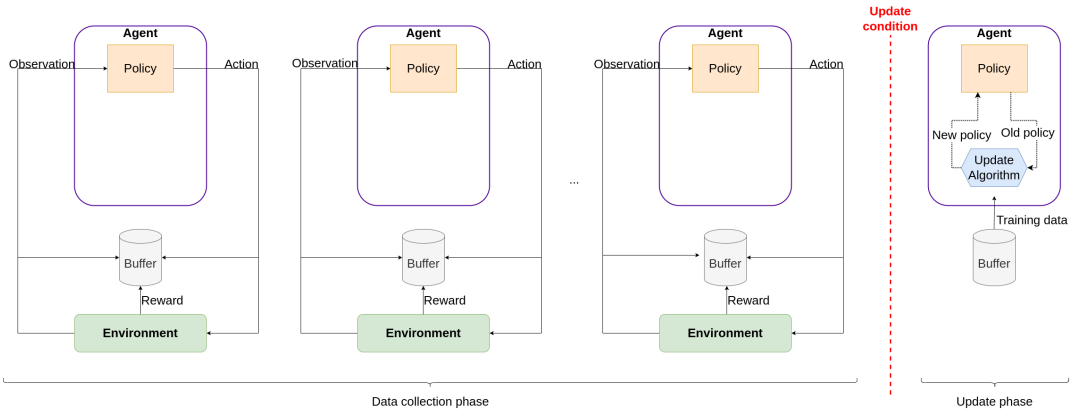


Figure 2.3: One iteration of the training pipeline of a RL agent.

2.2.6 Convergence condition

We mentioned that the training should continue until convergence. However, in RL, there is no straightforward way to determine whether an agent is actually learning or has learned the optimal policy (see Sutton et al., 2018). Although certain metrics, such as the total return of a trajectory at each episode, can be monitored to get an idea of how the training is progressing, they provide only partial information about the agent behavior. Thus, determining when to stop an RL training session remains an open question. Usually, the training is stopped when there are signs that the agent performances are no longer improving, for example if the policy parameters are not changing between iterations or the return for some test instances is not improving anymore.

2.2.7 Using a trained agent

Once training is completed, the agent can be used to solve new instances of the problem. The process is similar to the data collection phase but usually, instead of sampling the action according to the action probabilities, at each time step t the agent takes the actions with the highest probability $a_t = \arg\max_a \pi_\theta(a \mid s_t)$.

2.3 Deep reinforcement learning

In the previous section, we presented the process of defining a RL framework by describing the underlying MDP and how to solve it using a parametric approximation of the policy. In this section, we introduce the basics of DRL, the approach to RL that uses DNNs as function approximators. As stated above, a policy is a function that maps a state to a probability distribution over the actions. A common approach of DRL is to use DNNs to approximate the policy. The aim of this section is to explain the basics of DNNs and how their parameters are updated, in order to subsequently use them as function approximators for the policy π_θ.

The interest towards DRL stems from the capability of DNNs to capture the underlying structure of data like images (see He et al., 2016), time series (see Wu et al., 2023 or textual data (see Vaswani et al., 2017). The success in these fields led to the idea that Deep Learning, i.e. the subfield of Machine Learning that leverages DNNs as functions approximators, could be used to solve other hard problems, like CO problems. Furthermore, RL serves as a framework that seamlessly integrates the strengths of Deep Learning while retaining a solid theoretical foundation as an optimization framework.

The first step to understand DRL is to understand how DNNs work.

2.3.1 Deep neural networks

In this section, we aim to provide an accessible introduction to DNNs for those unfamiliar with the field, without going into excessive technical details of various DNN.

In the context of DNNs, we call *parameters* all the values that need to be learned during the training process (that, in the context of Section 2.2, corresponds with the values of θ in the parametric function π_θ), and we use the term *hyperparameters* for all the values that are set by the user before the training process. Some common hyperparameters are the number of episodes to play before the training process starts, the total number of episodes and all the values regulating the shape of the DNNs involved.

DNNs can be thought of as parametric function approximators created by the composition of almost everywhere differentiable functions with respect to the parameters. These functions are typically linear or simple non-linear transformations. Each of the functions in the composition is called with the generic term *layer*. In this composition there are two special layers, that are the input and the output layer. The input layer is the layer that takes as input the data to be processed, while the output layer is the layer that outputs the result of the computation. Every layer and every result that happens between the input and the output layer is preceded by the adjective *hidden*. A hidden layer is a layer that is neither the input nor the output layer and a hidden vector is the input or output of a hidden layer. Typically, the notation $\boldsymbol{h}$ is employed to represent hidden vectors.

There are several classes of layers, and the most common ones found in the NCO literature are listed below. For a more comprehensive coverage, interested readers may refer to Goodfellow et al., 2016 and for a review of the relationship between the DNN and the type of data or problem at hand to Battaglia et al., 2018. Indeed, the choice of a layer is determined by the kind of data that the layer is processing. For instance, the layer used for processing images is different from the layer used for processing sequences or graphs.

We now introduce the most common layers used in the NCO literature, and we briefly explain for which kind of data they are more suited. Every layer we introduce can be used as an input layer, as a hidden layer or as an output layer. As stated above, it is common to denote with $\boldsymbol{h}$ a hidden vector. On the other hand, when referring to the input or output of a single layer, it is more common to use letters x and y, respectively. For example, if we have a single vector as input and a single vector as output, we use $\boldsymbol{x}$ and $\boldsymbol{y}$ to denote the input and the output of the layer. If we have data represented as a sequence of vectors both as input and as output, we use $x = (\boldsymbol{x_1}\ \boldsymbol{x_2}\ \dots\ \boldsymbol{x_N})$ and $y = (\boldsymbol{y_1}\ \boldsymbol{y_2}\ \dots\ \boldsymbol{y_N})$ to denote the input and the output of the layer, where N is the length of the sequence. We use letter i to denote the index of a generic vector in the sequence. We also denote as n the dimension of each vector in the input sequence and m the dimension of each vector in the output sequence. Finally, if we have data represented as a graph G, with nodes labeled from 1 to N, we use letter i to denote a generic node. The entries of node i are denoted as $\boldsymbol{x_i}$ for the input and as $\boldsymbol{y_i}$ for the output. As it is common in deep learning, we use the term *features* to refer to the entries of the input vector.

The most common layers are:

- Linear layer: first introduced by Rosenblatt, 1958, a linear layer is a linear function that maps the input $\boldsymbol{x} \in \mathbb{R}^n$ to the output $\boldsymbol{y} \in \mathbb{R}^m$ where $n \times m$ is the size of the layer, see Figure 2.4.

  The dimension of the output vector m is a hyperparameter to set when defining the layer. A linear layer can be thought of as a matrix multiplication between the input vector and a matrix $\Theta \in \mathbb{R}^{m \times n}$, i.e. $\boldsymbol{y} = \Theta \boldsymbol{x}$. Entries of the matrix Θ are the parameters of the layers that are learned during the training process. This layer is the most generic and can be used every time data can be organized in a single vector.

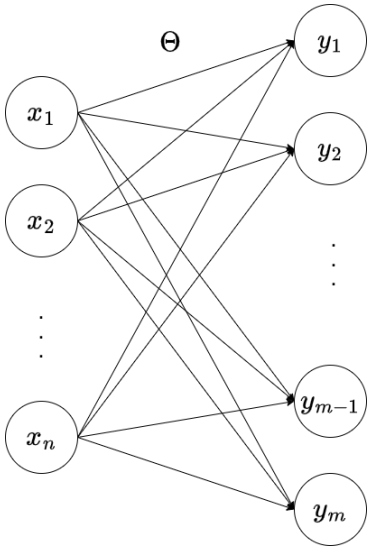


Figure 2.4: A linear layer. $\boldsymbol{x} \in \mathbb{R}^n$ is the input and $\boldsymbol{y} \in \mathbb{R}^m$ is the output.

- Activation layer: this layer was introduced together with the linear layer by Rosenblatt, 1958. An activation layer is a layer that applies a non-linear function to the input vector $\boldsymbol{x}$. The output of an activation layer is a vector $\boldsymbol{y}$ of the same size of the input vector $\boldsymbol{x}$ ($m = n$). Activation layers are used to introduce non-linearity in the DNN, and they are usually applied after a linear layer. A very useful activation layer is the softmax layer, that takes as input a vector $\boldsymbol{x} \in \mathbb{R}^n$ and outputs a vector $\boldsymbol{y} \in \mathbb{R}^n$ where the entries of $\boldsymbol{y}$ are non-negative and sum to 1

  $$y_k = softmax(\boldsymbol{x})_k = \frac{e^{x_k}}{\sum_{j=0}^{n} e^{x_j}}.$$

  The softmax layer is often used to map the output of a DNN to a vector of probabilities.

- Recurrent layer: introduced by Hopfield, 1982, a recurrent layer is often called *recurrent block* or *recurrent cell*. Recurrent layers are designed to handle sequences of data, such as time series or sentences. What makes these layers unique is their ability to remember previous inputs through a mechanism called the *hidden state*. This hidden state acts like a memory, storing relevant information from earlier steps in the sequence to help make decisions later on.

  A common example of their usage is to predict the next word in a sentence. At each step, the recurrent layer processes one word at a time and updates its memory with the new word. This allows to keep track of important information from the earlier words, like the subject and verb, to better predict the next one. Each step depends not only on the current word, but also on what the layer has remembered from previous words. The hidden state is updated by combining the current input (the word being processed) with the previous hidden state (what the network remembers). This updating happens at every time step, allowing the recurrent layer to process sequences of arbitrary length. This process is described in Figure 2.5.

  More technically, the input of a recurrent layer is a sequence of vectors. The recurrent layer acts sequentially on each vector of the sequence, and takes as input not

only the current vector but also the output computed by the same layer with the previous vector in the sequence. The output at each step is a vector called hidden state.

The dimension d of the hidden state, usually called *hidden state size*, is a hyperparameter to set when defining the layer. This hyperparameter is usually set experimentally by trial and error. At time step 1, the hidden state is usually initialized to a vector of zeros of dimension d.

A strength of recurrent layers is that they can be applied to sequences of different length, since they are not dependent on the length of the sequence. Notice that this is not true for linear layers, where the size of the input vector must be known a-priori to create the matrix of parameters.

Another strength of recurrent layers is that they can be used *online*, meaning that they can process a sequence of data one element at a time, without having to wait for the whole sequence to be available.

Recurrent layers are a class of layers, where each differs from the others by the set of operations they perform on the input data. The most famous recurrent blocks are the Long Short Term Memory (LSTM), introduced by Hochreiter et al., 1997, and the Gated Recurrent Unit (GRU), introduced by Cho et al., 2014.

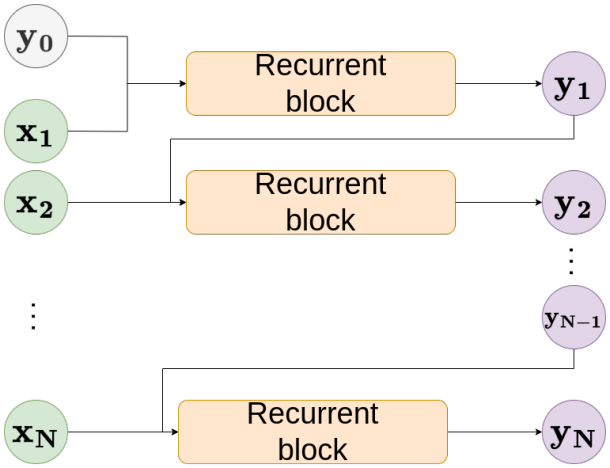


Figure 2.5: A recurrent block mapping a sequence of vectors $\{\boldsymbol{x_i}\}_{i=1}^{N}$ to the output sequence $\{\boldsymbol{y_i}\}_{i=1}^{N}$. At each iteration i, the input for a recurrent block is the element $\boldsymbol{x_i}$ in the input sequence and the output of the previous step $\boldsymbol{y_{i-1}}$. The output of the recurrent block is the next hidden state $\boldsymbol{y_i}$. $\boldsymbol{y_0}$ is the starting hidden state.

- Message passing layer: making their first appearance for applications in quantum chemistry (see Gilmer et al., 2017), message passing layers act on data that are described on a graph.

  They allow nodes in a graph to exchange information with their neighbors. Each node is initially described by a vector of features, like its coordinates. During the message passing step, every node sends its features to its neighbors and receives information from them in return. The node then updates its features by combining what it knew before (its own features) with what it learned from its neighbors (the neighbors features). The result is a new vector of features that should better

describe the node and its relationships in the graph. For example, it might contain information regarding the distance from its neighbors or the coordinates of the closest neighbors.

To recap, a message passing layer takes as input a graph G and outputs a new graph G' with the same nodes and edges but with updated node features, see Figure 2.6. As opposed to recurrent layers, message passing layers are not sequential, since they act on all the nodes of the graph at the same time (in parallel).

Message passing layers aim to leverage the invariance of data under geometric transformations and permutations. Simultaneously, they seek to capture the underlying graph structure, specifically the relationships between nodes as expressed by the edges. Hence, message passing layers are more suited when data are organized in a graph and each piece of data has a relationship with the other pieces, and these relationships can be expressed using the edges of the graph.

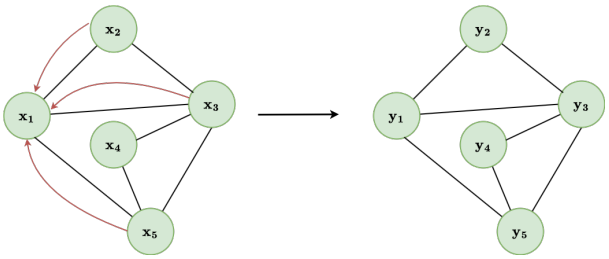


Figure 2.6: The effect of a message passing layer is to update the features for each node. For example, the layer computes the new features $\boldsymbol{y_1}$ using the features $\boldsymbol{x_1}$ of node 1 itself and of its neighbors $\boldsymbol{x_2}, \boldsymbol{x_3}$ and $\boldsymbol{x_5}$. The contribution of the neighbors of node 1 is represented with red arrows.

- Attention: Attention layers, or attention mechanisms, became famous after their use on the Transformers (see Vaswani et al., 2017), that signed the start of large language models. In this tutorial, we have simplified the description of attention layers and the terminology used does not align with that found in the literature.

  A common example of attention is in natural language translation. The attention layer takes as input two sets of data, for example words in English and words in Spanish. The attention layer gives a score to each (English word, Spanish word) couple, where a high score means that the two words have similar meanings.

  More technically, the input of an attention layer are two sets of vectors, the input data and the target data. If we take an input set with M vectors, a target set with N vectors, then the output of an attention layer is a matrix of weights with dimension $N \times M$. Each column of the matrix is a vector of non-negative numbers that sum to 1, that can be regarded as the probability distribution of the importance of the corresponding element of the input data for the corresponding element of the target data (see Figure 2.7).

  An important example of attention layers are self-attention layers (see Vaswani et al., 2017) where the target data are the input data themselves. For example, a self-attention layer used on a sentence should assign high importance to the entries corresponding to the subject-verb relationship, and low importance to subject-adverbs or subject-particles relationships.

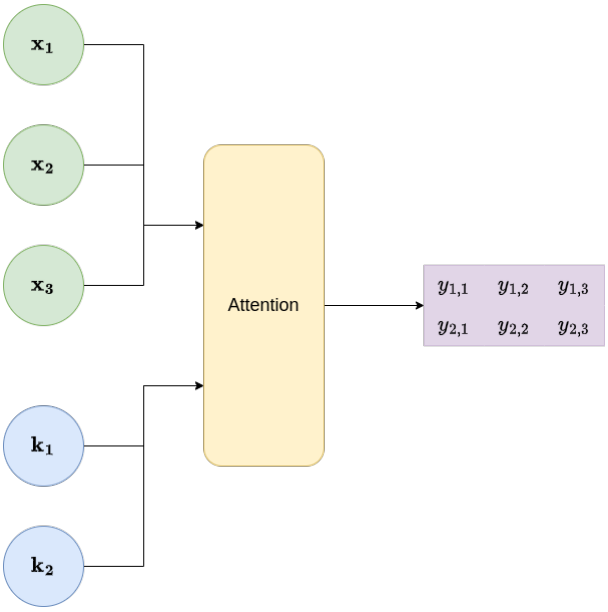


Figure 2.7: The effect of an attention layer to 3 input vectors $\boldsymbol{x_1}, \boldsymbol{x_2}, \boldsymbol{x_3}$ and 2 target vectors $\boldsymbol{k_1}, \boldsymbol{k_2}$. The output of the layer is a matrix of weights $\{y_{i,j}\}$ with dimension $N \times M$.

In general, an attention layer tries to find relevant relationships between the elements of the input data and the elements of the target data. They are used when pieces of data have a relationship between each other but this relationship is not known. Differently from recurrent layers where the relationship is given by the order in the sequence or message passing layers where the relationship is expressed using edges, the attention layer itself is responsible of finding the hidden structure in data.

The kind of layers used in a DNN determines the class of the DNN. Some of the most common classes of DNNs are listed below.

- Multi-Layer Perceptron (MLP): sometimes also called *vanilla neural network*, an MLP is a DNN that consists of the composition of linear layers and activation layers (see Figure 2.8). Since the MLP is composed solely of linear and activation layers, the input and output dimensions for this kind of DNN must be defined a priori.

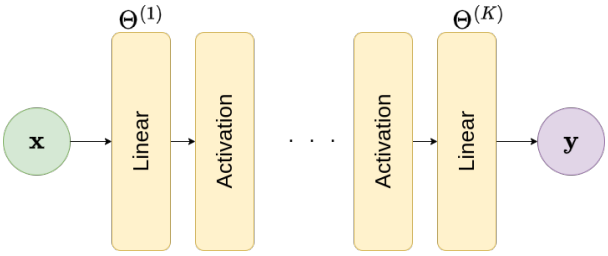


Figure 2.8: An MLP network made by alternating K linear layers and K activation layers.

- Recurrent Neural Network (RNN): this class of networks consists of the composition of linear, activation and recurrent layers. A simple example of RNN is shown in

Figure 2.9, where the recurrent layer is followed by a linear layer. The linear layer acts on the hidden state of the recurrent layer and outputs the result of the computation. The input and output of this DNN are still a sequence of vectors.

The RNN usually takes its name from the recurrent layer used. For example, an LSTM network is an RNN that uses LSTM layers.

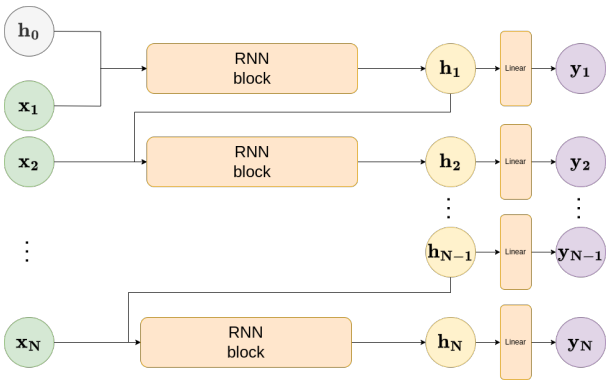


Figure 2.9: An RNN obtained by the composition of a recurrent layer and a linear layer.

- Graph Neural Network (GNN): GNNs operate on data described on a graph, where each node or edge of the graph has a set of features. This class of DNNs tries to capture and use the graph structure as an information for processing the data. GNNs usually work *node-wise* or *edge-wise*, aggregating information of nodes and edges with operations that do not depend on the order or number of nodes and edges considered, but only on the graph structure (i.e. its adjacency matrix). An important class of GNNs are *message passing* GNNs, that use message passing layers to update the features of the nodes of the graph. We do not discuss other kinds of GNNs in this paper. Message passing GNNs (that, from now on, we will call GNNs) exploit the graph structure of data and the invariance from geometric transformations that message passing layers have.

When multiple DNNs are combined, the way they interact is called *architecture* or *model*. For example, an RNN might be combined with an MLP, or a GNN can be combined with an attention layer.

A very common architecture in the field of NCO is the *encoder-decoder* architecture. In this architecture the first DNN, called *encoder*, is used to extract features from the input data, while the second one, called *decoder*, is used to generate the output data from the features extracted by the encoder. Usually the encoder is a DNN responsible to map the input data to a vector of different size able to capture relevant information about the data. This vector is called *embedding*. Since embeddings are regarded as high dimensional representations of the input data that the DNN has to learn, they are commonly called also *learned features*. An embedding is a vector that represents the most important features of the input data in a way that helps the DNN. It captures patterns, relationships, and important details, but these details are often abstract and difficult for humans to interpret directly. During training, the network adjusts the values in the embedding so that similar inputs have similar representations.

The decoder is also a DNN, and is responsible for translating the information contained in the embedding, possibly with additional data, back into a meaningful form, that is the final output.

For example, we could define an encoder-decoder architecture for the KP. The encoder might be an MLP that, at each time step t, takes as input the vector $\boldsymbol{x} = (u_1, w_1, u_2, w_2, \ldots, u_N, w_N, W_t)$ that concatenates all the features of each item and the residual capacity of the knapsack. Then the decoder might be an RNN followed by an attention layer. The RNN takes as hidden state the output of the encoder for the first step and the previous hidden state for all the other steps, and takes as input the features of the last added item. The attention layer takes as input the set of all the items features and as target the output of the RNN (i.e. the hidden state). The final output, at each step, is a probability vector indicating which item to add next.

Deciding what DNN is best suited for a problem is, at the current state of research, a matter of trial and error. The trial and error process begins with selecting the appropriate layers or DNNs based on the problem requirements. For instance, RNNs are a suitable starting point for tasks involving temporal dependencies, while GNNs are effective when the problem can be modeled on a graph. If understanding relationships between different inputs is essential, attention mechanisms may be a good starting choice. Ultimately, these DNNs must be combined to produce an architecture that outputs the desired output vector. Although some works aim to automate this process (see Elsken et al., 2019), in DRL it remains largely manual, typically beginning with architectures from the literature on similar problems and adapting them as needed. Once some candidate architectures have been identified, they are trained and tested to assess their performance by means of metrics that depend on the problem at hand. For example, in CO, the metrics might be the mean gap with respect to the optimal solution for known test instances or the number of times the chosen architecture gives a better objective value than a benchmark heuristic. Other relevant metrics include the time required for training to reach convergence and the time needed to generate a solution given a specific computing power. The interested reader might start from the surveys by Vesselinova et al., 2020 and Liu et al., 2023, where they present the DNNs used in the NCO literature for solving different CO problems.

2.3.2 Learning DNN parameters

As mentioned in Section 2.2.5, the parameters of a DNN are learned by minimizing a loss function. Let us assume that the policy π_θ is a DNN with parameters θ. Thus, learning means finding the proper values of parameters θ. Initially the parameters are set to random values, meaning that the agent will behave randomly. Then, during each update phase, the loss function is computed, and the parameters are updated using the gradient descent algorithm to minimize the loss function. The problem with DNNs is that the gradient of the loss function is hard to compute, since the DNN is the composition of many functions. The backpropagation algorithm, introduced by Rumelhart et al., 1986, is the algorithm used for approximating the gradient. Then, it is common to apply a first-order method to update the parameters θ.

Creating a DNN from scratch, implementing the backpropagation algorithm and performing the update steps are very complex tasks. For this reason, there exist many libraries that provide the building blocks for creating DNNs and that handle the optimization process automatically. The most common ones, written in Python, are PyTorch (see Paszke et al., 2019) and Jax (see Bradbury et al., 2018).

2.4 Neural combinatorial optimization

NCO is a research field focused on employing DRL for the development of heuristics for CO problems.

As discussed in Section 2.1, previous works have explored the use of DRL in CO. However, a clear definition for NCO is still missing. The term NCO was initially introduced in Bello et al., 2017. Joshi et al., 2022 further elaborated the term, stating that it involves the usage of DRL to create heuristics capable of generating solutions for CO problems. This can be achieved through either an autoregressive approach, where a partial solution is iteratively extended until a complete solution is obtained, or a non-autoregressive approach, where the complete solution is generated in a single step. The following discussion focuses on the autoregressive approach, while referring interested readers to Joshi et al., 2022 for the non-autoregressive approach.

NCO offers several potential advantages. Firstly, it can serve as a heuristic for problems where a conventional heuristic has not been developed yet. Moreover, a DRL-based heuristic may be specialized for instances that resemble the training data distribution, potentially outperforming generic heuristics. The choice of DRL over supervised learning is motivated by the requirement, for the latter one, of ground truth labels for many problem instances, which are typically obtained from exact methods or commercial solvers. Generating such solutions can be time-consuming, especially as the problem size increases. In contrast, RL learns through interaction with the environment, eliminating the need for labeled samples.

Having seen how to use DNNs to approximate a policy and how to train them using DRL in Section 2.3, we now focus on how to design all the elements of Figure 2.3 for NCO.

2.4.1 How to develop an NCO algorithm

Since NCO is a new and evolving research area, there exists no standardized procedure for developing an NCO algorithm.

The process of developing an NCO algorithm is theoretically similar to the one of applying a DRL algorithm for any other problem. A DRL algorithm provides a general framework with predefined steps that must be followed to ensure convergence, such as the definition of the loss function and update conditions. Other steps, instead, are more problem specific and need to be defined or revised each time, like the DNN architecture or the environment. For example, we might decide to use the same DRL algorithm to solve the KP, but try different DNNs at each trial. Indeed, research in NCO has primarily focused on finding DNN architectures that can best represent the structure of CO problems, followed by applying RL techniques to optimize these networks.

The distinctive nature of NCO arises from the fact that the agent operates within an environment represented by an instance of a CO problem. Describing the states, actions, and reward function is nontrivial, and using a DNN to process data representing a CO problem instance formulated as an MDP is not straightforward.

Here is a general overview of the steps involved in applying a DRL algorithm, with a focus on the peculiarities of NCO problems. Applying a DRL algorithm means outlining all the elements introduced in Figure 2.3. Therefore, to formulate an NCO algorithm, one must define the environment, the agent, and the interactions between the two, including the data buffer. We recall here that the DRL algorithm encompasses both the data

collection and the update phase.

- **Define the environment.**

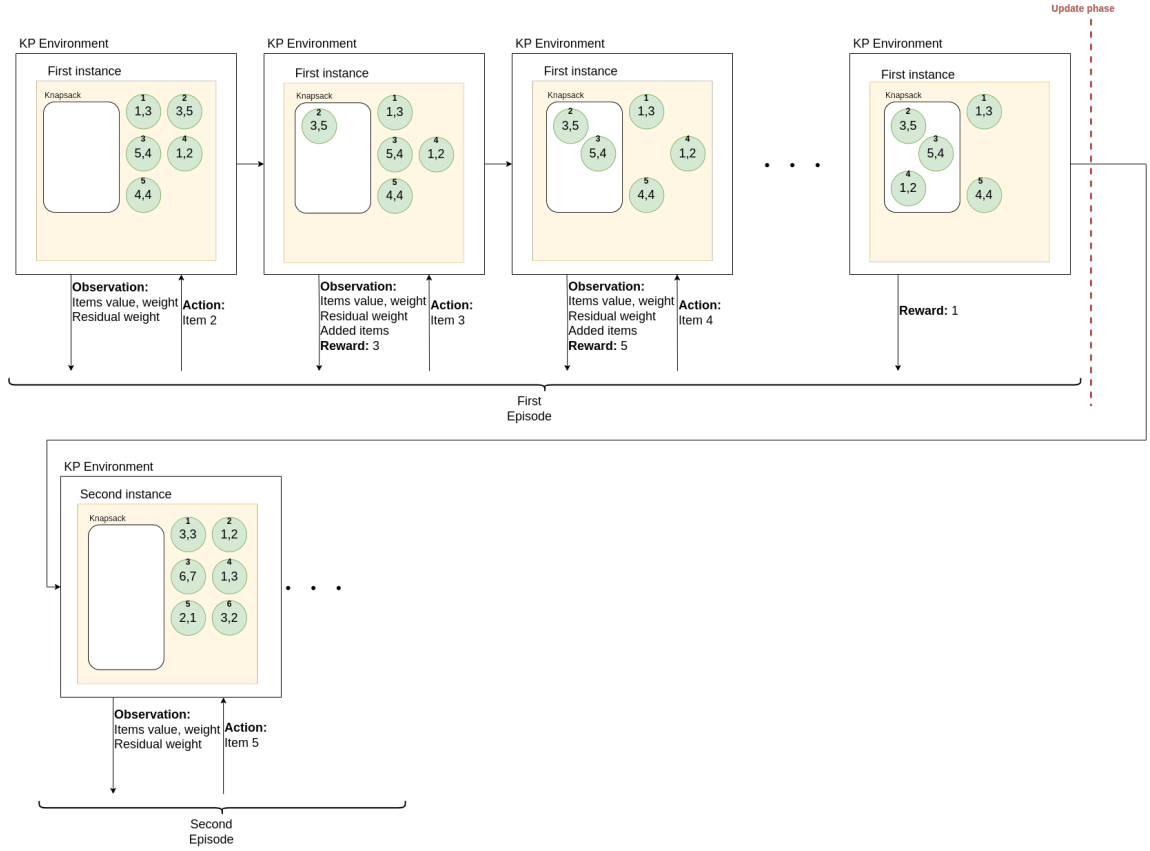


Figure 2.10: The evolution of the environment state for the KP during episodes. In this figure, we suppose that the update phase happens after each episode.

This aspect is generally not discussed in papers that present novel DRL algorithms, as they are often tested on games, where the environment is provided by well-known benchmarks. Similarly, many NCO papers overlook this point, as they revolve around problems that are easy to frame as MDP, like the KP. The behavior of the environment is depicted in Figure 2.10, where the CO problem at hand is a KP.

At the start of every episode, the environment provides a new problem instance as a state. Afterward, whenever the agent takes an action, the environment is responsible of the state transition and provides a new state, which is the instance of the problem with the partial solution found so far by the agent, along with a reward.

In the case of the KP, the environment is the piece of software that generates new instances or, given an action (an item to add to the knapsack), adds it to the set of already added items and computes the new residual capacity of the knapsack. Then, it communicates the updated state to the agent together with the value of the item it added as a reward.

MDP formulation.
Defining the environment requires treating the problem as an MDP, where the state

space is made of all the possible instances of the problem with a partial solution (possibly empty), the action space is made of all the possible actions that the agent can take. This is the first peculiarity of NCO problems: the state space is extremely large, as the agent is facing a different instance of the problem at each episode. Also, the available actions depend on the instance at hand, as the agent can only perform actions that are feasible for the instance.

Moreover, for NCO, the reward is strictly tied to the objective function, and is usually provided only at the end of the episode as the value of the objective function for maximization problems and the negative of the objective function for minimization problems (*sparse reward*). This kind of reward makes it harder for the agent to learn, as it is not clear which actions affected the final reward the most. Hence, it is also common to define a *dense reward*, that is a reward that is provided at each step of the episode, and that describes the quality of the action taken at that step. For instance, in the KP example we defined a dense reward, which is the value of the item added to the knapsack. Defining (*shaping*) a dense reward is not always trivial and a badly designed reward might lead to undesired behaviors of the agent. Also, notice that the definition of the reward is different from other DRL problems, as for NCO there is a clear objective function that the agent has to minimize or maximize.

Modifications to the MDP formulation, such as altering the state representation or reward structure, can influence the design of the DNN, the speed at which the agent learns, and the suitability of different RL algorithms. Unfortunately, to the best of our knowledge, there have been no study analyzing the differences between different MDP formulations for the same problem.

- **Define the agent.**
  The agent is responsible for the interaction with the environment and for learning from its experience. Interaction happens through the policy, that is a DNN that takes as input the state and outputs the action to perform. Learning happens through the update of the DNN parameters θ. The parameters are updated in order to maximize the expected return (or minimize the loss function). Therefore, to define the agent, one must define the DNN architecture, how actions are selected based on the DNN output and how the DNN parameters are updated (update algorithm).

  Defining the DNN architecture means deciding which DNN/DNNs to use and how it/they should process the input data to compute the output. For the implementation of the DNN architecture, a practitioner might resort to one of the packages mentioned in Section 2.2.5. The output of the DNN is a probability distribution over the actions, and usually the action is sampled from this distribution.

  The choice of the update algorithm highly depends on the DRL algorithm chosen. The connection between the DRL algorithm and the agent definition is so strong that an agent is usually called by the name of the DRL algorithm used. For example, an agent that is trained using the algorithm called REINFORCE (see Algorithm 1), is called REINFORCE agent. However, as stated above, an agent is also composed by the DNN architecture, and therefore the same DRL algorithm can be used with different DNN architectures, leading to different agents. The update algorithm is responsible for updating the DNN parameters, and specifies how the loss function

is computed and how the parameters of the DNN are updated.

To recap, an agent can be described by defining its policy (that is a DNN, mapping observations to distributions over actions, and the strategy to sample from this distribution) and the way it learns (that is the update algorithm).

For instance, a possible way to solve the KP using NCO is to define the policy as an MLP that takes as input a vector with all the features of the problem: values and weights of the items and residual capacity at each step. The output is the vector of probabilities for the next item to add. Since the only DNN involved is used to approximate the policy, we can use a policy based algorithm like REINFORCE to train the agent.

- **Define the interactions between the agent and the environment.**
  Choosing the DRL algorithm means choosing the blueprint for the algorithm. This involves all the choices about the agent-environment interaction, for example how data are stored, what is the update condition, what function the DNN approximates (the policy in our case) and, as stated in the previous point, the update algorithm. There does not exist a procedure to follow to choose the right algorithm, and it is often a matter of trial and error, even though some algorithms are more suited for some problems than others.

  There are many DRL algorithms available, the most famous ones for NCO being REINFORCE, Proximal Policy Optimization (PPO) by Schulman et al., 2017, Deep Deterministic Policy Gradient (DDPG) by Silver et al., 2014.

- **Setting hyperparameters.**
  All the above-mentioned choices involve hyperparameters that must be set. An exhaustive list of all the hyperparameters that can be set is not possible. They heavily depend on the DRL algorithm used and the DNNs involved.

It is worth noting that the steps outlined above provide a general understanding of how to develop an NCO algorithm and are not exhaustive. Each step presents various choices and considerations, and the developer must make decisions based on the specific problem at hand. Experimentation and fine-tuning, along with the selection of hyperparameters, play crucial roles in the development process and in the success of the algorithm.

2.4.2 Learning in NCO

A fundamental part in each DRL algorithm is the update algorithm. This phase is responsible for updating the parameters of the DNN based on the collected experiences.

A key challenge in NCO lies in evaluating the agent learning progress. As already mentioned, the state space in NCO problems is usually composed by all the possible instances and partial solutions of a given problem. The agent learning happens by collecting experiences from the environment and using those experiences to reinforce some state-action pairs and to punish others based on the reward. Since the agent has to interact with different instances, the reward it receives depends not only on the quality of the solution, but also on the instance at hand. For this reason, evaluating if an agent is improving or not based solely on the reward is not easy. To assess the performance of an agent, therefore, it is common in NCO to use a *baseline* for comparison. A baseline is an algorithm that can provide solutions to the problem quickly. Common baselines

are the greedy heuristic (if available for the problem) or another NCO agent. When an agent is solving an instance of a problem, the same problem is solved using the baseline and the cumulative reward obtained by the agent and the baseline are compared. If the agent performed better than the baseline, its actions are reinforced, otherwise they are punished. Therefore, when defining an NCO algorithm, it is fundamental also to explain how the baseline is computed.

2.4.3 Available software

Developing the environment involves taking low level decisions about how the agent receives a new state, what are the actions the agent can perform, how the chosen action affects the state, how the reward is calculated, how the new state is computed and finally what happens when an episode ends and a new one has to start. When the environment is coded using a simulator, there are tools available that simplify this process, with Gymnasium (see Towers et al., 2023) being the most well-known package for Python. By adhering to the package API, developers can define some simple functions that capture the relevant information about the environment, and the package handles the agent-environment interaction.

The implementation of a DRL algorithm is not trivial, and it is often better to rely on existing libraries. However, these libraries are primarily designed for solving DRL problems in games or physics simulators, and may not be easily adaptable to NCO problems. On the other hand, they are often open-source, making them easily modifiable. Some of the most famous ones include Stable Baseline 3 (see Raffin et al., 2021), Tianshou (see Weng et al., 2022) and SheepRL (see Angioni et al., 2023). These libraries simplify the NCO algorithm development process, because developers can rely on pre-existing and tested code, where the training process is already implemented. They also provide utilities such as buffers for storing collected data and loggers for monitoring the training, which facilitate the development.

2.5 NCO for the TSP

Thus far we have given a general overview of NCO. This section aims to provide a more in-depth understanding of NCO and its applications by focusing on the TSP. The choice of the TSP as a case study in NCO is motivated by several factors. These include:

- **Extensive study in NCO**: The TSP has been extensively studied within the NCO field (see Bello et al., 2017; Dai et al., 2017; Deudon et al., 2018; Kool et al., 2019; Kwon et al., 2020; Xin et al., 2020; Jacobs et al., 2021; Luo et al., 2022; Joshi et al., 2022; Son et al., 2023; Luo et al., 2024). Most of the methods developed in NCO have been tested on the TSP, making it the standard benchmark for evaluating the performance of new approaches.

- **Simplicity**: The TSP is relatively easy to define and understand. This simplicity allows researchers to focus more on the application of DRL techniques.

- **Long-standing problem**: The TSP has been a subject of research for a long time, resulting in the development of numerous heuristics that serve as benchmarks for evaluating the performance of DRL-based approaches, providing a basis for comparison and analysis.

In this section, we present three approaches that serve as foundations for understanding the field of NCO.

For all the approaches presented, the only difference is the agent definition part, and especially the DNN architecture and how actions are chosen. Hence, we first present the common choices shared by all the approaches, and then, for each of them, we present the DNNs involved and how they are used for selecting actions in an episode.

The structure of the section follows the one of Section 2.4, where the agent definition part is further explored for each approach in the corresponding subsection. In Section 2.5.1 we define the environment, i.e. the TSP problem, and how the state and the action are represented. In Section 2.5.2 we define the agent, i.e. the common choices shared by all the approaches, like the underlying DRL algorithm. In Section 2.5.3 we define the interaction between the agent and the environment, i.e. how the agent interacts with the environment and how the training algorithm works. The three approaches are presented in Sections 2.5.4, 2.5.5 and 2.5.6.

- In Section 2.5.4 we introduce a naive approach, not reported in the literature, that uses an MLP to approximate the policy. This approach is only applicable to fixed-size instances.

- In Section 2.5.5 we present a simplified version of the approach presented in Bello et al., 2017. Here an architecture, called *Pointer Network*, that combines RNNs and attention, is used to handle variable-sized instances.

- In Section 2.5.6, we present a simplification of the approach proposed in Kool et al., 2019 where GNNs are used to capture the graph structure of the TSP. The resulting architecture is called *Attention Model* (AM) and is the starting point for most of the recent papers in the field (see Kwon et al., 2020; Xin et al., 2020; Kim et al., 2022).

Finally, in Section 2.5.7 we compare the three approaches.

For all the approaches, we present a numerical example to better understand how they work. By examining these different approaches, we can gain insights into the challenges and advancements in applying DRL to solve the TSP.

2.5.1 Define the environment

We consider the Euclidean TSP. Given an instance with N customers labeled from 1 to N, we use a 2-dimensional vector $\boldsymbol{x_i}$ to denote the xy-coordinates in the 2D plane for the i-th customer. The distance between customer $\boldsymbol{x_i}$ and customer $\boldsymbol{x_j}$, denoted as $d_{i,j}$, is computed using the Euclidean distance formula:

$$d_{i,j} = ||\boldsymbol{x_i} - \boldsymbol{x_j}||_2.$$

We consider this problem as defined over a graph, where the nodes are the customers and the nodes features are their coordinates. We use the terms *node* and *customer* interchangeably throughout this discussion.

To simplify the problem, we focus on instances where the customers are placed on a unit square, i.e. $\boldsymbol{x_i} \in [0, 1] \times [0, 1]$. This restriction does not limit the generality of the problem since we can always scale the coordinates to fit this constraint.

The goal of the TSP is to find an Hamiltonian cycle of minimum length, namely the shortest tour that visits each node exactly once and returns to the starting node.

Throughout this section, to better understand how the policy is defined and used to compute a probability distribution over possible actions starting from a state representation, we will apply the three different approaches to the instance of the TSP shown in Figure 2.11.

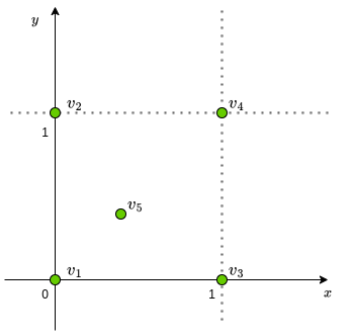


Figure 2.11: An instance of the TSP with 5 customers.

There are 5 customers, with the following coordinates:

$$\boldsymbol{x_1} = (\,0.0\;\;0.0\,), \boldsymbol{x_2} = (\,0.0\;\;1.0\,), \boldsymbol{x_3} = (\,1.0\;\;0.0\,), \boldsymbol{x_4} = (\,1.0\;\;1.0\,), \boldsymbol{x_5} = (\,0.3\;\;0.3\,).$$

Since the only difference between the approaches is how the policy is defined and how actions are selected, we focus on these aspects. All the other aspects, already described above, are the same for all the approaches.

To apply DRL to the TSP, first of all we need an MDP formulation of it.

MDP formulation.
The state of the environment contains the nodes features $\{\boldsymbol{x_i}\}_{i=1,\dots,N}$ and the list of already visited nodes in order of visit. An action is the label of a node that has not been visited yet. The reward is 0 for all the steps, except for the last one, where it equals the negative of the total distance of the tour. Such reward is a sparse reward, and we decided to use it since it is the most common choice in the literature (see Kool et al., 2019 and Kwon et al., 2020).

In this formulation we opted for using nodes coordinates to describe the problem, instead of the matrix of distances. The main reason for this choice comes from the fact that in the NCO literature, this is the most common choice (see Bello et al., 2017; Nazari et al., 2018; Kool et al., 2019 and Kwon et al., 2020). However, the distance matrix is needed whenever coordinates cannot fully describe the problem, for example in the case of the asymmetric TSP (see Drakulic et al., 2023). To the best of our knowledge, there is no contribution in the literature comparing the two approaches.

Low level decisions.
We need to define how the state and the action are represented.

For each time step $t = 0, 1, 2, \dots$, the state is represented using a matrix O of dimension $2 \times N$, where each column has the xy-coordinates of a node, and a list l_t of length t containing the labels of the already visited nodes.

Let $\sigma(t)$ for $t \in \{0, 1, \dots, N-1\}$ represent the label of the node that was visited at time step t, before the next action is chosen. Then, a partial solution at time t is represented by

$$l_t = \begin{cases} [] & \text{if } t = 0 \\ [\sigma(0), \sigma(1), \dots, \sigma(t-1)] & \text{if } t > 0 \end{cases}.$$

Each action represents the next node to visit. The agent performs exactly N actions before the episode ends. The horizon of the problem is $T = N - 1$, since time starts from 0 while the node labels start from 1. The transition function simply updates the list of visited nodes.

For every action, the reward is 0, except for the final step, where the reward equals the negative total distance of the completed tour.

$$r_t = \begin{cases} 0 & \text{if } t < T \\ -\sum_{i=0}^{T-1} d_{\sigma(i),\sigma(i+1)} - d_{\sigma(T),\sigma(0)} & \text{if } t = T \end{cases}.$$

When an episode ends, a new one starts with a new O matrix and an empty list l_0.

2.5.2 Define the agent

For all the examples, the DRL algorithm used is called REINFORCE. To use REINFORCE, we need to define a parametric policy π_θ. The definition of π_θ is the main difference between the approaches presented in Sections 2.5.4, 2.5.5 and 2.5.6. To define the policy, we will describe the DNN architecture and how data are fed to it.

For all the three approaches, given a state s_t, the policy outputs a probability distribution over the actions. The agent then chooses the action by sampling according to this distribution.

Since we want to avoid visiting the same node twice, we need to avoid sampling the actions that have already been taken. This is accomplished by setting the probability of already visited nodes to 0 before sampling from the probability distribution. This process is called *action masking*, since it is achieved using a so called *mask vector*. The mask vector is called $\boldsymbol{\mu_{t-1}}$ and is used to set the probability of already visited nodes to 0. This vector is computed using the list of visited nodes l_{t-1}. $\boldsymbol{\mu_{t-1}}$ has value 0 in the entries corresponding to the nodes that have been visited (the labels in l_t), and 1 otherwise. The masking happens by a point-wise multiplication of the vector of probabilities with the mask vector. Then, to ensure that the resulting vector still sums to 1, each entry of the vector is divided by the sum of all the entries in the vector. Given the vector of probabilities before masking p_t, the vector of probabilities after masking is computed as

$$\pi_t(a \mid s_t) = \frac{p_t^a \mu_t^a}{\boldsymbol{p_t} \cdot \boldsymbol{\mu_t}}$$

where ^{a} represents the a-th element of the vector.

2.5.3 Define the interaction between the agent and the environment

With REINFORCE the agent interacts with the environment until a given number K of episodes is reached. This number is a hyperparameter that needs to be set before training. Collected trajectories are stored in a buffer that is emptied after the training algorithm ends, as represented in Figure 2.3. When K trajectories are collected, the training algorithm starts. The algorithm takes the trajectories generated by the agent and updates the policy using a loss function that depends on the opposite of the total cumulative undiscounted reward for each trajectory (i.e. on the length of the tour). In the case of the TSP example presented in this section, the total cumulative reward is undiscounted

as it corresponds to the deterministic total distance traveled in the constructed solution, where a discount makes no sense.

We give here the pseudocode of the REINFORCE algorithm, where steps 5 and 6 are the data collection phase and step 8 is what is called the update algorithm in Figure 2.3.

Algorithm 1 REINFORCE

1: Initialize the policy parameters θ randomly
2: Initialize empty buffer B
3: **while** Convergence condition is not met **do**
4: **for** $k = 1, \ldots, K$ **do**
5: Generate trajectory τ_k using π_θ
6: Store τ_k in B
7: **end for**
8: Update θ according to the REINFORCE loss function using the trajectories in B
9: Empty B
10: **end while**

What remains to define for all the approaches is the DNN architecture and how data are fed to the DNN.

2.5.4 Fixed size instances

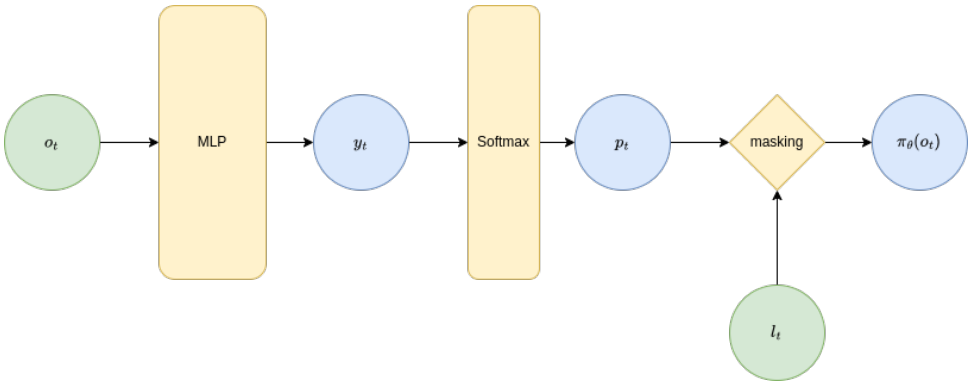


Figure 2.12: The MLP policy architecture for the TSP.

In this section, we discuss the use of an MLP to approximate the policy π for solving the TSP. The DNN takes as input the state of the environment and outputs a probability distribution over the actions.

We refer to Figure 2.12 for the architecture of the agent-environment interaction. Since the MLP cannot accept a variable number of inputs, we need to fix the size N of the instances in advance and we transform the matrix representation of the state to a vector by concatenating the columns of O and concatenating also the first visited node and the current visited node. At the first step, when no node has been visited yet, these values are substituted with a default vector $g = (-1.0 \quad -1.0)$. This results in a vector $\boldsymbol{o_t}$ of length $2N + 4$.

$$o_t = (\boldsymbol{x_1}; \boldsymbol{x_2}; \ldots; \boldsymbol{x_N}; \boldsymbol{x_{\sigma(0)}}; \boldsymbol{x_{\sigma(t-1)}})$$

where ; means concatenation.

The approximated parametric policy is denoted as π_θ, and returns the probabilities for the next node to visit. The policy is constructed using an MLP with input layer of size $2N + 4$, a variable number of hidden layers followed by an output layer of size N, which corresponds to the number of possible actions. The number and size of the hidden layers are hyperparameters that need to be set before training. The output of the final linear layer of the MLP is a vector $\boldsymbol{y_t}$ of size N, but this vector is not a vector of probabilities yet. To ensure that the output of the MLP is a probability function, we apply the softmax function as a final layer of the MLP.

To handle cases where the MLP outputs probabilities for nodes that have already been visited, we need to mask out those actions to avoid infeasible solutions. This is accomplished using the action masking procedure described in 2.5.2.

Notice that knowing the last visited node and the first visited node (which is also the final node to return to at the end of the tour), together with the masking process that removes already visited nodes, is sufficient information to describe the problem of visiting all the residual nodes from the current node to the starting node.

This example has been implemented in the Github repository https://github.com/DavideTr8/RL4TSP/, and the details of its implementation are discussed in Appendix A.

Numerical example

Let us consider the TSP defined in Section 2.5.1.

The initial state is given by the matrix O and an empty list of visited nodes. The vector $\boldsymbol{o_t}$ is obtained by concatenating the columns of O and the default values $g = (\,-1.0\;\;-1.0\,)$, :

$$\boldsymbol{o_0} = (\,0.0\;\;0.0\;\;0.0\;\;1.0\;\;1.0\;\;0.0\;\;1.0\;\;1.0\;\;0.3\;\;0.3\;\;-1.0\;\;-1.0\;\;-1.0\;\;-1.0\,).$$

We create an MLP setting the input layer to have dimension 14, since the state has dimension 14. We set also the hyperparameters of the hidden layers. For example, we set the number of hidden layers and the size of each hidden layer and the activation function. Finally, the output layer has dimension 5, since there are 5 possible actions. The last layer is a softmax activation layer. We treat this MLP as a function $MLP : \mathbb{R}^{14} \longrightarrow [0,1]^5$, that takes as input $\boldsymbol{o_t}$ and outputs $\boldsymbol{y_t}$. For example, $MLP(\boldsymbol{o_0})$ could be: $\boldsymbol{y_0} = \boldsymbol{p_0} = (\,0.1133\;\;0.1608\;\;0.4687\;\;0.0051\;\;0.2521\,)$. We can now sample from $\boldsymbol{\pi}(\boldsymbol{o_0})$ to obtain the label of the first node to visit, for example node 3. We update the state of the environment by adding node 3 to the list of visited nodes $l_t = [3]$. The vector of the observations becomes

$$\boldsymbol{o_1} = (\,0.0\;\;0.0\;\;0.0\;\;1.0\;\;1.0\;\;0.0\;\;1.0\;\;1.0\;\;0.3\;\;0.3\;\;1.0\;\;0.0\;\;1.0\;\;0.0\,).$$

We iterate the process above, but this time, since node 3 has been visited, we need to perform action masking. To do so, we apply the softmax function to $\boldsymbol{y_1}$, obtaining $\boldsymbol{p_1}$. Then, we set the probability of node 3 to 0 and we divide by the sum of the resulting vector to obtain a probability distribution again. For example, if $\boldsymbol{p_1} = softmax(MLP(\boldsymbol{o_1})) = softmax(\boldsymbol{y_1}) = (\,0.2647\;\;0.5201\;\;0.1311\;\;0.0119\;\;0.0722\,)$. Then we introduce $\boldsymbol{\mu_1} = (\,1.0\;\;1.0\;\;0.0\;\;1.0\;\;1.0\,)$. Finally, we obtain $\boldsymbol{\pi}(a|\boldsymbol{o_1}) = \frac{p_1^a \mu_1^a}{\boldsymbol{p_1} \cdot \boldsymbol{\mu_1}} = (\,0.3046\;\;0.5986\;\;0.0\;\;0.0137\;\;0.0831\,)$. Now the agent samples an action from this probability and gets action 1. Then, $l_2 = [3, 1]$,

$$\boldsymbol{o_2} = (\,0.0\;\;0.0\;\;0.0\;\;1.0\;\;1.0\;\;0.0\;\;1.0\;\;1.0\;\;0.3\;\;0.3\;\;1.0\;\;0.0\;\;0.0\;\;0.0\,).$$

and the process repeats.

The process ends when all the nodes have been visited. At this point, also the reward is computed, and the trajectory is stored in the buffer.

2.5.5 Instances with changing size

In this section, we present *Pointer Networks*, an architecture introduced in Vinyals et al., 2015 to tackle CO problems, and we describe a simplified version of the approach proposed in Bello et al., 2017, that uses a Pointer Network and DRL to solve the TSP. The main reason for choosing Pointer Networks over MLP is that Pointer Networks can handle variable-size instances. Specifically, a Pointer Network is an encoder-decoder architecture that combines RNNs and attention. The encoder is an RNN (RNN_e) and the decoder is an RNN (RNN_d) followed by an attention mechanism (A) (see Figure 2.13).

The data in a Pointer Network are processed as follows. Since the input to an RNN must be a sequence, we consider the state O as a sequence of nodes features $(\boldsymbol{x_i})_{i=1,...,N}$. Moreover, we use the list of visited nodes l_{t-1} to create the mask vector $\boldsymbol{\mu_{t-1}}$. The encoder receives the sequence of nodes features $(\boldsymbol{x_i})_{i=1,...,N}$ and the initial hidden state $\boldsymbol{h_0}$, and produces a sequence of hidden states, denoted as $\left(\boldsymbol{h_i^{enc}}\right)_{i=1,...,N} \in \mathbb{R}^d$, where d is the dimension for the hidden state, which is a hyperparameter to set.

The value of d is arbitrarily set by experience, trial and error.

The first part of the decoder is also an RNN that, at each time step t, takes as input the last visited node $\boldsymbol{x_{\sigma(t-1)}}$ (we recall that the partial solution at time t is denoted as $(\sigma(0), \sigma(1), ..., \sigma(t-1))$), with a *starting signal* $\boldsymbol{g}$ when no node has been visited, and outputs a vector $\boldsymbol{h_t^{dec}} \in \mathbb{R}^d$. After RNN_d, there is an attention layer that operates using $\boldsymbol{h_t^{dec}}$ as input data and the set of all the embeddings $\left\{\boldsymbol{h_i^{enc}}\right\}_{i=1,...,N}$ as target data. The output of this attention mechanism is an $N \times 1$ matrix that can be regarded as a vector of probabilities over the nodes. This probability distribution is used to sample the next node to visit.

To summarize, the encoder is run once at the beginning to process all the nodes in order, then the decoder is run at each step to produce the next node to visit, which is also the input of the decoder itself in the next step. This architecture is called a Pointer Network (see Figure 2.13) because its output serves as a pointer to an element of the input. To ensure feasible solutions, the masking procedure is applied to set the probability of already visited nodes to 0 after the attention is computed, as described in Section 2.5.2.

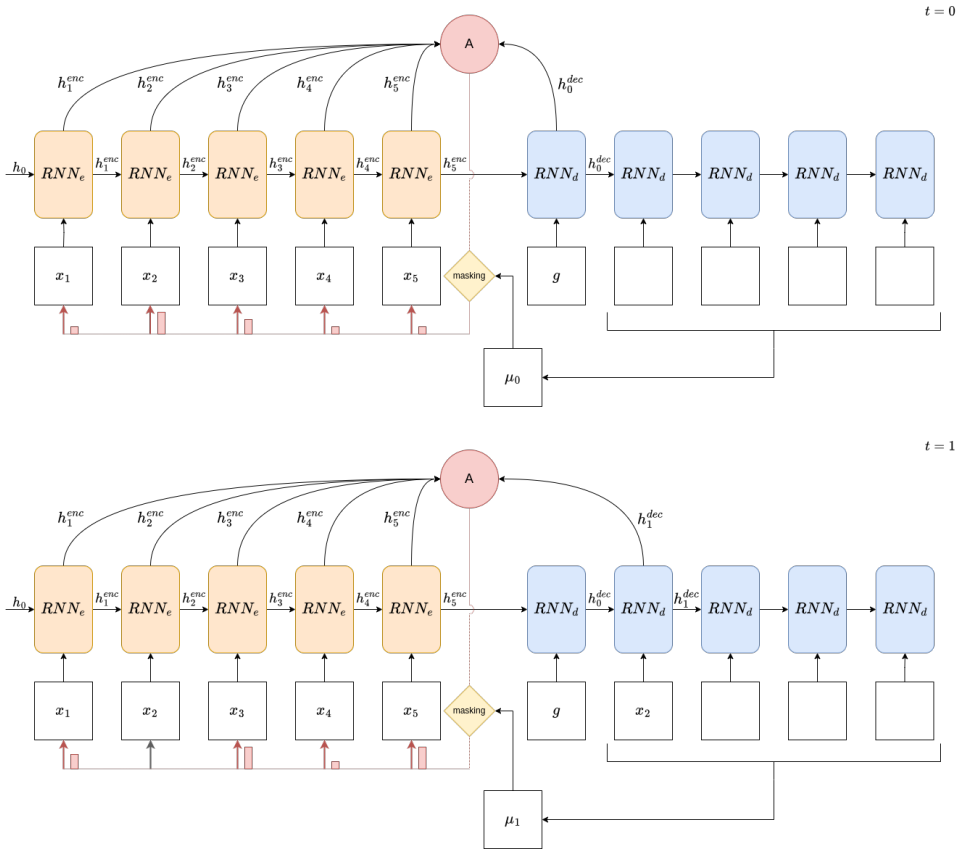


Figure 2.13: Two consecutive steps of a Pointer Network in a problem with 5 customers: the encoder RNN is in orange, the decoder RNN is in blue and the attention mechanism is in red. At time step $t = 2$, the probability of visiting $\boldsymbol{x_2}$ is set to 0 since it has already been visited at step $t = 1$. $\boldsymbol{g}$ is the starting signal introduced above. The black arrows represent the input of the various DNNs. The orange arrows represent the output of the attention mechanism, where the red bar on the right of an arrow indicates the probability of choosing that node.

Numerical example

Let us consider the TSP instance defined in Section 2.5.1.

We consider an encoder RNN_e and a decoder RNN_d that are RNNs with hidden states of a given size d. This is a hyperparameter that needs to be set before training.

Since RNNs need the previous state as input, we need to define a starting state for the encoder, that we call $\boldsymbol{h_0^{enc}}$. We set $\boldsymbol{h_0}$ to a vector of zeros of size d. Then, we get

$$\begin{aligned}\boldsymbol{h_1^{enc}} &= RNN_e(\boldsymbol{x_1}, \boldsymbol{h_0}),\\ \boldsymbol{h_2^{enc}} &= RNN_e(\boldsymbol{x_2}, \boldsymbol{h_1^{enc}}),\\ &\dots\\ \boldsymbol{h_5^{enc}} &= RNN_e(\boldsymbol{x_5}, \boldsymbol{h_4^{enc}}).\end{aligned}$$

Now, at time step 0, we compute $\boldsymbol{h_0^{dec}} = RNN_d(\boldsymbol{g}, \boldsymbol{h_5^{enc}})$. The starting signal $\boldsymbol{g}$ is a vector filled of -1.0 of size d. We compute the attention of $\boldsymbol{h_0^{dec}}$ over all the embeddings $\{\boldsymbol{h_i^{enc}}\}_{i=1,..,N}$. At this point, we get $attention(\boldsymbol{h_0^{dec}}, \{\boldsymbol{h_i^{enc}}\}_{i=1,...,5}) = \pi(\cdot|O) = (0.15\ \ 0.3\ \ 0.2\ \ 0.25\ \ 0.1)$, then we sample using this vector and obtain $a_0 = 2$.

Now, since we have selected $\boldsymbol{x_2}$ as the first node to visit, we compute $\boldsymbol{h_1^{dec}} = RNN_d(\boldsymbol{x_2}, \boldsymbol{h_0^{dec}})$. $\boldsymbol{h_1^{dec}}$ depends on the node we have just visited and on the memory of the previous state of the decoder. We apply the attention mechanism to obtain $\pi(\cdot|O, a_0) = masking(attention(\boldsymbol{h_1^{dec}}, \{\boldsymbol{h_i^{enc}}\}_{i=1,\dots,5}), \boldsymbol{\mu_1}) = (\,0.31\;\;0.0\;\;0.18\;\;0.11\;\;0.4\,)$.

Sampling from this probability distribution, we get $a_1 = 5$. Notice that the probability computed above is conditioned over O and the previous action a_0, since the encoder outputs have the information about the observation and the decoder outputs have the information about the previous actions thanks to the RNN memory. In general, at each time step t, the probability of visiting $\boldsymbol{x_i}$ is conditioned over the observation O and the previous actions $a_0, \dots, a_{t-1}$.

The process ends when all the nodes have been visited. At this point, the reward is computed and the trajectory is stored in the buffer.

2.5.6 Instances represented on a graph

The previous method used RNNs to address the challenge of variable-size instances, but RNNs may not be well-suited for CO problems that lack a sequential structure. The order in which nodes are presented to the RNN can impact its output. Since the TSP can be viewed as a problem defined on a graph G, GNNs can be a more suitable choice than RNNs. GNNs consider the underlying graph structure, providing better solutions for such problems.

In this section, we present a simplification of the approach presented in Kool et al., 2019, where the AM architecture is used in place of the Pointer Network. The AM is again an encoder-decoder architecture. The encoder is a GNN instead of an RNN, and the decoder is composed solely by the attention mechanism. In the AM an additional vector, called *context vector*, is used to keep track of the partial solution.

We explain in more detail the architecture of the AM in the following. The encoder part of the architecture is composed of a GNN that takes as input the graph G describing the TSP instance at hand and outputs a new graph G' where each node i has updated nodes features $\boldsymbol{h_i}$. The dimension of the new nodes features is a hyperparameter that needs to be set. As we did with the Pointer Network, we call $\{\boldsymbol{h_i}\}_{i=1,\dots,N}$ the embeddings of the nodes. Using these embeddings, a *graph embedding* $\bar{\boldsymbol{h}} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{h_i}$ is computed, which is a vector representation of the whole graph (see Figure 2.14). This vector is used to keep an overall representation of the graph in the context vector. Like in the Pointer Network, the encoder is run only once at the beginning.

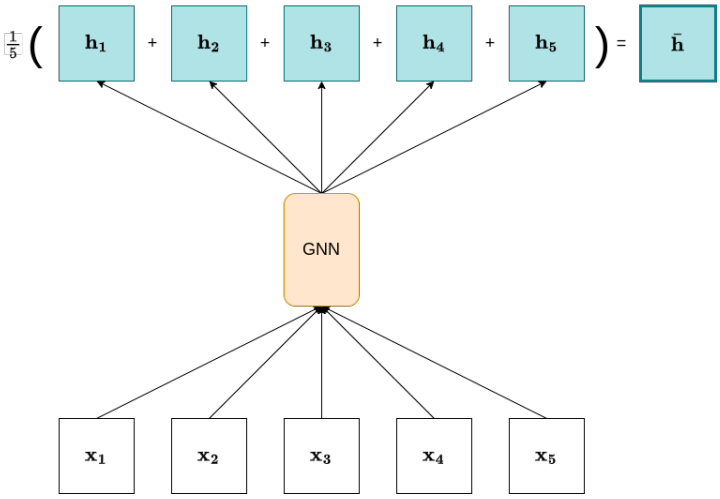


Figure 2.14: The encoder part of the architecture. $\boldsymbol{h_i}$ is the embedding of each node i and $\bar{\boldsymbol{h}}$ is the graph embedding.

The decoder part acts sequentially and consists solely of an attention mechanism between the node embeddings and the context vector $\boldsymbol{h_t^c}$. The context vector is updated at each step and is necessary to keep track of relevant information for taking decisions. For the TSP, this vector is created by concatenating the embedding of the whole graph, the embedding of the first visited node and the embedding of the last visited node (these last two embeddings are substituted with a default vector $g = (\,-1 \quad -1\,)$ when no node has been visited)

$$\boldsymbol{h_t^c} = \begin{cases} (\,\bar{\boldsymbol{h}}; \boldsymbol{g}; \boldsymbol{g}\,) & \text{if } t = 0 \\ (\,\bar{\boldsymbol{h}}; \boldsymbol{h_{\sigma(0)}}; \boldsymbol{h_{\sigma(t-1)}}\,) & \text{if } t > 0 \end{cases}.$$

At each time step t, the decoder takes as input data the context vector $\boldsymbol{h_t^c}$ and as target data the set of all the embeddings $\{\boldsymbol{h_i}\}_{i=1,\dots,N}$ (see Figure 2.15). As in Section 2.5.5, the output of the attention can be regarded as the probability for each node to be the next node to visit. Once again, the probability of already visited nodes is set to 0 using the mask vector $\boldsymbol{\mu_t}$. This probability distribution is used to sample the next node to visit. This process is iterated until the whole solution is found, that is when the list l_t has length $N-1$. Similarly as in the previous method, the reward is computed at this point.

With the Pointer Network, the partial solution at time step t was kept in the memory of the RNN. Here, the context vector $\boldsymbol{h_t^c}$ is used to keep the embeddings of the first and last visited node. The rest of the partial solution is relevant for masking out the nodes that have already been visited.

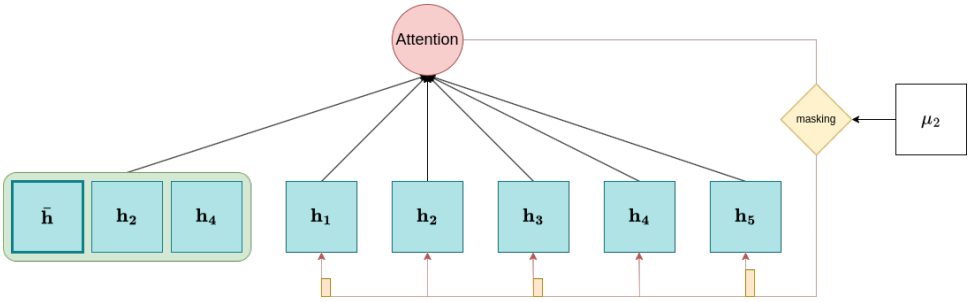


Figure 2.15: The decoder part of the architecture when the partial solution found so far is $(2\ \ 4)$ in a TSP with 5 nodes. The context vector is represented with the green rectangle containing $\bar{\boldsymbol{h}}$, $\boldsymbol{h_2}$ and $\boldsymbol{h_4}$, the embeddings of each node are the blue squares. The yellow bar represents the probability of visiting a given node, with the probability of visiting $\boldsymbol{x_2}$ and $\boldsymbol{x_4}$ set to 0 since they have already been visited.

Numerical example

Let us consider the TSP instance defined in Section 2.5.1. If we apply the GNN to the graph G representing the TSP instance, we obtain a graph G' where each node has features $\{\boldsymbol{h_i}\}_{i=1,\dots,5}$. The graph embedding $\bar{\boldsymbol{h}}$ is computed by averaging all the node embeddings, that is $\bar{\boldsymbol{h}} = \frac{1}{N}\sum_{i=1}^{N} \boldsymbol{h_i}$.

To apply the decoder, we need to compute the context vector $\boldsymbol{h_t^c}$. For time step 0, we set $\boldsymbol{h_0^c} = (\bar{\boldsymbol{h}}; (-1.0\ \ -1.0); (-1.0\ \ -1.0))$, where $(-1.0\ \ -1.0)$ are the dummy values of the default vector g and ; indicates concatenation. Then, we compute the attention of $\boldsymbol{h_0^c}$ over all the nodes embeddings $\boldsymbol{h_i}$ to obtain the probability of visiting each node $\pi(\cdot \mid O, \boldsymbol{h_0^c}, \boldsymbol{\mu_0}) = (0.3\ \ 0.45\ \ 0.17\ \ 0.06\ \ 0.02)$.

We sample from this probability distribution and get $a_0 = 2$. Hence, we set $\boldsymbol{h_1^c} = (\bar{h}; \boldsymbol{x_2}; \boldsymbol{x_2})$, and we repeat the decoding process, getting $\pi(\cdot \mid O, \boldsymbol{h_1^c}, \boldsymbol{\mu_1}) = (0.51\ \ 0.00\ \ 0.15\ \ 0.08\ \ 0.24\ \ 0.02)$ where the probability of visiting $\boldsymbol{x_2}$ is set to 0 since it has already been visited. We sample again and get $a_1 = 5$. We set $\boldsymbol{h_2^c} = (\bar{\boldsymbol{h}}; \boldsymbol{x_2}; \boldsymbol{x_5})$ and repeat the process until the whole solution is found. At this point, the reward is computed and the trajectory is stored in the buffer.

2.5.7 Comparison

In this section we analyze and compare the approaches presented above.

The approach presented in Section 2.5.4 has a number of limitations. The main drawback is given by the fact that if we change the size of the instances, we need to create a new MLP and train it from scratch. Another reason why this approach is not adopted, is the fact that in the MLP architecture presented there is not a strong connection between an action and the node representing that action. For example, if we take an instance and change the order of its nodes, the problem stays the same, but the MLP will see it as a new instance. If we compare it with the pointing mechanism introduced with the attention, this does not happen as the attention points directly to the nodes feature to select an action and, if we change the labels of the nodes, the probability of each node would stay unchanged. Also, if we rotate the plane in which the nodes are placed, translate it, scale it or apply any isometric transformation the solution of the TSP stays the same, while the MLP will treat them as different instances.

To solve the problem of instances with changing size, the approach presented in Section 2.5.5 uses an RNN in the encoder since RNNs can handle variable-sized instances. At the same time, the fact that the decoder is an RNN allows us to keep track of the partial solution found so far. Finally, self-attention mechanism, used in the Pointer Network presented in Section 2.5.5, can also handle instances of different sizes. Moreover, the attention mechanism is able to relate the action to the node, since the output of the model is not the label of the action but a pointer to the node itself.

While RNNs can handle variable size instances, there are some drawbacks. In the RNN approach, to be able to use the Pointer Network, we need to compute the embedding of all the nodes once and then sequentially decode them at each time step. Thus, it is assumed that the nodes features remain unchanged after each step, since if the features changed the embedding would also change. This assumption does not hold, for example, in stochastic and dynamic settings, where the agent lacks knowledge of the transition and reward functions. Extending this architecture to these problems is not straightforward.

Nevertheless, the flexibility of Pointer Networks allows them to be used with other CO problems, like the knapsack problem. In this case, the Pointer Network can be used to select items to include in the knapsack, stopping the process when the total weight exceeds the capacity, as done in Bello et al., 2017. Finally, embedding the nodes in a sequence implies that the order of presentation to the network matters and thus, the graph structure of the TSP is not fully exploited.

The model presented in Section 2.5.6 solves the latter problem but also offers more flexibility. The fact that GNNs are not sequential allows us to take into consideration the possibility of recomputing the embedding of the instance at each step, allowing the model to be used in problems where the features might change at each step. On the other hand, the action choice is only conditioned by the information provided by the context vector and, thus, the agent lacks knowledge of the whole partial solution. Constraint satisfaction is guaranteed only by the masking procedure and, thus, no learning takes place with regard to the constraints. The work by Kool et al., 2019 demonstrates the flexibility of this architecture by applying it not only to the TSP, but also to the Vehicle Routing Problem (VRP), the Split Delivery VRP, the Prize Collecting TSP (PCTSP), the Orienteering Problem and a stochastic version of the PCTSP.

Indeed, this model became the starting point of much of the research around NCO. Notice, however, that this model might not be well suited for the KP, due to the lack of a graph structure.

After the AM was developed, researchers focused more on solving open issues of NCO, like scalability and symmetry exploitation. These issues are discussed in Section 2.6.

2.6 Issues and future directions

In the previous section, we discussed the application of NCO to the solution of the TSP and how the same approach can be applied to other CO problems.

NCO has evolved from the work presented in Kool et al., 2019, and subsequent research has attempted to address some of the open issues that will be discussed in this section. A more in-depth study on some open points in NCO, with a focus on the TSP, is presented in Liu et al., 2023.

Problems with changing features: As we have seen in Section 2.5, modern NCO approaches use an encoder-decoder architecture where the encoder is run only once at

the start of a new episode, and then the decoder is responsible for taking actions using the embeddings computed by the encoder.

This approach makes the assumption that, after every action, there is no substantial change in the state of the environment, and that only the partial solution changes from one time step to the next. This is not always the case. Dynamic CO problems are a notable example, where the data describing the instance might change at every time step. Then, the embedding computed at time step 0 would no longer be a valid representation of the problem instance at future time steps. A solution for this problem might be to re-compute the embedding at each time step. This issue is addressed in Xin et al., 2020 and Peng et al., 2020, where the authors propose to recompute the embedding of the nodes to account for the partial solution.

Symmetries exploitation: One of the prominent challenges in NCO is the presence of symmetries in CO problems. These symmetries might appear in the problem definition (e.g. changing the node labels of a TSP instance should not affect the solution) or in the solution itself (e.g. the solution [1, 2, 3, 4, 5] for a TSP with 5 nodes is the same as the solution [5,1,2,3,4]). Different CO problems have different symmetries, hence it becomes difficult to find a general way to exploit them.

In Kwon et al., 2020 the authors propose a way to exploit symmetries in the solution of some routing problems by creating multiple solutions using the same policy. Each solution is generated by increasing the randomness for the first action, to encourage the agent to start the solution creation from different nodes. By doing so, the agent should learn that the starting node is not relevant in the construction process. At the same time, they propose a way to exploit symmetries in the problem by using *instance augmentation*. This technique consists in creating, starting from one instance, multiple instances that are symmetric with the first one and to train the agent on the symmetric instances too. By doing so, the agent should learn the concept of symmetries. An example of instance augmentation for the TSP is to apply rotation and translation to the instance.

The investigation of symmetry in instances and solutions is further explored in Kim et al., 2022, where instead of generating multiple solutions for the same instance, the authors change the loss function of the AM to account for rotational symmetries and for the solution symmetry, for problems defined over a graph.

Scalability: Another significant challenge in NCO is *scalability*, which refers to the ability of NCO algorithms to effectively solve instances of different sizes from those used during training. For example, can an agent trained to solve TSP instances with up to 100 nodes effectively handle instances with 200 nodes? This is the most addressed problem in the literature about NCO for routing problems. This issue is addressed in Joshi et al., 2022, where they study different approaches for the scalability problem, and in Son et al., 2023, through a *meta-learning* approach that creates a policy that adapts to different instance sizes.

Generalization: Generalization is the ability of an agent to perform well on new problems and has two different aspects. The first one is the ability of an agent to perform well on instances that are different from the ones used for training. For example, can an agent trained on TSP instances generated by placing nodes using a uniform distribution on the unit square perform well on instances where all the nodes are placed on a grid? Or using a Gaussian distribution centered in (0.5, 0.5) instead of the uniform distribution? These aspects are mostly discussed in Liu et al., 2023.

The second aspect is the ability to quickly adapt to new problems. For example, can an agent that has learned to solve the TSP be adapted to solve a similar problem, such

as the PCTSP, without requiring complete retraining from scratch? To the best of our knowledge, there is no research on this topic so far. However, generalization, especially on this second aspect, is a very powerful feature and one of the aims of NCO: the possibility to lift the burden of designing a new heuristic whenever a problem changes, and perform a quick retraining instead. Nevertheless, this goal for NCO is still far from being reached due to the next open issue of NCO, which is solution feasibility.

Feasibility: Feasibility of solutions is a major concern in NCO. Currently, feasibility can only be guaranteed for problems where a masking procedure can be applied. However, for problems like the TSP with Time Windows (TSPTW), to ensure solution feasibility is a challenging matter because the agent might construct a partial solution where there is no feasible action left. So far, mostly soft constraints have been considered for routing problems with time windows (see Zhang et al., 2020; Lin et al., 2022), and, to the best of our knowledge, only Bi et al., 2024 address the construction of feasible solutions for problems where masking can not be applied.

A prominent area of research nowadays is the use of DRL for solving constrained MDPs (see Altman, 2021), which are a generalization of MDPs that can be used to model problems with constraints.

Evaluation of solutions: Finally, a problem more related to the literature itself than to the challenges of NCO is how to evaluate NCO solutions. Since NCO agents work as heuristics, the most straightforward comparison is to use classical heuristics as benchmarks. However, there are differences between NCO agents and classical heuristics.

The first difference is the training part. While a trained NCO agent can generate solutions as quickly as a heuristic, the training time needs to be considered. This can go from a few hours to many days, requiring a lot of computational power.

The second difference is the data-driven nature. Many studies (see, for example, Bello et al., 2017; Kool et al., 2019; Kwon et al., 2020) compare their outcomes solely on instances generated from the same distribution of the training set. However, classical heuristics for well-studied CO problems, like the TSP or VRP, are designed to work on generic instances. This is not true for NCO yet.

The third difference is that NCO has been developed recently and does not focus on a single CO problem. Instead, NCO algorithms can be easily adapted to different problem classes with minimal adjustments and training. However, NCO is often tested on well-studied problems like TSP or VRP, for which specialized heuristics have been developed and refined over decades.

These challenges underscore the research effort required to enhance the capabilities and applicability of NCO algorithms.

2.7 Why NCO

NCO is still in its infancy and, nowadays, is mostly used to solve deterministic CO problems. However, it is important to understand why NCO can be beneficial for the CO community and how, addressing the challenges mentioned in Section 2.6, it could transform NCO into a powerful tool for the CO community.

Data driven nature: NCO, due to its data driven nature, was born to create ad hoc heuristics for problems in which the instances share some commonalities. For example, if a company needs to solve a VRP where all the customers to visit are very similar every day, NCO might be able to learn a heuristic more tailored to the instances of the

company.

Generalization: Another aspect is the generalization ability, which is currently one of the most studied aspects in NCO. With generalization, we mean the ability for an agent, trained for a given problem, to solve instances of a similar problem without (or with a very short) retraining. If achieved, this property would be very important in practical implementation of CO solutions, where it is common that a problem definition changes over time. Nowadays, if a classical heuristic is developed and the problem changes, it is responsibility of the developer to design a new one that can solve the new problem.

Liu et al., 2023 tested the generalization capabilities of some NCO algorithm by training them on instances where the nodes are uniformly spread in the 0-1 square and testing them on instances where nodes tend to be clustered, and vice-versa, showing how the performances of the algorithm tend to degrade when the training and testing instances have different properties. Moreover, they show how training the algorithm on a mix of both instance types would increase the overall performances of all the algorithms tested.

Berto et al., 2024 instead introduces an agent that is trained to solve many variations of the VRP without retraining, with performance comparable to other NCO algorithms.

Solution generation speed: This latter aspect allows us to highlight another strength of NCO: the speed at which solutions are generated. Indeed, NCO algorithms are divided in two phases: the training phase and the inference phase. While training might take some time before the convergence of the algorithm, once it is done, the algorithm can be used to generate solutions in a fraction of a second, making it as fast as modern heuristics or even faster. The training phase usually happens only once, and then the learned heuristic can be used on new instances without re-training. For instance, Drori et al., 2020 shows how, once trained, a NCO algorithm can produce solutions for the TSP in a time that is linear in the instance size.

Beyond linear deterministic problems: There are other interesting research directions in the field of NCO. For instance, the theoretical background of NCO makes it a good approach for stochastic and dynamic problems, both discrete and continuous, especially for instances of large size. The same is true for non-linear problems, since NCO does not require an explicit formulation of an objective function, the latter being implicitly learned through reward signals from the environment. For example, Iklassov et al., 2024 solves the stochastic VRP using NCO, showing how NCO can learn to predict the stochasticity of the environment and to take actions accordingly.

There are some important milestones to reach before NCO can be actually used as a tool for solving CO problems. First of all, NCO can not solve most of the constrained problems. Constraints like hard time windows in a TSPTW, where the feasibility of a solution cannot be assessed a priori, are still not solvable by NCO. An interesting attempt is made by Bi et al., 2024 where, using a Lagrangian approach, the feasibility of a solution is included as an objective of the problem. In this way, the authors are able to solve the TSPTW with hard constraints obtaining feasible solutions between 85% to 100% of the time, depending on the instance characteristics. Moreover, NCO still struggles to reach the performance of classical heuristics, especially for small to medium size instances. Hence, NCO is usually tested on very high dimensional problems (see Fu et al., 2021), where classical heuristics struggle to find good solutions in reasonable computational times.

Conclusions

This tutorial introduces the concepts that are instrumental to the understanding of NCO, which is the field aimed at designing heuristics for CO problems using DRL. We have used the TSP and the KP as examples to show different approaches and discuss the current and future challenges in this field. While we only presented the TSP and the KP, NCO has been applied to several other CO problems encompassing routing problems, scheduling problems, classical graph problems, and many more.

Several papers about NCO are emerging. They mostly revolve around finding models and techniques that can overcome its limits.

We believe that this tutorial can serve as a starting point for researchers that want to approach NCO and are not familiar with DRL, especially for the operational researchers who can both benefit and positively contribute to the development of NCO.

Acknowledgments

The authors are grateful to the anonymous referees for their careful review and constructive comments on previous versions of this paper.

References

Altman, E. (2021). *Constrained Markov decision processes*. Routledge.

Angioni, D., F. Belotti, R. Can Malli, and M. Milesi (2023). *SheepRL*. Version 0.5.4.

Battaglia, P. W., J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, Ç. Gülçehre, H. F. Song, A. J. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. R. Allen, C. Nash, V. Langston, C. Dyer, N. M. O. Heess, D. Wierstra, P. Kohli, M. M. Botvinick, O. Vinyals, Y. Li, and R. Pascanu (2018). "Relational inductive biases, deep learning, and graph networks". In: *ArXiv* abs/1806.01261.

Bello, I., H. Pham, Q. V. Le, M. Norouzi, and S. Bengio (2017). "Neural Combinatorial Optimization with Reinforcement Learning". In: *5th International Conference on Learning Representations, ICLR 2017*.

Bengio, Y., A. Lodi, and A. Prouvost (2021). "Machine learning for combinatorial optimization: a methodological tour d'horizon". In: *European Journal of Operational Research* 290, pp. 405–421.

Berner, C., G. Brockman, B. Chan, V. Cheung, P. Dębiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, et al. (2019). "Dota 2 with large scale deep reinforcement learning". In: *arXiv preprint arXiv:1912.06680*.

Berto, F., C. Hua, N. G. Zepeda, A. Hottung, N. Wouda, L. Lan, K. Tierney, and J. Park (2024). "RouteFinder: Towards Foundation Models for Vehicle Routing Problems". In: *ICML 2024 Workshop on Foundation Models in the Wild (Oral)*. `https://github.com/ai4co/routefinder`.

Bi, J., Y. Ma, J. Zhou, W. Song, Z. Cao, Y. Wu, and J. Zhang (2024). "Learning to Handle Complex Constraints for Vehicle Routing Problems". In: *Advances in Neural Information Processing Systems*. Ed. by A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang. Vol. 37. Curran Associates, Inc., pp. 93479–93509.

Bradbury, J., R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang (2018). *JAX: composable transformations of Python+NumPy programs*.

Cho, K., B. van Merriënboer, Ç. Gul˙çehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio (2014). "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 1724–1734.

Dai, H., E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song (2017). "Learning combinatorial optimization algorithms over graphs". In: *Advances in Neural Information Processing Systems* 2017, pp. 6349–6359.

Deudon, M., P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau (2018). "Learning heuristics for the tsp by policy gradient". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 15th International Conference*. Vol. 15, pp. 170–181.

Drakulic, D., S. Michel, F. Mai, A. Sors, and J.-M. Andreoli (2023). "BQ-NCO: Bisimulation Quotienting for Efficient Neural Combinatorial Optimization". In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine. Vol. 36. Curran Associates, Inc., pp. 77416–77429.

Drori, I., A. Kharkar, W. R. Sickinger, B. Kates, Q. Ma, S. Ge, E. Dolev, B. L. Dietrich, D. P. Williamson, and M. Udell (2020). "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)*, pp. 19–24.

Elsken, T., J. H. Metzen, and F. Hutter (2019). "Neural Architecture Search: A Survey". In: *Journal of Machine Learning Research* 20.55, pp. 1–21.

Fawzi, A., M. Balog, A. Huang, T. Hubert, B. Romera-Paredes, M. Barekatain, A. Novikov, F. J. R. Ruiz, J. Schrittwieser, G. Swirszcz, D. Silver, D. Hassabis, and P. Kohli (2022). "Discovering faster matrix multiplication algorithms with reinforcement learning". In: *Nature* 610, pp. 47–53.

Fu, Z.-H., K.-B. Qiu, and H. Zha (May 2021). "Generalize a Small Pre-trained Model to Arbitrarily Large TSP Instances". In: *Proceedings of the AAAI Conference on Artificial Intelligence* 35.8, pp. 7474–7482.

Gilmer, J., S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl (2017). "Neural message passing for Quantum chemistry". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, pp. 1263–1272.

Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*. MIT press.

Gupta, P., M. Gasse, E. B. Khalil, M. P. Kumar, A. Lodi, and Y. Bengio (2020). "Hybrid models for learning to branch". In: *Advances in Neural Information Processing Systems* 2020, pp. 18087–18097.

He, K., X. Zhang, S. Ren, and J. Sun (2016). "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778.

Hochreiter, S. and J. Schmidhuber (1997). "Long Short-Term Memory". In: *Neural Computation* 9, pp. 1735–1780.

Hopfield, J. J. (1982). "Neural Networks and Physical Systems with Emergent Collective Computational Abilities". In: *Proceedings of the National Academy of Sciences of the United States of America* 79.8, pp. 2554–2558.

Iklassov, Z., I. Sobirov, R. Solozabal, and M. Takáč (Nov. 2024). "Reinforcement Learning for Solving Stochastic Vehicle Routing Problem". In: *Proceedings of the 15th Asian Conference on Machine Learning*. Ed. by B. Yanıkoğlu and W. Buntine. Vol. 222. Proceedings of Machine Learning Research. PMLR, pp. 502–517.

Jacobs, T., F. Alesiani, and G. Ermis (2021). "Reinforcement Learning for Route Optimization with Robustness Guarantees". In: *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, pp. 2592–2598.

Joshi, C. K., Q. Cappart, L.-M. Rousseau, and T. Laurent (2022). "Learning the travelling salesperson problem requires rethinking generalization". In: *Constraints* 27, pp. 70–98.

Kaufmann, E., L. Bauersfeld, A. Loquercio, M. Müller, V. Koltun, and D. Scaramuzza (2023). "Champion-level drone racing using deep reinforcement learning". In: *Nature* 620.7976, pp. 982–987.

Kim, M., J. Park, and J. Park (2022). "Sym-nco: Leveraging symmetricity for neural combinatorial optimization". In: *Advances in Neural Information Processing Systems* 35, pp. 1936–1949.

Kool, W., H. Van Hoof, and M. Welling (2019). "Attention, Learn to Solve Routing Problems!" In: *International Conference on Learning Representations*.

Kwon, Y.-D., J. Choo, B. Kim, I. Yoon, Y. Gwon, and S. Min (2020). "POMO: Policy Optimization with Multiple Optima for Reinforcement Learning". In: *Advances in Neural Information Processing Systems* 33, pp. 21188–21198.

Lin, B., B. Ghaddar, and J. Nathwani (2022). "Deep Reinforcement Learning for the Electric Vehicle Routing Problem With Time Windows". In: *IEEE Transactions on Intelligent Transportation Systems* 23, pp. 11528–11538.

Liu, S., Y. Zhang, K. Tang, and X. Yao (2023). "How Good is Neural Combinatorial Optimization? A Systematic Evaluation on the Traveling Salesman Problem". In: *IEEE Computational Intelligence Magazine* 18, pp. 14–28.

Lodi, A. and G. Zarpellon (2017). "On learning and branching: A survey". In: *Top* 25, pp. 207–236.

Luo, F., X. Lin, F. Liu, Q. Zhang, and Z. Wang (2024). "Neural combinatorial optimization with heavy decoder: Toward large scale generalization". In: *Advances in Neural Information Processing Systems* 36.

Luo, J., C. Li, Q. Fan, and Y. Liu (2022). "A graph convolutional encoder and multi-head attention decoder network for TSP via reinforcement learning". In: *Engineering Applications of Artificial Intelligence* 112, p. 104848.

Mazyavkina, N., S. Sviridov, S. Ivanov, and E. Burnaev (2021). "Reinforcement learning for combinatorial optimization: A survey". In: *Computers & Operations Research* 134, p. 105400.

Mirhoseini, A., A. Goldie, M. Yazgan, J. W. Jiang, E. M. Songhori, S. Wang, Y.-J. Lee, E. Johnson, O. Pathak, A. Nazi, J. Pak, A. Tong, K. Srinivasa, W. Hang, E. Tuncer, Q. V. Le, J. Laudon, R. Ho, R. Carpenter, and J. Dean (2021). "A graph placement methodology for fast chip design". In: *Nature* 594, pp. 207–212.

Mnih, V., A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu (2016). "Asynchronous methods for deep reinforcement learning". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*. ICML'16. JMLR.org, pp. 1928–1937.

Mnih, V., K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis (2015). "Human-level control through deep reinforcement learning". In: *Nature* 518, pp. 529–533.

Nazari, M., A. Oroojlooy, L. Snyder, and M. Takac (2018). "Reinforcement Learning for Solving the Vehicle Routing Problem". In: *Advances in Neural Information Processing Systems* 31.

Paszke, A., S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. (2019). "Pytorch: An imperative style, high-performance deep learning library". In: *Advances in Neural Information Processing Systems* 32.

Peng, B., J. Wang, and Z. Zhang (2020). "A deep reinforcement learning algorithm using dynamic attention model for vehicle routing problems". In: *Artificial Intelligence Algorithms and Applications: 11th International Symposium, ISICA 2019*, pp. 636–650.

Powell, W. (2022). *Reinforcement Learning and Stochastic Optimization: A Unified Framework for Sequential Decisions*. John Wiley & Sons, Ltd.

Puterman, M. (2014). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley Series in Probability and Statistics. Wiley.

Raffin, A., A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann (2021). "Stable-Baselines3: Reliable Reinforcement Learning Implementations". In: *Journal of Machine Learning Research* 22, pp. 1–8.

Rosenblatt, F. (1958). "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65 6, pp. 386–408.

Rumelhart, D. E., G. E. Hinton, and R. J. Williams (1986). "Learning representations by back-propagating errors". In: *Nature* 323, pp. 533–536.

Scarselli, F., M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini (2009). "The Graph Neural Network Model". In: *IEEE Transactions on Neural Networks* 20, pp. 61–80.

Schrittwieser, J., I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, et al. (2020). "Mastering atari, go, chess and shogi by planning with a learned model". In: *Nature* 588, pp. 604–609.

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov (2017). "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347*.

Silver, D., G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller (2014). "Deterministic policy gradient algorithms". In: *International Conference on Machine Learning*, pp. 387–395.

Son, J., M. Kim, H. Kim, and J. Park (2023). "Meta-SAGE: scale meta-learning scheduled adaptation with guided exploration for mitigating scale shift on combinatorial optimization". In: *International Conference on Machine Learning*, pp. 32194–32210.

Sutton, R. S. and A. G. Barto (2018). *Reinforcement Learning: An Introduction*. Second. MIT press.

Towers, M., J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis (2023). *Gymnasium*.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin (2017). "Attention is all you need". In: *Advances in neural information processing systems* 30.

Vesselinova, N., R. Steinert, D. F. Perez-Ramirez, and M. Boman (2020). "Learning Combinatorial Optimization on Graphs: A Survey With Applications to Networking". In: *IEEE Access* 8, pp. 120388–120416.

Vinyals, O., M. Fortunato, and N. Jaitly (2015). "Pointer Networks". In: *Advances in Neural Information Processing Systems* 28.

Weng, J., H. Chen, D. Yan, K. You, A. Duburcq, M. Zhang, Y. Su, H. Su, and J. Zhu (2022). "Tianshou: A Highly Modularized Deep Reinforcement Learning Library". In: *Journal of Machine Learning Research* 23, pp. 1–6.

Wu, H., T. Hu, Y. Liu, H. Zhou, J. Wang, and M. Long (2023). "TimesNet: Temporal 2D-Variation Modeling for General Time Series Analysis". In: *International Conference on Learning Representations*.

Xin, L., W. Song, Z. Cao, and J. Zhang (2020). "Step-wise deep learning models for solving routing problems". In: *IEEE Transactions on Industrial Informatics* 17, pp. 4861–4871.

Ye, H., J. Wang, Z. Cao, H. Liang, and Y. Li (2023). "DeepACO: Neural-enhanced Ant Systems for Combinatorial Optimization". In: *Advances in Neural Information Processing Systems*.

Zhang, R., A. Prokhorchuk, and J. Dauwels (2020). "Deep Reinforcement Learning for Traveling Salesman Problem with Time Windows and Rejections". In: *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8.

Zhao, J., M. Mao, X. Zhao, and J. Zou (2021). "A Hybrid of Deep Reinforcement Learning and Local Search for the Vehicle Routing Problems". In: *IEEE Transactions on Intelligent Transportation Systems* 22, pp. 7208–7218.

Appendix A

In this appendix, we present the code available in our GitHub repository https://github.com/DavideTr8/RL4TSP/.

This repository contains a Python implementation of the example discussed in Section 2.5.4, where the parametric policy is represented by an MLP. Here, we provide a detailed breakdown of the implementation. The explanation on how to run the example and plot the results are available in the `README.md` file of the repository.

The core packages used are PyTorch by Paszke et al., 2019 and Gymnasium by Towers et al., 2023.

A.1 Structure of the code

To keep the codebase simple and organized, we created a separate Python class for each main component: the environment, the agent, and the REINFORCE algorithm. Additionally, we included a fourth file, `baseline_policies.py`, which defines baseline policies that can be used with the REINFORCE algorithm.

A.2 Environment

The environment is implemented using the Gymnasium package, which allows users to define custom environments by creating a class that extends the `Env` class provided by Gymnasium.

The class for the TSP environment is named `TSPEnv`. Its first method, `__init__`, initializes the class and sets various attributes, such as the number of nodes (`self.n_nodes`), the observation space (`self.observation_space`) and the action space (`self.action_space`) for the environment. Additional attributes are defined for rendering and setting the random seed.

The two key methods in this environment class are the `reset` and `step` methods.

Reset

The reset method initializes an episode by creating a new instance of the problem and returning the observation for time step $t = 0$ to the agent.

In this example, the reset method is defined as follows:

```
def reset(self):
    self.nodes = np.random.rand(2, self.n_nodes)
    self.visited = []
    return {"nodes": self.nodes, "visited": self.visited}
```

The method generates random nodes within the $[0, 1] \times [0, 1]$ square as a $2 \times$ `self.n_nodes` matrix, which is our O matrix. At the same time the reset method sets the list of visited nodes as an empty list, corresponding to l_0. The `return` command sends the dictionary containing the nodes and the list of visited nodes to the agent.

Step

The step method takes the action selected by the agent as input. It returns the updated state, the reward, and the done signal—a boolean flag indicating whether the episode has ended.

In this example, the step method is defined as:

```
def step(self, action):
    self.visited.append(action)
    if len(self.visited) == self.n_nodes:
        done = True
    else:
        done = False

    if done:
        reward = 0
        for i in range(self.n_nodes - 1):
            reward -= np.linalg.norm(
                self.nodes[:, self.visited[i]] - self.nodes[:, self.visited[i + 1]]
            )
        reward -= np.linalg.norm(
            self.nodes[:, self.visited[-1]] - self.nodes[:, self.visited[0]]
        )
    else:
        reward = 0
    next_state = {"nodes": self.nodes, "visited": self.visited}
    return next_state, reward, done
```

The method first updates the list of visited nodes. If the number of visited nodes equals the total number of nodes in the instance, the done signal is set to `True`. If the episode is not over, the reward is set to 0. Otherwise, the reward is calculated as the negative length of the tour selected by the agent. Finally, the method returns the new state, reward, and done signal.

Other methods

The environment class also includes additional methods for rendering and setting the random seed. These methods are omitted here for brevity, as they are not essential to understanding the primary functionality of the class.

A.3 Agent

The agent class is named `MLPAgent`.

It extends the PyTorch `Module` class, which is used to handle DNNs. The agent class is responsible for taking actions according to its policy. For simplicity, the agent policy is defined directly within the MLPAgent class.

The first method is the `__init__`, which sets up the DNN layers.

Set the DNN layers

```
def __init__(self, n_nodes, hidden_dim):
    super().__init__()
    self.model = torch.nn.Sequential(
        torch.nn.Linear(n_nodes * 2 + 4, hidden_dim),
        torch.nn.Tanh(),
        torch.nn.Linear(hidden_dim, hidden_dim),
        torch.nn.Tanh(),
        torch.nn.Linear(hidden_dim, n_nodes),
```

```
        )
        self.softmax = torch.nn.Softmax(dim=-1)
```

The initial `super` call is used to inherit from the base `Module` class, although this is beyond the scope of this tutorial.

The remaining lines define the DNN layers. The DNN consists of a sequence of Linear layers, each followed by an activation function called tanh (hyperbolic tangent), which is applied point-wise to the vector. The final layer is an activation layer that uses the softmax function.

Set how data are processed by the DNN

The `forward` method defines how the input data are processed by the DNN.

```
def forward(self, state):
    O_matrix = torch.from_numpy(state["nodes"]).float()
    l_t = state["visited"]
    if len(l_t) > 0:
        first_node = O_matrix[:, l_t[0]]
        current_node = O_matrix[:, l_t[-1]]
    else:
        # dummy values for the first action.
        # since nodes are in the [0, 1]x[0, 1] square,
        # we know that no node can be (-1, -1)
        first_node = -torch.ones(2)
        current_node = -torch.ones(2)

    o_t = torch.cat(
        [O_matrix.T.flatten(), first_node, current_node]
    ).unsqueeze(0)
    y_t = self.model(o_t)

    # creates a vector of 0s with the same
    nu_t = torch.zeros_like(y_t)
    length of y_t
    nu_t[:, l_t] = -1e6
    pi_t = self.softmax(y_t + nu_t)
    return pi_t
```

This code follows the explanation provided in Section 2.5.4, unless for a detail, that is the masking procedure. with one difference: the masking procedure. In the code snippet, the mask is represented by ν and has value 0 for unvisited nodes and value $-\infty$ (approximated as $-1e6$) for visited nodes. Additionally, this mask is applied before the softmax function.

We can demonstrate that this masking approach is equivalent to the masking method described in Section 2.5. Specifically, we aim to show that

$$\frac{p_t^i \mu_t^i}{\boldsymbol{p_t} \cdot \boldsymbol{\mu_t}} = softmax(\boldsymbol{y_t} + \boldsymbol{\nu_t})^i$$

Starting with the definition of softmax, we have:

$$softmax(\boldsymbol{y_t} + \boldsymbol{\nu_t})^i = \frac{e^{y_t^i} e^{\nu_t^i}}{\sum_{j=1}^{N} e^{y_t^j} e^{\nu_t^j}}. \tag{2.4}$$

We call $p_t^i = e^{y_t^i}$ and $\mu_t^i = e^{\nu_t^i}$ and we get:

$$\frac{e^{y_t^i} e^{\nu_t^i}}{\sum_{j=1}^{N} e^{y_t^j} e^{\nu_t^j}} = \tag{2.5}$$

$$\frac{p_t^i \mu_t^i}{\sum_{j=1}^{N} p_t^j \mu_t^j} = \tag{2.6}$$

$$\frac{p_t^i \mu_t^i}{\boldsymbol{p_t} \cdot \boldsymbol{\mu_t}}. \tag{2.7}$$

The masking technique used in the repository improves numerical stability.

The output of the forward method is a vector of probabilities for each action, that is our $\pi(a \mid s)$.

Action selection

The `select_action` method runs the `forward` method to obtain action probabilities and then samples an action based on these probabilities. It also returns the logarithm of the probability for the selected action, as this value is used in the loss function of the REINFORCE algorithm.

A.4 Baseline policies

The following code implements the *greedy* policy. This policy selects the closest feasible node as the next action at each step.

```
def greedy_policy(nodes):
    nodes = torch.from_numpy(nodes)
    length = 0
    current_node = nodes[:, 0]
    visited = [0]

    # for each node, select the closest node
    for i in range(1, nodes.shape[1]):
        # compute the distance to all the nodes from the current node
        distances = torch.norm(nodes - current_node.unsqueeze(-1), dim=0)
        distances[visited] = 1e6
        # pick the closest node
        closest_node = torch.argmin(distances).item()
        # update the current node
        current_node = nodes[:, closest_node]
        # add the distance to the length
        length += distances[closest_node]
        visited.append(closest_node)
    # add the distance from the last node to the first node
    length += torch.norm(current_node - nodes[:, 0])
    return length, visited
```

The greedy policy serves as the baseline policy during training.

A.5 REINFORCE

This is the core component of the repository, where all previously defined elements are combined to collect data and train the DNN.

The `REINFORCE` class is responsible for the training, which is handled by the `train` method.

To align the theoretical and practical aspects, we have divided the method into two phases: the data collection phase and the update phase.

Data collection phase

The data collection phase is implemented as a *for* loop around the `collect_one_episode` method, defined as follows:

```
def collect_one_episode(self):
    state = self.env.reset() #`reset` generates the starting state
    done = False
    log_probs = []
    rewards = []
    while not done:
        action, log_prob = self.agent.select_action(state)
        #`step` updates the state according to the action
        state, reward, done = self.env.step(action)
        rewards.append(reward)
        log_probs.append(log_prob)
    self.buffer.append((rewards, log_probs, state))
```

First, a new problem instance is created using the environment `reset` method. The agent then interacts with the environment until the done signal is received. At this point, the trajectory—comprising the state, action *log-probabilities* (the logarithm of the selected action probability), and rewards—is stored in the buffer. We save the log-probabilities instead of the actions themselves, as these values are needed in the REINFORCE loss function.

Update phase

During the update phase, the data stored in the buffer are used to update the parameters of the DNN.

```
def update(self):
    loss = 0
    # iterate over all stored trajectories
    for state, log_probs, rewards in self.buffer:
        # compute reward for a greedy policy
        greedy_policy_length, _ = greedy_policy(state["nodes"])

        returns = []
        G = 0
        for r in rewards[::-1]:
            G = r + self.gamma * G
            returns.insert(0, G)
```

```
            returns = torch.tensor(returns)

            log_probs = torch.stack(log_probs)
            score = G + greedy_policy_length
            loss -= torch.sum(log_probs) * score
            self.scores.append(score.item())
        self.optimizer.zero_grad()
        # performs backpropagation, i.e., computes the gradients
        (loss / len(self.buffer)).backward()
        self.optimizer.step()
        self.buffer = []  # reset the buffer
```

In the update phase, the method iterates over each stored trajectory in the buffer. For each trajectory, it first computes the reward for a greedy policy.

Next, it calculates the returns G_t by summing discounted rewards in reverse order. The log-probabilities are then stacked, and the score is calculated by adding G_t to the greedy policy length (the baseline). The total loss is computed as the negative sum of the log-probabilities weighted by the score.

Finally, the optimizer performs backpropagation to update the DNN parameters, after which the buffer is cleared.

Chapter 3

Machine Learning in Dynamic Routing Problems: A Survey

3.1 Introduction

In the previous chapter, we showed how, in the last 10 years, the fields of Operations Research (OR) and Machine Learning (ML) have increasingly intertwined, with a focus on how Deep Reinforcement Learning (DRL) can be used to solve routing problems in a static setting. In this chapter, we elucidate how ML has contributed to the subfield of dynamic routing problems by surveying its applications in the solution pipeline.

While the literature on static problems is thriving, a survey of ML approaches to dynamic problems is more coherent with the problem we will tackle in the next chapters.

Since the word "dynamic" is used with different meanings, we begin the chapter by briefly articulating what we mean by it in Section 3.2. The survey, then, continues with Section 3.3, wherein we describe the inclusion criteria. The main corpus of this survey, Section 3.4, is divided in two parts: the first one will present ML integrations within classic OR methods, starting by surveying approaches relying more heavily on OR methods and later approaches that lean more towards learning techniques; the second part will present ML approaches used to solve OR problems end-to-end. Finally, in Section 3.5, we will give our perspectives on directions for future research.

The rationale behind dividing the central Section 3.4 into two parts is to make it clear where the researchers have placed ML within the decision pipeline. In this way, we clarify how learning has been used, and can be used, to provide foresight (predictions, values, dispatch sets) while the optimizer guarantees structure (feasible routing, allocation, operator selection). We detail, for each paper, the optimization method, the learning model and training regime, and the integration point, enabling us to make more precise comparisons. In end-to-end ML approaches, the control logic is provided entirely by a learned policy or value function, without calling a combinatorial solver inside the decision loop; in these cases, performance hinges on the environment, reward design, and training.

This separation is consistent with, and indebted to, Ojeda Rios et al., 2021's taxonomy of exact, heuristic/metaheuristic, Approximate Dynamic Programming/Reinforcement Learning (ADP/RL), and hybrid solution streams. Our reorganization can be read as an adaptation of that taxonomy with a focus on learning approaches. Hence, most end-to-end works surveyed in this work correspond to Ojeda Rios et al., 2021 ADP/RL category, while what we call hybrid methods gathers their hybrid and heuristics/metaheuristics

categories. When we say "hybrid methods", we mean those in which a learned component cooperates with an optimization module that is invoked to enforce feasibility or to search the combinatorial space. The result preserves coverage while aligning the exposition with the practical question of how ML and optimization co-design improves routing under uncertainty.

One thing that emerged from this survey is that exact approaches mainly serve as methodological baselines on restricted instance sizes, or as proof-of-concept formulations that clarify the impact of specific operational constraints, as in Sabar et al., 2019 and Liu et al., 2022. They remain important because they can be used to establish rigorous lower or upper bounds that other methods can be compared against, even if their practical scope is limited (see, for instance, Kullman et al., 2022).

Earlier exact strands (branch-and-price with resource-constrained shortest paths) likewise reached optimality only on small/medium scales but shaped modern heuristics and evaluation practices—see Petris et al., 2025 for a tutorial on Branch-and-Price, Branch-and-Bound, and Branch-Price-and-Cut algorithms.

In general, exact methods now serve chiefly as precise reference points, clarifying constraint costs and providing benchmarks, while scalability remains their key limitation.

3.2 Dynamic, but how?

In this section, we will discuss what we mean when we say that a problem is dynamic. In doing so, we will try to separate a given problem—defined as a qualitative description—from its analytical representations and the solution methods these entail; in other words, a problem should be understood as the qualitative description of an instance (e.g., a vehicle that must visit all clients and return to the starting point), that intuition must be translated into a model, which then is solved in a given way.

As a starting point for our definition of dynamic, we report the definition of Dynamic Vehicle Routing Problem (DVRP) proposed by Psaraftis et al., 2016, which is: a Vehicle Routing Problem (VRP) is characterized as dynamic if the input on the problem is received and updated concurrently with the determination of the route. If all problem inputs are received before route determination and do not change thereafter, the VRP is static.

This means that, when talking about dynamic problems, we do not have all the information available but we must decide how to solve the problem given the partial information we have. From this definition, we can infer a first element of this class of problems: dynamic problems bear some form of informational scarcity. It should be noted that, according to the taxonomy of Psaraftis et al., 2016, there can be problems that have information scarcity but are not dynamic, as is the case of Static and Stochastic (SS) problems; the typical example is a variation of the Traveling Salesman Problem (TSP), known as Probabilistic TSP (PTSP), wherein a route must be found *a priori* knowing that a customer will be at a certain position with a given probability p—in this case, too, there is lack of information but the problem is static.

A second element of a dynamic problem is time. In fact, while we implement the (possibly partial) solution that we created, new information about the instance we are solving arrives. If there are no assumptions made or no prior knowledge about what probability distribution the unfolding information might follow, Psaraftis et al., 2016 talk about a Dynamic and Deterministic (DD) problem; whereas, if information is assumed or

known to be following a certain probability distribution, it is a Dynamic and Stochastic (DS) problem.

Subsequent surveys (Ojeda Rios et al., 2021; Mardešić et al., 2024) maintained this classification, the first contributing by adding a "Solution Methods" part to the taxonomy and the latter by slightly expanding on the resulting one. However, by admission of Psaraftis et al., 2016, the DD label is "misleading" in that it might lead to think that the problem is more tractable than the DS, when in fact that is not necessarily the case. For reasons that will become apparent later, we prefer to use the term "uncertain" for dynamic problems that do not use stochastic modeling, but we keep the DD notation for consistency with the existing literature.

Notice that these definitions cover only the initial step of the modeling process—that is, the high-level description of the problem—yet, they do not address how a dynamic problem could be modeled or solved. In fact, after defining a problem one should define its analytical representation. Hence, a number of possible solution methods follow. This distinction, which will seem obvious to more experienced readers, is necessary to underline how a dynamic problem, modeled accordingly, may still be solved in a static fashion. Consider, for example, a routing problem subject to operational constraints (e.g., capacities or time windows) and with an objective such as minimizing travel cost or delay, in which the length of edges changes at each time step without assumed or known probability distributions, i.e., a DD routing problem. One could still solve this problem using a heuristic that only considers the initial information, without adjusting the solution as information updates. Such a heuristic is highly unlikely to produce an optimal result, yet it would yield some results. Although this is a simple example, a similar choice might stem from perfectly reasonable or practical considerations, such as the need to manage limited computational power or to ensure the existence of analytical solutions.

Traditionally, stochastic modeling has been applied in the attempt to bridge this informational gap by relying on probability distributions to tackle uncertainty.

Another way to address uncertainty is known as "uncertainty-based" approaches. These methods have been used, in the OR literature, to solve problems where the probability distributions of uncertain parameters could not be known—thus, being akin to DD problems, for all practical purposes. In those cases, methods like Robust Dynamic Programming (RDP) or Distributionally Robust Dynamic Programming (DRDP) may be used because they allow to hedge against uncertainty by operating under "worst-case scenario" conditions.

In recent years, advances in ML and Deep Neural Networks (DNNs)—namely, deep learning—have prompted researchers to move out of worst-case scenario assumptions or any other assumption. A particularly promising approach in this sense is DRL. Indeed, a large share of recent work uses DRL, often *model-free* (as opposite to model-based), to approach dynamic and uncertain problems. In this family of methods, the agent learns directly from interactions with the environment without assuming knowledge of its underlying dynamics. Broadly, model-free DRL divides into three categories: *value-based* approaches, which estimate action values to derive policies, for example Deep Q-Networks (DQN) and their refinements such as Double DQN (DDQN) or Double Dueling DQN (D3QN); *policy-based* approaches, which optimize policies directly through gradient methods (e.g., REINFORCE); Actor–Critic (AC) algorithms, for example Advantage Actor Critic (A2C), Proximal Policy Optimization (PPO) and Soft-Actor Critic (SAC), that use both a value estimation and a policy function.

3.3 Inclusion criteria

The sources of literature are twofold. Some come from Mardešić et al., 2024 and others are the result of a Scopus query that is detailed in Appendix A. In both cases, these were the filtering criteria:

- For most papers, publication in a journal ranked Q1 by Scimago (i.e., the top 25% of journals);
- For ML-intensive papers, publication in a conference ranked A*/A by Computer Research and Education Association of Australasia's CORE Conference Ranking;
- All papers should be no older than 2021;
- All papers should focus on some use of ML to solve a dynamic and stochastic or uncertain problem.

Let us provide some context for these choices. The first filtering criterion is motivated by the need to restrict the scope of research, limiting it to papers that underwent the strictest review process. We made an exception for the ML world because, more often than not, the extremely high-pace research environment in that field is ill-suited for publication in academic journals review process; hence, time constraints make conference papers a more viable alternative. Thus, we chose to mirror the first criterion by referencing a well-known ranking system, including only the most selective conferences. When it comes to the third filtering criterion, we pick up where Ojeda Rios et al., 2021 survey left off. Finally, as we detailed in the previous section, we are only interested in dynamic and stochastic or uncertain problems.

3.4 Dynamic and stochastic or uncertain routing problems

In their simplest formulation, routing problems involve one or more vehicles tasked with visiting all, or a subset of, nodes in a given graph. The objective then is to find the best tour, i.e., the tour that minimizes a cost. This cost can be of various kinds: the length of the tour, the time needed to complete the tour, the amount of waiting time, the emissions of greenhouse gas. Sometimes, instead of minimizing a cost, routing problems might be defined as maximization of some kind of prize—such as, profits or the service level to clients. Despite the simple formulation, the problem is all but trivial to solve, particularly in dynamic settings. Moreover, note that each element of the description above can be modified to increase the complexity of the problem—for instance, multiple vehicle may be utilized with various constraints on vehicle capacities or node visit timing.

As anticipated in the introduction, this section is divided into two parts—the first one covers hybrid methods while the second surveys end-to-end applications of ML to OR problems.

For each paper, we report the problem solved, the OR algorithm, the ML model, the training paradigm, and how the two are integrated (if applicable). In the hybrid approaches subsection, items are ordered from the most OR-intensive use of ML to the least, so the transition toward ML-dominant approaches is gradual.

Table 3.1 offers a synopsis of the specific problem at hand in each paper (Problem), which element of the formulation are stochastic (Stochasticity), which OR and ML approaches the authors deploy to find a solution (OR, ML), how the algorithm got trained (if applicable) (Training), and whether they relied more on OR or ML approaches (Pipeline). In this case, papers are ordered alphabetically.

Problem acronyms used in Table 3.1 are shortly described in Table 3.2. The other acronyms are briefly expanded upon below:

These are the acronyms used in Stochasticity:

- DEM denotes demand uncertainty, covering stochastic request, order, passenger, as well as stochastic demand volumes.
- TRV denotes travel-time or network-state uncertainty, including time-dependent or stochastic travel times, speeds, and traffic flows.
- SVT denotes service-time uncertainty, including preparation, handling, or turnaround durations at stops.
- RES denotes resource or energy uncertainty, including stochastic energy consumption and battery state of charge.
- SUP denotes supply availability uncertainty, including fluctuating availability of couriers, vehicles, or third-party capacity.
- BEH denotes behavioral or acceptance uncertainty, including stochastic acceptance or participation probabilities.

OR methods are as follows:

- PC-HGS denotes Prize-Collecting Hybrid Genetic Search.
- Ass.–reb.–hire denotes assignment–rebalancing–hiring optimization.
- BD-CVH and BD-AVH denote Balanced Dynamic Closest Vehicle Heuristic and Balanced Dynamic Assignment Vehicle Heuristic.
- TSM–TW denotes a TSP-with-Time-Windows optimizer used to generate feasible route candidates.
- HGS denotes Hybrid Genetic Search.
- Online LNS denotes a problem-specific online Large Neighborhood Search.
- Structured dispatcher denotes a custom feasibility-enforcing dispatcher.
- B&P and B&C denote Branch-and-Price and Branch-and-Cut.
- SPPRC and ESPPRC denote Shortest Path Problem with Resource Constraints and its Elementary variant, respectively.
- Sched. DP denotes a scheduling dynamic program on routes or fragments.
- Lightweight scheduler denotes a simple feasibility-enforcing dispatcher.

Table 3.1: Summary by authors, problem, stochastic elements, OR/ML components and integration.

Paper	Problem	Stochasticity	OR	ML	Training	Pipeline
Basso et al., 2022	SDEVRP	TRV; RES	—	MLP	(Safe)RL	End-to-end
Baty et al., 2024	DVRPTW (EMN 2022)	DEM	PC-HGS*	MLP	SL	Hybrid-OR
Beirigo et al., 2022	AMoD ridesharing	DEM; TRV; SUP	Ass.–reb.–hire*	MLP	ADP	Hybrid-OR
Bono et al., 2021	MA-DVRP	DEM; TRV	—	Attn	RL†	End-to-end
Chen et al., 2025	DTSP-TDS	TRV	—	GNN+Attn	RL	End-to-end
Chen et al., 2022	SDD (with dron.)	DEM; TRV; SVT	—	MLP	RL	End-to-end
Chen et al., 2023	SDD (with fair.)	DEM	—	MLP	RL**	End-to-end
Du et al., 2021	Simult. PD-TSP	DEM	—	Attn	RL	End-to-end
Garn, 2021	BD-mTSP	DEM	BD-CVH* / BD-AVH*	Reg	SL	Hybrid-OR
Gubena et al., 2025	Courier	DEM; TRV; BEH	TSM–TW	Attn	SL+RL	Hybrid-ML
Jahanshahi et al., 2022	Meal del. DVRP	DEM; TRV; SVT	—	MLP	RL	End-to-end
Kullman et al., 2022	MoD EV	DEM; TRV; RES	—	MLP	ADP	End-to-end
Kwon et al., 2022	DVRPTW (EMN 2022)	DEM	HGS	MLP	RL	Hybrid-OR
Liu et al., 2022	FSTSP	TRV	—	MLP	RL	End-to-end
Neria et al., 2024	DPA-F (fairness)	DEM	Online LNS*	MLP	RL	Hybrid-OR
Santiyuda et al., 2024	MO-DMV-PD	DEM; TRV	—	Attn	RL**	End-to-end
Silva et al., 2023	Crowdshipping	DEM; TRV; SUP	Structured dispatcher*	MLP	RL	Hybrid-OR
Sippel et al., 2025	PDPTW-HoS	—	B&P/B&C (SP-PRC/ESPPRC)* + sched. DP	Clf	SL	Hybrid-OR
Sun et al., 2025	DFPDP-F	DEM; TRV; SVT	Lightweight scheduler*	Seq	SL	Hybrid-ML
Wu et al., 2024	(CBus)	DEM; TRV	—	Attn	RL†	End-to-end
Xiang et al., 2024	DMV-PD	DEM; TRV; SUP	—	Attn	RL	End-to-end
Xu et al., 2023	DPDPTL	DEM	ALNS	Tab	RL (online)	Hybrid-OR
Zhang et al., 2023	DTSP/DPDP	DEM; TRV	—	Attn	RL	End-to-end
Zhou et al., 2023	Courier accept/dispatch	DEM; TRV	—	MLP	SL+RL	End-to-end

Legend: * = custom algorithm; ** = multi-objective learning; † = multi-agent learning

Table 3.2: Glossary of Routing Problems in Table.

Acronym	Meaning
SDEVRP	Stochastic dynamic electric vehicle routing with online demand, time-varying travel times, and battery/charging decisions under energy-risk constraints.
AMoD	Autonomous mobility-on-demand ride-pooling with dispatch, rebalance, and optional third-party hiring to meet service-level contracts under uncertain demand.
MA-DVRP	Multi-agent dynamic vehicle routing in which vehicles act as decentralized agents coordinating assignments and routes in a shared, time-evolving environment.
DTSP-TDS	Dynamic traveling salesman with time-dependent and stochastic edge times; tours are resequenced online as network conditions evolve.
SDD	Same-day delivery platform control that jointly decides order acceptance and courier assignment under time windows, service times, and fairness or service-level goals.
Simult. PD-TSP	Simultaneous pickup–delivery TSP with precedence and capacity constraints (e.g., lockers/handling), solved with feasibility-aware decoding in dynamic settings.
BD-mTSP	Balanced dynamic multiple-TSP emphasizing real-time assignment with explicit workload balancing across vehicles as requests arrive online.
Courier	Courier routing with online requests where candidate routes must satisfy TW/precedence; policies gate between optimizer-built and learned sequences.
Meal del. DVRP	Meal-delivery dynamic VRP coupling order acceptance and dispatching with food preparation/service-time uncertainty and delivery time windows.
DVRPTW (EMN 2022)	Dynamic VRP with time windows from the EURO meets NeurIPS benchmark——used as the problem setting by Baty et al., 2024 and Kwon et al., 2022——where requests appear online and must be inserted feasibly under tight decision budgets.
MoD EV	Mobility-on-demand with electric vehicles (EV) integrating accept/reposition/recharge under stochastic travel times and energy consumption with SoC feasibility.
FSTSP	Flying sidekick TSP coordinating a truck and a drone with stochastic travel times and strict launch–rendezvous feasibility.
DPA-F	Dynamic pickup–allocation with fairness objectives over sites, deciding near-term routes and allocations while respecting time/capacity constraints.
MO-DMV-PD	Multi-objective dynamic multi-vehicle pickup–delivery balancing efficiency and fairness/service metrics via preference-conditioned control.
Crowdshipping	Last-mile delivery with crowdshippers where a learned policy proposes accept/assign decisions and a structured dispatcher enforces feasibility.
PDPTW-HoS	Pickup–delivery with time windows and hours-of-service regulations, requiring duty-time scheduling on routes/fragments and associated feasibility cuts.
DFPDP-F	Dynamic flexible pickup–delivery with fairness constraints and lightweight online scheduling under stochastic orders, prep times, and travel times.
CBus	Customized bus routing with online passenger demand, multi-stop assignment, and capacity/schedule feasibility at network scale.
DMV-PD	Dynamic multi-vehicle pickup–delivery with rolling feasibility masks for TW/precedence/capacity and centralized coordination across vehicles.
DPDPTL	Dynamic pickup–delivery with transshipments and LIFO loading, enforcing synchronization and loading discipline across legs.
DTSP/DPDP	Dynamic TSP and dynamic pickup–delivery focusing on rapid resequencing under evolving arrivals and time-dependent travel.
Courier accept/dispatch	Courier platforms' accept/dispatch problem combining admission control and assignment under uncertain arrivals and travel/service-time limits.

- ALNS denotes Adaptive Large Neighborhood Search.

Whereas ML ones are the following:

- MLP denotes a plain feedforward DNN (Multi-Layer Perceptron) used for policies, value functions, prizes, or priorities, without attention, recurrence, or graph message passing.
- Attn denotes attention-based models that weigh candidates context-dependently.
- GNN denotes Graph Neural Networks with message passing on vehicle–request graphs; GNN+Attn combines message passing with attention-based models.
- Seq denotes sequence models without attention (e.g., RNN/LSTM/GRU or parallel encoder with serial decoder) used to output action/order sequences.
- Reg denotes non-neural regression (e.g., continuous-approximation) used as a surrogate or predictor inside the pipeline.
- Clf denotes a classifier (binary or multiclass) used for pruning (e.g., route/fragment filtering).
- Tab denotes a tabular approach (lookup/Q-table) without a neural approximator.

Training and pipeline acronyms are as follows.

- SL denotes Supervised Learning.
- SL+RL denotes combined Supervised and Reinforcement signals (e.g., supervised pretraining with RL refinement or mixed losses).
- RL denotes Reinforcement Learning; + "(online)" denotes RL updated during execution on the instance; "(Safe)" + denotes RL with explicit safety or feasibility constraints.
- ADP denotes Approximate Dynamic Programming.
- End-to-end denotes controllers that do not call a combinatorial optimizer at inference.
- Hybrid-OR denotes OR-dominant hybrids with ML as an auxiliary component.
- Hybrid-ML denotes ML-dominant hybrids with optimization guardrails.

3.4.1 Hybrid methods

This section follows a gradual shift in how ML influences the solution method. First, in the OR dominant class of solution methods, we find papers that use ML to reduce the search space for the OR optimizer, i.e., Sippel et al., 2025; Garn, 2021. Next, operator tuning, as implemented by Xu et al., 2023, where learning adjusts the frequency or choice of destroy–repair moves inside a metaheuristic. Then value or cost shaping, where predictive models add surcharges or credits to guide the objective toward high-payoff decisions, such as in Neria et al., 2024; Beirigo et al., 2022; Kwon et al., 2022. Then, Silva et al., 2023 exemplifies an optimizer-in-the-loop approach. At each policy-improvement step, a Mixed

Integer Linear Program (MILP) selects a first-stage visit order using the current value estimate learned by a DRL value network. The DRL component is trained by alternating policy evaluation on simulated scenarios——where fixed recourse rules produce returns used to fit the value network——and policy improvement via the MILP. After that, in Baty et al., 2024, we look into differentiable combinatorial optimization, where an OR method sits inside the learning loop so gradients can shape routing choices. Finally, we reach feasibility-masked ML policies that rely on action masking and light OR guardrails to enforce constraints while the network selects among candidates without invoking a full reoptimizer, as in Gubena et al., 2025; Sun et al., 2025; Pan et al., 2023. These methods prioritize learned selection over embedded OR reoptimization, using masks or simple dispatching to keep choices feasible.

Hybrid methods: OR-dominant

In this subgroup, an optimization or metaheuristic engine performs the combinatorial work, while a learned component supplies forecasts, surrogates, or operator guidance that steer the optimizer.

A strongly OR-dominant example is the one introduced by Sippel et al., 2025. In their work, the authors pair exact and metaheuristic solvers with a minimal ML screening step to solve a Pickup and Delivery Problem with Time Windows and Hours-of-Service constraints (PDPTW-HoS). By screening we mean a conservative preselection that quickly filters candidates by passing only likely high-value or feasible ones to the optimizer. The first OR component is a route-based set-partitioning model solved by Branch-and-Price with a (Elementary) Shortest-Path Pricing Resource-Constrained subproblem (SPPRC/ESPPRC). The second OR component is a fragment-based formulation solved by Branch-and-Cut. Hours-of-service feasibility is handled by a scheduling dynamic program on routes/fragments together with lifted feasibility/optimality cuts. The ML component is a supervised binary classifier trained on instances with known solutions to flag routes or fragments likely to appear in an optimal solution (the approach is model-agnostic and does not hinge on a specific binary classifier). The ML and OR components integrate in the pre-optimization phase as pruning: the classifier filters the route/fragment pool before column/fragment generation and subsequent exact or metaheuristic solving, shrinking the problem without altering the core search.

Garn, 2021 addresses the Balanced Dynamic multiple-TSP (BD-mTSP) with two online heuristics—the Balanced Dynamic Closest Vehicle Heuristic (BD–CVH) and the Balanced Dynamic Assignment Vehicle Heuristic (BD–AVH)——that construct and update routes in real time while explicitly balancing workload across vehicles. Around these online rules, the authors build Continuous-Approximation (CA) regressions that learn to predict route length from system descriptors (e.g., number of vehicles and customers, measures of dynamism). The CA models are trained in a supervised fashion on solutions produced by the heuristics over a broad testbed, yielding calibrated surrogates with mean absolute percentage error below 3%. Integration is advisory rather than prescriptive: the learned CA surrogates inform parameter tuning and provide quick, strategic performance estimates, whereas the actual routing decisions at run time are executed by BD–CVH/BD–AVH without a learned controller in the loop.

In the Dynamic Pickup–Delivery setting with Transshipments Loading (DPDPTL) and Last-In-First-Out (LIFO), Xu et al., 2023 fuse online learning with a classical Neighborhood Search (NS) so that the search adapts to precedence and synchronization require-

ments on the fly. The search backbone is an Adaptive Large Neighborhood Search (ALNS) seeded by a Clarke–Wright savings construction. ALNS cycles through destroy/repair operators to explore the solution space while enforcing transshipment timing and LIFO feasibility. Instead of fixing operator frequencies, the authors attach a Q-learning controller that learns which destroy/repair move to apply next. After ALNS executes the chosen operator and reoptimizes, the controller receives a reward signal (improvement and feasibility) and updates the operator's value, gradually shifting probability mass toward moves that yield progress under LIFO and synchronization constraints. Training is purely online, so the policy adapts during the run to instance-specific structure. Integration is in-loop: Q-learning selects the operator, ALNS executes and repairs, and the feedback updates the Q-table before the next iteration. Empirically, this adaptive scheme helps the search escape local minima and maintain feasibility relative to a non-learning ALNS, particularly in instances where transshipment synchronization and LIFO rules make standard neighborhoods brittle.

In a Dynamic Pickup–Allocation problem with Fairness setting (DPA-F), Neria et al., 2024 couple a Large Neighborhood Search (LNS) with a learned value signal so that exploration is steered before decisions are committed. The optimization side is an online LNS that proposes candidate subroutes together with ending inventories and allocations under time and capacity constraints; the learning side is a DNN that estimates the downstream value of each state–action pair. Training proceeds via RL tailored to an action space too large to enumerate, so the value model is fitted from simulated trajectories rather than supervised labels. Integration happens at the candidate-evaluation step: at each decision epoch the LNS generates multiple feasible moves and the network scores them, biasing the search toward high-value subroutes that improve both throughput and equity. On instances with 38–180 sites drawn from Berlin Foodbank data and synthetic sets, the reported averages show gains of 57.9% over a myopic selector, 28.2% over a single-scenario lookahead solved by a constrained LNS, and 41.6% over an operations-inspired rule.

The problem in Beirigo et al., 2022 is autonomous ridesharing under uncertain demand with service-level contracts for heterogeneous user groups and the option to hire idle third-party vehicles on demand. The objective is to dispatch, rebalance, and occasionally hire so as to meet delay/rejection penalties while controlling operating and hiring costs on a high-resolution Manhattan network. The OR core is a repeated assignment–rebalance–hiring program that decides vehicle–request matches, empty repositions, and when to hire external vehicles under capacity and service rules. The ML side supplies foresight by estimating the marginal value of a vehicle at each time–location state via a value-function approximator trained with approximate value iteration on simulated trajectories. Integration is objective shaping: the learned marginal values enter the optimizer as surcharges/credits, biasing it toward high-impact dispatch and rehiring decisions while striving to honor service-level guarantees. In operation, the loop is simulate to update values, solve the optimization with the current value surcharges, implement the first-period decisions, and iterate.

In stochastic last-mile delivery with crowdshipping, Silva et al., 2023 combine a learned controller with a structured dispatcher. A DRL policy, trained in simulation to cope with uncertain arrivals, proposes accept/assign actions; a dispatch module then solves the assignment and routing subproblems under time-window and capacity constraints, enforcing feasibility and finalizing matches. This policy-in-the-loop interaction outperforms both sample-average and distributionally robust optimization baselines, indicating that learned guidance can raise service quality while the optimizer preserves constraint

fidelity.

Next, we present two papers ranking second and first, respectively, in the EURO meets NeurIPS 2022 competition; see Kool et al., 2023 for the problem specification and to read the other contribution, reported only in Table 3.2 for conciseness' sake.

The winner of that competition is what we could call an optimization–enriched ML approach, proposed by Baty et al., 2024. The dynamic problem is reformulated as a prize-collecting VRP with Time Windows (VRPTW) solved by a Hybrid Genetic Search (HGS) metaheuristic; the prize vector is produced by a DNN trained to assign higher prizes to customers that should be dispatched. Since the final solution depends both on the input problem and on the prize vector, the training must consider the function that takes as input the problem and outputs the solution obtained with HGS as a layer of the DNN over which to optimize, called CO layer. This DNN is trained in a supervised imitation setting from an anticipative policy, using a custom loss function that makes the CO layer amenable to gradient-based learning. Integration occurs via the CO layer: at inference, the DNN outputs prizes and HGS solves the induced prize-collecting instance to yield dispatch-and-route decisions; during training, the CO layer's output is compared to the anticipative target and the loss is optimized by backpropagation with stochastic gradient descent. This division of labor—DNN for foresight, HGS for feasibility and combinatorial search—yielded the winning solution in the EURO meets NeurIPS 2022 dynamic VRPTW and remained competitive in extended tests beyond the competition, delivering online-compatible performance against greedy and sampling-based baselines.

The runner-up (Team SB) Kwon et al., 2022 flips the emphasis: RL decides what to dispatch each epoch, and the optimizer decides how to serve it. A state-of-the-art Hybrid Genetic Search (HGS) constructs time-window–feasible routes, but two deep networks–—one assigning request priorities, another selecting the dispatch subset—are trained by policy-gradient reinforcement learning on a simulator and feed learned signals into HGS. Integration is twofold: the priority network reshapes costs in the HGS objective, and the subset network filters the candidate requests before routing; HGS then performs the combinatorial search given these learned inputs.

Hybrid methods: ML-dominant

We now move from OR-dominant hybrids to ML-dominant hybrids that rely on feasibility-masking.

Gubena et al., 2025 pair a feasibility-masking policy learned through DDQN (i.e., a controller they call i-DeepRoute) with a MILP model devised to enforce operational constraints (they call TSM-TW) and a precedence-aware particular form of attention based architecture—i.e., a transformer. Hybridization stems from the embedding of a mixed-integer TSM-TW optimizer as a base learner that constructs a feasible route candidate at inference; the policy then scores and selects between this optimizer-produced route and the learned transformer route. On real Dada Express data, the method reduces location square deviation by 29.14%, lowers edit distance by 48.33%, raises complete-match rate by 19.20%, and improves rank correlation by 6.67% relative to state-of-the-art learning baselines and, per metric, the strongest comparator including TSM-TW, while using 85% fewer parameters and 57.7% less training time.

By contrast, Sun et al., 2025 tackle the Dynamic Flexible Pickup and Delivery Problem with Fairness (DFPDP-F) with a sequence model at the core. A parallel-encoder/serial-decoder is trained offline on synthetic and realistic order streams to generate candidate

action sequences, while a scheduling mechanism selects the vehicle with the lowest current time and enforces time windows and fairness rules. The result is a sequencer-proposes, scheduler-enforces workflow that finds feasible plans quickly and achieves more balanced task allocations than classical heuristics.

On the far end of the spectrum, Pan et al., 2023 propose a ML–dominant hybrid for the Dynamic and Uncertain Capacitated VRP (DU-CVRP) that uses an encoder–decoder with graph embeddings and an Asynchronous Advantage Actor-Critic (A3C) learner (see Mnih et al., 2013), enforcing capacity via penalty shaping rather than an OR reoptimizer. On classic CVRP benchmarks with different spatial distributions, namely uniformly distributed versus clustered customers, the policy achieves 1.58–9.75% gaps to OPT on the uniformly distributed set with a mean of 5.79% and per-instance times around 0.45–1.22 s—that is, the trained model produced solutions achieving the stated optimality gaps in roughly 0.45–1.22 s per instance. On the clustered set selected results are 3.73% at 0.265 s and 7.03% at 0.359 s, with a worst case 12.70% at 0.421 s, all versus OPT maintaining matched fleet sizes. Overall, the contribution is an end-to-end DRL dispatcher with minimal OR auxiliary structure and performance reported as gaps and times on standard benchmarks (Pan et al., 2023).

Comparison

Viewed as a whole, the hybrid subset does not identify a single dominant integration mechanism so much as a set of trade-offs between where learning sits and what the OR core guarantees. Screening and pruning reduce search effort without changing the solver's guarantees, but their utility depends on labeled data and can be brittle when the screened features shift across cities or time, while gains arise mainly from compute savings rather than qualitatively different routing behavior (Sippel et al., 2025; Garn, 2021). Operator tuning adapts metaheuristics online with limited engineering burden and preserves feasibility by construction, yet it exposes the search to reward-selection biases and may overfit operator frequencies to specific synchronization patterns (Xu et al., 2023). Value or cost shaping injects anticipative signals into objectives and can surface multiobjective preferences such as fairness or service reliability, but it raises risks of rewarding or penalizing the same underlying effect twice and requires explicit guardrails to prevent infeasible or myopic choices under nonstationary demand (Neria et al., 2024; Beirigo et al., 2022; Kwon et al., 2022). Differentiable combinatorial optimization tightly couples prediction and routing by embedding the solver in the learning loop. It requires loss functions that align with the discrete objective and yield stable gradients, typically via perturbed or surrogate objectives that approximate how small changes in scores alter the solver's output (Baty et al., 2024). Because each training step invokes the embedded solver, the total training time grows with the solver's own complexity and the instance size. As a result, inference after training is relatively light, but a large share of computation is paid upfront during learning.

On the ML-dominant side, feasibility-masking policies provide expressivity with minimal reoptimization, yet their reliability hinges on mask completeness and the stability of learned scores, especially when penalty terms substitute for explicit constraint handling (Gubena et al., 2025; Silva et al., 2023; Sun et al., 2025; Pan et al., 2023).

In sum, the corpus supports a neutral reading. Placing learning outside the solver (screening, tuning) favors stability and maintainability but may cap upside when anticipative behavior is crucial. Placing learning inside the objective or through a differentiable

solver raises potential gains at the cost of data fidelity, design complexity, and sensitivity to shift. ML-dominant feasibility-masking offers a lightweight middle ground provided feasibility defenses are explicit rather than implicit in rewards. Under current evidence, none of these choices is universally superior, and mechanism selection is best viewed as a design decision about where to locate uncertainty and where to keep guarantees.

3.4.2 End-to-end ML applications

This section follows more closely Ojeda Rios et al., 2021 and covers end-to-end ADP/RL controllers where inference does not call a combinatorial optimizer. We present results by technique to emphasize what each architectural choice buys in dynamic settings.

Necessary concepts

A major component in RL/DRL is training methods, i.e., how the agent learns to behave. We begin this section by briefly reporting the major families of training methods, also presented in Table 3.3.

Table 3.3: RL/DRL algorithms with citing papers and problem contexts.

Algorithm	**Papers**	**Problems**
A2C	Liu et al., 2022	FSTSP
AC	Bono et al., 2021	MA-DVRP
Approximate Policy Iteration	Silva et al., 2023	Crowdshipping
Custom algorithm	Zhou et al., 2023	Courier
DDQN	Gubena et al., 2025; Jahanshahi et al., 2022	Courier; Meal del. DVRP
DQN	Chen et al., 2022; Chen et al., 2023; Du et al., 2021; Jahanshahi et al., 2022; Liu et al., 2022	SDD (with dron.); SDD (with fair.); Simult. PD-TSP; Meal del. DVRP; FSTSP
Dueling DDQN–PER (D3QN–PER)	Jahanshahi et al., 2022	Meal del. DVRP
Multi-Objective RL	Santiyuda et al., 2024	MO-DMV-PD
Q-learning	Basso et al., 2022; Neria et al., 2024, Xu et al., 2023	SDEVRP; DPA-F; DPDPTL
REINFORCE	Chen et al., 2025; Kwon et al., 2022; Wu et al., 2024; Xiang et al., 2024; Bono et al., 2021; Zhang et al., 2023	DTSP-TDS; DVRPTW (EMN 2022); CBus; DMV-PD; MA-DVRP; DTSP/DPDP

Value-based methods estimate action values and select greedy actions with respect to those estimates. Q-learning is an off-policy temporal-difference algorithm that updates a tabular or parametric estimate toward the Bellman optimality target, typically with ϵ-greedy exploration to ensure coverage of the action space (Watkins et al., 1992). Deep Q-Networks approximate $Q_\theta(s, a)$ with a DNN and stabilize learning via experience replay and a slowly updated target network to form bootstrap targets for the temporal-difference loss (Mnih et al., 2013). Double DQN reduces the overestimation bias by decoupling action selection and evaluation across the online and target networks, yielding more accurate targets and improved stability (van Hasselt et al., 2015). Dueling architectures further decompose action values into a shared state-value and an advantage head, which is advantageous when many actions have similar effects, and are often combined with Double DQN and prioritized replay in practical variants (Wang et al., 2015).

Policy-gradient methods optimize a stochastic policy by ascending an estimate of the performance gradient. REINFORCE performs Monte Carlo policy-gradient updates based on episodic returns, commonly with a learned baseline to reduce variance while

keeping the estimator unbiased (Sutton et al., 1999). Actor–critic algorithms instantiate this baseline with a learned value function, enabling lower-variance, more sample-efficient updates and supporting on- or off-policy variants (Grondman et al., 2012). Advantage Actor–Critic leverages an advantage estimate to center the policy gradient and synchronizes policy and value updates, a design that is frequently used in routing settings with coupled timing and feasibility constraints.

Multi-agent policy gradient extends these ideas to systems with multiple decision makers. A common design is centralized training with decentralized execution, where a centralized critic conditions on joint or aggregated information during learning, while each agent executes its local policy at inference; REINFORCE-style estimators and actor–critic updates are adapted to aggregate per-agent trajectories into a global gradient signal (Grondman et al., 2012).

Multi-objective reinforcement learning introduces preference conditioning to trade off among conflicting goals. A two-stage scheme pretrains a single-objective policy and then freezes most of the network while training a preference-conditioned head or hypernetwork to trace Pareto operating points across objectives such as efficiency and fairness, allowing fast adaptation without retraining the full model (Santiyuda et al., 2024).

Finally, stochastic optimizers used to train these models are not RL algorithms themselves but underpin gradient-based learning. Stochastic Gradient Descent and Adam are standard choices for minimizing the chosen surrogate losses in value and policy networks and are ubiquitous in deep reinforcement learning implementations (Bottou, 2010; Kingma et al., 2014).

Before delving into the discussion, however, we think that a brief introduction to the notion of "masking" may help readers unfamiliar with ML jargon. Masking denotes feasibility-aware action taking that removes illegal actions from the candidate set before the policy evaluates or samples them. Operationally, masked actions receive a prohibitive score so that the agent never selects them, yielding constraint satisfaction without invoking a full optimizer at inference. Typical instances include precedence and capacity masks in simultaneous pickup–delivery TSP with lockers, where infeasible next stops are zeroed out during Q-evaluation (Du et al., 2021). Rolling feasibility masks forbid vehicle–request pairs that break pickup–before–delivery, time windows, or ride-time bounds in dynamic multi-vehicle pickup–delivery with crowdshippers (Xiang et al., 2024). Pointer decoders for dynamic TSP mask already visited nodes and time-window violations while attending to time-dependent stochastic travel times (Chen et al., 2025). Accept/assign controllers mask drone–vehicle pairings that violate rendezvous or service-time constraints, pruning the action space before value computation (Chen et al., 2022). Feasibility-masking filters candidate routes that fail TSP–TW feasibility checks so the selector only compares viable options, as in Gubena et al., 2025.

For dynamic TSP–style sequencing, Chen et al., 2025 provide the clearest step forward by coupling DRL with a dynamic graph temporal–attention encoder that explicitly models time-dependent and stochastic travel times. The encoder attends over node–edge features with time indices so that the policy can weight candidate moves by predicted future travel conditions, and a pointer-style decoder outputs the next city under a feasibility mask. In controlled studies, this approach improves both tour quality and solving time relative to rolling-horizon heuristics and prior RL baselines, indicating that temporal structure at the edge level is the binding constraint. Earlier attention-based AC models also help on moderate instance sizes. Zhang et al., 2023 use an encoder–decoder with masked attention that prevents illegal moves and yields $O(n)$ scoring per decision, reporting more

than 5% tour-cost reductions against constructive and rollout baselines on DTSP/DPDP (up to 40 customers). Du et al., 2021 embed multi-head attention inside a DQN so that the Q-network aggregates local context over candidates while a feasibility mask enforces pickup–delivery rules and capacities, enabling fast $O(n)$ evaluations that beat stochastic-optimization rules in a simultaneous pickup–delivery TSP with lockers.

Value-based accept/dispatch for same-day and courier platforms shifts the bottleneck from routing to admission and assignment under uncertainty. Zhou et al., 2023 first distill a base policy by supervised imitation from historical courier choices and then refine it with RL, serving on average 12.07% more customers than a Dijkstra dispatcher and 8.38% more than Nearest Neighbor (NN) in high- and low-density settings, respectively, on matched networks and loads. The model maps system snapshots to accept/assign actions using a feasibility-aware decoder so that assignments remain within time-window and capacity limits while the reward emphasizes served volume and timeliness. Chen et al., 2022 cast joint drone–vehicle accept/assign as a DQN, representing candidate pairings in the state and using action masks to exclude infeasible matches, which consistently dominates threshold and insertion-cost heuristics across vehicle counts and demand intensities. Jahanshahi et al., 2022 demonstrate an end-to-end DRL controller that directly outputs dispatch decisions from state features with masks for feasibility, improving timely service versus reoptimization-style baselines at similar decision budgets.

Truck–drone joint control tests whether RL can coordinate heterogeneous assets under stochastic travel times. Liu et al., 2022 formulate a joint policy over truck routing, drone launches, and retrieval scheduling, with the state encoding relative positions and remaining times and the action constrained by launch–rendezvous feasibility masks. The learned controller reports delivery-time reductions up to 28.65% on the stochastic flying-sidekick TSP versus a mixed-integer program and a dynamic local search, and a 32.68% reduction in a city-scale case study using real traffic, all on matched instances and time budgets.

Centralized and multi-objective policies address feasibility and fairness when multiple vehicles compete for tasks. Xiang et al., 2024 use a centralized attention model with a rolling feasibility mask for Dynamic Multi-Vehicle Pickup-and-Delivery (DMV-PD) with crowdshippers and obtain the lowest total cost under matched sampling scales against learning and heuristic baselines. A centralized attention model is a single controller that observes the full system state and computes joint assignment scores by attending over all vehicles and tasks at once, which enables coordinated choices that decentralized per-vehicle policies may miss. The rolling mask zeroes out infeasible vehicle–request pairs due to pickup–delivery precedence, time windows, or capacity so the policy cannot select illegal actions during inference.

Santiyuda et al., 2024 train a two-stage Multi-Objective RL (MORL) with heterogeneous attention and hypernetworks that approximates Pareto fronts in minutes and improves hypervolume and ϵ^+ relative to multi-objective evolutionary algorithms (MOEAs) across multi-city streams. MORL optimizes several objectives simultaneously by learning policies that trade off among them, often via a preference vector that selects operating points along the frontier. In this design the hypernetwork conditions on the preference vector to generate policy parameters, with a first stage that learns single-objective experts and a second stage that distills them into a conditional policy spanning the Pareto set.

Mobility-on-demand, EV logistics, and transit planning scale the state and action spaces. Kullman et al., 2022 learn value approximations that jointly coordinate accept,

reposition, and recharge decisions and outperform a strong rolling reoptimizer while staying consistent with dual bounds on Manhattan-derived instances. The controller is an ADP design with a centralized value function over zone–time features and battery states, where actions are decoded with feasibility masks for charging availability and energy limits and exploration trades off short-term revenue against future serviceability. Basso et al., 2022 deploy safe RL with offline value estimation to cut energy consumption under chance-constrained feasibility in stochastic EV routing. Safety is implemented via a constraint layer and action masking that prevent state-of-charge violations while the critic learns a risk-aware value proxy from logged trajectories, and the policy chooses charging and routing actions under stochastic consumption. Wu et al., 2024 design a multi-agent encoder–decoder that replans customized bus routes online and lowers network cost with improved adaptiveness versus heuristic controls. Agents correspond to buses and are trained with centralized critics and decentralized execution, using attention to aggregate passenger and stop features and masks to enforce capacity and schedule constraints during decoding. At network level, Lin et al., 2021 show that distributed Multiple-Agent Reinforcement Learning (MARL) alleviates congestion in a software-defined Internet of Vehicles (IoV) by coordinating local routing policies, using neighbor-aware critics to stabilize learning under partial observability. Multi-agent reinforcement learning trains multiple decision makers that interact in the same environment and learn policies jointly or cooperatively. A common design is centralized training with decentralized execution, where critics or value estimators access joint or neighbor-aggregated information during learning, while each agent acts locally at inference using only its partial view. In networked routing this translates into agents choosing local actions that reduce delay and spillback while aligning with global flow objectives, with feasibility masks enforcing capacity and signal constraints. Bono et al., 2021 demonstrate that attention mechanisms in multi-agent DRL remain competitive in dynamic stochastic routing by letting each agent weight nearby requests and vehicles before committing to actions under feasibility masks.

Methodology head-to-head and scope transfer clarify when and why to use RL. Akkerman et al., 2025 compare value- and policy-based control with linear and neural approximators and find no uniform winner, with neural models advantaged by strong state interactions and linear models competitive in simpler dynamics. The study fixes simulators and decision budgets and varies the function class and exploration scheme, revealing predictable trade-offs between sample efficiency, stability, and action-space search.

Comments

Reading across the end-to-end stream, the common design is to replace explicit reoptimization with a learned value or policy, while retaining feasibility via masking and simple decoders. Temporal encoders that expose time-varying edge conditions consistently appear as the most effective architectural choice when sequencing dominates, as in the dynamic TSP variants by Chen et al., 2025; Zhang et al., 2023; Du et al., 2021. Where the operational bottleneck is admission rather than routing, value-based accept/dispatch policies shift the control point upstream and produce steady service gains under realistic loads (see, for instance, Zhou et al., 2023; Chen et al., 2022; Jahanshahi et al., 2022). Heterogeneous fleets benefit when the action space is structured around joint feasibility of coupled assets, with truck–drone policies showing the clearest returns under stochastic travel times, as in Liu et al., 2022.

Coordination at scale is handled either by centralized attention with rolling masks, which enforces joint feasibility while scoring all vehicle–task pairs, or by multi-objective conditioning that trades efficiency against fairness or service guarantees Xiang et al., 2024; Santiyuda et al., 2024. In networked settings, multi-agent reinforcement learning with centralized training and decentralized execution helps mitigate non-stationarity and partial observability, and remains competitive when combined with attention and neighbor-aware critics Bono et al., 2021. Applied to mobility-on-demand and EV logistics, ADP and safe-RL layers show that constraint handling and energy risk can be embedded without full combinatorial solvers at inference Kullman et al., 2022; Basso et al., 2022; Wu et al., 2024.

Three caveats temper these gains. First, reliability depends on mask completeness and reward shaping, and several policies degrade under distribution shift when travel times or arrival processes move away from training conditions (Ojeda Rios et al., 2021). Second, evidence is heterogeneous across datasets and decision budgets, so cross-paper rankings remain fragile even when instance families overlap (Ojeda Rios et al., 2021). Third, comparative studies indicate there is no uniformly best learner or function class, with neural and linear approximators trading off sample efficiency, stability, and search burden in predictable ways (Akkerman et al., 2025).

3.5 Conclusions and future research

This chapter aimed to show how machine learning can complement operations research in dynamic settings by providing predictive components that anticipate demand, travel times, or state transitions, while the OR layer enforces feasibility and converts predictions into admissible decisions.

Pure ML approaches typically ensure feasibility through masking or feasibility-aware decoding, which prunes illegal actions from the policy's action set at inference time. However, masking presupposes that invalid actions can be identified cheaply and locally (Kool et al., 2019; Du et al., 2021; Xiang et al., 2024; Chen et al., 2025; Chen et al., 2022; Gubena et al., 2025), and it struggles when feasibility depends on long-horizon couplings and global side constraints (Ojeda Rios et al., 2021), or state variables that are only partially observed (Lin et al., 2021; Bono et al., 2021). As a result, end-to-end ML policies are brittle on problems with strict, multi-constraint structure, while hybrid ML+OR pipelines remain applicable because feasibility is guaranteed by the optimization component regardless of the learned module's errors.

Across the surveyed studies the problem formulations, data-generating processes, and evaluation protocols vary widely, which impedes one-to-one comparisons. Differences in horizon length, uncertainty modeling, the nature and tightness of constraints, and even seemingly minor choices affect reported performance.

Within ML-only methods, the emphasis is on DRL and almost exclusively on model-free algorithms. Policy gradient and Value based methods dominate because they don't have the extra-complexity of learning both the policy and the value network at the same time. Model-based RL remains underrepresented despite its potential to reason about constraints and to improve sample efficiency via learned dynamics or planning modules.

Regarding architectures, GNNs are well suited to routing and network design because they encode permutation invariance and locality, yet their adoption in the reviewed papers is still limited compared with MLPs. The latter remain the most common backbone for

both policies and value functions, likely due to simplicity, stable training, and historical code availability, even though GNNs offer stronger inductive biases for combinatorial structure. Bridging this gap, for instance by pairing GNN encoders with lightweight decoders or OR layers, appears promising for sample efficiency and generalization across instance sizes.

Multi-objective formulations can be handled within DRL by scalarization of objectives and preference-conditioned policies to approximate Pareto fronts (Santiyuda et al., 2024). Constrained RL with explicit safety layers or penalty methods is also used in practice (Basso et al., 2022; Wu et al., 2024). These techniques allow the learner to trade off cost, service quality, emissions, or risk dynamically, but they require careful reward shaping and explicit monitoring of constraint violations to avoid hidden feasibility erosion (Ojeda Rios et al., 2021).

Future work should prioritize standardized dynamic benchmarks with public simulators, clearly separated training, validation, and test regimes, and mandatory reporting of feasibility metrics alongside objective values. Advances are likely from tighter ML–OR integration such as differentiable optimization layers, primal–dual or safe RL that treats constraints explicitly, and model-based or planning-augmented RL to improve data efficiency. Architecturally, greater use of GNNs and problem-structure-aware DNNs can improve the learning process, the generalization of the method to similar instances and its scalability in the dimension of the problem. Finally, reproducibility practices including fixed seeds, ablations for masking and OR components, compute and data budgets, and cross-instance generalization tests are essential to convert isolated case studies into comparable evidence. A notable example of an attempt in this regard is the work of Berto et al., 2025, where different NCO papers are implemented in the same repository, with a unified framework, and tested on the same computing infrastructure for easy and fair benchmarking.

References

Akkerman, F., M. Mes, and W. van Jaarsveld (2025). "A Comparison of Reinforcement Learning Policies for Dynamic Vehicle Routing Problems with Stochastic Customer Requests". In: *Computers and Industrial Engineering* 200, p. 110747.

Basso, R., B. Kulcsár, I. Sanchez-Diaz, and X. Qu (2022). "Dynamic Stochastic Electric Vehicle Routing with Safe Reinforcement Learning". In: *Transportation Research Part E: Logistics and Transportation Review* 157, p. 102496.

Baty, L., K. Jungel, P. Klein, A. Parmentier, and M. Schiffer (2024). "Combinatorial Optimization-Enriched Machine Learning to Solve the Dynamic Vehicle Routing Problem with Time Windows". In: *Transportation Science* 58.4, pp. 708–725.

Beirigo, B. A., F. Schulte, and R. R. Negenborn (May 2022). "A Learning-Based Optimization Approach for Autonomous Ridesharing Platforms with Service-Level Contracts and On-Demand Hiring of Idle Vehicles". In: *Transportation Science* 56.3, pp. 677–703.

Berto, F., C. Hua, J. Park, L. Luttmann, Y. Ma, F. Bu, J. Wang, H. Ye, M. Kim, S. Choi, N. G. Zepeda, A. Hottung, J. Zhou, J. Bi, Y. Hu, F. Liu, H. Kim, J. Son, H. Kim, D. Angioni, W. Kool, Z. Cao, Q. Zhang, J. Kim, J. Zhang, K. Shin, C. Wu, S. Ahn, G. Song, C. Kwon, K. Tierney, L. Xie, and J. Park (2025). "RL4CO: An Extensive Reinforcement Learning for Combinatorial Optimization Benchmark". In: *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V.2*. KDD '25. Toronto ON, Canada: Association for Computing Machinery, pp. 5278–5289.

Bono, G., J. Dibangoye, O. Simonin, L. Matignon, and F. Pereyron (2021). "Solving Multi-Agent Routing Problems Using Deep Attention Mechanisms". In: *IEEE Transactions on Intelligent Transportation Systems* 22.12, pp. 7804–7813.

Bottou, L. (2010). "Large-Scale Machine Learning with Stochastic Gradient Descent". In: *Proceedings of COMPSTAT'2010*. Ed. by Y. Lechevallier and G. Saporta. Heidelberg: Physica-Verlag HD, pp. 177–186.

Chen, D., C. Imdahl, D. Lai, and T. Van Woensel (2025). "The Dynamic Traveling Salesman Problem with Time-Dependent and Stochastic Travel Times: A Deep Reinforcement Learning Approach". In: *Transportation Research Part C: Emerging Technologies* 172.

Chen, X., M. Ulmer, and B. Thomas (2022). "Deep Q-learning for Same-Day Delivery with Vehicles and Drones". In: *European Journal of Operational Research* 298.3, pp. 939–952.

Chen, X., T. Wang, B. W. Thomas, and M. W. Ulmer (July 2023). "Same-Day Delivery with Fair Customer Service". In: *European Journal of Operational Research* 308.2, pp. 738–751.

Du, Y., S. Fu, C. Lu, Q. Zhou, and C. Li (2021). "Simultaneous Pickup and Delivery Traveling Salesman Problem Considering the Express Lockers Using Attention Route Planning Network". In: *Computational Intelligence and Neuroscience* 2001.1, p. 5590758.

Garn, W. (2021). "Balanced Dynamic Multiple Travelling Salesmen: Algorithms and Continuous Approximations". In: *Computers and Operations Research* 136, p. 105509.

Grondman, I., L. Busoniu, G. A. D. Lopes, and R. Babuska (Nov. 2012). "A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients". In:

IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) 42.6, pp. 1291–1307.

Gubena, M., Y. Zhang, Z. Ma, X. Ma, and Y. Yang (2025). "Double Deep Q-Learning Network for Package Pickup and Delivery Route Prediction". In: *IEEE Transactions on Consumer Electronics*, pp. 1–1.

Jahanshahi, H., A. Bozanta, M. Cevik, E. M. Kavuk, A. Tosun, S. B. Sonuc, B. Kosucu, and A. Başar (May 2022). "A Deep Reinforcement Learning Approach for the Meal Delivery Problem". In: *Knowledge-Based Systems* 243, p. 108489.

Kingma, D. P. and J. Ba (2014). "Adam: A Method for Stochastic Optimization". In: *arXiv pre-print arXiv:1412.6980*.

Kool, W., L. Bliek, D. Numeroso, Y. Zhang, T. Catshoek, K. Tierney, T. Vidal, and J. Gromicho (2023). "The EURO Meets NeurIPS 2022 Vehicle Routing Competition". In: *Proceedings of the NeurIPS 2022 Competition Track*. Vol. 220. Proceedings of Machine Learning Research. PMLR, pp. 35–49.

Kool, W., H. Van Hoof, and M. Welling (2019). "Attention, Learn to Solve Routing Problems!" In: *International Conference on Learning Representations*.

Kullman, N., M. Cousineau, J. Goodson, and J. Mendoza (2022). "Dynamic Ride-Hailing with Electric Vehicles". In: *Transportation Science* 56.3, pp. 775–794.

Kwon, Y.-D., J. Choo, D. Bae, J. Kim, and A. Hottung (2022). *Cost Shaping via Reinforcement Learning for Vehicle Routing Problems*. EURO Meets NeurIPS 2022 Vehicle Routing Competition (dynamic VRPTW) technical report, Team SB. URL: `https://github.com/ortec/euro-neurips-vrp-2022-quickstart/raw/main/papers/Team_SB.pdf` (visited on 08/31/2025).

Lin, K., C. Li, Y. Li, C. Savaglio, and G. Fortino (2021). "Distributed Learning for Vehicle Routing Decision in Software Defined Internet of Vehicles". In: *IEEE Transactions on Intelligent Transportation Systems* 22.6, pp. 3730–3741.

Liu, Z., X. Li, and A. Khojandi (2022). "The Flying Sidekick Traveling Salesman Problem with Stochastic Travel Time: A Reinforcement Learning Approach". In: *Transportation Research Part E: Logistics and Transportation Review* 164, p. 102816.

Mardešić, N., T. Erdelić, T. Carić, and M. Đurasević (2024). "Review of Stochastic Dynamic Vehicle Routing in the Evolving Urban Logistics Environment". In: *Mathematics* 12.1, 28.

Mnih, V., K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller (2013). "Playing Atari with Deep Reinforcement Learning". In: *arXiv pre-print arXiv:1312.5602*.

Neria, G. and M. Tzur (2024). "The Dynamic Pickup and Allocation with Fairness Problem". In: *Transportation Science* 58.4, pp. 821–840.

Ojeda Rios, B. H., E. C. Xavier, F. K. Miyazawa, P. Amorim, E. Curcio, and M. J. Santos (Oct. 2021). "Recent Dynamic Vehicle Routing Problems: A Survey". In: *Computers & Industrial Engineering* 160, p. 107604.

Pan, W. and S. Q. Liu (Jan. 2023). "Deep reinforcement learning for the dynamic and uncertain vehicle routing problem". In: *Applied Intelligence* 53.1, pp. 405–422.

Petris, M., C. Archetti, D. Cattaruzza, M. Ogier, and F. Semet (Mar. 2025). "A Tutorial on Branch-Price-and-Cut Algorithms". In: *4OR* 23.1, pp. 1–52.

Psaraftis, H. N., M. Wen, and G. Kontoravdis (2016). "Dynamic Vehicle Routing Problems: Three Decades and Counting". In: *Networks* 67.1, pp. 3–31.

Sabar, N. R., A. Bhaskar, E. Chung, A. Turky, and A. Song (Feb. 2019). "A Self-Adaptive Evolutionary Algorithm for Dynamic Vehicle Routing Problems with Traffic Congestion". In: *Swarm and Evolutionary Computation* 44, pp. 1018–1027.

Santiyuda, G., R. Wardoyo, R. Pulungan, and V. Yu (2024). "Multi-Objective Reinforcement Learning for Bi-Objective Time-Dependent Pickup and Delivery Problem with Late Penalties". In: *Engineering Applications of Artificial Intelligence* 128, p. 107381.

Silva, M., J. P. Pedroso, and A. Viana (2023). "Deep Reinforcement Learning for Stochastic Last-Mile Delivery with Crowdshipping". In: *EURO Journal on Transportation and Logistics* 12, p. 100105.

Sippel, L., M. Forbes, and J. Menesch (2025). "Algorithms for Pickup and Delivery Problems with Hours of Service Constraints". In: *Computers and Operations Research* 183, p. 107123.

Sun, Z., R. Tian, J. Wu, X. Lu, and J. Wang (2025). "A Fast Solution Method for the Dynamic Flexible Pickup and Delivery Problem with Task Allocation Fairness for Multiple Vehicles". In: *Neurocomputing* 639.C.

Sutton, R. S., D. McAllester, S. Singh, and Y. Mansour (1999). "Policy gradient methods for reinforcement learning with function approximation". In: *Proceedings of the 13th International Conference on Neural Information Processing Systems*. NIPS'99. Denver, CO: MIT Press, pp. 1057–1063.

Van Hasselt, H., A. Guez, and D. Silver (2015). "Deep Reinforcement Learning with Double Q-learning". In: *arXiv pre-print arXiv:1509.06461*.

Wang, Z., T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas (2015). "Dueling Network Architectures for Deep Reinforcement Learning". In: *arXiv pre-print arXiv:1511.06581*.

Watkins, C. J. C. H. and P. Dayan (May 1992). "Q-Learning". In: *Machine Learning* 8.3–4, pp. 279–292.

Wu, B., X. Zuo, G. Chen, G. Ai, and X. Wan (2024). "Multi-Agent Deep Reinforcement Learning Based Real-Time Planning Approach for Responsive Customized Bus Routes". In: *Computers and Industrial Engineering* 188.

Xiang, C., Z. Wu, J. Tu, and J. Huang (2024). "Centralized Deep Reinforcement Learning Method for Dynamic Multi-Vehicle Pickup and Delivery Problem with Crowdshippers". In: *IEEE Transactions on Intelligent Transportation Systems* 25.8, pp. 9253–9267.

Xu, X. and Z. Wei (2023). "Dynamic Pickup and Delivery Problem with Transshipments and LIFO Constraints". In: *Computers and Industrial Engineering* 175, p. 108835.

Zhang, Z., H. Liu, M. Zhou, and J. Wang (2023). "Solving Dynamic Traveling Salesman Problems With Deep Reinforcement Learning". In: *IEEE Transactions on Neural Networks and Learning Systems* 34.4, pp. 2119–2132.

Zhou, C., J. Ma, L. Douge, E. Chew, and L. Lee (2023). "Reinforcement Learning-based Approach for Dynamic Vehicle Routing Problem with Stochastic Demand". In: *Computers and Industrial Engineering* 182.C.

Appendix A
Scopus Search Strategy and Data Handling

In this appendix, we share our search strategy according to the PRISMA-S convention, so to ensure our results are reproducible.

Database name

Database searched: **Scopus** (Elsevier). URL: `https://www.scopus.com`.

Multi-database searching

Databases searched simultaneously on a single platform: *None.*

Study registries

Study/clinical trial registries searched: *None.*

Online resources and browsing

Other online or print sources purposefully searched/browsed): *None.*

Citation searching

Cited/citing reference methods used (e.g., Scopus citation index, reference list checks): *We screened the reference list of Mardešić et al., 2024 on 20 July 2025. We extracted 8 cited records that met our journal eligibility rule and imported them manually on Zotero.*

Contacts

Authors/experts/manufacturers contacted to locate additional studies or data: *None.*

Other methods

Additional methods or sources: *None.*

Full search strategies (copied exactly as run)

```
KEY ( ( "dynamic vehicle rout*" OR "DVRP*" OR "dynamic traveling salesman*"
OR "dynamic TSP*" OR "real-time rout*" OR "real-time vehicle rout*" OR
"real-time traveling salesman*" OR "real-time TSP*" OR "dynamic pickup*" OR
"dynamic deliver*" OR "pickup and deliver*" OR "simultaneous deliver* and
pickup*" OR "on-demand rout*" OR "real-time optim*" OR "online vehicle
rout*" ) AND ( "machine learning" OR "reinforcement learning" OR "deep
reinforcement learning" OR "GNN" OR "graph neural network*" ) ) AND PUBYEAR
> 2020
```

Limits and restrictions

- Date range: *2021 to 24 July 2025 (last query re-run).*
- Document types included/excluded: *We included only papers published in journals classified Q1 by Scimago, this filtering was done manually.*
- Language restrictions: *English only.*
- Subject areas/source types/OA filters/country or affiliation limits (if any): *None.*
- Justification for any limits used: *To the best of our knowledge, the last survey on the topic published in a Scimago Q1 journal dated back to 2021 (Ojeda Rios et al., 2021); hence, the time range.*

Search filters

Validated/published search filters used: *None.*

Prior work

Search strategies adapted or reused from prior reviews (with citations): *None.*

Updates

Method(s) for updating the search: *Manually re-run one week later.*

Dates of searches

Initial search run: **16 July 2025, 21:25** (Time zone: Europe/Rome).
Re-run. *24 July 2025, 11:50* (Time zone: Europe/Rome).

Peer review of the search

Search peer review conducted: *No.*

Total records

Total records identified from Scopus at time of search: **184**.

De-duplication

Deduplication software/process: *Since papers taken from Mardešić et al., 2024 were input manually in Zotero, there was no need to de-duplicate.*

Chapter 4

Context-Aware Traveling Salesman Problem: definition and solution approach

Introduction

Imagine you are a driver who has to deliver packages to a set of customers in a known city. You start from the office and must visit all the customers during your day of work. Once all the deliveries are finished, you must return to the office and your shift ends. For this reason, you want to finish the delivery as soon as possible, minimizing the total travel time (and the working hours). You are given a map service to check the traffic conditions on the roads. Based on the traffic reported by the map service and on the context in which you are operating (e.g., time of day, day of the week and weather), you decide the first customer to visit. Once you have decided, you inform the customer of your imminent arrival and start driving towards it. Once you arrived and delivered the package, you check the map service again to see how the traffic conditions have changed, you reconsider also the context in which you are operating and decide the next customer to visit. You repeat this process until you have visited all the customers and returned to the office.

Solving this problem optimally is not trivial. Indeed, the problem we just described is a variant of the Traveling Salesman Problem (TSP) (Dantzig et al., 1954), a well-known combinatorial optimization problem that is NP-hard, meaning that no polynomial-time algorithm is known to exist for solving it in the general case. The variant we just described is called Dynamic Traveling Salesman Problem (DTSP) (Bertsimas et al., 1991), and it differs from the classical TSP in that the information about the problem are not all available, but they are revealed as the solution unfolds. While the problem has a very simple description, it is even more challenging to solve than the classical TSP, due to uncertainty about the future. Nonetheless, it is a problem of great practical importance, as it models many real-world scenarios, such as delivery services, ride-sharing, and logistics. While this problem might have seemed academic in the past, nowadays, thanks to the availability of real-time data and advances in computational capabilities, it has become increasingly relevant.

While the DTSP has been already studied in the literature, different papers made various assumptions and simplifications about the "dynamic" aspects of the problem. The version we present is unique by being as close as possible to the real world scenario described above. In the DTSP we present, the goal is minimizing the total travel time of a tour that visits all the customers and returns to the starting point. The cost of traversing

an edge varies continuously over time, and its actual value is available only after the edge is traversed. Changes in the cost depend both on external factors, in our case the time of the day, the day of the week and the weather, and on internal factors, in our case the distribution of traffic in the network, possible car crashes and the unpredictable behavior of the other drivers. This set of changing features defines the *context* in which the salesman must solve the problem. Following the notation introduced in Chapter 3, the problem we want to solve can be defined as an Uncertain Dynamic Traveling Salesman Problem (UDTSP), that we call *Context-Aware Traveling Salesman Problem* (CA-TSP).

This formulation is motivated by real-world considerations. Travel times between customers are never fixed, as many happenings could cause disturbances. Additionally, nowadays, thanks to map services and the availability of real-time traffic data, the current state of traffic is available to the driver, but its evolution is not known in advance and is usually hard to predict. In our formulation, customers need to be informed of the imminent arrival, so that they can prepare for the delivery and for this reason the driver cannot change the customer to visit once it has been chosen. While this might seem a limitation, it is a realistic assumption for many real-world scenarios—e.g., technicians that have to visit customers to repair devices. We point out that this assumption can be relaxed with minor changes of the environment, but we decided to focus on the case of no rerouting for simplicity. Relaxing this constraint would only require redefining the time step—for example, by incrementing it at fixed intervals (e.g., every five minutes) or upon specific events (e.g., when the agent reaches a junction in the road network). We assume also that the number of customers is known a priori, even if our approach should work also in the case of dynamic customers. The map in which the nodes are located is a road network, and is fixed for all the instances. Customers can only be located along the edges of this road network.

To solve this problem, we propose a solution approach based on Neural Combinatorial Optimization (NCO) (Bello et al., 2017). With NCO, we aim to learn a policy that adapts to the dynamic nature of the problem and is able to make decisions in real-time, not only considering the current state of the environment but also anticipating future changes—without prior assumptions, such as the probability distribution of the edge costs. A major contribution of our approach is the usage, in the environment, of a traffic simulation software to train and test the agent using a realistic scenario. Indeed, the ability to apply NCO in simulated environments is increasingly valuable. For example, the rise of digital twins——software replicas of industrial processes designed to analyze, predict, and simulate "what-if" scenarios——is transforming industries. Digital twins can serve as the environment in which agents interact, enabling optimization of real-world industrial processes. Moreover, the ability to use realistic data mitigates one of the problems of Reinforcement Learning (RL) solutions, called *Sim2Real*, where the behavior of trained agent is usually not maintained when deployed in real world scenario due to the differences between the training data and the real world data.

The chapter is structured as follows. Section 4.1 introduces the TSP. In Section 4.2 we examine how the class of DTSP has been defined in the literature and we propose a new model for the DTSP. We introduce the problem notation and the MDP formulation for this problem in Section 4.3. Finally, Section 4.4 describes our solution approach to the DTSP.

4.1 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is a classical problem in combinatorial optimization. Informally, it can be stated as follows: given a set of customers and the pairwise distances between them, the objective of the TSP is to determine the shortest possible route that visits each customer exactly once, and returns to the starting point.

Formally, the TSP can be framed as a problem on a graph. Let $G = (V, A)$ be a graph where $V = \{1, 2, \ldots, n\}$ denotes the set of nodes (i.e., the customers) and $A = \{a_{ij} = (i, j)\}_{i,j \in V \times V}$ is the set of directed edges (i.e., the routes between customers). Each edge a_{ij} is associated with a weight d_{ij}, representing the distance, or more generally the cost, of traversing it to go form node i to node j. The goal is to find a Hamiltonian circuit (i.e., a closed tour visiting each node exactly once) that minimizes the total cost of the traveled route.

We can define the TSP as a Mixed-Integer Linear Program (MILP). Using binary decision variables $x_{ij} \in \{0, 1\}$, where $x_{ij} = 1$ if the tour includes the edge (i, j) from node i to node j, and 0 otherwise, the problem can be written using the Dantzig-Fulkerson-Johnson formulation (Dantzig et al., 1954):

$$\min \quad \sum_{i=1}^{n} \sum_{\substack{j=1 \\ j \neq i}}^{n} d_{ij} x_{ij} \tag{4.1}$$

$$\text{s.t.} \quad \sum_{\substack{i=1 \\ i \neq j}}^{n} x_{ij} = 1 \qquad \forall j = 1, \ldots, n \tag{4.2}$$

$$\sum_{\substack{j=1 \\ j \neq i}}^{n} x_{ij} = 1 \qquad \forall i = 1, \ldots, n \tag{4.3}$$

$$\sum_{i \in Q} \sum_{\substack{j \in Q \\ j \neq i}} x_{ij} \leq |Q| - 1 \qquad \forall Q \subset V, |Q| \geq 2 \tag{4.4}$$

$$x_{ij} \in \{0, 1\} \qquad \forall (i, j) \in A \tag{4.5}$$

The constraints 4.2 and 4.3 ensure that each node is entered and exited exactly once, respectively. Constraints 4.4, usually called subtour elimination constraints, ensures that the tour is a Hamiltonian circuit, i.e., it visits each node exactly once and returns to the starting node. The constraints 4.5 ensure that the variables x_{ij} are all binary.

The distance matrix $D = [d_{ij}]_{i,j \in A}$ captures the distances between all pairs of nodes. If D is symmetric, i.e., $d_{ij} = d_{ji}$ for all $i, j \in A$, the TSP is said to be symmetric. Otherwise, the problem is referred to as the Asymmetric TSP (ATSP).

Symmetric TSP often uses Euclidean or Manhattan distances. In contrast, real-world TSP is frequently asymmetric, due to one-way streets, traffic conditions, or cost differences that depend on the direction of the edge. Indeed, the cost may represent not only spatial separation but also travel time, fuel consumption, or other application-specific objectives.

The TSP is known to be a NP-hard problem, meaning no known polynomial-time algorithm can solve it optimally. Consequently, a variety of heuristics have been developed, as exact algorithms are often impractical for large instances. One of the most

effective algorithms for the symmetric TSP is the Lin-Kernighan heuristic with Helsgaun implementation (LKH), introduced by Helsgaun, 2000, which is a local search algorithm that iteratively improves the solution. The ATSP, being a generalization of the TSP (the TSP is a special case of the ATSP in which $d_{ij} = d_{ji}$), is also NP-hard. Heuristics were, hence, developed for this case, too—for instance, see Rego et al., 2011 for a variation of the LKH algorithm that tackles the ATSP.

Due to its fundamental nature, the TSP and its variations appears in numerous real-world applications (Applegate et al., 2006), ranging from circuit design to DNA sequencing. Notably, it plays a critical role in logistics, as it is the basis of routing problems.

4.2 Dynamic TSP

The TSP, in its classical formulation, is a static deterministic problem: it assumes that the nodes in the graph and the cost between every pair of nodes are known in advance, as discussed in Chapter 3. However, this assumption rarely holds in real-world scenarios, where the set of nodes and the cost between them may vary dynamically during the tour.

For example, consider a TSP instance where nodes represent customers to visit and the cost of traveling between two customers corresponds to the travel time. Travel times may fluctuate due to changing traffic conditions, causing the cost of traveling from one customer to another to vary dynamically. Additionally, the set of customers to be visited may also change during the tour: some customers might be removed, or new ones may be added to the itinerary, as it was first analyzed by (Bertsimas et al., 1991).

When either the cost or the set of nodes changes during execution, and the information becomes available only progressively, the problem is referred to as the *Dynamic Traveling Salesman Problem (DTSP).* Notice how this definition exclude a TSP variant that might seem very similar to the DTSP, i.e., the Time Dependent TSP (TDTSP). The TDTSP assumes that the cost between two nodes varies over time, but the variation is known in advance—see Adamo et al., 2024 for a recent review on this variation of the TSP. Hence, according to the taxonomy proposed by Psaraftis et al., 2016, the TDTSP should be considered as a *Static and Deterministic (SD)* problem. By labeling our formulation as uncertain, we wish to underline the fundamental difference in the model assumption—that is, the lack of any knowledge about how external conditions evolve over time; for a more in-depth discussion of this topic, see Section 3.2 in Chapter 3.

While the DTSP more closely reflects real-world applications, addressing it generally involves predicting how the instance would evolve and, possibly, continuously updating the solution as new data becomes available. This poses two requirements for solution methods: to be able to predict how the problem will evolve and to update the solution in a short amount of time.

The DTSP is not a uniquely defined problem, but rather a class of problems whose formulation depends heavily on the nature and timing of information availability. Specifically, variations of the DTSP differ based on what is known about the problem in advance, and how and when new information is revealed during the course of the tour and on the level of certainty of new information.

The importance of the DTSP is widely recognized and is studied since the late 1980s. But, maybe due to its complexity, it has not been as extensively studied as the static TSP. One of the first works in this area is the survey about Dynamic Vehicle Routing Problems by (Psaraftis, 1988), which presents the DTSP and some of its variants, and

discusses the challenges in solving it. Another early work that tackles the DTSP is (Bertsimas et al., 1991), who study a dynamic variant of the TSP, called Dynamic Traveling Repairman Problem, in which service requests (i.e., nodes) arrive randomly over time and the objective is to minimize total waiting time. The authors derive theoretical lower bounds for system performance and propose several policies to tackle the problem.

As discussed in Chapter 3, exact methods for solving the DTSP are often computationally not applicable for large instances, due to the problem complexity and the need to adapt to changing conditions. For this reason, most research has focused on heuristic and metaheuristic approaches capable of adapting to new information efficiently. A notable early contribution using swarm intelligence is found in (Boers et al., 2001), where the authors propose a modified Ant Colony System to handle instances in which a single node can be dynamically inserted or removed to the set of nodes to visit. This insertion or removal happen at a fixed iteration of the algorithm. They design three different pheromone update strategies to account for the time-varying nature of the problem.

One of the first approaches addressing DTSP with time-dependent distances is proposed by (Zhou et al., 2003), who developed an evolutionary algorithm tailored to adapt to dynamic cost changes. In their formulation, the location of nodes changes every 0.5 to 2 seconds, and the new position is changed according to a Gaussian distribution. The solution is built at the starting time, and it is updated at discrete time intervals.

Another approach dealing with time-varying distances is presented by Cheong et al., 2012. The authors assume that 90% of the travel times in the edges are known in advance, while the remaining 10% are revealed only upon traversal. This subset of edges can be in one of three states: non congested, slightly congested, or heavily congested. Based on the state of the edge, a different probability distribution describes the travel time that the edge will have once traversed. Moreover, the probability of congestion is known in advance, and computed using historical data. The authors model this problem as a Markov Decision Process (MDP) and the state of the problem (i.e., which edges are currently congested) is updated at each time step. It follows that, this problem can be considered a dynamic problem, as information is revealed progressively, and the solution approach adapts to the changing conditions. Additionally, the problem is also stochastic, as a probability distribution over unknown information is given in advance. To solve it, the authors use an extension of the A* algorithm, the AO* algorithm, that iteratively refines the solution according to the revealed information. Hence, they compute the solution as new information is revealed, using a dynamic approach.

A different version of the DTSP is presented by (Toriello et al., 2014). They also deal with the problem of changing travel times, and the travel time between two nodes is known only probabilistically. In contrast with the previous example, they assume that, once a node is reached, the value of all its outgoing edges is known with certainty. This problem is also dynamic in its nature, and has both stochastic and deterministic elements, as the value of the outgoing edges is known with certainty, while the remaining edges are only known probabilistically. To tackle this problem, they formulated it as a dynamic program that seeks to minimize expected tour cost, and developed an approximate linear programming (ALP) approach to circumvent the typical curse of dimensionality associated with dynamic programming. This ALP leads to a price-directed policy and yields a provable lower bound on performance. The approach used to solve the problem is a dynamic approach, as it updates the solution at each time step based on newly revealed arcs cost.

In the DTSP formulation of Groba et al., 2015, the initial position of the nodes is

known in advance but changes while time passes. Their future position is predicted using Newton motion equation, but it is not known with certainty. They propose a genetic algorithm to solve the DTSP with unknown future positions.

Strak et al., 2019 propose a DTSP where distances change based on a Gaussian distribution and the actual distance is revealed only when the edge is traversed. Moreover, the set of nodes might change over time. They design an Ant Colony System (ACS) that can adapt to the dynamic changes without restarting the search process. Instead of reinitializing pheromone trails when changes occur, the algorithm incrementally updates pheromone information and solution candidates based on the new problem state. This allows ants to reuse prior knowledge while reacting to newly revealed distances or node changes.

More recently, the field of NCO has started tackling the DTSP. In Zhang et al., 2023, the authors propose a neural policy that selects the next node to visit at each time step, based on the current state of the problem. They introduce two architectures inspired by the Attention Model (AM) (Kool et al., 2019), and test them on DTSP and Dynamic Pickup and Delivery Problem (DPDP) instances. To model realistic time-dependent travel costs, the authors collect real-world data for 100 locations in Beijing, recording travel times between pairs of nodes every two hours. These values are interpolated using cubic splines and perturbed with Gaussian noise to simulate uncertainty. Problem instances are then generated by sampling subsets of the locations.

Another paper in the field of NCO that addresses the DTSP is the work of Chen et al., 2025. They call the problem they tackle Stochastic Time Dependant DTSP (DTSP-TDS). In their scenario, the travel times between nodes are time-dependent and stochastic, and vary according to the departure time. They are modeled using a Gamma distribution with known expectation and standard deviation. At each decision step, the actual travel times for the current state are realized, while those for future moves remain uncertain. Moreover, the planning horizon is discretized into equal-length time intervals with the assumption that travel times remain constant within each interval. To solve this problem they propose an NCO solution, with a new architecture inspired by the AM, called Dynamic Graph Temporal Attention (DGTA). This DNN takes as input the position of nodes, the expected travel times at each time interval and information about the partial solution constructed so far, and outputs the probability for each node to visit. They test their approach on synthetic data generated using an approach similar to Cordeau et al., 2014.

These examples illustrate that the DTSP encompasses a spectrum of problem variants, and that its definition can vary significantly based on the specific assumptions and requirements of the application at hand. As such, a general solution framework must be tailored to the specific variant under consideration.

The following section describes our formulation of the DTSP, namely the CA-TSP.

4.3 Problem definition

Since the CA-TSP is a dynamic problem, we need to introduce a time variable, that is represented by t. The time t represents the current second of the day, and is an integer variable that goes from 0 to 86400 (i.e., the seconds in 24 hours). To describe the CA-TSP we use two graphs. The first graph is the road network, which is a weighted directed graph $G^r(t)$. In this graph, each node is a junction and each edge is a road

between two junctions. We will call this graph the *road network graph.* Each node in the road network graph is represented by its xy-coordinates on the plane. To describe the continuously changing traffic conditions, at each time t, each edge e^r_{ij} has associated the current estimated travel time of the edge $l_{ij}(t)$, the number of vehicles $n_{ij}(t)$ on the edge and their average speed $v_{ij}(t)$. The second graph $G^c(t)$ is the *customers graph*, which is a weighted directed graph where each node is a customer or the starting point for the salesman (i.e., the depot). Each node is represented by its xy-coordinates on the plane, and each edge e^c_{ij} between two nodes represents the full path between the two nodes. Each of these edges has associated a weight which is the current travel time $c_{ij}(t)$ between the two nodes—i.e., the time it would take to go from the first node to the second if the traffic conditions would not change through time.

To complete the problem description, we also need to consider external factors that influence the traffic conditions in the road network. The rain is represented by a binary variable w, which is 1 if it is raining and 0 otherwise. Finally, the distinction between workdays and weekends is represented by a binary variable d, which is 1 if it is a workday and 0 otherwise.

The problem is to find a tour that starts from the depot and visits all the customers before returning to the depot, minimizing the total travel time of the tour.

We can define this problem as an MDP. Since MDPs are defined in discrete time, we need to define how the time steps of the MDP relate to the actual time t. In our formulation, a time step happens every time the salesman arrives at a customer location. This introduces two different time dimensions: the actual time (measured in seconds and denoted by letter t) and the time step. The duration of a time step is not fixed, and it depends on the travel time between the current node and the next node. Since the maximum number of time steps we consider in our environment is N, as a time step advances every time we visit a new customer, we introduce $\tau = 1, 2, ..., N$ to denote the time steps of the problem. We call $t(\tau)$ the actual time at time step τ.

Let S be the set of possible states of the problem. Each state $s \in S$ is described by two graphs and a vector. The first graph is the road network graph $G^r(t(\tau))$, which describes the current state of the road network at time $t(\tau)$. The second graph is the customers graph $G^c(t(\tau))$, which describes the current state of the customers graph at time $t(\tau)$. The vector contains the external factors, i.e., the current time of the day $t(\tau)$, the weather w and the day of the week d. Moreover, the state contains the current position of the salesman, represented using the label of the current node, and the set of customers that have already been visited.

The set of possible actions A is a set that contains the possible nodes to visit. Once a node has been chosen, the agent must traverse the edge leading to that node and there is no possibility of rerouting.

The reward $r(\tau)$ at each time step τ is the negative of the travel time needed to reach the current node from the previous node, i.e., $r(\tau) = -(t(\tau) - t(\tau - 1))$ (we recall here that RL is framed as a maximization problem, while the TSP as a minimization problem).

Finally, to describe the MDP, we need to define the state transition probability function $p(s'|s, a)$, which is the probability of reaching state s' from state s after taking action a. This function depends on how the traffic evolves through time and on the behavior of the other drivers, but we decide not to make any assumption on it, but that it depends on the current traffic level, the weather and the day type. Apart from that, we consider it unknown.

Notice that, in the CA-TSP formulation, we made some assumptions. We assumed

that the set of nodes is known in advance and does not change during the tour. However, the cost of traveling between two nodes changes over time, and its actual value is revealed only when the edge is traversed. We assumed that the nodes are located in a road network, and can only be located along the edges of this road network. The changes in the cost depend on both external and internal factors. There is no probability distribution over the unknown information, and no assumption is made on how the cost will evolve over time. While we made these simplifying assumptions to define our problem, the majority of them are not restrictive: they can be relaxed or modified, with minimal changes to the solution approach we present in Section 4.4.

4.4 Solution approach

In this work, we aim to show that NCO provides a flexible and robust framework for solving the DTSP. A key strength of NCO lies in its ability to learn directly from data rather than relying on problem-specific assumptions. This makes it particularly well-suited to dynamic environments, where traffic conditions or customer demands may change over time. By training directly on problem instances, the approach learns patterns of the problem and relationships between variables. Moreover, when new factors such as additional traffic influences are introduced (e.g., holidays or special events like football matches), the solution does not need to be redesigned or its assumptions re-validated. It is enough to extend the input to include these factors and retrain the agent on the updated environment. In this sense, NCO shows its potential as a promising direction for solving complex, real-world routing problems.

To make the environment more realistic, we develop an environment that uses a traffic simulator to model the evolution of travel times in a road network. This integration of a simulator within the environment is a key aspect of our work, distinguishing it from previous approaches of NCO. While this integration enables more realistic training scenarios, it also introduces challenges for the applicability of RL techniques. Indeed, synthetic data can be generated rapidly, making large datasets readily available for training machine learning agents. However, when data is generated using a simulator, the agent must still learn effectively with limited samples, as simulation time often remains a significant constraint.

The agent interacts with the environment by choosing the next node to visit at each time step, based on the current state of the problem. During training, the agent is presented with different instances of the problem, each one with different nodes positions and context. The agent learns a policy that minimizes the total travel time of the tour by adapting to the changing conditions.

4.4.1 Agent

The core objective of our work is to develop an agent capable of learning a policy that adapts to the dynamic nature of the DTSP and makes real-time decisions that minimize the total travel time of the tour.

A distinctive feature of our agent is its ability to process and reason over both the road network graph and the customers graph. This is a significant departure from previous works in NCO, where typically only the customer graph is considered, and the underlying road network is ignored or abstracted away. While a related idea is explored by Zhang

et al., 2023, their approach integrates traffic information directly into the customer graph rather than reasoning over two separate graphs.

By explicitly modeling the road network and its dynamic state, our agent can make more informed decisions that account for real-world constraints such as traffic congestion, road connectivity, and varying travel times.

Another important aspect is the dynamic nature of the environment. Traffic conditions on the roads change over time and are influenced by the external factors. Our agent must therefore learn not only to react to the current state but also to anticipate how the environment will evolve in the future. This requires the agent to develop a form of temporal reasoning, planning actions that may not be immediately optimal but lead to better outcomes as conditions change.

To illustrate the importance of this anticipatory behavior, consider the following scenario. Suppose a salesman must visit three customers, labeled A, B, and C, and is currently located at A. The initial travel times are as follows: $A \to B$ is 10min, $A \to C$ is 20min, $B \to A$ is 20min, $B \to C$ is 10min, $C \to B$ is 20min, and $C \to A$ is 20min. At time step $\tau = 0$, it is 7:00am and the seemingly optimal choice is to visit B first, since $A \to B$ is the shortest immediate route. However, suppose that B is located in the city center and, after visiting B, a traffic jam occurs on the road from B to C as there are schools in that road. The time from B to C increases from 10min to 40min, while all other travel times remain unchanged. The total travel time for the route $A \to B \to C \to A$ becomes $10min + 40min + 20min = 70min$. Alternatively, if the salesman knows that the edge from B to C is likely to be congested in the near future, he might choose to visit C first. In this case, the travel times would be $A \to C = 20min$, $C \to B = 20min$, $B \to A = 20min$, for a total of $20min + 20min + 20min = 60min$. This example shows that the action which appears optimal based solely on the current traffic information may not be the best choice when future changes in the environment are considered. In this case, visiting C first would have resulted in a lower total travel time, highlighting the necessity for the agent to reason about the evolution of traffic conditions and to plan accordingly.

Our agent is composed of two main elements. The first one is the "brain" of the agent, composed of Deep Neural Networks (DNNs) that are trained to predict the best action for a given state. We will describe the architecture of these DNNs in detail in Section 2.3.1.

The second element determines how the agent learns, that is the training algorithm, which in our case is Proximal Policy Optimization (PPO) (Schulman et al., 2017), a state-of-the-art RL method. The choice of PPO is motivated by its stability in dynamic environments, which is crucial for our problem setting. Moreover, unlike other approaches in the literature that employ REINFORCE with a baseline (see Chen et al., 2025; Zhang et al., 2023), PPO does not require solving the same instance with an additional baseline algorithm for comparison. This is particularly important because evaluating a baseline would require restarting the environment from the same state as the agent and running it until the end of the episode. Consequently, at every time step the environment would need to be simulated twice, once for the agent and once for the baseline, substantially increasing the computational cost of training and rendering the approach impractical.

In the rest of this section, we introduce some theoretical background needed to better understand our agent implementation, that is presented in Chapter 5. We start this section by presenting the DNN involved in the agent creation. Then, we proceed presenting the details of the PPO algorithm.

DNNs

The agent is composed mainly of two kind of DNNs, which are used to reason over graph data.

The first core DNN component we describe is the Multi-Head Attention (MHA) mechanism (Vaswani et al., 2017). As with the standard attention mechanism introduced in Chapter 2, MHA assigns weights that indicate how strongly each node should take other nodes into account. The difference is that, instead of computing a single set of weights, it computes several sets in parallel, through several independent *attention heads.* Each head focuses on different kinds of relationships within the data——for instance, one head might focus on local interactions while another captures longer-range dependencies. By combining these different perspectives, the mechanism provides a richer description of the relationships in the graph. The results of all the heads are then concatenated and passed through a final linear transformation creating a combined representation that jointly encodes diverse types of relational information.

The second DNN we explain is a GNN called GATv2 (see (Brody et al., 2021)), which is a variant of the Graph Attention Network (GAT) introduced by (Veličković et al., 2017).

GATv2 is a message passing GNN. GNNs were already described in Chapter 2, but we provide a more in depth analysis of GATv2. The key innovation of GATv2 is its ability to adaptively compute attention coefficients for each node in the graph, allowing it to focus on the most relevant neighbors during message passing. To do so, it uses a learnable attention mechanism that incorporates both node and edge features.

To be more rigorous, given a node with features $\mathbf{x}_i$ (these features might be already an embedding from a previous layer) and edges features $\mathbf{e}_{ij}$ for each neighbor j, a GATv2 layer computes the updated node features $\mathbf{x}'_i$ as follows:

$$\mathbf{x}'_i = \sum_{j \in V} \alpha_{i,j} \mathbf{W_1} x_j$$

where $\mathbf{W_1}$ is a learnable matrix of parameters and $\alpha_{i,j}$ are defined as follows:

$$\text{softmax}\left(\mathbf{a}^T \text{LeakyReLu}\left(\mathbf{W_2 x_i} + \mathbf{W_1 x_j} + \mathbf{W_3 e_{ij}}\right)\right)$$

where $\mathbf{W_2}$ and $\mathbf{W_3}$ are matrices of learnable parameters, $\mathbf{a}$ is a vector of learnable parameters, and we recall that $\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$. LeakyReLu, instead, is an activation function defined as

$$\text{LeakyReLu}(x) = \begin{cases} x & \text{if } x > 0 \\ \kappa x & \text{if } x \leq 0 \end{cases}$$

where κ is a small positive constant (e.g., 0.01) that allows a small, non-zero gradient when the unit is not active.

Additionally, we describe a new kind of layer, not discussed in Chapter 2. *LayerNorm* is a layer that normalizes the output of a layer across the feature dimension for each input sample. To clarify, given an input vector $\mathbf{x} = [x_1, x_2, \ldots, x_N]$, its mean and variance are computed as:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i, \quad \sigma^2 = \frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2$$

The output of LayerNorm is then:

$$y_i = \gamma \cdot \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

where γ and β are the learnable parameters of this layer, usually called gain and bias, respectively. This layer is usually performed before an activation function, to improve training convergence and model robustness.

Finally, we detail some additional strategies that we use to make the training more robust.

The first one is called *skip connection*. Introduced by He et al., 2016, it consist on summing the input of a layer to is output or to the output of a subsequent layer, when input and output share the same dimensions. This mitigates the problem of *vanishing gradients*, which happens when the initial layer of DNNs receive small updates when the DNN is very big.

The second strategy has been developed by Srivastava et al., 2014 and is called *dropout*. During training some DNN neurons are deactivated. This allows the DNN to develop important features extraction in many "parts" of the network and prevents overfitting. The impact of this strategy can be tuned by changing the *dropout rate*, i.e., the fraction of neurons deactivated at each training step.

Training algorithm

The agent is trained using the PPO algorithm, a widely adopted method in RL that belongs to the class of Actor-Critic (AC) algorithms. AC algorithms are a class of training algorithms that combines the usage of a policy network (the actor) and a value network (the critic). These two neural networks are optimized simultaneously: the actor is trained to give higher probability to actions that are more favorable given the current state, and the critic is trained to estimate what the expected return (value) from the current state is.

A famous AC algorithm is called Advantage Actor Critic (A2C). This algorithm highly resembles REINFORCE with a baseline (see Chapter 2). Instead of judging an action comparing its return to the one of the baseline, A2C compares it with the value predicted by the critic. The difference between the return and the value is called the *advantage*, which measures how much better or worse the action performed relative to expectations. The policy is then updated proportionally to this advantage, reinforcing actions that outperformed the critic estimate and discouraging those that did not. Since A2C is prone to unstable training, as the policy might change drastically during updates, potentially forgetting what it had learned—an effect known as *catastrophic forgetting*—PPO improves upon A2C by introducing a mechanism to limit the size of policy updates, thereby, enhancing training stability and sample efficiency. This is achieved by introducing, alongside the objective function that the agent optimizes over, a "trust region" that constraints the policy updates.

To better understand how PPO works, let us introduce the policy network π_θ, where θ are its trainable parameters, and the value network V_ϕ, with trainable parameters denoted by ϕ. Since parameters changes at each training step, then at a given training step, if we want to explicitly refer to the parameters before and after the update, we can use the subscript old for old parameters.

Then, we can introduce the *ratio* between the new and old policies for each action-state pair, defined as

$$\bar{r}_\tau(\theta) = \frac{\pi_\theta(a_\tau|s_\tau)}{\pi_{\theta_{\text{old}}}(a_\tau|s_\tau)}.$$

Intuitively, this ratio indicates how much the probability of taking action a_τ in state s_τ has changed after the policy update. If the ratio is close to 1, it means the policy has not changed much, while a ratio far from 1 indicates a significant change.

Since we want to ensure that the policy update is not too drastic, we can use this ratio to create a constraint on the policy update. To do so, we can introduce a hyperparameter ϵ, which defines the size of the trust region and the maximum allowed change in the policy. This leads to the definition of the *clipped surrogate objective*, the distinctive feature of PPO. Given an action a_τ taken in state s_τ at time τ, the clipped ratio is defined as

$$\bar{r}_\tau^{\text{clip}}(\theta) = \text{clip}(\bar{r}_\tau(\theta), 1-\epsilon, 1+\epsilon).$$

Now, given a trajectory (i.e., state-action pairs collected during one episode), we can compute the returns at each time step τ as

$$\hat{R}_\tau = \sum_{k=\tau}^{N} \gamma^{k-\tau} r_k,$$

where r_k is the reward received at time step k and $\gamma \in [0, 1]$ is the discount factor.

The return is not, by itself, a good estimate of the quality of an action, as an action can have high return just because the agent was already in a good state, but its action might be suboptimal. Moreover, the magnitude of the rewards can vary across different instances, making it difficult to compare the quality of actions based solely on their returns. For this reason, we introduce the advantage function, which estimates how much better or worse an action is compared to the average action in that state. The advantage function is defined as

$$\hat{A}_\tau = \hat{R}_\tau - V_\phi(s_\tau),$$

where $V_\phi(s_\tau)$ is the value function estimate for state s_τ.

Intuitively, if the advantage function is positive, it means the action taken was better than what the value network expected, and the action at that time step is enforced. If the advantage function is negative, it indicates that the action taken at that time step was worse than expected, and the action is discouraged. The amount of enforcement or discouragement is proportional to the magnitude of the advantage function.

Finally, we can define the objective function of PPO (the function we want to maximize) as:

$$J^{\text{CLIP}}(\theta) = \mathbb{E}_\tau \left[\min \left(\bar{r}_\tau(\theta)\hat{A}_\tau,\ \text{clip}(\bar{r}_\tau(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_\tau \right) \right],$$

This objective function tells us that we want to maximize the expected advantage while ensuring that the policy update remains within a trust region defined by ϵ.

This objective ensures that the new policy does not deviate excessively from the old policy, thus preventing large, destabilizing updates.

As Deep Learning problems are usually framed as minimization problems, we define the policy loss as the negative of the objective function:

$$L^{\text{POL}}(\theta) = -J^{\text{CLIP}}(\theta).$$

Simultaneously, since PPO uses a value function to guide the policy training, it is fundamental that the value that the value network outputs is accurate. To achieve this, the value function parameters are trained to minimize the difference between the value network outputs and the returns obtained by the agent. This is typically done using the mean squared error between the predicted value and the empirical return:

$$L^{\mathrm{VF}}(\phi) = \mathbb{E}_\tau \left[\left(V_\phi(s_\tau) - \hat{R}_\tau \right)^2 \right],$$

This may induce a self-reinforcing feedback loop wherein the agent converges to suboptimal behaviors, the value function learns the corresponding (suboptimal) returns, and, since advantages use this value, the policy further reinforces, rather than penalizing, them.

For this reason, many PPO implementations also introduce the *entropy loss*. The agent is encouraged to keep the entropy of the policy distribution high, which promotes exploration and prevents premature convergence to suboptimal policies. The entropy loss is typically defined as:

$$L^{\mathrm{ENT}}(\theta) = -\mathbb{E}_\tau \left[H(\pi_\theta(\cdot|s_\tau)) \right],$$

where $H(\cdot)$ denotes the entropy of the policy distribution, computed as

$$H(\pi_\theta(\cdot|s_\tau)) = -\sum_a \pi_\theta(a|s_\tau) \log \pi_\theta(a|s_\tau).$$

The final loss function is

$$L(\theta, \phi) = L^{\mathrm{CLIP}}(\theta) + C_V L^{\mathrm{VF}}(\phi) + C_E L^{\mathrm{ENT}}(\theta)$$

where C_V is a hyperparameter that controls the trade-off between the value function loss and the policy loss, and C_E is a hyperparameter that controls the strength of the entropy regularization.

The optimization is performed over several iterations, called *training steps*. Each training step is composed of the following 2 sub-steps: the data collection step and the update epoch. During the update epoch, the set of collected data is divided in batches and several steps of optimization are performed over each batch. Notice that, at each training step, the ratio between the new and old policy should move away from 1, as the new policy learns different behaviors. The use of multiple training steps with limited batch size and of the clipped surrogate objective function ensures that the policy updates remain stable and do not lead to large, destabilizing changes.

Algorithm 2 Proximal Policy Optimization

Initialize policy parameters θ and value function parameters ϕ
for each iteration **do**
    Collect trajectories by running the current policy π_θ in the environment
    Compute returns and advantage estimates using the value function V_ϕ
    for each epoch **do**
        for each batch **do**
            Compute the clipped surrogate objective $L^{\text{CLIP}}(\theta)$
            Compute the value function loss $L^{\text{VF}}(\phi)$
            Compute the entropy loss $L^{\text{ENT}}(\theta)$
            Combine losses into: $L(\theta, \phi)$
            Update θ and ϕ in order to maximize $L(\theta, \phi)$ (backpropagation algorithm + SGD)
        end for
    end for
end for

Above we mentioned how advantages are computed as the difference between the returns and the value function estimates. In reality, in our implementation, we used a different definition of advantages based on the work of (Schulman et al., 2015), called Generalized Advantage Estimation (GAE), which provides a low-variance, low-bias estimate of the advantage function.

Conclusions

In this chapter, we presented our formulation for the CA-TSP and the approach used to solve it. After reviewing the DTSP solutions approach available in the literature, we introduced the CA-TSP as a model for DTSP problems where the dynamic evolution depends on the context in which the instance is inserted. Then, we discussed how to frame this problem as an MDP. To solve the MDP, we decided to use DRL, and we introduced the theoretical components for our solution framework, introducing the DNNs adopted and the PPO algorithm, employed for training. The choices of model architecture and learning strategy were motivated both by their theoretical properties and their suitability for handling the dynamic nature of the CA-TSP.

In the next chapter, we present the implementation details, describing the simulation environment, and especially the simulator used, and the components of the agent.

References

Adamo, T., M. Gendreau, G. Ghiani, and E. Guerriero (2024). "A review of recent advances in time-dependent vehicle routing". In: *European Journal of Operational Research* 319, pp. 1–15.

Applegate, D. L., R. E. Bixby, V. Chvatal, and W. J. Cook (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton University Press.

Bello, I., H. Pham, Q. V. Le, M. Norouzi, and S. Bengio (2017). "Neural Combinatorial Optimization with Reinforcement Learning". In: *5th International Conference on Learning Representations, ICLR 2017*.

Bertsimas, D. J. and G. Van Ryzin (1991). "Stochastic and Dynamic Vehicle Routing in the Euclidean Plane with Multiple Capacitated Vehicles". In: *Operations Research* 39.4, pp. 601–615.

Boers, E. J., H. Kuiper, B. L. Happel, and I. G. Sprinkhuizen-Kuyper (2001). "A Pheromone-Based Discussion of the Dynamic Traveling Salesman Problem". In: *Applications of Evolutionary Computing*. Springer, pp. 409–418.

Brody, S., U. Alon, and E. Yahav (2021). "How attentive are graph attention networks?" In: *arXiv preprint arXiv:2105.14491*.

Chen, D., C. Imdahl, D. Lai, and T. Van Woensel (2025). "The Dynamic Traveling Salesman Problem with Time-Dependent and Stochastic Travel Times: A Deep Reinforcement Learning Approach". In: *Transportation Research Part C: Emerging Technologies* 172.

Cheong, C. W. and A. Lim (2012). "Dynamic Traveling Salesman Problem Using Markov Decision Process". In: *Proceedings of the 2012 International Conference on Machine Learning and Computing*, pp. 239–243.

Cordeau, J.-F., G. Ghiani, and E. Guerriero (2014). "Analysis and branch-and-cut algorithm for the time-dependent travelling salesman problem". In: *Transportation science* 48.1, pp. 46–58.

Dantzig, G., R. Fulkerson, and S. Johnson (1954). "Solution of a large-scale traveling-salesman problem". In: *Journal of the operations research society of America* 2.4, pp. 393–410.

Groba, C., A. Sartal, and X. H. Vazquez (2015). "Solving the Dynamic Traveling Salesman Problem Using a Genetic Algorithm with Trajectory Prediction". In: *Expert Systems with Applications* 42.22, pp. 8793–8800.

He, K., X. Zhang, S. Ren, and J. Sun (2016). "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778.

Helsgaun, K. (2000). "An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic". In: *European Journal of Operational Research* 126.1, pp. 106–130.

Kool, W., H. Van Hoof, and M. Welling (2019). "Attention, Learn to Solve Routing Problems!" In: *International Conference on Learning Representations*.

Psaraftis, H. N. (1988). "Dynamic vehicle routing problems". In.

Psaraftis, H. N., M. Wen, and G. Kontoravdis (2016). "Dynamic Vehicle Routing Problems: Three Decades and Counting". In: *Networks* 67.1, pp. 3–31.

Rego, C., D. Gamboa, F. Glover, and C. Osterman (2011). "Traveling Salesman Problem Heuristics: Leading Methods, Implementations and Latest Advances". In: *European Journal of Operational Research* 211.3, pp. 427–441.

Schulman, J., P. Moritz, S. Levine, M. Jordan, and P. Abbeel (2015). "High-dimensional continuous control using generalized advantage estimation". In: *arXiv preprint arXiv:1506.02438*.

Schulman, J., F. Wolski, P. Dhariwal, A. Radford, and O. Klimov (2017). "Proximal policy optimization algorithms". In: *arXiv preprint arXiv:1707.06347*.

Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1, pp. 1929–1958.

Strak, Ł. and R. Skinderowicz (2019). "Self-Adaptive Ant Colony System for the Traveling Salesman Problem". In: *Expert Systems with Applications* 120, pp. 109–119.

Toriello, A., W. B. Haskell, and M. Poremba (2014). "The Traveling Salesman Problem with Stochastic Travel Times". In: *Transportation Science* 48.3, pp. 382–398.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin (2017). "Attention is all you need". In: *Advances in neural information processing systems* 30.

Veličković, P., G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio (2017). "Graph attention networks". In: *arXiv preprint arXiv:1710.10903*.

Zhang, Z., H. Liu, M. Zhou, and J. Wang (2023). "Solving Dynamic Traveling Salesman Problems with Deep Reinforcement Learning". In: *IEEE Transactions on Neural Networks and Learning Systems* 34.4, pp. 2119–2132.

Zhou, A., L. Kang, and Z. Yan (2003). "Solving Dynamic TSP with Evolutionary Approach in Real Time". In: *Proceedings of the 2003 Congress on Evolutionary Computation*. Vol. 2. IEEE, pp. 951–957.

Chapter 5

Environment and Agent details

Introduction

In this chapter, we describe the implementation of our proposed solution to the Context-Aware Traveling Salesman Problem (CA-TSP). Building on the theoretical foundations and problem formulation introduced in Chapter 4, we now turn to their practical implementation. Our goal is to show how the CA-TSP can be implemented in an environment that uses a traffic simulator and how to structure the agent that has to learn to solve this problem.

We begin by detailing the simulator used to model the traffic environment, with a particular focus on the process of input data generation. Next, we explain how the simulator interfaces with the environment, describing how states are constructed and how communication between the agent and the environment is managed during training. Finally, we outline the architecture of the Deep Neural Networks (DNNs) that form the core of the agent and discuss how data from the simulator are processed and adapted into suitable network inputs.

5.1 Simulator

Our environment is constructed using the SUMO traffic simulator (Lopez et al., 2018). SUMO is a software used for traffic simulation that, given a road network and the routes that each vehicle should follow, plus some configuration parameters, simulates the movement of vehicles on the roads—including traffic lights, speed limits, drivers behaviors and other factors.

In this section, by *route*, we refer both to the ordered list of edges in the road network a vehicle should follow and to when a vehicle should enter the simulation, i.e., its *departure time.* Once a vehicle enters the simulation, it follows the list of edges in its route until it reaches the destination, then it is removed from the simulation.

We chose SUMO for several reasons. Firstly, its flexibility, but also the availability of tools designed to simplify input creation and, most importantly, the possibility to interact with the simulation in real-time. This last feature is crucial for our problem, as it allows us to insert the salesman in the simulation and let him interact with the environment, receiving information about the current state of the system and updating his strategy accordingly.

In order to use SUMO, we first need to create a road network and a set of routes

for the vehicles that enter the simulation. In the following of this section we present the details of the creation of this input data.

5.1.1 SUMO input data creation

We describe here the steps we followed to generate these inputs for the simulation, while motivating our choices. We start by presenting how we created the road network and then explain the traffic generation process.

Map generation Among SUMO tools, `OsmWebWizard` allows users to download maps from OpenStreetMap (OSM) (OpenStreetMap contributors, 2017) and convert them into a SUMO-compatible road network.

`OsmWebWizard` Graphical User Interface (GUI) allows users to select an area through a bounding box on the world map. The resulting map can be customized to include different road types (e.g., highways, secondary roads, etc.) and other options. By choosing a real World map (and not a synthetic one), we intend to remark the applicability of our method to real world problems.

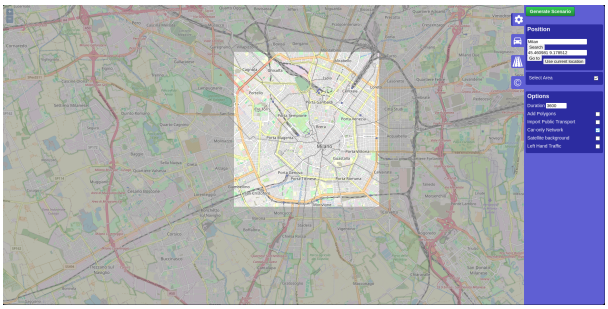


Figure 5.1: OsmWebWizard GUI.

From the configuration sidebar, we choose to include only car roads, excluding other types, such as pedestrian and bike paths, as they are not relevant for our simulation. We also remove polygons from the map, since they serve only aesthetic purposes. While OsmWebWizard can generate traffic data, we did not leverage them—they are hardly comparable to real traffic patterns.

The main output of OsmWebWizard is an XML file. This file contains all the roads for the selected area, with information regarding the number of lanes for each road, the speed limit, the length and the vehicles that are allowed to travel on it; together with information about junctions, traffic lights, roundabouts and other elements that all together represents the road network of the selected map. The format of this XML is ready to be imported into SUMO.

The road network comprises solely roads that are contained in the region we selected when generating the map. For this reason, it might happen that two roads are not connected, meaning that there is no way for a car to go from the first road to the second—e.g., because the connection happens outside the border of the area we selected. This connectivity is essential to ensure that every generated customer can be reached from any other customer, given that customers are spawned randomly. Thinking of the road network as a graph, this means that each edge must be reachable from any other edge. But there is a slight difference between a directed graph and a road network. In a directed graph, given a node, an edge entering the node and another edge exiting the

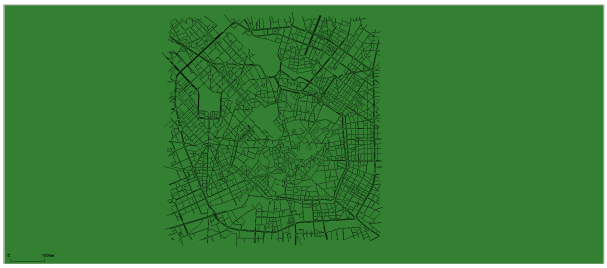


Figure 5.2: An example of a map loaded in SUMO.

node, the two edges are necessarily connected. In a road network, that is not necessarily the case. Consider, for example, the junction in Figure 5.3. In this case, each road (edge) entering the junction (node) is connected solely to the road (edge) exiting the junction (node) in the same direction, thus, regardless of where you are coming from, you can only continue straight.

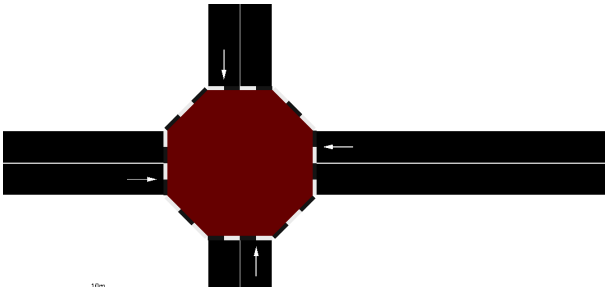


Figure 5.3: A junction with four roads entering and four roads exiting, where only going straight is allowed.

To ensure that every customer can be reached from any other customer, we first identify all the edges such that, if we take any other edge in the network, they are not reachable. We achieve this by using a Python script that parses the .net.xml file and creates a *dual graph* for our road network. In this graph, each edge is converted to a node; we create a directed edge from node i to node j, if edge j can be reached from edge i passing through the node (junction) that they share. At this point, we find all the strongly connected components of the dual graph, i.e., the subgraphs where every node is reachable from every other node. Then, we keep only the biggest strongly connected component—the one that contains the most nodes. We use this information to remove all the edges that are not part of the biggest strongly connected component from the original map. This means that the resulting map is a directed graph where every edge is reachable from any other edge, and by extension, every customer can be reached from any other customer, no matter where they are located on the map.

The cleaned map still requires further adjustments to be suitable for our purposes. In simulations, vehicles do not behave exactly like real-world vehicles; they follow predefined logic, which can lead to traffic jams, especially at complex junctions. The map from OSM may include junctions that, while accurate to the real road network, are difficult for simulated vehicles to navigate. To address this, we use the `netconvert` tool to sanitize the map, making it more navigable for vehicles. Some of these changes include converting complex junctions into roundabouts, automatically detect possible ramps and guess the position of traffic lights.

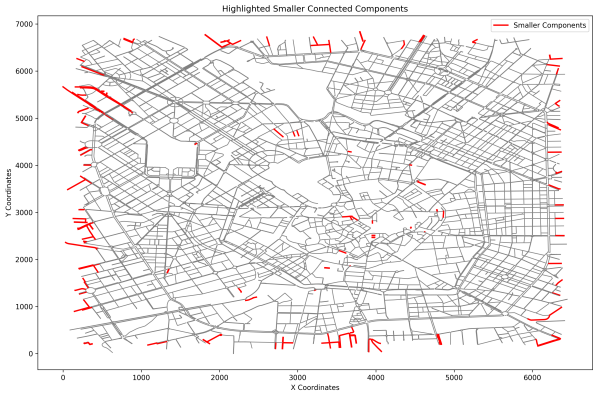


Figure 5.4: Map with the smaller components highlighted in red. These edges are removed from the map.

Traffic generation Once the map is ready, the second input for the simulation is the file with the routes that each vehicle should take. This file determines the traffic in the simulation. Our objective is to simulate a full day of traffic, starting at 00:00:00 and ending at 23:59:59; throughout the course of the day, dynamic changes should take place depending on the day of the week, the time of the year and the weather.

We generate traffic by partitioning the network into Traffic Assignment Zones (TAZ), each defined as a portion of the map containing a set of edges. To simplify the TAZs creation, we simply create TAZs using a grid-based approach. We divide the map into a grid of squares, and edges with the midpoints in the same square are assigned to the same TAZ. This step is easily achieved using the `gridDistricts` tool—a SUMO tool that creates the grid of TAZs for an input map given the width of the squares in the grid.

Figure 5.5: The grid of TAZs.

For every ordered pair of zones, we specify a *relation.* Creating a relation means assigning a volume of vehicles flowing from the origin TAZ to the destination TAZ over the simulation. In SUMO, zones and relations are declared in two .xml files: one lists each TAZ and its member edges, and the other enumerates a list of ordered pairs of TAZs along with the number of vehicles and the simulation duration.

The relations between TAZ pairs are established using a custom Python script, which assigns a fixed number of vehicles to each pair of TAZs. For instance, if we have three TAZs labeled A, B, and C, and the number of vehicles is set to 100 for each relation,

the script creates nine relations: $A \to A$, $A \to B$, $A \to C$, $B \to A$, $B \to B$, ..., $C \to C$. During the simulation, 100 vehicles will flow from A to B, 100 from A to C, and so forth. This process results in an *Origin-Destination (OD) matrix* in *tazRelation* format. In general, given n TAZs, the number of relations is n^2 and the total number of vehicles for the simulation is $n^2 * vehicles$.

SUMO can compute the route for each vehicle at runtime, starting from the OD matrix. But when there are many vehicles, this becomes time expensive. To expedite the simulation, we use the `od2trips` tool, which converts the OD matrix into a set of predefined *trips.* A trip consists of a vehicle, a departure time, a starting edge, and an ending edge, but it does not specify the exact route the vehicle will take. Once the trips are defined, we can compute the actual routes using the `duarouter` tool. This tool employs the Dijkstra algorithm to compute the fastest route for each vehicle (without considering traffic). The outcome of this process is an XML file, which contains the routes, i.e., the sequence of edges, that each vehicle must follow.

This file, along with the .net.xml file and a configuration file for the simulation, serves as the input for the SUMO simulation.

Scenario generation In the previous section, we discussed how to generate traffic based on a Origin-Destination (OD) matrix over TAZs. However, this approach results in a constant flow of vehicles entering the simulation at each hour, which does not accurately reflect the daily traffic patterns observed in real life. In reality, traffic volume fluctuates significantly throughout the day, with pronounced peaks during rush hours and much lower volumes during the night. Capturing these temporal variations is crucial for creating realistic and challenging scenarios for the agent.

To address this, we create different scenarios that represent various days of the week and weather conditions. This diversity is important because it exposes the agent to a wide range of possible environments states, improving its robustness and generalization capabilities.

To that end, we utilize the `timeline` parameter of the `od2trips` tool. This parameter allows us to specify the percentage of vehicles entering the simulation during each hour of the day, thereby shaping the temporal distribution of traffic.

To simulate the different distribution of traffic in workdays and weekends, we construct a function that models how many vehicles enter the simulation at each hour. This function is obtained by summing Gaussian distributions centered at different hours of the day and with different standard deviations. The Gaussians are also scaled by different factors, to reflect the relative intensity of the morning, midday, and evening peaks. We then sample this function at hourly intervals and normalize the resulting values so that their sum equals 1. The outcome is a list of 24 values, each representing the proportion of vehicles entering the simulation in a given hour.

To illustrate, we define the following example scenarios:

- **Workday**: Three Gaussian distributions with means at 8:00 AM, 12:00 PM, and 5:00 PM, and standard deviations of 9000s, 1800s, and 7200s, respectively. This configuration captures the typical morning and evening rush hours, as well as a smaller midday peak.

- **Weekend**: Two Gaussian distributions with means at 10:00 AM and 6:00 PM, and standard deviations of 12600s and 7200s. This reflects the tendency for traffic to peak later in the day on weekends.

These distributions are shown in Figure 5.6.

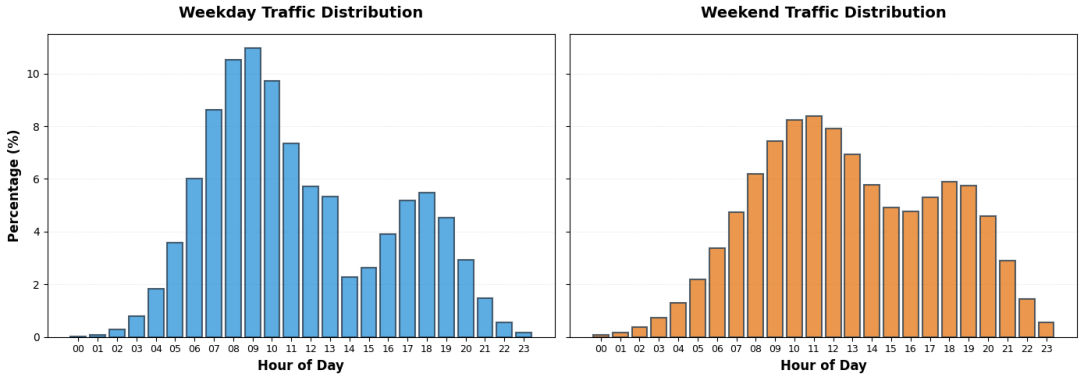


Figure 5.6: Percentages of vehicles entering the simulation (y-axis) at each hour of the day (x-axis). The functions are constructed as a sum of Gaussians to model daily traffic peaks for workdays and weekends respectively.

Thus, we can flexibly model different traffic patterns by adjusting the number, position, and scale of the Gaussian components. For instance, we can represent the reduced traffic on weekends by shifting the peaks to later hours and lowering their amplitude. Rush hour allocation is based on common sense, according to which traffic tends to be more intensive around office hours during workdays and around social gatherings in the weekend; for our intents and purposes, it is not relevant whether the *actual* rush hour peak in Milan on average is at 8:24 or 8:00, as along as the pattern is somewhat realistic. Similarly, we can simulate the effect of holidays or special events by modifying the function accordingly.

In addition to the day of the week, we also consider the impact of weather conditions on traffic patterns. To simulate the effect of adverse weather, such as rain, we reduce the maximum speed of vehicles and roads by 10% and increase the total number of vehicles, reflecting the tendency for people to use the car more and drive more cautiously in poor conditions.

For each combination of day type and weather, we generate a different scenario. This result in the 4 distinct scenarios, one for each combination of day type (workday-weekend) and weather condition (sunny-rainy). Each scenario corresponds to a unique .rou.xml file (defining the vehicle routes) and a corresponding osm.net.xml file (defining the road network, which may have different speed limits depending on the weather conditions).

5.2 Environment creation

In the preceding sections, we discussed the process of generating input data for the simulator. These steps are not directly part of the RL process, but they are essential for creating a realistic and controlled environment for the agent.

To implement the environment, we need to create a piece of software that creates CA-TSP instances using the data created as above, simulates their execution, and provides the necessary interfaces for the RL agent to interact with it.

To achieve this, we leverage the Gymnasium (Towers et al., 2023) library, which is a widely adopted Python toolkit for developing RL environments. Gymnasium provides a simple, consistent communication protocol between the environment and the agent that

makes it easier to create environments that adhere to well-defined standards. In addition, it offers a rich ecosystem of pre-built tools to parallelize training and evaluation. The parallel execution is one of the most important features for our purpose, because it allows for efficient training of agents in environments where episodes are long or computationally expensive, as is the case with traffic simulations.

The core abstraction in Gymnasium is the `Env` class, which defines the methods and properties that any RL environment must implement. To define the environment, we need to extend the Gymnasium `Env` class and implement two primary methods: `reset` and `step`. Both the `reset` and `step` methods return what we call an *observation*, that is the set of information the environment sends to the agent at each time step.

5.2.1 Reset

The `reset` method is a fundamental component of any Gymnasium environment. It is responsible for initializing the environment at the start of each episode and returning the initial observation to the agent. A well-designed `reset` method ensures that each episode begins from a valid and potentially diverse state, crucial for effective training of RL agents.

In our environment, the `reset` method is responsible to load the road network (that is fixed for all the episodes), generate a new set of customers on the edges of the road network, and initialize the salesman position and the start time for its route. Moreover, it must decide the external conditions and compute the initial traffic conditions. Finally, it must initialize the set of already visited nodes with the current position of the salesman.

All these pieces of information compose the initial observation, which is then returned to the agent, marking the beginning of a new episode. — In our environment, the `reset` method performs several key steps to set up a new episode:

1. **Select a random scenario.** To select the set of external conditions and the traffic distribution over the network, the `reset` method randomly selects one of the pre-generated scenarios from the `scenarios` directory. In this scenario is also present the road network, which may have different speed limits depending on the weather conditions.

2. **Start the SUMO simulation using the selected scenario.** The SUMO simulator is launched with the configuration files corresponding to the chosen scenario. This includes the road network, the vehicle routes, and any scenario-specific parameters such as speed limits or weather effects.

3. **Generate a list of customers to visit.** The set of customers for the episode is determined by randomly selecting a number of edges from the road network graph and then choosing a random number in the interval $[0, l_e]$ where l_e is the length of the selected edge. Each node position is then converted to xy-coordinates in the plane.

4. **Determine a random starting node for the salesman.** The initial position of the salesman is chosen randomly from the list of customers.

5. **Set a random start time for the salesman route.** To further increase the diversity of episodes, the starting time of the salesman is selected at random from a predefined list of possible start times. This list includes all hours intervals between

05:00:00 and 19:00:00, covering the main periods of daily traffic variation. By varying the start time, we ensure that the agent experiences different traffic conditions, even within the same scenario.

After completing these steps, the simulation starts and it pauses when time reaches the start time. At this point, we insert the salesman vehicle in the simulation and the initial information is constructed, using the traffic data at the time at which the simulation paused.

5.2.2 Step

The `step` method takes as input an action, it is responsible for updating the environment state according to the current state and the action received. This method is central to the RL loop, as it defines how the agent decisions affect the environment and the resulting feedback.

In our environment, the `step` method moves the salesman to the next customer based on the selected action, marking the new customer as visited and recording the amount of time taken to reach it.

After performing these steps, the environment constructs the new observation using the updated environment state, similar to the `reset` method. It also returns the reward (the negative of the recorded travel time) and an auxiliary signal, called *done signal*, that indicates if the episode has ended.

The reward function is a critical component of any RL environment—it defines the objective that the agent is trying to optimize. In our setting, the reward is designed to encourage the agent to minimize the total travel time required to perform a full tour.

As mentioned above and in Chapter 2, the reward at each step is the negative of the travel time needed to reach the next customer. This type of reward is called *dense* because the agent receives feedback after every action, rather than only at the end of the episode. Dense rewards are generally preferred in RL, as they provide more frequent learning signals and facilitate faster convergence.

At the final step, however, when the agent has visited the last customer and must return to the starting location, the reward includes both the travel time for visiting the last customer and the travel time for returning to the first customer.

Example:
Suppose there are 4 nodes A, B, C, D. At the beginning of the episode, the agent is currently at depot A, chooses to visit customer B next, and the travel time from A to B is 12 minutes. The reward for this step is -12. Then it decides to visit customer C, and the travel time from B to C is 10 minutes, so the reward for this step is -10. Finally, it has to visit customer D and to go back to customer A. The travel time from C to D is 8 minutes and the travel time from D to A is 4 minutes. Hence, for the final step, it receives a reward of $-(8 + 4) = -12$. A representation of how the episode evolves is depicted in Figure 5.7.

In the TSP literature, rewards are often defined sparsely: the agent receives a single terminal reward equal to the negative cost of the completed tour. In dynamic settings, however, dense rewards are preferable because they provide stepwise feedback at each decision point, thus they allow the agent to assess the quality of its actions using information available at the current and subsequent time steps.

In our environment, the `step` method performs the following sequence of operations:

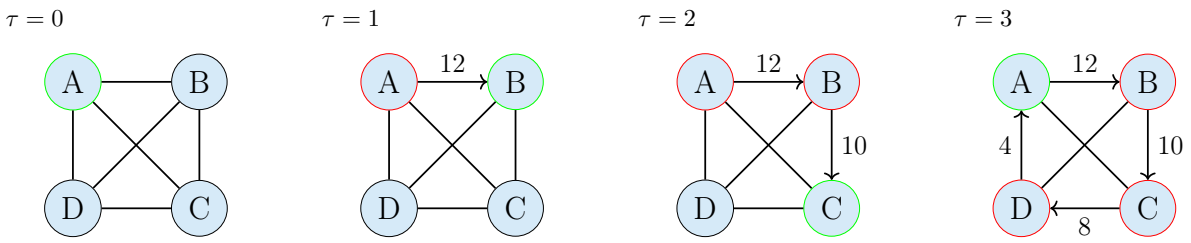


Figure 5.7: Figure that represents the rewards obtained at each step in an example with four nodes.

1. **Instruct the simulation to move the salesman to the next customer.** The agent action is interpreted as the selection of the next customer to visit. Using the `libsumo` library, we command the SUMO simulator to move the salesman vehicle from its current position to the chosen customer location. After the command is received by SUMO, the simulator computes the fastest route to the next customer according to current traffic condition using the Dijkstra algorithm. This is the route that the salesman vehicle will follow, until it reaches the next customer. There is not the possibility of changing this route once it has been set.

2. **Wait for the salesman to arrive at the next customer.** The simulation advances in time until the salesman reaches the destination. During this period, the environment updates the position of the vehicle, taking into account any delays caused by traffic jams, speed limits, traffic lights or other dynamic factors.

3. **Record the travel time.** Upon arrival, the environment records the actual travel time required to reach the next customer. Notice that this value might be different from the value used by the Dijkstra algorithm to compute the fastest route, because of the evolving traffic conditions. This travel time is used to compute the reward for the agent, which is the negative of the travel time (since the objective is to minimize total travel duration).

4. **Mark the current customer as visited.** The environment updates its internal state to reflect that the current customer has been visited. This information is used to mask out already visited customers in future action selections, ensuring that the agent does not revisit the same location.

5. **Check for episode termination.** The episode ends when all customers have been visited and the salesman has returned to the starting location. If the episode has ended, the environment signals this to the agent and the SUMO simulation is closed.

The `step` method is designed to closely mimic the real-world process of making sequential routing decisions in a dynamic environment. For example, if the agent chooses to visit a customer located in a congested area during rush hour, the travel time will be longer, resulting in a lower (more negative) reward. Conversely, if the agent anticipates future traffic conditions and selects a more efficient route, it will be rewarded with shorter travel times. This design encourages the agent to learn not only to minimize immediate travel costs but also to plan ahead, taking into account the evolving state of the traffic network.

We recall that the observation returned by both the `reset` and `step` methods includes the following components:

- **The current position of the salesman.** This indicates where the agent is located in the city at the current time step.
- **The positions of the customers.** The xy-coordinates for the locations of all customers that need to be visited in the current episode.
- **The expected travel time between customers.** For each pair of customers, the environment provides the estimated travel time, as computed by the SUMO simulator. These are the travel times estimated by the Dijkstra algorithm. These values are an indication of the current traffic conditions and are updated at each time step.
- **The road network with real time traffic.** The full structure of the road network is included, alongside real-time information about the state of the traffic on each road segment. Traffic is represented by augmenting each edge with features such as the current number of vehicles or their average speed at that time.
- **The current time of day, day of the week, and weather conditions.**
- **The list of customers that have been visited.**

To efficiently encode this rich set of information, we use a combination of graph-based and vector representations:

- **Road network graph:** A weighted directed graph where each node represents a junction and each edge represents a road segment. Node features include the coordinates of the junction, while edge features include the length of the edge, the number of vehicles currently on it, and the average speed of those vehicles.
- **Customers graph:** Another weighted directed graph, where each node corresponds to a customer and each edge represents the full path between two customers. Node features are the coordinates of the customers, and edge features are the current estimated travel times between customers, as provided by SUMO.
- **Salesman context information:** A vector with the current position (xy-coordinates) of the salesman and a list of the indexes of the already visited customers.
- **Global context vector:** A fixed-length vector encoding the current time, day type and weather.

In practice, each graph is represented using three matrices:

- A node feature matrix, with one row per node and one column per feature.
- An edge index matrix, with two columns (source and target node indices) and one row per edge.
- An edge feature matrix, with one row per edge and one column per feature.

We call n_r the number of nodes and e_r the number of edges in the road network graph, and n_c the number of nodes and e_c the number of edges in the customers graph. We refer to the number of features per node in the road network graph as f_{nr}, and to the number of features per edge in the road network graph as f_{er}, while we call f_{nc} the number of features per node in the customers graph, f_{ec} the number of features per edge in the customers graph and, finally, f_g the number of features in the global context vector. Then, the observation includes:

- N_r: $n_r \times f_{nr}$ node feature matrix for the road network, where each row corresponds to a junction and each column corresponds to a feature.
- I_r: $e_r \times 2$ edge index matrix for the road network, that is a matrix with entries (i, j) for each edge from node i to node j.
- E_r: $e_r \times f_{er}$ edge feature matrix for the road network, where each row corresponds to an edge and each column corresponds to a feature in the road network.
- N_c: $n_c \times f_{nc}$ node feature matrix for the customers graph, where each row corresponds to a customer and each column corresponds to a feature.
- I_c: $e_c \times 2$ edge index matrix for the customers graph, that is a matrix with entries (i, j) for each edge from node i to node j in the customers graph.
- E_c: $e_c \times f_{ec}$ edge feature matrix for the customers graph, where each row corresponds to an edge and each column corresponds to a feature.
- A global context vector of length f_g.

This structured representation will be the key for the use of graph neural networks (GNNs) and other advanced DNN architectures that can exploit the relational and spatial structure of the problem, mimicking the graph representation used by PyTorch Geometric (Fey et al., 2025).

5.2.3 Action

While it is the agent that performs actions, the agent must communicate them in a way that it is comprehensible by the environment. It is common practice, when defining an environment, to define the space of possible actions, i.e., the *action space.*

In our case, at each time step, the agent must select the next customer to visit from the set of unvisited customers. Formally, if $V = \{1, 2, \ldots, N\}$ is the set of all customers, and V_u is the subset of unvisited customers, then an action is an integer in V_u, (i.e., V_u is the action space). At the beginning of each episode, the agent is placed at a randomly selected customer location a_0, so $V_u = V \setminus \{a_0\}$ and the first action a_1 is to choose which of the remaining customers to visit next. At the second step, the agent will be at the location of the customer selected in the first action, and the set of unvisited customers will be $N_u = N \setminus \{a_0, a_1\}$, and so on. When all customers have been visited (i.e., $V_u = \emptyset$), the environment automatically sends the salesman back to the first visited customer to complete the tour.

5.3 Agent implementation details

All the DNNs are implemented using PyTorch (Ansel et al., 2024) and PyTorch Geometric (Fey et al., 2025) libraries. These frameworks provide efficient primitives for GNNs and support for batching, parallelism, and GPU acceleration, which are essential for scaling to large problem instances and for efficient training.

5.3.1 Policy

The policy network is the central component of the agent, responsible for mapping the current observation to a probability distribution over possible actions, i.e., selecting the next customer to visit. Given the complexity and dynamic nature of the environment, our policy is designed as an encoder-decoder architecture that can effectively process both graph-structured and contextual information.

The encoder-decoder paradigm is well-suited for problems where the input consists of multiple, potentially heterogeneous sources of information. In our case, the encoder is composed of two separate GNNs, each specialized for a different aspect of the environment.

The first encoder, that we call *City Encoder*, processes the road network graph. Its purpose is to capture the structure of the city and the current state of traffic on each road segment. Its output is a single vector, referred to as the *city context vector.*

The second encoder, that we call *Customer Encoder*, processes the customers graph, and should represent the customers to visit and the spatial relationship between them. Its output is a graph with the same nodes and edges of the original customers graph, but with updated node embeddings.

Finally, the decoder takes both the city context vector and the updated customer graph as input and the vector of global features to produce a probability distribution over the next customer to visit. The full architecture is represented in Figure 5.8.

Both the encoder and the decoder are run at each time step, allowing the model to incorporate the most recent information about the environment and the agent state. This is opposed to what usually happens in static NCO algorithms, where the encoder is run only at the first step and then the decoder is run using the fixed embedding and a context vector (Kool et al., 2019).

All the networks should account for the graph structure of the input data, ensuring that the relationships between nodes (e.g., customers and road segments) are preserved and leveraged throughout the processing pipeline.

City Encoder

The City Encoder is a fundamental building block of our policy network. It is designed to process the entire road network graph and distill its complex, dynamic information into a single, informative vector representation——i.e., the context vector. This context vector is intended to summarize the global state of the city, capturing both its structural properties and the current traffic conditions; it is later used by the decoder to inform the agent routing decisions.

The challenge in encoding a real-world road network lies in its scale and heterogeneity. A city road network can contain thousands of nodes (junctions) and edges (road segments), each with its own set of features. These features are not static: some, like

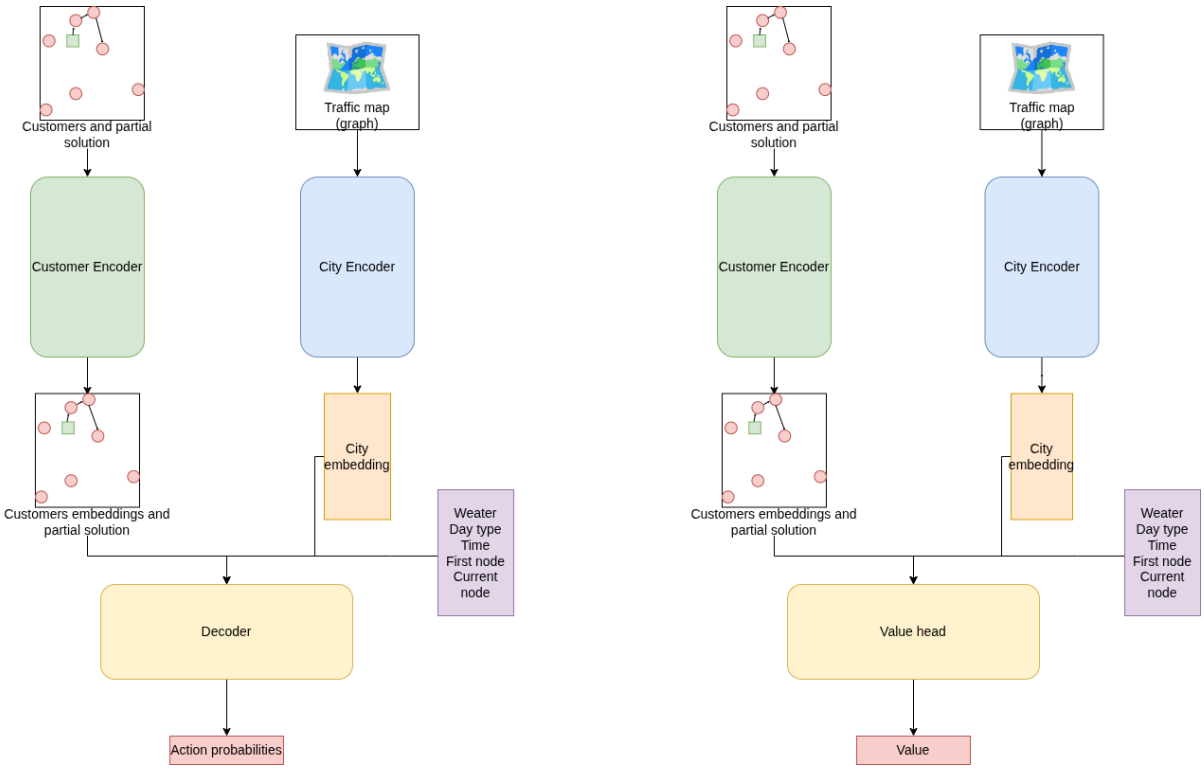


Figure 5.8: Figure representing the whole network architecture. The policy network is represented on the left and the value network on the right.

the length of a road, remain constant, while others——such as the number of vehicles on a segment or the average speed——change dynamically as the simulation progresses. Simply flattening or averaging these features would result in a significant loss of information, especially regarding the relationships and interactions between different parts of the network.

To avoid such loss, we employ a multi-layer Graph Attention Network (GATv2) (Brody et al., 2021) as the core of the City Encoder. The GATv2 architecture is particularly well-suited for this task because it allows the model to learn which nodes and edges are most relevant in the current context. Through its attention mechanism, the network can dynamically focus on congested areas, critical junctions, or other features that are likely to impact the agent decisions. For example, if a particular intersection is experiencing heavy traffic, the attention mechanism can assign it greater importance, ensuring that this information is reflected in the final context vector.

The encoding process begins by feeding the node and edge features of the road network into the first GATv2 layer. Each node updates its embedding by aggregating information from its neighbors, with the attention mechanism determining the weight of each neighbor contribution. This process is repeated across several layers, allowing information to propagate through the network—enabling the model to capture both local and global patterns. For instance, a node final embedding may reflect its immediate traffic conditions as well as the state of nearby intersections plus the overall flow of traffic in its vicinity.

After the final GATv2 layer, we apply a *global mean pooling* operation. This step aggregates the node embeddings into a single, fixed-size vector with the mean of all the node embeddings. With this operation we obtain a single vector from the entire road

network. The mean pooling ensures that the context vector is invariant to the number of nodes, making the encoder robust to different city layouts and scales. The resulting city context vector is a compact summary of the entire road network, encoding information about congestion, connectivity, and other structural and dynamic properties.

As already mentioned, the use of GATv2, as opposed to simpler GNN architectures, is motivated by its flexibility and expressiveness. In a dynamic environment, like urban traffic, the importance of different roads and intersections can change rapidly. The attention mechanism in GATv2 allows the encoder to adapt to these changes on the fly, highlighting the most relevant parts of the network for each decision step. This is crucial for the agent to develop strategies that are sensitive to the current state of the city, such as avoiding newly formed traffic jams or exploiting temporarily decongested routes.

In summary, the City Encoder transforms a high-dimensional, dynamic, and relational input——i.e., the road network graph——into a single vector that should encapsulate the most salient information for the agent decision-making process.

Customer Encoder

The Customer Encoder is the second major component of our policy encoder architecture, and it is specifically designed to process the customer graph——i.e., the set of delivery points and the relationships between them. While the City Encoder provides a global summary of the entire road network and its traffic state, the Customer Encoder focuses on the local structure and dynamic relationships that are directly relevant to the agent immediate routing decisions.

Recall that, in the context of the CA-TSP, each customer is represented as a node in a graph, with features representing its geographic coordinates. The edges between customers represent the current estimated travel times, which are computed by the SUMO simulator and reflect the real-time traffic conditions on the underlying road network. This abstraction allows the agent to reason about the cost of moving between any pair of customers, taking into account the latest information about congestion and delays.

The main challenge for the Customer Encoder is to capture not only the individual properties of each customer but also the complex, time-varying relationships between them. For example, two customers might be geographically close, but if a major road between them is congested, the effective travel time could be much higher than expected. Conversely, a customer that is farther away in terms of distance might be more easily reachable if the connecting roads are clear. The encoder must therefore learn to integrate both spatial and dynamic information to produce meaningful representations for each customer.

To achieve this, in this instance too, we employ a multi-layer Graph Attention Network (GATv2) (Brody et al., 2021). The GATv2 layers allow each customer node to aggregate information from its neighbors, with the attention mechanism dynamically weighting the importance of each connection based on the current travel times and other edge features. This means that, at each layer, a customer embedding is updated not just by its own features, but also by the most relevant information from other customers——especially those that are easy or difficult to reach under current traffic conditions.

Unlike the City Encoder, which aggregates all node embeddings into a single context vector, the Customer Encoder preserves the individual embeddings for each customer node. This is because, in the decision-making process, the agent needs to evaluate the desirability of visiting each unvisited customer as a potential next step. The final output

of the Customer Encoder is, thus, a set of node embeddings, one for each customer, where each embedding encodes both the customer own features and its dynamic relationships to all other customers in the current episode.

This design enables the policy decoder to compare the context vector (from the City Encoder) with each customer embedding, allowing the agent to make informed, context-sensitive choices about which customer to visit next. For example, if a particular customer is currently difficult to reach due to traffic, its embedding will reflect this, and the decoder can assign it a lower probability. Alternatively, if a customer is well-connected and easily accessible, its embedding will highlight this advantage.

The use of GATv2 is motivated by the same reasons as for the City Encoder, i.e., its flexibility and expressiveness.

In summary, the Customer Encoder transforms the raw, dynamic customer graph into a set of rich, context-aware embeddings. These embeddings serve as the foundation for the agent action selection process, enabling it to reason about the current and future costs of visiting each customer and to adapt its strategy as the environment evolves.

Decoder

The Decoder is the final and decisive component of our policy architecture. It is responsible for translating the rich, structured information encoded by the City Encoder and Customer Encoder into actionable probabilities for the agent. In essence, the Decoder determines, at each decision point, which customer the agent should visit next, given the current state of the environment.

The design of the Decoder stems from the need to integrate multiple sources of information: the global context of the city (as captured by the context vector from the City Encoder), the local and relational information about each customer (from the Customer Encoder), and the global scenario features. The challenge is to combine these heterogeneous data streams in a way that allows the agent to make context-sensitive, adaptive decisions that reflect both the current and anticipated future states of the environment.

To achieve this, the Decoder employs a MHA. The core idea is to compute, for each unvisited customer, an attention score that reflects the desirability of visiting that customer next, given the current context.

The process begins by projecting the context vector——obtained by concatenating the city embedding and the global information vector——into the same embedding space as the customer embeddings. This projection ensures that the context and customer representations are compatible and can be meaningfully compared.

For each customer, the Decoder computes an attention score between the context vector and the customers embedding. This score can be interpreted as a measure of how well the current context "matches" the prospect of visiting that customer next. For instance, if the context vector encodes heavy congestion in a certain part of the city, customers located in or near that area may receive lower scores, while those in less congested regions may be favored.

A crucial step in this process is masking: customers that have already been visited are assigned a score of $-\infty$, ensuring that they are never selected again within the same episode. This masking is essential for enforcing the constraints of the TSP and for guiding the agent to construct valid tours.

Once all attention scores have been computed and masked, the Decoder applies a softmax function to convert the scores into a probability distribution over the set of valid

(unvisited) customers. This distribution represents the agent policy by specifying, for the current state, the probability of selecting each possible next customer. During training, the agent samples from this distribution to encourage exploration; during evaluation, it typically selects the customer with the highest probability (greedy selection) to maximize performance.

The MHA allows the Decoder to consider multiple “perspectives” or “subspaces” when evaluating each customer, enhancing its ability to capture complex dependencies and interactions in the data. This is particularly valuable in the DTSP, where the optimal choice may depend on subtle combinations of global context, local traffic, and the evolving set of remaining customers.

By integrating the outputs of the City Encoder, Customer Encoder, and global context vector, the Decoder enables the agent to make decisions that are both globally informed and locally adaptive. It can, for example, learn to avoid customers that are temporarily hard to reach due to traffic, prioritize those that are likely to become less accessible in the near future, or exploit windows of opportunity created by transient changes in the environment.

In summary, the Decoder serves as the decision-making "head" of the policy network, synthesizing all available information to produce a probability distribution over possible actions. Its attention-based design ensures that the agent choices are sensitive to both the current and anticipated future states of the environment, enabling robust and adaptive solutions to the dynamic TSP.

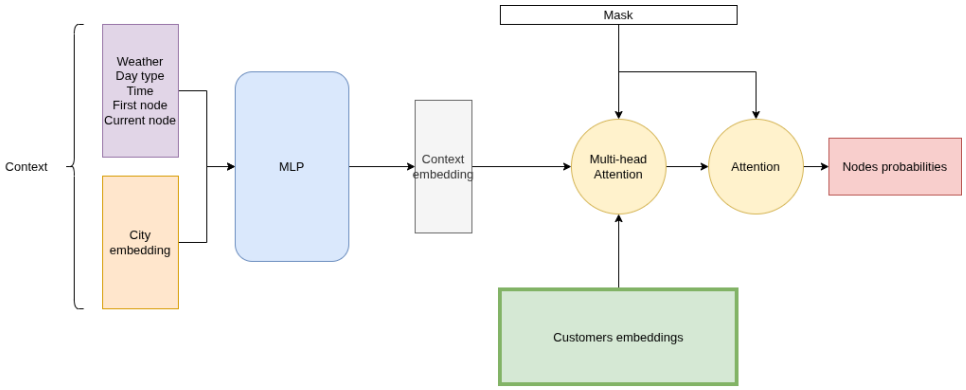


Figure 5.9: Flow of the decoder layer.

The process works as detailed in Figure 5.9:

- The context vector is projected into the same embedding space as the customer embeddings using an MLP, creating the context embedding.
- For each customer, an MHA is computed as a function of the context embedding and the customers embeddings.
- A final, single head attention, is computed to create the probability distribution over the remaining customers.
- Both the MHA and the attention are masked to exclude already visited customers, ensuring that only valid actions are considered.

The customer with the highest probability is selected as the next node to visit, either by sampling (during training) or by taking the argmax (during evaluation).

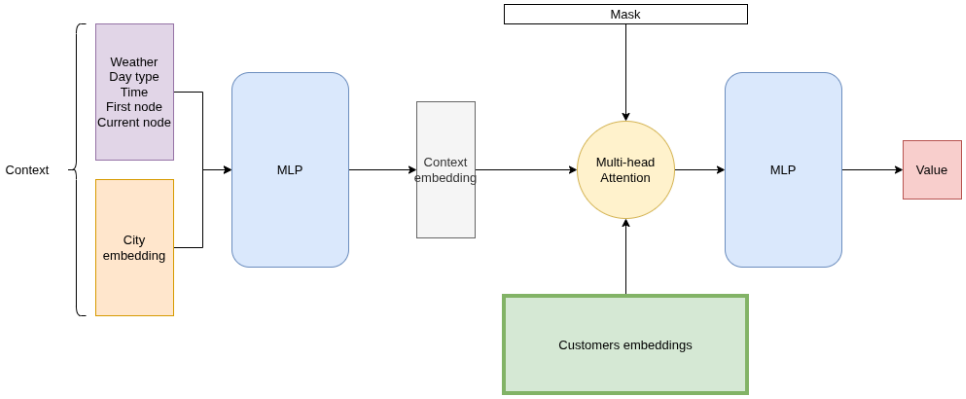


Figure 5.10: Flow of the value network head.

5.3.2 Value Network

The value network plays a central role in the learning process, by providing an estimate of the expected return (or value) for any given state. In the context of PPO, the value function is used to compute the advantage of each action, which in turn guides the policy updates and helps stabilize learning. The advantage of an action is defined as the difference between the expected return and the value of the current state.

Structurally, the value network closely mirrors the design of the policy network, as it must process the same rich, graph-structured, and contextual information to produce a meaningful estimate (see Figure 5.10). A key difference is that, instead of outputting a probability distribution over possible actions, the value network outputs a single scalar value for each state, representing the expected cumulative reward the agent can obtain from that point onward.

The process begins by constructing a context vector, just as in the policy network. This context vector is formed by combining the city embedding (from the City Encoder), the global information vector, and the first and current customer embeddings (from the Customer Encoder). This context vector is then projected into the same embedding space as the customer embeddings produced by the Customer Encoder.

Then, instead of the Decoder, we have the Value head. The Value head computes an attention score for each node, reflecting its relevance in the current state, exactly as in the Decoder. The attention scores are then masked to exclude already visited customers, ensuring that only valid, unvisited nodes contribute to the value estimate.

Rather than using these scores to select an action, the value network uses them to aggregate information from the customer embeddings. Specifically, it applies a softmax over the attention scores to obtain a set of weights, which are then used to compute a weighted sum of the customer embeddings. This operation produces a single, context-aware embedding that summarizes the most relevant aspects of the current state, taking into account both the global context and the local structure of the remaining customers.

Finally, this aggregated embedding is passed through a small Multi-Layer Perceptron (MLP)——typically consisting of a linear layer, a nonlinear one (such as tanh), and a final linear layer—to produce the scalar value estimate for the state. This value represents the agent prediction of the total future reward it can expect to collect, starting from the current state and following its current policy.

The design of the value network ensures that it is sensitive to the same factors as the policy: the current traffic conditions, the spatial arrangement and connectivity of

the customers, the global scenario context, and the set of customers that remain to be visited. By leveraging the same graph-based and attention mechanisms, the value network can provide accurate and informative value estimates, which are essential for effective advantage estimation and stable policy learning.

5.4 Mapping observations and actions

An issue we needed to address was the mapping of how SUMO represents networks to the graph structure expected by the policy network. Indeed, SUMO represents edges and nodes using a unique ID, while PyTorch Geometric expects edges and nodes in a matrix format, as described in Section 5.1.

To address this, we created a mapping from the SUMO edge and node IDs to the indices of the PyTorch Geometric matrix representation. This mapping is called by the agent every time it receives an observation from the environment, and it is used to convert the SUMO data into the required format. It is again called by the environment when it receives an action from the agent, converting the action chosen by the policy network into the format expected by SUMO.

Conclusions

In this chapter, we detailed the implementation of our proposed CA-TSP solution. We described how to generate input data for the simulator and how to integrate them in the environment to model traffic dynamics, in turn, generating the environment states, which are then used by the agent to decide its next action. We also illustrated the design of the DNNs, explaining how the state of the environment is processed to create the next action by the policy and a prediction on the expected return by the value network.

This implementation adhere to the standard agent-environment framework of DRL approaches, allowing for the usage of tools that highly simplify the training process and the scalability of our solution.

In the next chapter, we present the experimental setup, the benchmark we used for comparison, and report the results obtained from extensive testing of the implemented system.

Acknowledgments

Map data copyrighted OpenStreetMap contributors and available from
https://www.openstreetmap.org.

References

Ansel, J., E. Yang, N. Gimelshein, N. Gimelshein, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala (2024). "PyTorch 2.0: Our next generation release that brings compiler support to PyTorch". In: *arXiv preprint arXiv:2303.08774*.

Brody, S., U. Alon, and E. Yahav (2021). "How attentive are graph attention networks?" In: *arXiv preprint arXiv:2105.14491*.

Fey, M., J. Sunil, A. Nitta, R. Puri, M. Shah, B. z. Stojanovi c, R. Bendias, A. Barghi, V. Kocijan, Z. Zhang, X. He, J. E. Lenssen, and J. Leskovec (2025). "PyG 2.0: Scalable Learning on Real World Graphs". In: *Temporal Graph Learning Workshop @ KDD*.

Kool, W., H. Van Hoof, and M. Welling (2019). "Attention, Learn to Solve Routing Problems!" In: *International Conference on Learning Representations*.

Lopez, P. A., M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner (2018). "SUMO-Simulation of Urban Mobility: An Overview". In: *Proceedings of the 21st SUMO Conference*. EPiC Series in Engineering, pp. 1–13.

OpenStreetMap contributors (2017). *Planet dump retrieved from https://planet.osm.org*. `https://www.openstreetmap.org`.

Towers, M., J. K. Terry, A. Kwiatkowski, J. U. Balis, G. d. Cola, T. Deleu, M. Goulão, A. Kallinteris, A. KG, M. Krimmel, R. Perez-Vicente, A. Pierré, S. Schulhoff, J. J. Tai, A. T. J. Shen, and O. G. Younis (2023). *Gymnasium*.

Chapter 6

Numerical results

Introduction

This chapter presents the empirical validation of the Neural Combinatorial Optimization (NCO) agent proposed in Chapter 5. We evaluate its performance in the Context-Aware Traveling Salesman Problem (CA-TSP) environment, aiming to answer two key questions:

1. Can the agent learn to leverage environmental features to make anticipatory decisions that outperform myopic strategies?

2. How does the agent performance scale in a complex, realistic simulation compared to heuristic and optimization-based benchmarks?

The chapter is structured as follows. We first describe the experimental setup, detailing the map, traffic generation process, and key environment parameters. Next, we specify the agent Deep Neural Network (DNN) and the Proximal Policy Optimization (PPO) hyperparameters used.

The analysis of our results begins with a preliminary, small-scale experiment designed to isolate and demonstrate the agent ability to anticipate future conditions based on contextual features. We then move to the core of our evaluation, where we compare the agent performance against several benchmarks in the full SUMO-based environment. The chapter concludes with a discussion of our findings and their implications for using NCO in dynamic routing problems.

The code for reproducing this experiments and allowing practitioners to test new algorithms against a CA-TSP environment will be released on GitHub together with the publication of the manuscript.

6.1 Environment details

This section details the experimental setup used to train and evaluate our RL agent. As outlined in Chapter 5, the environment is built upon the SUMO traffic simulator to model the Context-Aware Traveling Salesman Problem (CA-TSP) within a realistic urban setting where traffic levels depend on the time of day, the day of the week and the weather, that defines the context in which the salesman operates. We specify the key components of this setup: the map chosen for the experiments, the process for generating time-varying traffic, and the parameters defining the problem instances.

Figure 6.1: Map of the area selected in Milan, Italy after cleaning.

6.1.1 Map chosen

For our experiments, we selected a suburban area of Milan, Italy, bounded by the coordinates (45.427647, 9.159004) and (45.485226, 9.249268). This region was chosen for its complex road network topology, featuring a mix of residential streets, main roads, and highways, which provides a realistic testbed for the agent routing capabilities. In contrast, the city center was avoided due to its numerous pedestrian zones, which would constrain routing options.

After processing the map data to remove irrelevant roads and ensure network connectivity, the final road network consists of 566 nodes (intersections) and 1104 edges (road segments), as shown in Figure 6.1. The map covers an area of approximately 3.0 km x 1.7 km, a size chosen to balance realistic complexity with the computational constraints of the SUMO simulation and agent training.

6.1.2 Traffic generation

Traffic is generated by creating 6 Traffic Assignment Zones (TAZs) within the selected area, as shown in Figure 6.2. The size of each TAZ is 1600x1600 meters. This size was chosen after testing different configurations to produce significant traffic during peak hours without causing gridlock (i.e., a situation in which traffic is unable to move freely around some junctions).

Each of the 6 TAZs has a traffic relation with every TAZ, including itself, resulting in 36 relations. The number of vehicles generated for each relation is determined by the scenario:

- **Sunny Workday**: The base number of vehicles is 350 for each relation.
- **Rainy Workday**: The number of vehicles is increased by 10% to 385 to simulate higher traffic volumes due to adverse weather.
- **Sunny Weekend**: The number of vehicles is decreased by 20% to 280 to reflect lighter weekend traffic.

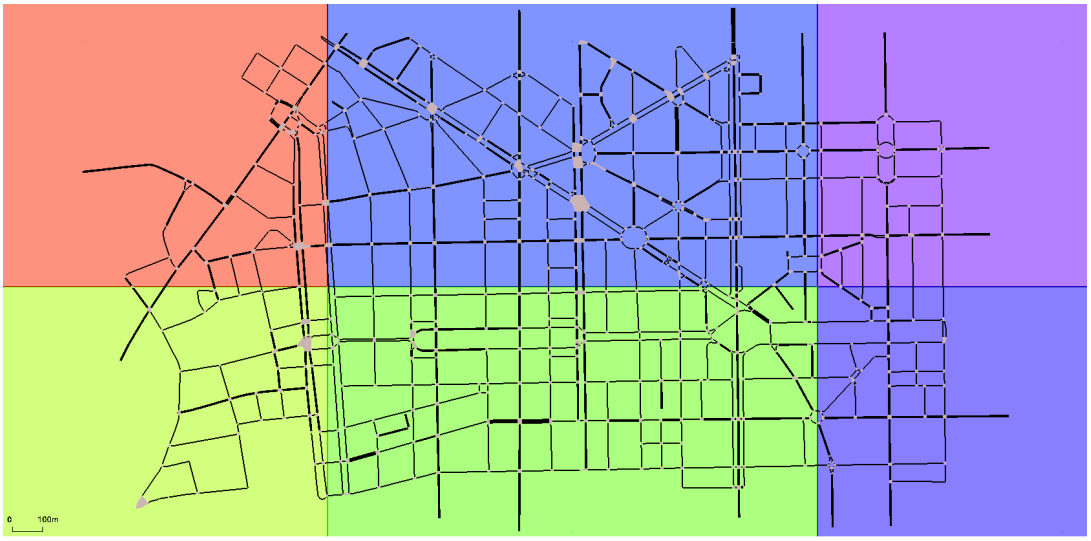


Figure 6.2: Map of the area selected in Milan, Italy with the 6 TAZs used for traffic generation.

- **Rainy Weekend**: The number of vehicles is first increased by 10% for the weekend effect and then decreased by 20% for the rain effect, resulting in 308 vehicles per relation ($350 \times 0.8 \times 1.1$).

To model localized traffic patterns, certain "hot" areas were defined. On workdays, traffic is intensified on the eastern portion of the map by doubling the number of vehicles for the mutual relations between the two easternmost TAZs. Conversely, on weekends, traffic is intensified on the west side by doubling the vehicle count for the mutual relations between the two westernmost TAZs.

From these TAZ relations, individual trips are generated. The total number of trips is determined by the scenario (workday or weekend), while their temporal distribution is dictated by the time of day. Specifically, we define an hourly percentage of the total trips to be generated.

These trips serve as inputs to generate routes—that is done by computing the fastest path between the origin and destination of each trip using Dijkstra algorithm. The total number of vehicles for each scenario is:

- Sunny workday: 14000 vehicles
- Rainy workday: 15400 vehicles
- Sunny weekend: 11200 vehicles
- Rainy weekend: 12360 vehicles

with the resulting traffic distribution as

Additionally, scenario-specific driving behavior parameters, such as maximum speed, acceleration, and deceleration, are configured for the vehicles using SUMO parameters.

Figure 6.4 illustrates the spatial distribution of traffic at different times of the day for a given scenario, while Figure 6.5 compares traffic patterns across different scenarios at the same time of day. Each edge is colored based on the average delay experienced by vehicles, defined as the time lost due to traveling below their desired speed. Green means

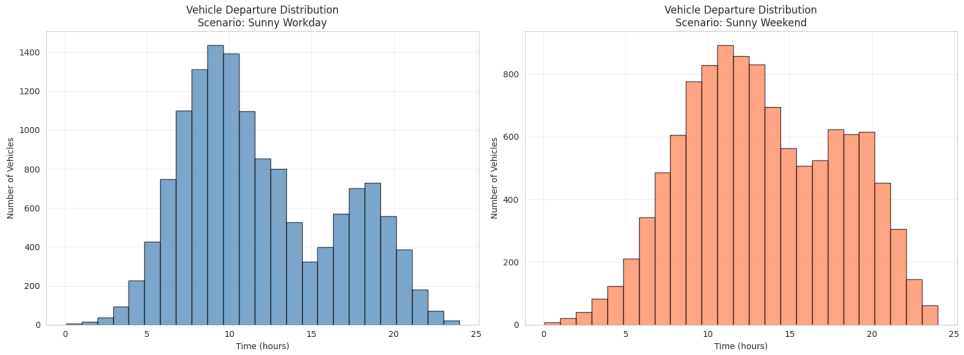


Figure 6.3: Number of vehicles entering the simulation for each hour interval.

Table 6.1: Simulation parameters for weather conditions.

Parameter	**Sunny**	**Rainy**	**Explanation**
sigma	0.5	0.6	Noise/randomness in driver behavior.
accel	2.6	1.5	Maximum acceleration rate of vehicles.
decel	4.5	2.5	Maximum deceleration rate of vehicles.
tau	1.0	1.4	Driver reaction time.
speed_factor	1.0	0.8	Factor scaling the desired speed of vehicles.
speed_dev	0.1	0.2	Standard deviation of the speed variation.
min_gap	2.5	3.0	Minimum gap (distance) maintained between vehicles.

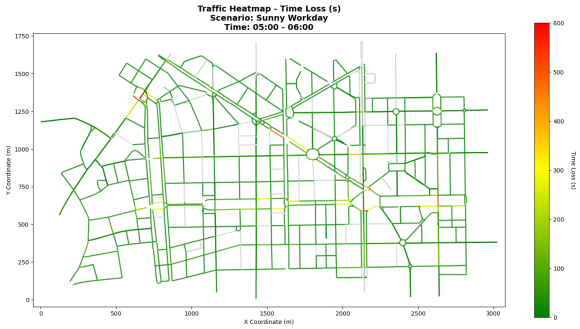
a low value for lost time while red means that the time lost was 10 minutes or more. These heatmaps reveal distinct traffic dynamics, with congestion patterns shifting in response to the time of day and other scenario conditions. Moreover, the global number of vehicles in the simulation at each instant, including the mean and peak number of vehicles in the network, are shown in Figure 6.6. From this image, we can see that the shape of the plot resembles the plots in Figure 6.3, but slightly shifted to the right as entering vehicles sum to vehicles already in the simulation. Moreover, the plot is very spiky, as the exact insertion and removal time for each vehicle depends on the simulation logic.

These plots show that the traffic generation process produces realistic and time-varying patterns, that differ across scenarios. It is important to note that all vehicle routes are precomputed before the simulation begins and remain fixed throughout. This means that vehicles entering the simulation cannot adapt their route to the traffic, as this would significantly slow down the simulation. At the same time, this is not a realistic behavior—drivers adapt their route to the traffic they face.

Moreover, while this methodology provides a reasonable approximation of real-world traffic dynamics, it does not account for non-recurrent events such as accidents or roadworks.

6.1.3 Environment parameters

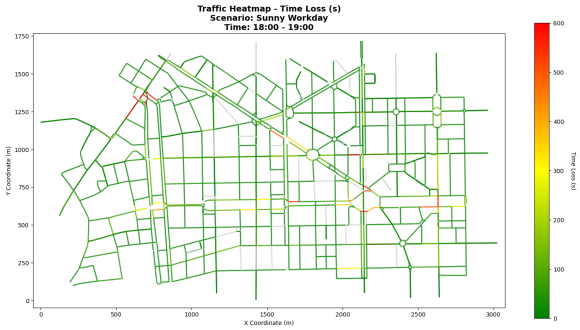
Each problem instance consists of 10 nodes: a depot and 9 customers. At the start of each episode, nodes are randomly spawned on the road network. The placement process for each node begins by uniformly sampling an edge (i, j) from the network. To prevent nodes from being located too close to junctions, a position is then sampled from a uniform distribution over the central 80% of the edge, specifically within the interval $[0.1 \cdot l_{ij}, 0.9 \cdot l_{ij}]$, where l_{ij} is the edge length, as we noticed that spawning them near



(a) Traffic at 5:00

(b) Traffic at 9:00

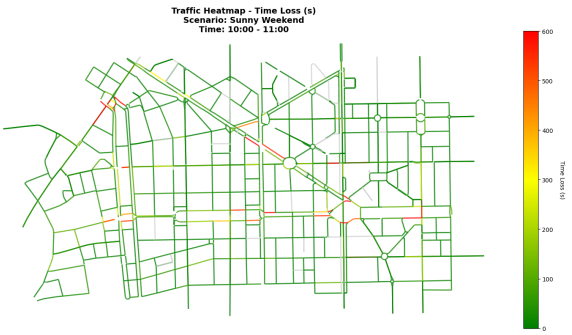
(c) Traffic at 13:00



(d) Traffic at 18:00

Figure 6.4: Heatmaps showing the spatial distribution of traffic at different times of the day for a given scenario.

(a) Traffic at 10:00 in a Sunny Workday

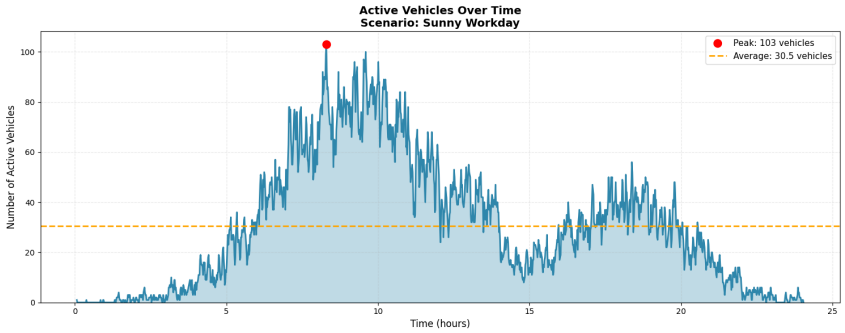


(b) Traffic at 10:00 in a Sunny Weekend

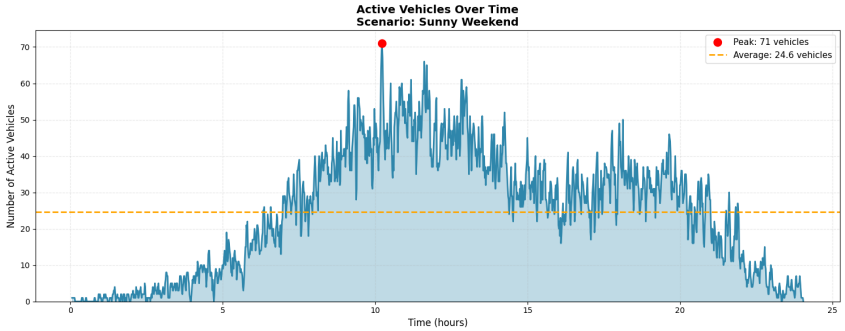
(c) Traffic at 10:00 in a Rainy Workday

(d) Traffic at 10:00 in a Rainy Weekend

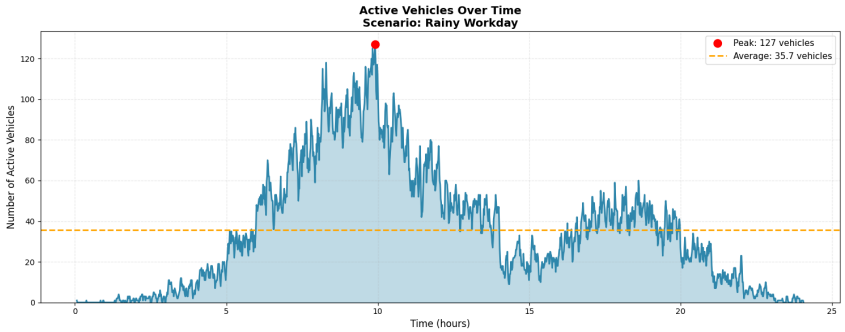
Figure 6.5: Heatmaps comparing traffic patterns across different scenarios at the same time of day.



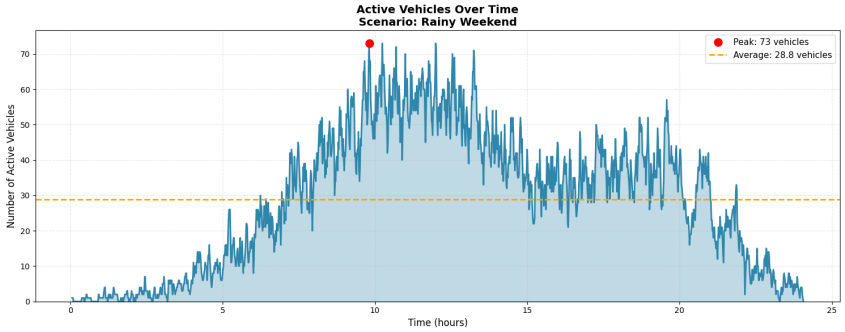
(a) Sunny Workday



(b) Sunny Weekend



(c) Rainy Workday



(d) Rainy Weekend

Figure 6.6: Number of vehicles per hour of the day for each scenario: sunny workday, sunny weekend, rainy workday, and rainy weekend.

Figure 6.7: Example of an episode with 9 customers to visit (red dots). The agent starts at the blue dot and must visit all customers and return to the starting point.

junctions could cause gridlocks. The depot is selected randomly from the 10 nodes, with the remaining 9 serving as customers. Nodes are added as Points of Interest in the SUMO simulation, allowing the agent to navigate to them, as shown in Figure 6.7, where the positions of the customers is denoted by red dots and the depot by a blue dot.

When the agent reaches a customer, it remains stationary for 20 seconds to simulate service time, before proceeding to the next destination. Each simulation run starts at 00:00 and ends when the salesman returns to the depot. The agent is introduced into the simulation at a random time between 05:00 and 19:00 to ensure it operates during periods with meaningful traffic. To establish a realistic traffic state, the simulation always runs starting from 00:00, allowing traffic to evolve naturally. We observed that truncating this warm-up period by starting the simulation closer to the insertion time of the salesman (e.g., starting the simulation 4 hours before the insertion time) resulted in significantly different and less realistic traffic patterns.

Table 6.2: Recap of the environment settings and SUMO configuration.

Parameter	**Value / Description**
Number of nodes	10
Reward function	Negative of the time to reach a customer (in minutes)
Insertion time	Random value $\in [5, 19]$
Stop duration	20 seconds
Min. path algorithm	Dijkstra
Scenarios	
`is_raining`	Sets the rain parameters (see Table 6.6)
`is_weekend`	Sets the weekend parameters (see Table 6.6)
SUMO Configuration	
`processing.ignore-route-errors`	`true`
`processing.tls.actuated.jam-threshold`	`30`
Other parameters	Default

6.2 Agent details

This section details the agent implementation. We describe the architecture of the DNNs, the hyperparameters for the PPO algorithm, and key implementation choices. Furthermore, we outline the training procedure and the computational resources used.

6.2.1 Neural network architecture

The DNN architecture follows the design outlined in Chapter 5. For our experiments, the embedding size for all layers is set to 128. The customer encoder consists of a single GNN message passing layer, while the city encoder uses three GNN message passing layers. The policy decoder is equipped with a Multi-Head Attention (MHA) with 4 heads. The value decoder also uses a MHA with 4 heads, followed by a final MLP to produce the value estimate. The customer and city encoders are shared between the policy and value networks to promote learning common representations. The LeakyReLU activation function is used throughout the networks. To enhance training stability and prevent overfitting, we incorporate several standard techniques. LayerNorm and skip connections are applied within both the GNN and MHA blocks. Additionally, a dropout rate of 0.2 is used in these same blocks.

6.2.2 PPO parameters

The training process leverages 8 parallel environments for data collection. In each iteration, every environment gathers 3 full episodes, resulting in a total of 24 episodes. With 9 decisions per episode (for 9 customers), this amounts to 216 transitions collected per iteration. These transitions are stored in a buffer and used for the network updates. The update phase consists of 8 epochs, where the collected data is processed in mini-batches of 32 samples. The total training procedure runs for 4000 iterations.

Key PPO hyperparameters, described in Section 4.4.1, are set as follows. The clipping parameter ϵ for the surrogate objective is set to 0.2. To prevent exploding gradients, we apply gradient clipping with a maximum norm of 0.5. The final loss function combines the policy loss with a value function loss and an entropy bonus, weighted by coefficients $C_V = 0.1$ and $C_E = 0.1$, respectively. Advantage estimation is performed using Generalized Advantage Estimation (GAE) with $\gamma = 0.999$ and $\lambda = 0.95$. Advantages are normalized at the mini-batch level.

For optimization, we use the AdamW optimizer (Loshchilov et al., 2017) with a learning rate of 10^{-3}, $\beta_1 = 0.9$, $\beta_2 = 0.999$, and a weight decay of 10^{-2}. For the interested reader, these parameters can be found in the original paper of the AdamW optimizer.

6.3 Experimental results

In this section, we present our experimental results. The goal is to assess whether the proposed NCO agent can leverage contextual and temporal information to improve route decisions under dynamic conditions. First, we introduce the benchmarks used to compare our agent. Before proceeding with the main experiment with the SUMO simulator, we report a preliminary experiment based on the example from Chapter 4, where the cost of traveling from B to C increases after the first step on workdays. This test illustrates

that the agent learns to exploit contextual features to anticipate future costs and adapt its strategy. For this experiment, we use slightly different hyperparameters, which are detailed later. We then present the findings on the main experiment.

6.3.1 Benchmarks

To evaluate our agent, we compare it against three benchmarks.

The first baseline, which we call *Rolling Greedy*, selects at each decision step the nearest unvisited customer using the current expected travel times and recomputes this choice after every visit. As PPO agent, this benchmark sees the current state of the traffic at each step, but it does not use any features to anticipate future conditions. The second baseline, *Greedy 2-Opt*, takes a snapshot of travel times at the moment the salesman is inserted into the simulation, builds a greedy tour, and then applies 2-Opt to improve it by iteratively re-arranging segments to check if this shortens the tour. This improvement is performed only once at the beginning of the episode and the solution is not updated thereafter. The number of 2-Opt iterations is fixed to 100. The third one, that we call *Snapshot-MILP* solves the static TSP obtained by taking a snapshot of travel times at insertion using a Mixed Integer Linear Programming (MILP) formulation and the exact solver CBC (see Forrest et al., 2024). Then, it keeps that tour fixed for the entire episode. Similarly, we define *Freeflow-MILP*, a benchmark that just like Snapshot-MILP solves a MILP formulation with an exact solver, but as data uses the travel times as if there is no traffic, namely the free-flow time.

While these benchmarks might seem simplistic, it is important to notice all, except for the Freeflow-MILP, have access to the expected travel time between nodes at solution construction time, and they do not reason over average travel times or other estimated measures. This is a big advantage compared to many real-world scenarios, where only historical data are available. Indeed, a method like the Snapshot-MILP in bigger dimension would quickly become infeasible to be run online and therefore use snapshot data. Only Freeflow-MILP would be feasible at that point, since it uses a static estimation of travel times.

6.3.2 Preliminary experiment

We recall the example introduced in Chapter 4. In this case there is no map, only three nodes A, B, and C with the time 0 costs shown in Figure 6.8, and A is the depot.

If it is a workday, the cost of traveling from B to C changes after the first step to 40, otherwise it stays the same throughout the episode.

When the scenario is a workday, the optimal solution is A $\rightarrow$ C $\rightarrow$ B, for a total cost of $20 + 20 + 20 = 60$. When the scenario is a weekend, the optimal solution is A $\rightarrow$ B $\rightarrow$ C, for a total cost of $10 + 10 + 20 = 40$. A myopic algorithm that ignores features and future evolution always chooses A $\rightarrow$ B $\rightarrow$ C, yielding $10 + 40 + 20 = 70$ on workdays.

All benchmark algorithms end up choosing A $\rightarrow$ B $\rightarrow$ C regardless of the scenario. Conversely, our RL agent learns to use the features to anticipate future costs and adapt its strategy. After training for 1000 iterations, the agent consistently outperforms the benchmarks, as shown in Figure 6.9.

At first the agent occasionally performs even worse than the benchmarks, but as training progresses it learns to exploit the features and converges to the optimal policy.

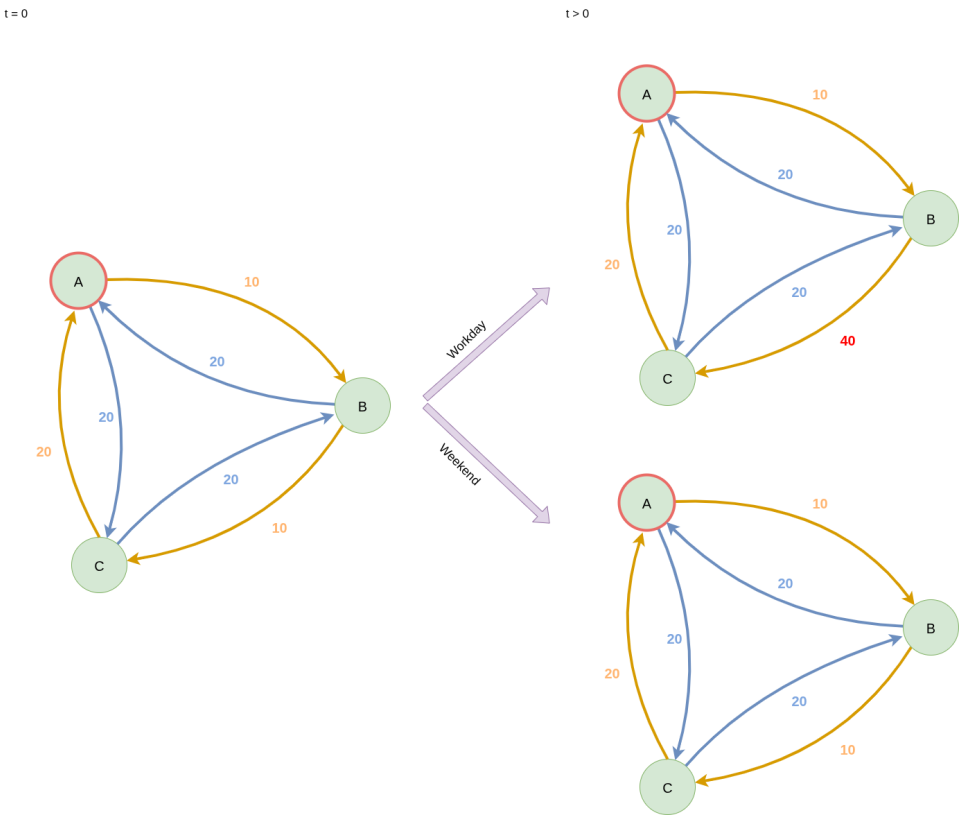


Figure 6.8: Illustration of how the preliminary example evolves.

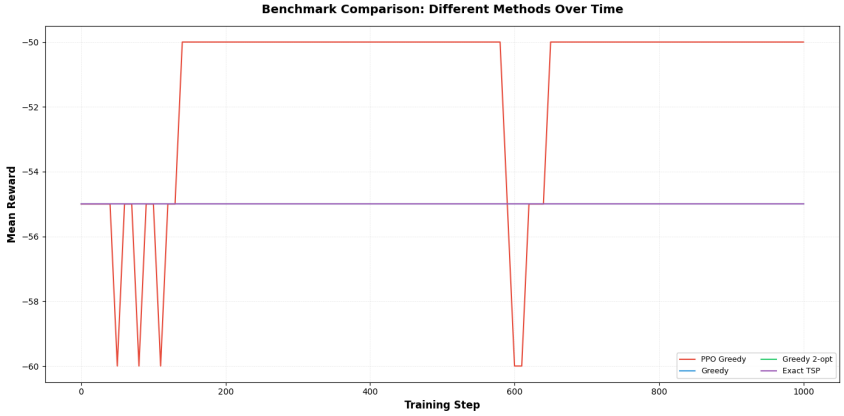


Figure 6.9: Training on the preliminary experiment.

There is a small example of *catastrophic forgetting* around iteration 600, but the agent quickly recovers and returns to the optimal policy.

Tested on two instances, one per scenario, the agent achieves the optimal solution in both cases, as reported in Table 6.3. Notice that we only included Snapshot-MILP and not Freeflow-MILP in the results, as the two solution methods are the same in this example.

As a thought experiment, consider also using an exact method that reasons on average travel times or worst case scenario optimization. Also in that case, it would still be a solution method that always take one of the two tours no matter the context, not leading to an optimal strategy.

Table 6.3: Performance comparison of different algorithms on the preliminary experiment.

Algorithm	Workday return	Weekend return
PPO	**−60**	**−40**
Rolling Greedy	−70	−40
Greedy 2-Opt	−70	−40
Snapshot-MILP	−70	−40

This controlled test confirms that the agent architecture and training setup have the capacity to learn context-dependent strategies when the environment exhibits clear non-stationary patterns. The next experiment investigates how this ability transfers to a more realistic scenario.

6.3.3 Main experiment

We now present the results of the main experiment on the selected map.

Training

For the main experiment, we trained the agent for 5000 iterations, on a server equipped with two AMD EPYC 9534 64-Core Processors, providing a total of 256 threads, an NVIDIA H100 PCIe GPU with 80 GB CUDA 12.8 and 750 GB of RAM. The training took approximately 48 hours. The training time is mostly dominated by the time taken by SUMO to run the simulations, as the DNN forward and backward passes are relatively fast due to the small size of the networks, as can be seen in Figure 6.10.

In all results, we report the performance of our PPO agent using two action selection strategies: a greedy strategy (selecting the action with the highest probability) and a sampling strategy (sampling from the learned probability distribution).

Indeed, we can see that the rollout collection phase takes between 28 and 34 seconds per iteration, while the network update takes from fractions of a second up to 4 seconds, without considering outliers.

The training curve is shown in Figure 6.11, where we can see that the agent loss decreases and the average objective increases as training progresses.

During training, for monitoring the performances of our agent, we keep track of the results it obtains on a fixed set of 10 instances never seen during training, and compare those results against Snapshot-MILP, Rolling Greedy and Greedy 2-Opt, as we can see in Figure 6.12.

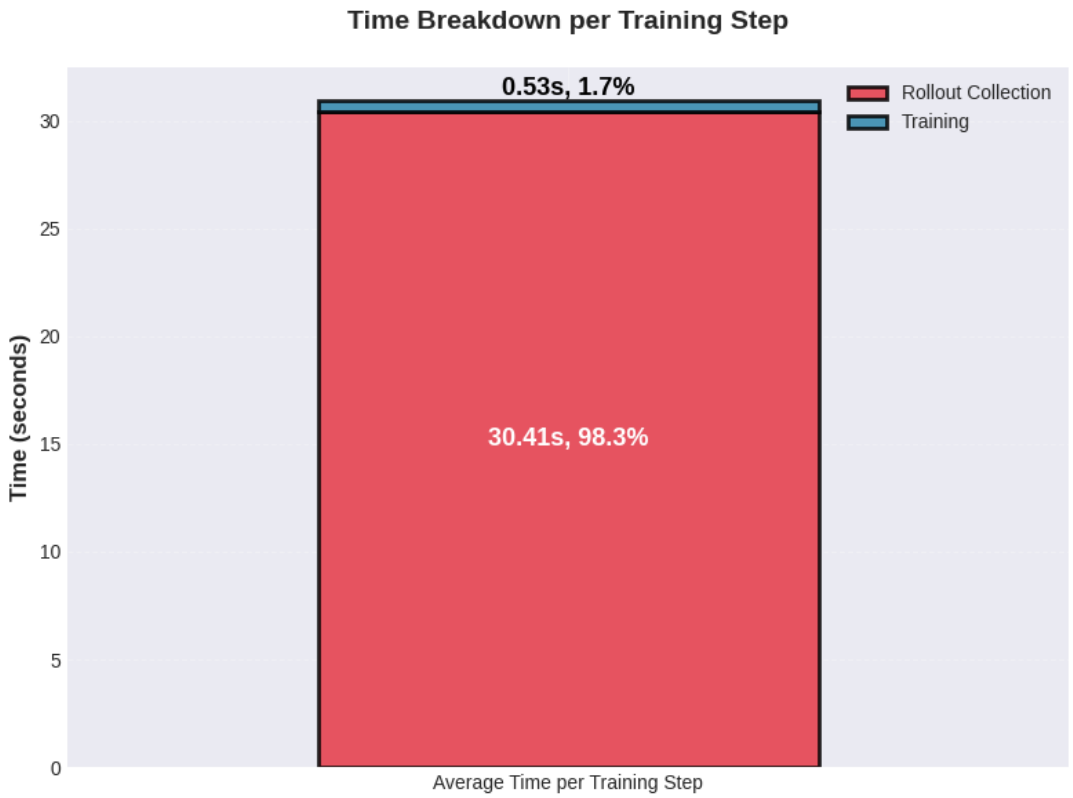
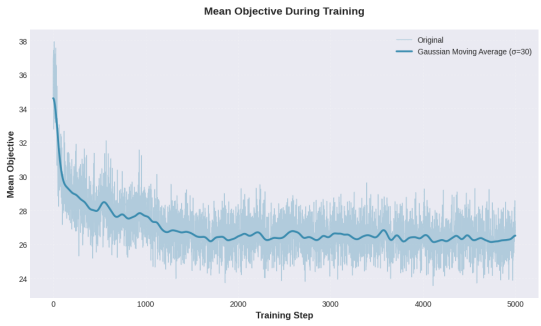


Figure 6.10: Training time breakdown per iteration.

(a) Total loss of PPO.



(b) Mean objective obtained by the agent.

Figure 6.11: Total loss and mean objective as training progress.

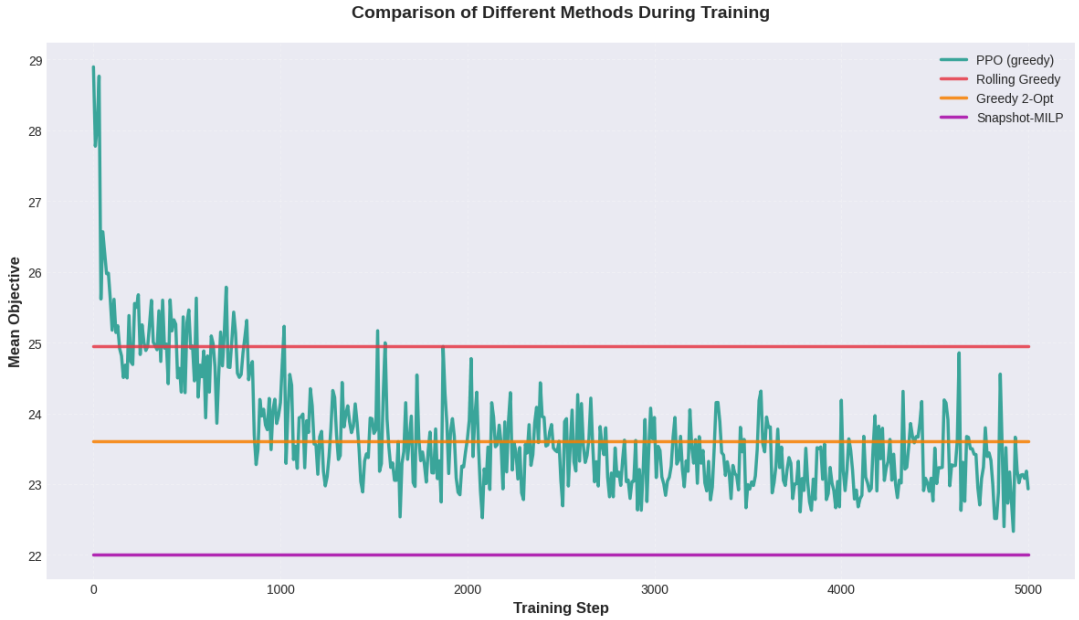


Figure 6.12: Training objective comparison.

From this plot, we can see how the agent is slowly approaching the objective of the Snapshot-MILP in this 10 instances and that, even after 48 hours it was still learning. Indeed, looking at the plot of the single loss function in Figure 6.13, we can see how the value loss quickly decreased to small values but the policy loss still has oscillations around 0.5 in magnitude and the entropy loss has not yet reached zero.

To confirm this, we can also analyze how many times the policy loss is being clipped to avoid going out of the trust region. Looking at Figure 6.14, we notice that there are still big changes in the policy, even after 5000 iterations of training.

Testing

Once training is over, we test the resulting agent on 100 test instances, again generated randomly.

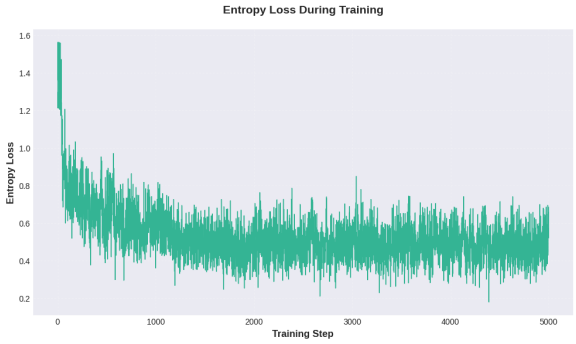
Table 6.4: Summary statistics of algorithm performance (absolute values).

	Mean	**Min**	**Max**
PPO (greedy)	26.169	52.050	17.233
PPO (sampling)	27.393	56.400	18.483
Rolling Greedy	26.883	56.817	17.883
Greedy 2-Opt	26.183	46.750	18.250
Snapshot-MILP	24.024	46.750	15.800
Freeflow-MILP	**23.615**	**46.700**	**15.600**

Results from 6.4 show that our agent outperforms both the Rolling Greedy and the Greedy 2-Opt. algorithms in a comparison between the mean objective obtained. It performs almost on par (48 wins, 49 losses and 3 ties) with the best heuristic, that is Greedy 2-Opt, when looking at the head-to-head results in Figure 6.15. An algorithm wins if the objective obtained for a given instance is lower than the one of the competing algorithm. It can be observed that PPO is the best heuristic method when compared to

(a) Value loss

(b) Policy loss



(c) Entropy loss

Figure 6.13: Training loss functions over iterations.

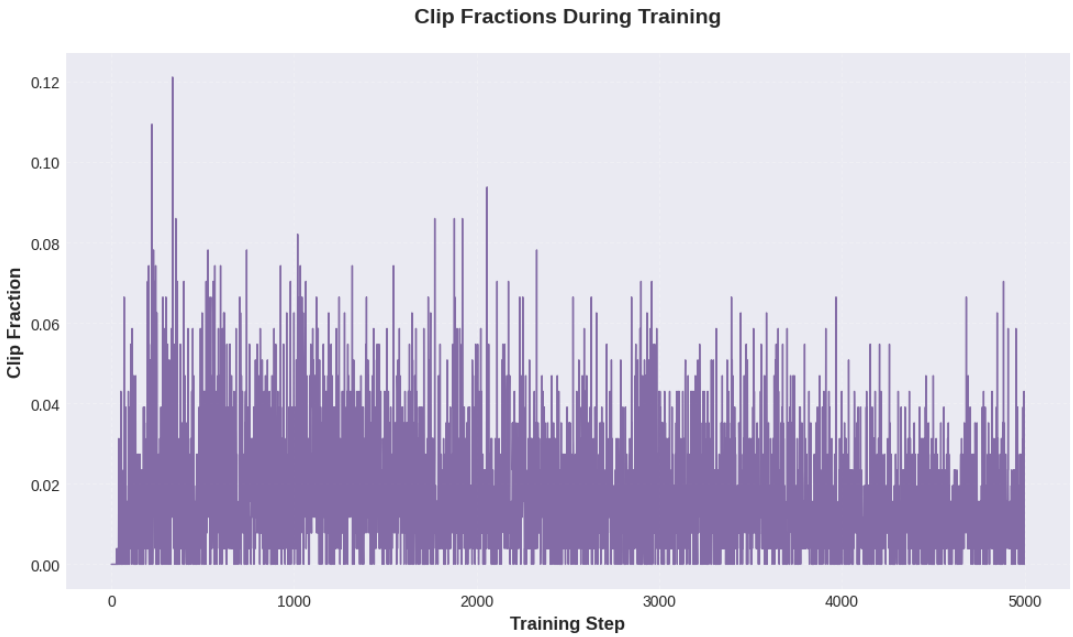


Figure 6.14: Clipping fraction during training.

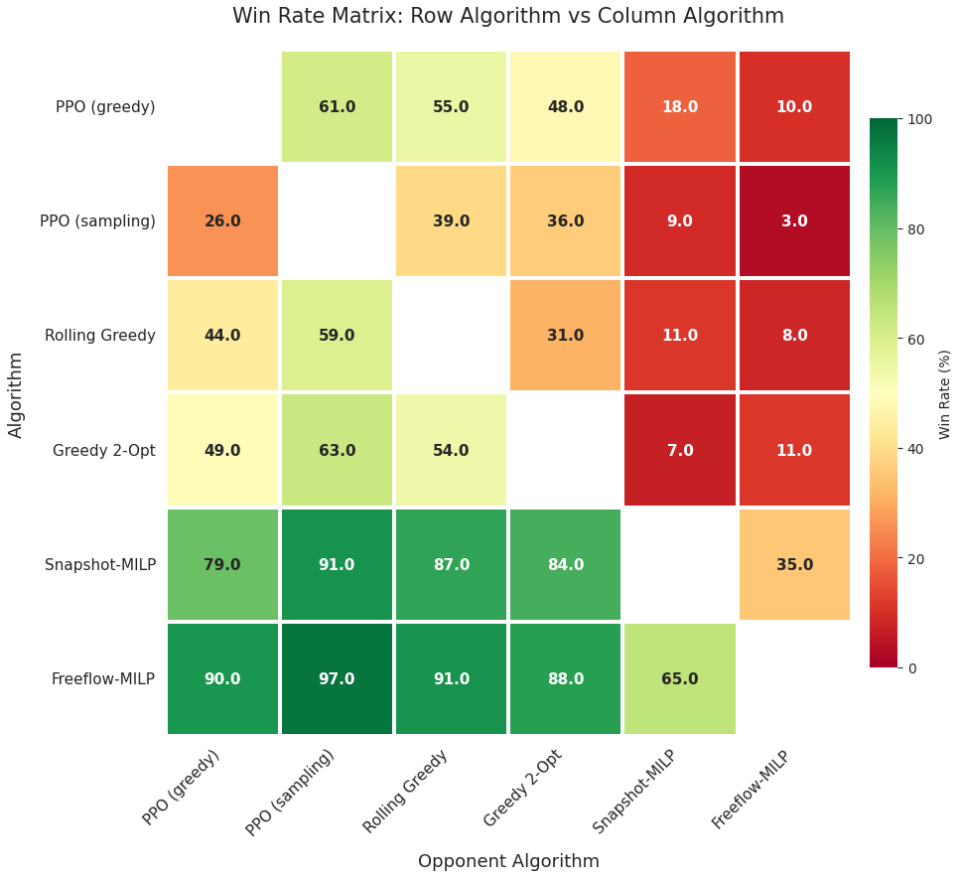


Figure 6.15: Head-to-head tournament results. Each cell shows the wins for the row algorithm against the column algorithm. If the sum of the wins of two competing algorithms is less than 100%, the remaining percentage represents ties.

the Snapshot-MILP (15 wins, 82 losses and 3 ties) but it performs slightly worse than Greedy 2-Opt if compared against Freeflow-MILP (9 wins for PPO against 11 wins for Greedy 2-Opt.) The two exact methods, however, outperform all the heuristics, including PPO.

Comparing also the time needed to reach compute the solution, we get the results shown in Figure 6.5

Table 6.5: Time breakdown by computation and simulation phases.

	Total (s)	**Computation (s)**	**Comp %**	**Simulation (s)**	**Sim %**
PPO (greedy)	26.764	0.707	2.64	**26.057**	97.36
PPO (sampling)	26.850	0.723	2.69	26.127	97.31
Rolling Greedy	**26.098**	**0.000**	0.00	26.098	100.00
Greedy 2-Opt	28.241	0.115	0.41	28.126	99.59
Snapshot-MILP	28.826	1.516	5.26	27.310	94.74
Freeflow-MILP	28.974	1.738	6.00	27.237	94.00

We can see that the Freeflow-MILP and Snapshot-MILP are the more expensive algorithm in term of time. The rolling greedy is the fastest, as it only needs to compute the nearest neighbor at each step. PPO takes approximately half of the time of the Snapshot-MILP. While the difference between PPO and Snapshot-MILP time seems small, it is important to notice that the MILP solution time grows exponentially with the number of nodes, while PPO time grows linearly.

Finally, we analyze how the different algorithms perform based on the different context features. From Table 6.6, we can see that there is not a big difference in terms of performances between different scenario, as the gap from the best known solution is similar in all cases. Interestingly, a big difference is instead visible in other algorithms, especially Snapshot-MILP. Indeed, we can see that the gap is small (0.92%) on sunny weather but relatively high (2.83%) on rainy days. Similarly, on workdays the gap is high (2.58%) but relatively small on weekends (1.07%). This is unexpected, as we have seen that rainy days and workdays have a higher traffic volume with more traffic spikes, and we would expect the performances of an algorithm that can not access the traffic level to perform worse than one that can see it (even if just when entering the simulation).

Table 6.6: Performance analysis by weather conditions and day type in average minutes; the relative gap to the best performing algorithm (Freeflow-MILP) is reported in parentheses.

	Weather		**Day Type**	
	Clear (53 instances)	**Raining (47 instances)**	**Workday (46 instances)**	**Weekend (54 instances)**
PPO (greedy)	22.95 (+11.05%)	29.79 (+10.62%)	25.12 (+11.46%)	27.06 (+10.31%)
PPO (sampling)	24.37 (+17.88%)	30.80 (+14.37%)	26.34 (+16.84%)	28.29 (+15.34%)
Rolling Greedy	23.30 (+12.70%)	30.93 (+14.82%)	25.77 (+14.33%)	27.83 (+13.46%)
Greedy 2-Opt	22.97 (+11.14%)	29.80 (+10.64%)	25.04 (+11.07%)	27.16 (+10.73%)
Snapshot-MILP	20.86 (+0.92%)	27.59 (+2.43%)	23.12 (+2.58%)	24.79 (+1.07%)
Freeflow-MILP	**20.67**	**26.93**	**22.54**	**24.53**

6.4 Analysis

The preliminary example verified that the PPO is able to exploit the context to predict the future evolution of traffic, and use that information to learn the optimal policy. In that setting, feature-based reasoning is necessary and clearly beneficial. Conversely, an algorithm that cannot use context, is not able to find the optimal solution, even if it is a dynamic algorithm, as the case of the rolling greedy.

When training and testing the agent in the simulation, we notice that it is able to learn how to produce good results, outperforming both the greedy with 2-Opt and the rolling greedy, but not the Snapshot-MILP. This shows that, while our agent is able to adapt to changing traffic conditions and make effective routing decisions, as showed in the preliminary experiment, the traffic simulator produced only minor fluctuations over the 20-30 minute horizon of the salesman route. As a result, the dynamic component of the problem remained weak, and static methods such as the Snapshot-MILP and Freeflow-MILP retained the best performances. Since PPO can not reach the same performance of MILP based approaches, we can infer that the agent is not able to learn to produce solutions that are as good as the optimal ones in a static scenario. This is also confirmed by the literature, where NCO is usually used to solve problems in big dimensions, where MILP cannot be used; but for small instances, NCO still has a relevant gap to optimal solutions. When we tried to increase the traffic in the simulation, instead, we encountered gridlock problems and high instability in training due to unrealistic traffic patterns caused by zones of the map with very high traffic that would take hours to resolve. For these reasons we are willing to test our NCO agent with more customers and with a bigger map in which traffic can spread more easily, but this will require extra optimization to reduce the simulation time or algorithms more sample efficient than PPO, like model based methods.

The substantial difference between PPO (greedy) and PPO (sampling) can be interpreted as further evidence that training was not yet complete. Indeed, when an agent has learned to act optimally within an environment, its policy typically becomes nearly deterministic, concentrating most of the probability mass on the best-performing action. This observation, together with the indicators discussed in Section 6.3.3, suggests that even after 48 hours, the training process was still ongoing. This outcome has both a positive implication—-that extended training might yield a stronger agent—-and a negative one, as the algorithm failed to achieve convergence within such a long training period.

To improve the efficiency of the training, we have also tried to increase the number of Environments in parallel and increase the minibatch size accordingly. We noticed that this lead the value network to overfit to the early data collected, even if we reduced its learning rate, modified its architecture or played with the entropy regularization factor V_E. This resulted in a poor training signal for the policy network, that was not able to learn a good policy.

A surprising result was that the Freeflow-MILP outperformed the Snapshot-MILP, indicating that the optimal solution obtained without considering traffic performed better than the one based on current traffic levels. This might be explained by the fact that the traffic level is generally low and almost negligible in most cases, effectively acting as noise in the Snapshot-MILP data. This result highlights one of the main challenges of this work: generating meaningful traffic patterns while keeping the simulation time within reasonable limits.

The present results, thus, show that while an NCO agent can exploit dynamic traffic

patterns, the benefits are limited in scenarios with minimal traffic variation. Future work could increase temporal variance (e.g., longer tours or simulated peak-hour traffic) or exploit curriculum strategies to expose the agent to richer temporal patterns.

Conclusions

In conclusion, we have shown how to implement an NCO agent for the Context-Aware Traveling Salesman Problem using a realistic traffic simulator. The agent architecture effectively integrates contextual features to inform routing decisions. In a controlled preliminary experiment, the agent successfully learned to anticipate future traffic conditions based on contextual cues, outperforming myopic benchmarks. However, in the main experiment with the SUMO simulator, while the agent outperformed heuristic baselines, it fell short of matching the performance of a static MILP solutions. This outcome highlights the challenges of leveraging dynamic traffic patterns when such variations are limited within the simulation.

In future work, we plan to extend this study in several directions. First, we aim to experiment with larger maps and a greater number of customers to increase the problem dynamic complexity and better assess the agent ability to adapt to evolving traffic conditions. To do so, we need to optimize the rollout collection phase to accelerate training and explore more sample-efficient RL algorithms or model-based approaches to improve convergence. To help with convergence, we will also investigate curriculum learning strategies to progressively expose the agent to increasingly complex traffic patterns, as well as pretraining techniques using synthetic data or simplified environments before fine-tuning in the full SUMO simulation. Finally, we plan to analyze the learned policies to understand how contextual features influence decision-making and conduct ablation studies to isolate the contributions of different architectural components and training strategies.

References

Forrest, J., T. Ralphs, S. Vigerske, H. G. Santos, J. Forrest, L. Hafer, B. Kristjansson, jpfasano, EdwinStraver, Jan-Willem, M. Lubin, rlougee, a-andre, jpgoncal1, S. Brito, h-i-gassmann, Cristina, M. Saltzman, tosttost, B. Pitrus, F. MATSUSHIMA, P. Vossler, R. @. SWGY, and to-st (Aug. 2024). *coin-or/Cbc: Release releases/2.10.12.* Version releases/2.10.12.

Loshchilov, I. and F. Hutter (2017). "Decoupled weight decay regularization". In: *arXiv preprint arXiv:1711.05101.*

Chapter 7

Conclusions

This thesis explored the use of Neural Combinatorial Optimization (NCO) for solving Combinatorial Optimization (CO) problems. In particular, it examined how NCO can be applied to static settings and then extended to dynamic environments that closely model real routing problems.

Several types of dynamic routing problems have been studied in the literature, depending on the assumptions made about the problem and its uncertain elements. Since the Traveling Salesman Problem (TSP) lies at the foundation of most routing problems, this work focused on its dynamic formulation. To model a TSP in which travel times between pairs of points are uncertain and influenced by contextual factors, we proposed a new dynamic formulation, the Context-Aware Traveling Salesman Problem (CA-TSP). The CA-TSP formulation captures key aspects of real-world scenarios while making minimal assumptions about how the dynamics of an instance evolve over time.

We created an NCO agent capable of reasoning about context-dependent environments and integrated it into a traffic simulation to introduce realism. In this environment, traffic evolution depends not only on time but also on contextual factors such as weather conditions and day type (workday or weekend). Using a toy example, we showed that the CA-TSP cannot be solved optimally by classical methods, whereas an NCO agent successfully learned the optimal strategy. We then used the traffic-simulation-based environment to evaluate the approach on a set of randomly generated instances and compared its performance with both heuristic and exact methods.

Although the challenges of building a simulation that is at once realistic and computationally efficient limited our results, the experiments provided valuable insights into the practical trade-offs involved in bringing NCO closer to real-world settings. The findings underscored the open challenges that remain when moving from synthetic data to real-world applications. In practical scenarios, limitations such as restricted data availability, high simulation costs, and poor sample efficiency can severely constrain scalability. Overcoming these challenges is crucial for making NCO methods viable in real-world contexts and for addressing one of the persistent pitfalls of Reinforcement Learning (RL) approaches——their limited adaptability when the training data are not representative of reality.

At the same time, the findings of this study open several promising directions for future research. These include improving the sample efficiency of learning algorithms (i.e., how quickly they learn under scarce data), developing more realistic yet computationally tractable simulation environments, leveraging historical or expert data to guide early training phases, and exploring neural architectures that can better capture the complexity

of dynamic optimization problems.

In conclusion, this thesis does not mark an endpoint, but the beginning of a more in-depth study into an emerging class of solution methods that, in our view, can greatly expand the toolkit available to Operations Research specialists for tackling complex, real-world problems.