Master in Business Analytics and Data Science
Department of Business Administration

Thesis Title

Combinatorial Optimization using Machine Learning

By
Vasileios Kostakis Kassandros

A thesis submitted for the degree of MSc in
Business Analytics and Data Science

December 2022

Acknoledgments

I feel the need to express how grateful I am to my supervisor Konstantinos Kaparis for accepting my request to supervise my thesis and then proposing to work on this beautiful and challenging research area. Moreover, his directions and help were vital for the completion of my master's studies. I also wholeheartedly thank my dear friend Nikolas Feto with whom I had endless conversations regarding my thesis subject which led to many improvements in the thesis content.

ABSTRACT

Combinatorial optimization is a subfield of mathematical optimization that contains several hard problems with numerous real-life applications. The traditional way of solving combinatorial optimization problems relies on decisions taken based on expert knowledge and expert-designed heuristics. In recent years, a promising research line has brought an alternative way to light. That way to automate decision-making for combinatorial optimization using machine learning. In this thesis, we provide general information on this research line but focus more on direct ways of leveraging machine learning to solve combinatorial optimization problems. Moreover, we create a reinforcement learning framework that learns a greedy constructive heuristic for the following graph combinatorial optimization problems: minimum vertex cover, maximum independent set and travelling salesman. We discuss all the necessary background in mathematics and machine learning in order to make this work an independent study.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

In optimization problems, the objective is to find an optimal solution. A solution to an optimization problem can be a real number, a permutation of objects, or even a graph. These problems are either continuous optimization problems or discrete optimization problems. Continuous optimization problems are formulated using variables from an interval of real numbers, whereas discrete optimization problems are formulated using integer variables. Both categories of optimization problems find application in industrial fields such as supply chain and logistics. In this thesis we discuss discrete optimization problems. In particular we focus on combinatorial optimization problems, a subcategory of discrete optimization problems. In these problems the objective is to find an optimal combination of objects within a finite collection of objects. Unfortunately, such optimization problems are hard to solve in provable optimality, both in theory and practice, and belong among the "21 Karp's NP-hard problems"

A great deal of research is devoted to the construction of methods to solve these problems. These methods fall into exact methods, approximation algorithms, and heuristics. Each method class has its advantages and weaknesses. When the problem scale is large, exact methods become inefficient. Approximation algorithms and heuristics have shortcomings as well. Usually, they are case specific - designed and not adaptable to multiple problems. In other words, one method for solving a specific combinatorial optimization problem may be of no use for solving another. Moreover, heuristics might not offer any quality guarantees for the obtained solution. Despite these shortcomings, heuristics are of great importance to optimization experts.

In order to devise algorithms that are adaptable to solve multiple combinatorial optimization problems, leveraging tools from an alternative scientific area seems to be a promising way. This area is machine learning and it has been successfully applied in tackling combinatorial optimization problems. In 1985, Hopfield [Hopfield and Tank, 1985] was the first to use an artificial neural network to solve the travelling salesman problem, a well-known combinatorial optimization problem. However, this line of research became active in recent years and now has various applications ranging from creating constructive greedy heuristics such as

[Khalil et al., 2017] to improving sub-routines and processes of state-of-the-art exact solvers such as [Gasse et al., 2019]. Typically, either supervised learning or reinforcement learning is used. Supervised learning requires obtaining large amounts of optima labels which is difficult when dealing with NP-hard optimization problems. For this reason, reinforcement learning, which works with the maximization of reward functions, is often preferred.

In our work, we follow the reinforcement learning paradigm and devise a framework to solve three combinatorial optimization problems. Moreover, we train an agent to learn a constructive greedy heuristic for the following problems: maximum vertex cover, maximum independent set, and travelling salesman problem. The algorithm works as follows: a neural network predicts a value of utility for each graph node that can join the partial solution, and then the node with the highest value joins the partial solution. This process terminates when the target problem is solved. We compare our algorithm with a similar RL- framework and standard heuristics like the nearest neighbour heuristic for the TSP.

Chapter 2

PREREQUISITES

This first chapter introduces some basic notions of mathematics which are necessary for comprehending the following chapters. Nonetheless, we assume some familiarity with basic terms of fields such as linear algebra, calculus, and probability theory. Presenting the basics of those fields would lead to a drift from the original goal of this work. So, what we will discuss here is restricted to a very limited list of elements of graph theory. Furthermore, an even more limited list of elements of computational complexity will also be discussed.

2.1 Graphs

Due to their nature, many CO problems have a graph representation and thus in this introductory paragraph, we outline a few fundamental graph-related notions. There is a whole field of mathematics dedicated to the study of graphs called graph theory. Graphs offer a decent way to model abstract entities and their relationships. Subsequently, graphs are quite useful to many sciences such as computer science, physics, biology and even linguistics. A Graph is a mathematical structure that is composed of edges and vertices. A more formal definition of graph is the following:

Definition 2.1 *A graph is an ordered pair G=(V,E). Where V is the set of vertices and E is the set of edges* $E = \{(x, y) : x, y \in V \text{ and } x \neq y\}$

A definition along with a drawing of a graph will give a first intuition on graphs. Usually, graphs are visualized as follows:

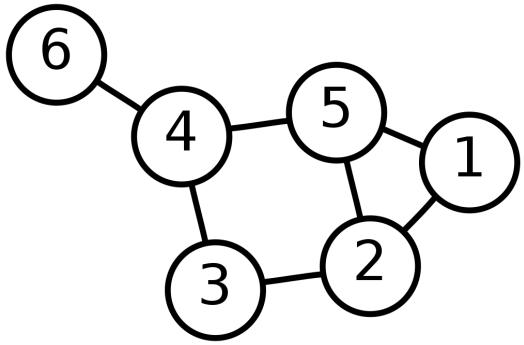

Figure 2.1: A graph from [Wikipedia, 2022a]

An edge (x, y) joins two vertices of the graph. Graphs can be distinguished into two categories: undirected graphs and directed graphs. The difference between those two categories of graphs lies on the edges: Undirected graphs have symmetric edges (that is, $(u, v) = (v, u)$ for all edges), and directed graphs have asymmetric edges ((u, v) and (v, u) edges are distinct). From now on, when it is not stated whether a graph is directed or undirected it is implied that it is undirected. Whenever two vertices u, v are joined by an edge $e = (u, v)$, u, v are called **adjacent** or **neighbours**.

Definition 2.2 *The set of all neighbours of a vertex u is called neighbourhood of vertex u and it is denoted* $N[u]$. *The number* $|N[u]|$ *is called degree of vertex u.*

For directed graphs, vertices have also in-degrees and out-degrees which are the number of edges that "end" on the vertex and "start" on the vertex, respectively. For example the edge $e = (u, v)$ starts from u and ends on v, in the literature u is called head and v is called tail. Every graph can contain subgraphs, a graph $G = (V, E)$ contains a subgraph $H = (V(H), E(H))$ if and only if $V(H) \subset V(G)$ *and* $E(H) \subset E(G)$. H is called an induced subgraph of G if it is a subgraph of G and $E(H) = \{(x, y) : x, y \in H(V)$ *and* $x \neq y\}$. A **complete graph** is a graph that each one of its vertices is adjacent to all other vertices. A **circuit** is a graph $G(V, E)$ with $V = \{u_1, ..., u_k\}$ *and* $E = \{e_1, ..., e_k\}$ if and only if the sequences $u_1, e_1, u_2, ...u_k, e_k, u_1$ is a closed walk. By walk we mean that each edge e_i joins the vertices before and after it in the sequence above, a closed walk means that the starting and the ending vertex coincide. The length of a circuit is the number of its edges. If a graph G contains a subgraph which is a circuit then one can say that there is a circuit in G. A spanning circuit in G is called a **Hamiltonian circuit** or alternatively a tour. If a graph G contains a Hamiltonian circuit then G

is called Hamiltonian Graph. An undirected graph is connected if, for every couple of its vertices there is a path from u to v. A path is a walk from u to v, that is a sequence $u, e_1, u_1, ..., u_k, e_{k+1}, v$. Similar to the circuit but in that case the walk is not closed. A graph is called **tree** if it does not contain any circuits and is connected. All the vertices of a tree of degree at most 1 are called **leaves**. Other popular terms for this kind of graph are **parent or predecessor** and **root**. A node a is a parent of b if there is an edge $e = (a, b)$. The root node of a tree graph is a node that has no parent. In a graph $G = (V, E)$, a **cut** is a set of edges which is produced by a set $X \subset V$ in this way $\{(u, v) : u \in X, v \in V - X\}$. The following graph related terms are important for the next chapters: A **matching** in an undirected graph G is a set of pairwise disjoint edges. A **vertex cover** in G is a set $S \subseteq V$ of vertices such that every edge of G is incident to at least one vertex in S. There are two types of graphs which are particularly useful. These graphs are the bipartite graphs and Eulerian graphs. The latter is named after the legendary mathematician Euler. Euler solved the famous problem of Königsberg bridges using those graphs. An **Eulerian walk** in a graph is a walk that uses each edge exactly once. If such a walk exists in a graph then the graph is called semi-eulerian.
A graph is called **Eulerian** if and only if all its vertices are of even degree. A graph $G = (V, E)$ is a **bipartite graph** if there are $A, B \subseteq V(G)$ with $A \cap B = \emptyset$ and $A \bigcup B = V$, $E(A) = E(B) = \emptyset$ and $E(A \bigcup B) = V(G)$.
A way to tell if a graph is bipartite is using the following theorem: *A graph is bipartite if and only if has no circuits of odd length*. There is a large number of theorems in graph theory. However, we do not include any other theorems as it would be unnecessary. Before concluding this section, we introduce random graphs, a type of graphs which facilitates running experiments on graph combinatorial optimization problems.

A **random graph** is a graph in which some of its components/properties are defined at random. These components can be the number of vertices, edges and hence connections between vertices. There are models to generate such graphs. We give some information on the so-called **Erdos-Renyi model**. This model has two distinct variants which where independently created in the same year. One variant the **G(n,M)** was created by Erdos- Renyi [RENYI, 1959] and the other variant **G(n,p)** by Gilbert [Gilbert, 1959]
In **G(n,M)** a graph with n vertices and M edges is generated completely at random. That is, the graph is drawn at random from the $\binom{\binom{n}{2}}{M}$ possible graphs that can be formed from n vertices by selecting at random M edges from $\binom{n}{2}$ possible edges.

G(n,p) generates a random graph with n vertices where each edge exists with probability p. The number of edges of a graph generated by that variant is a random variable with expected value $p\binom{n}{2}$

2.2 Complexity

The combinatorial optimization problems that we discuss in this thesis belong to the so-called NP-hard problems. That is a set of problems hard to solve in provable optimality both in theory and in practise. We briefly review a few fundamental notions for our analysis.

Definition 2.3 *Let $f, g : \mathbb{N} \to \mathbb{R}+$ be two functions. It is said that $f = O(g)$ when there are two constants $a, b > 0$ such that $f \leq ag(n) + b, \forall n \in \mathbb{N}$ and for g we say that $g = \Omega(f)$. If $g = O(f)$ and $f = O(g)$ then we say that $g = \Theta(f)$ and $f = \Theta(g)$.*

The Big O in the first part of the definition is the famous **big O notation**. When $f = O(g)$ we can say g is an asymptotic bound of f . When $g = \Theta(f)$ we can say that these two functions have the same growth rate. we can say that these two functions have the same growth rate. This notation is used in the analysis of algorithms. In the context of computational complexity, an algorithm can be defined in an abstract mathematical way. Here we define algorithms in a simple way : An algorithm is consists of inputs and well-defined instructions to be executed in steps such that for each input the algorithm produces an output. In algorithm analysis, the running time of an algorithm is quite important. The running time of an algorithm is the amount of time a computer takes to run the algorithm. Calculating the exact running time of an algorithm is not easy. So, it is common to study asymptotic time instead of the exact time. The asymptotic time is about how the running time of an algorithm changes when the input size grows.

Definition 2.4 *Let $f : \mathbb{N} \to \mathbb{R}+$ be a function and A an algorithm which takes inputs from a set B. If there are constants $a, b > 0$ such that the algorithm's computation terminates after at most a $f(size(x)) + b$ steps for all input x. Then we say that the running time of A is $O(f)$.*

Usually, an algorithm accepts as an input a list of numbers. These numbers are transformed and stored in the binary system. The input size *size(x)* of an instance

x that contains rational numbers is the number of bits needed for the binary representation. An algorithm with rational input is said to run in **polynomial** time if there is an integer k such that it runs in $O(n^k)$ time, where n is the input size, and all numbers in intermediate computations can be stored with $O(n^k)$ bits. So, the algorithm must run in $O(n^k)$ steps and $O(n^k)$ bits should be enough for storing all the numbers at all steps. Some algorithms that run in polynomial time are quicksort ($O(nlog(n))$ and bubblesort (O(n^2)). In the literature sometimes polynomial time algorithms are called efficient. Within computational complexity, an algorithm that transforms a problem into another problem is called a **reduction** algorithm. When there is an algorithm for transforming Problem P into P' then it is said P reduces to P'.

2.2.1 P vs NP

Problems that can be solved in polynomial time by a deterministic algorithm are called problems in **P**. However, most of the combinatorial optimization problems are considered computationally hard since no exact polynomial-time algorithm has been devised for solving them yet. **NP-class** is the class of all decision problems that can be solved by a non-deterministic algorithm in polynomial time. A decision problem P is **NP-complete** when it belongs to NP and any problem in NP can be reduced to P in polynomial time.

A problem is called **NP-hard** if every problem in NP can be reduced to it in polynomial time. A decision problem is called **NP-complete** if it is in NP and every other problem in NP can be reduced to it in polynomial time. When we say an optimization problem is $NP - complete$, we mean that its corresponding decision problem is NP-complete. Many interesting COPs such as **Karp's 21 NP-complete problems** belong to the class NP-complete. In [Karp, 1972] Karp uses **Cook's theorem** and shows that these 21 COPs belong to NP-complete class. That is done by showing that there is a polynomial time reduction from the **SAT** problem to each one of these 21 problems. The SAT problem (boolean satisfiability problem) was proved to be in NP- complete class (Cook's theorem) in [Cook, 1971].

Chapter 3

COMBINATORIAL OPTIMIZATION

Combinatorial Optimization (CO) is a subfield of the mathematical field of optimization. This subfield is focused on problems with discrete solution spaces. CO problems (COPs) seek to find the best combination of objects in order to optimize some objectives under some constraints. As an example, we can give the following problem: the best set of vertices (cost-wise) to traverse in order to transmit from a node u to a node v in a graph G. More examples of COPs, not necessarily defined on graphs, will be presented in this chapter. We will define COP as in [Mazyavkina et al., 2021].

Definition 3.1 *Let V be a set of elements and $f : V \mapsto R$ be a cost function. Combinatorial optimization problem aims to find an optimal value of the function f and any corresponding optimal element that achieves that optimal value on the domain V.*

The set V is finite, but its cardinal number can be so large that can make exhaustive search an impractical solution method. For example, an instance of size n of the travelling salesman problem, which is presented later, has $n!$ different possible solutions and that makes exhaustive search not a valid option for realistic instance sizes. To solve these problems, special algorithms have been developed that fall into either the category of the exact algorithms or approximate algorithms. The most known exact methods are branch and bound [Land and Doig, 1960], cutting plane [Gomory, 1960], and branch and cut. The last method is ,in a sense, a combination of the first two. All of them require solving linear relaxation problems (LP relaxation). Linear relaxation problems are just linear problems (LP) or better **Linear Programming problems**. Linear programming problems are problems involving the optimization of linear functions over a space formed from linear constraints. Linear programming is essential for the field to which this thesis belongs.

LP is proved to be in P complexity class [Karmarkar, 1984]. Interestingly enough, the most common algorithm used to solve LP problems is exponential. That algorithm is the Simplex and, though it is exponential, it is regarded as very efficient in

practice as opposed to the ellipsoid algorithm which is a polynomial algorithm and an inefficient one as stated in [Hart et al., 1987]. For more on linear programming and simplex algorithm, one can counsel [Bertsimas and Tsitsiklis, 1997].

3.1 Prototypical Combinatorial Optimization Problems (COPs)

We now discuss a few fundamental combinatorial optimization problems that are among Karp's 21 NP-complete problems and which form the basis of our analysis in chapter 7. Specifically, we present the following problems: *Knapsack Problem(KP)*, *Minimum Vertex Cover (MVC)*, *Maximum Independent set (MIS)* and *Travelling Salesman Problem (TSP)*. These four problems are studied in relevant work such as: [Khalil et al., 2017] which focuses on MVC, TSP, max cut and set cover problems but not on MIS and KP. MIS and KP are tackled using machine learning in other papers as many other COPs such as Vehicle Routing (VRP), Graph Coloring (GCP), Maximum Clique Problem (MCP) and others. One can find out what COPs have been studied in [Vesselinova et al., 2020, Mazyavkina et al., 2021, Bengio et al., 2021]. Each one of these papers is an overview of relevant research, the first is only about graph COPs while the others are more general.
We present the formulations of the problems as Integer Linear Programming problems (ILPs) along with some comments-details below. While it is common among COPs to have multiple versions, only a single version of each problem is presented here.

3.1.1 Knapsack Problem

This problem can be stated as follows: given a set of items where each of them has a weight and a value, determine the best combination of items in order to maximize the accumulative value. The knapsack has a fixed capacity and that implies a constraint regarding the total weight of the item combination. There are many variants of the problem. For brevity, we do not discuss all of them. The variant that is described below is called *0-1 knapsack problem* and for each item, a binary variable indicates whether the item is included or not in the knapsack.

Firstly, we define the parameters and the decision variables. Due to the fact that there is only a single family of constraints the formulation of the problem is short and simple. Even though KP's formulation is simple, KP is NP hard and has many applications in the real world.

ILP Formulation

Let us assume we have N items. Let $u_1, ..., u_N$ be the values of those N items, let

$w_1, w_2, ..., w_N$ be the set of their weights and $x_1, x_2, ..., x_N$ be the decision variables where

$$x_i = \begin{cases} 1 \\ 0 \end{cases}$$

W is the budget parameter (i.e the capacity of the knapsack), x_i takes the value 1 if the i_{th} item is included in the knapsack and 0 if it is not.

Our objective here is to maximize the value of the knapsack without surpassing its capacity. In other words, to include as much value as we can. So, a natural way to formulate the problem is:

$$maximize \sum_{i=1}^{i=N} u_i x_i$$
$$subject\ to \sum_{i=1}^{i=N} w_i x_i \leq W$$

Some other well known variations of the problem are the following: Unbounded KP (UKP) and multidimensional KP (MKP). The unbounded variant is the same with the one described above with the difference that in the unbounded variant the knapsack can include multiple copies of each item. The multidimensional KP, as the name suggests, considers more dimensions than just the weight dimension. For example, a knapsack can have limited volume. For each dimension, a constraint is added to the formulation of the problem. More on KP and its variants can be found in [Pisinger and Toth, 1998]

3.1.2 Minimum Vertex Cover (MVC)

MVC is another COP which is known to be a NP -Hard problem. Also, MVC is well studied and a number of algorithms have been devised in order to solve it, some of them are presented below.

Definition 3.2 *Given a graph G, find a subset of nodes $S \subset V$ such that every edge is covered i.e $(u, \upsilon) \in E$ if-f $u \in S$ or $\cup \in S$ and $|S|$ is minimized.*

ILP formulation

Let $G = (V, E)$ be the graph associated with the problem. For every $(u, \upsilon) \in E$ we

define the variables x_u, x_υ and we get the variable set. Given all the information the formulation of the problem as an integer linear program is the following:

$$minimize \sum_{u \in V} x_u$$
$$\text{subject to } x_u + x_\upsilon \geq 1 \; \forall (u, \upsilon) \in E$$
$$x_u = \begin{cases} 1 \\ 0 \end{cases}, \forall v \in V$$

The formulation is really simple since there is only a single family of constraints. The formulation for the weighted version of the problem where each node has some weight w_u is the same as the one written above adding w_u to the objective function.

3.1.3 Maximum Independent Set (MIS)

MIS, in contrast with MVC, seeks a subset S of vertices of a graph G such that each vertex in S is no adjacent to any other vertex of S. Mathematically that is simply:

$$\text{Given a graph } G = (V, E) \text{ and S} \subset V$$
$$\text{S is independent if and only if } SxS \cap E = \emptyset$$

So, MIS seeks the independent set S with the maximum cardinal number within graph G. The formulation of the problem is similar to other combinatorial problems discussed in the text.

ILP formulation

$$\text{maximize} \sum_{u} x_u$$
$$\text{subject to } x_u + x_v \leq 1, \forall (u, v) \in E$$
$$x_v = \begin{cases} 1 \\ 0 \end{cases}, \forall v \in V$$

3.1.4 Travelling Salesman Problem (TSP)

TSP is another interesting COP with many applications. TSP and its generalizations are applied in many areas such as logistics supply chain management, planning and scheduling. One can find an overview of the problem's applications

in [Matai et al., 2010]. Apart from the applications, TSP itself as a COP is very interesting. As it is usual for COPs, TSP has some distinct variants. Two of them are: graph TSP (GTSP) and the Euclidean TSP where the elements are 2-dimensional points in the euclidean metric space. The problem derives its name from the real life situation where a salesman wants to pass by a number of cities and return to their base city in an optimal way. As previously we give an integer programming formulation of that problem as well. That formulation is known as **Dantzig–Fulkerson–Johnson formulation.** [Dantzig et al., 1954]

Let V=1,2,3,...,N be the labels of the cities. Let c_{ij} be the cost of transportation from city i to city j:

$$x_{ij} = \begin{cases} 1, & \text{if the salesman goes from i to j city} \\ 0, & \text{if he does not} \end{cases}$$

$$minimize \sum_{i=1}^{n} \sum_{j \neq i \in Q} c_{ij} x_{ij}$$

$$subject\ to$$

$$\sum_{i=1, i \neq j}^{i=n} x_{ij} = 1, \forall j \in V$$

$$\sum_{j=1, i \neq j}^{j=n} x_{ij} = 1, \forall i \in V$$

$$\sum_{i \in Q}^{n} \sum_{j \neq i, j \in Q} x_{ij} \leq |Q| - 1, for\ all\ Q \subset V$$

The last type of constraints are called sub tour elimination constraints. TSP is hard both in theory and in practice. That gets more comprehensible seeing the table below about the first computational studies on TSP according to [Waterloo, 2016]. These studies were on the euclidean variant of the problem and none contained more than 120 cities.

However, advancements in theory and computational power made possible quite larger instances to be solved. In [Applegate et al., 2011] it is reported that an instance of 85.900 vertices has been solved using the TSP solver Concorde [Applegate et al., 2006]
.

cities	by	year
49	Dantzig et. al	1954
57	R.Karp	1957
120	M.Groetschel	1977

Table 3.1: First solutions to real TSP instances

3.2 Solution Methods

In this section, we are briefly introducing a number of distinct solution methods for COPs. Some of these methods can be applied to every COP (e.g branch and bound) and others work just for a specific problem. Solution methods can produce either exact or approximate solutions. One way to obtain an approximate solution is by using a heuristic method. Heuristics provide suboptimal solutions faster than exact algorithms. On some occasions, the use of heuristics is recommended.

3.2.1 Branch and Bound (BB)

We start with the BB method [Land and Doig, 1960]. The main idea of the method is to divide the problem solution space into smaller solution spaces (subspaces, whose union gives the problem solution space). These sub-spaces are constantly evaluated until the solver has a solution for the problem. That process starts with solving the relaxation LP (which is the initial problem without the integrality constraints). If the relaxation has an integer solution then the process terminates, otherwise, if there is at least one non-integer variable, then one of the non-integer variables is selected for branching. Branching leads to splitting the initial problem into two sub problems. Each of the sub problems formulations is given by adding one constraint to its parent problem. These constraints are $x_k >= \lceil x_k \rceil$ and $x_k \leq \lfloor x_k \rfloor$.

One can illustrate BB with a tree whose nodes correspond to BB subproblems and the root node corresponds to the initial problem. Some interesting decision problems arise in the process of BB; the solver has to decide on which variable to branch or which node to pick to examine at each step of the process. There are several ways to decide on these matters. The simplest one for selecting variable is to select the one that is "more fractional". Moreover, making good decisions has a considerable effect on how large the tree will be and hence on the algorithm's running time. That is why several rules have been developed to help the decision maker to make good decisions. Some of those methods are **strong branching** for variable selection and the **depth first search** strategy for node selection both of which are explained in [Wolsey, 2020]. Closing this subsection we give the pseudocode of

BB for minimization problems (one can easily modify the pseudocode to work for maximization problems). The following pseudocode is taken from [Sun et al., 2020]

Algorithm 1: Branch and Bound

Input a MIP in the form of equation (1) ;

1 Initialize $S = \{P_{LP}\}$ P_{LP} in form of equation (2) and set $x^* = \phi$ and $c = \infty$;

2 IF $S = \phi$ exit and return $x*$ and c ;

3 Select and pop a LP relaxation Q from S ;

4 Solve Q get solution $\hat{x}$ and optimal value $\hat{c}$;

5 IF $\hat{c} \geq c*$ go to 2 ;

6 IF $x \in X$ set $x* = \hat{x}$ and $c* = \hat{c}$, go to 2;

7 Select variable j, split Q into 2 subproblems add them to S then go to 3

The required equations follow :

$$\min_{\mathbf{x}\in\mathbb{R}^n} \left\{\mathbf{c}^T\mathbf{x} : A\mathbf{x} \leq \boldsymbol{b}, \ell \leq \mathbf{x} \leq \boldsymbol{u}, x_j \in \mathbb{Z}, \forall j \in J\right\} \tag{1}$$

$$\min_{\mathbf{x}\in\mathbb{R}^n} \left\{\mathbf{c}^T\mathbf{x} : A\mathbf{x} \leq \boldsymbol{b}, \boldsymbol{\ell} \leq \mathbf{x} \leq \boldsymbol{u}\right\} \tag{2}$$

3.2.2 Dynamic Programming

Dynamic Programming (DP) is a mathematical optimization algorithmic technique as well as a computer programming technique. DP is an effective way to solve problems which share a common characteristic. They can be broken down into simpler sub-problems and, by solving these subproblems recursively, a solution to the problem is obtained. For the problems that can be solved in that way, we say that they have the optimal substructure. COPs tend to have these desirable properties and hence DP is a way to solve them. The basis of this method is the so called **principle of optimality** which is defined in [Bellman, 1957] as "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision." The main characteristics of DP consist of stages, states, value functions and recursive relationships.

As said, DP problems can be structured into multiple subproblems, and each one of

the subproblems constitutes a stage. The solution of a stage affects the characteristics of the next stage. The states are the necessary information for seeing the effects of the current decision on future decisions. The recursive relationship relates the value function of a stage with the ones of the next stage. This relationship allows solving the problem recursively by solving a stage problem and then including a stage problem at a time until the overall optimum is obtained. Those concepts become more clear when seeing the **Held -Karp DP algorithm** for TSP [Held and Karp, 1962].

This algorithm in each stage i, for each set S of i cities, computes the shortest path from city b to all cities e such that $e \notin S, e \neq 1$ passing by all cities in S. The main components can be defined as follows: the stages are numbered as $i = 0, 1, 2..., n-1$, the states are tuples of the form (city, set of cities (S(i)), f_i denotes the optimal value function. Let b be the base city of the salesman. We denote with a_{kj} the cost of transportation from city k to city j and we denote with $S(i)$ a set of i in number cities. The value function and the recursive relationship are defined as:

$$f_i(e, S(i)) := \{\text{The minimum cost of transportation from the base city to e city visiting all the cities in the set S(i)}\}, \text{or}$$
$$f_i(e, S(i)) = \min_{k \in S(i)} [f_{i-1}(k, S(i) - \{k\}) + a_{ke}]$$

In the first stage of the algorithm we initialize the function values as follows:

$$f_0(e, -) = a_{be} \text{ , for all cities e}$$

3.2.3 Heuristics

All of the problems we have discussed are computationally hard. For large sized COP instances, the running time of exact algorithms is large. That problem can be tackled by heuristics. However, heuristics do not provide optimality guaranties and they may well be stuck in local optima. In this subsection, we briefly present some heuristics for the COPs we discussed in the previous subsection. Mainly, we discuss **constructive heuristics** and **greedy heuristics**. In constructive solutions, one starts with an empty set (initial solution) and adds elements sequentially. The decision on what element to add is based on some selection criterion. Greedy heuristics use selection criteria which are based on some local optimality such as the best improvement on the objective function. These criteria are myopic and can lead to worse future additions and hence to worse solutions.

Let us now give the **greedy node algorithm** for the MVC problem. This algorithm simply selects the node with the highest degree at each step. Then, erases all the

edges connected to that node until the edge set is covered. So, let $G = (V, E)$ be a graph.

Algorithm 2: MVC greedy node algorithm

```
S ← ∅
while S not a cover do
    Pick node v with highest degree in the active graph and add it S ;
    erase all edges incident to v ;
return S
```

Another similar algorithm is the one called: 2-OPT approximation algorithm for minimum vertex cover. The name of the algorithm makes much more sense when one knows that this algorithm finds vertex covers with less or equal to 2 x (minimum vertex cover). There are 2-opt algorithms for many other COPs as well. We move on to discuss a heuristic for TSP.

Nearest neighbour algorithm for the TSP :

The idea again in this algorithm is fairly simple. At each step, the algorithm adds the node which is closer to the previous added one. Let $V = \{v_1, ..., v_N\}$ be the set of vertices (or cities). We assume the base city is denoted v_1. This algorithm's running

Algorithm 3: Nearest Neighbour TSP

```
1 Initialize S = ∅
2 Begin from the base city and add v_1 to S
while |S| < N do
    Add v* from v : v ∈ V − S s.t distance v* with the previous city is the
     minimum
```

time in worst case scenario is $O(n^2)$. So, it approximates the optimal solution in a polynomial time. As we have already mentioned brute force attack has O(n!) and hence there is a huge difference. We continue to discuss an algorithm for the MIS problem.

A greedy algorithm for MIS

In a similar way, we can give a greedy algorithm for the MIS problem. This one is called **Minimum Degree Heuristic** and can be executed by following the two steps below :

- Select vertex with minimum degree
- Remove all its neighbors

So, in every iteration the solver selects the node of minimum degree available for selection and then removes all its neighbours from the list of available for selection nodes. This algorithm has an approximation ratio $(\Delta+2)/3$ where Δ is the maximal degree of the graph. [Wikipedia, 2022b]

The algorithms we briefly introduced are used as baselines in the last chapter of this thesis. In the last chapter, we present our applications, which we will see are nothing but "learned heuristics ". That means an artificial agent learns itself a heuristic using machine learning in contrast with traditional heuristics, such as the ones above, which are designed entirely by humans. In relevant work that inspired this thesis, a great number of human designed heuristics are used as baselines. For example, in [Khalil et al., 2017] the following heuristics are used: Minimum Spanning Tree (MST), Farthest insertion, Cheapest insertion, Closest insertion, Christofides and 2-opt, along with the nearest Neighbor heuristic which we use too.

Chapter 4

NEURAL NETWORKS

Machine learning is the field of computer science that specializes in constructing algorithms that are able to make accurate predictions without being explicitly programmed. Instead of being explicitly programmed these algorithms are automatically improved by using data. That happens through experience, trial and error and in a way that is similar to how humans learn. There are three types of machine learning: Supervised machine learning, unsupervised learning and reinforcement learning. Each category is suitable for some problems and has its own distinct algorithms. A tool that appears useful to all three categories is Neural Networks (NNs) and it is the topic of the next chapter.

The human brain contains an astronomical number of neurons or nerve cells, and each neuron is connected to a great number of other neurons. The cell body (soma), dendrites and an axon constitute a typical neuron. Usually, dendrites and soma receive signals and pass signals down to the axon and the signal ends up in a dendrite of another neuron [Stangor and Walinga, 2014]. Artificial Neural Networks (ANN) imitate that process. The basic components of an ANN are neurons. Neurons in ANNs can have multiple input and output neurons in the previous and next layer, respectively. Every neuron receives signals from its input neurons of the previous layer, aggregates them, and then passes the aggregated signal through an activation function. The activation function decides, based on the aggregated signal, whether the neuron will be activated. If the neuron is activated, then a "strong" signal (high value) will be passed forward to its output nodes in the next layer, otherwise a low value will be passed forward. The neurons are organized in consecutive layers. Each neuron of the network can be connected to a random number of neurons. Below, there are typical graphical representations of simple neural networks.

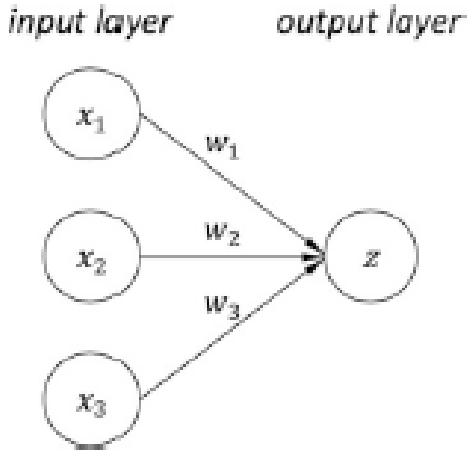

Figure 4.1: Perceptron illustrated as in [Dong et al., 2020]

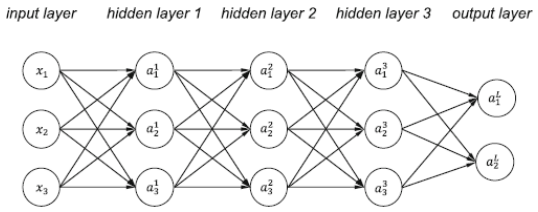

Figure 4.2: MLP illustration [Dong et al., 2020]

The neural networks in figures 4.1, and 4.2 are simple neural networks which are called **perceptron and multilayer perceptron (MLP)** respectively. When the neurons of a layer are connected to all neurons of the next layer, then that layer is called a **dense** layer. The layers of a neural network that lie between the input and the output layers are called **hidden layers** because they are not accessible from outside the network. How these networks work is described mathematically below :

Let

$$W = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \tag{4.1}$$

contain the weights of the input layer to the first hidden layer. Where w_{ij} is the synaptic weight between i_{th} neuron of the hidden layer and j_{th} input. Then each hidden layer gets as inputs the outputs from the previous layer, which using matrix algebra can be calculated by the following equation.

$$H_1 = W_1 x^t + b_1 \tag{4.2}$$

where W is the weight matrix of (4.1), $x = \begin{pmatrix} x_1 & x_2 & x_3 \end{pmatrix}$ is the inputs vector, b is the bias (it can be omitted) and t denotes the transpose matrix. Continuing in a forward way, we can calculate the outputs of the neural network using the equations below.

$$H_2 = W_2 H_1 + b_2 \tag{4.3}$$

$$H_3 = W_3 H_2 + b_3 \tag{4.4}$$

$$O = W_4 H_3 + b_4 \tag{4.5}$$

The process we just described is known as **forward propagation**. Up to this point, we have discussed only linear neural networks. These networks are useful, but usually, more complicated networks are used. Non-linear neural networks can be created by using activation functions.

4.1 Activation Functions

We briefly discuss now some common activation functions. Activation functions are element-wise, differentiable functions, and can add non-linearity to neural networks. The first activation function we are going to discuss is softmax which is also knows as normalized exponential activation function. It takes as input a vector of real numbers and outputs a probability distribution. Let S denote softmax S: $\mathbb{R}^n \to [0,1]^n$

$$S(z)_i = \frac{e^{z_i}}{\sum_{k=1}^{k=n} e^{z_k}} \tag{4.6}$$

The equation 4.6 shows how softmax normalizes the input vector and produces a probability distribution. It is very clear that $S(z)_i \in [0,1]$ and $\sum_i S(z)_i = 1$. In practice, softmax is mostly used in classification problems. Specifically, it is used in the output layer of the network in order to transform the output into a probability distribution. We now discuss an activation function called sigmoid.

4.1.1 Sigmoid

is another well known activation function. The sigmoid function takes as input any real number and outputs it in the (0,1) interval.
Sigmoid: $\mathbb{R} \to (0,1)$

$$f(x) = \frac{1}{1+e^{-x}} \tag{4.7}$$

Its derivative satisfies the next equation

$$f^{'}(x) = \frac{e^{-x}}{(1+e^{-x})^2} = f(x)(1-f(x)) \tag{4.8}$$

Let us now proceed to a very similar function that is called hyperbolic tangent or tanh.

4.1.2 Hyperbolic tangent

is an activation function similar to sigmoid because is defined over all real numbers and squeezes them in the interval $(-1, 1)$

$$tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}} \tag{4.9}$$

As expected it is differentiable, and its derivative is given below

$$tanh^{'}(x) = 1 - \frac{(1 - e^{-2x})^2}{(1 + e^{-2x})^2} = 1 - tanh^2(x) \tag{4.10}$$

4.1.3 ReLU

ReLU stands for Rectified Linear Unit. ReLU is an activation function that performs well on several machine learning tasks and is often preferred instead of the two activation functions discussed below. That is because ReLU is easier to implement and compute, and its gradients are more consistent, which helps in network optimization.

$$ReLU(x) = \begin{cases} x & , \text{if } x > 0 \\ 0 & , \text{if } x \leq 0 \end{cases} \tag{4.11}$$

ReLU consists of two linear parts. It is differentiable for every $x \in \mathbb{R}^*$. The equation below give us ReLu's derivative.

$$ReLU(x)' = \begin{cases} 1 & , if\ x > 0 \\ 0 & , if\ x < 0 \end{cases} \tag{4.12}$$

In practice, ReLU is far more often used than sigmoid or tanh. That is because of the advantages that we have discussed over the other two. Furthermore, the sigmoid is used only in the output layer in binary classification problems. The sigmoid is a good choice for that task since it squeezes its input into a value in $(0, 1)$, which can be converted to a label class (0 or 1) easily. In hidden layers, ReLu is preferred over sigmoid. Tanh outputs in the interval (-1,1), so it can be used in the output layer in order to generate image pixels or it can be used in hidden layers to provide non linearity.

4.1.4 Vanishing-Exploding gradients

We are interested in derivatives of activation functions because they are important for the process of optimizing and tuning neural networks. The behaviour of derivatives of activation functions can affect the process of optimizing a neural network. When optimizing a neural network, two problems called **exploding gradients**, and **vanishing gradients**, respectively, can possibly occur. The former means that the gradients get larger and larger, while the latter means that the gradients get smaller and smaller approaching zero. Some activation functions are more susceptible to leading to vanishing or exploding gradients. For instance, sigmoid and tanh are more susceptible to vanishing gradients than ReLU.

4.2 Training neural networks

In order to train neural networks a plethora of methods have been developed and there are many tools that facilitate this process. Some of these tools are briefly discussed below.

4.2.1 Cost Functions

An important tool in training neural networks is **cost functions**. Cost functions provide a way to measure the performance of machine learning models. They measure the extent a prediction is close to the real value. Some cost functions are the following

Mean Squared Error:

$$MSE(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{4.13}$$

Due to its quadratic formula, MSE is affected more by high errors compared to other cost functions such as the next one. So, if the predictions are quite close to the real values then the MSE will be small. If some of the predictions differ significantly from the real values then the MSE will be large.

The Mean Absolute Error:

$$MAE(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i| \tag{4.14}$$

Mean absolute error and mean squared error are popular cost functions for regression problems. To tackle classification problems, different cost functions are used. The most known is

Cross Entropy for binary classification:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log \hat{y}_i + (1-y_i)\log\left(1-\hat{y}_i\right)\right) \tag{4.15}$$

Where y_i is the actual binary value and $\hat{y}_i$ is the predicted probability that i_{th} data entry belongs to class 1. Apart from the cost functions, gradient methods are a very important component of the training phase of neural networks.

4.2.2 Gradient Methods

try to make the neural network learn its parameters θ (weights w and biases b) with respect to some cost function, which is minimized. Calculating the cost function's minimum is usually very difficult to be done analytically when the neural networks are complex. For that reason, methods like gradient descent are used. This method approaches the cost function's minimum by doing small steps toward it. In order to do this, we apply to the network parameters the following update rule

$$\theta \leftarrow \theta - a\nabla_\theta L \tag{4.16}$$

Where α is the learning rate, and it is the pace at which the network learns its parameters. Now we discuss how these gradients ($\nabla_\theta L$) are calculated. That is done using the following method:

Backpropagation algorithm traverses the network from the output layer to the input layer in order to calculate the gradients with respect to its parameters. The calculation requires the use of the chain rule heavily because a neural network consists of a number of composite functions. A brief description of how backpropagation works is given below.

- l=1,2,3,..,l are the indexes of the network layers.
- $z^l = W^l a^{l-1} + b^l$ is the weighted output of the l-th layer
- $C = \frac{1}{2}||y - a^L||_2$ is the cost function
- $\delta^l = \frac{dC}{dz^l}$
- $a^l = f(z^l)$ where f is an activation function

To calculate the gradient with respect to the weights of l-th layer we first have to calculate δ^l

$$\delta^l = \frac{dC}{dz^l} = \frac{dC}{dz^{l+1}} \frac{dz^{l+1}}{da^l} \frac{da^l}{dz^l}$$

$$\frac{dC}{dW^l} = \frac{dC}{dz^l} \frac{dz^l}{dW^l}$$

Calculating $\frac{dC}{dw^l}$ requires first calculating $\frac{dC}{dz^{l+1}}$ so the process of calculating gradients has to start from the last layer and then move backwards. Gradient descent on every step computes the cost function with respect to the whole training data set. The larger the training data set is the more costly the computation becomes. Usually, instead of gradient descent, stochastic gradient descent (SGD) is used. SGD computes the cost function on randomly selected samples from the training data which are called minibatches. For more detailed information on the discussed concepts above, one

Algorithm 4: Stochastic Gradient Descent (SGD)

Input: ϑ-parameters of the network, N- number of iterations.

for i=1,..., N **do**
  calculate the cost function C of a minibatch
  backpropagate to calculate $\frac{dC}{d\theta}$
  $\nabla\theta \leftarrow -a\frac{dC}{d\theta}$
  $\theta \leftarrow \theta + \nabla\theta$
end for
return the parameters θ

can take a look at [Zhang et al., 2021, Dong et al., 2020].

We proceed to next section which has to do with an interesting type of neural network.

4.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) [McClelland and Rumelhart, 1987] are neural networks which were made in order to deal with sequential data. Sequential data can be the price of a stock and, generally time series data, the words of a sentence in natural language. In sequential data, the parts of the sequence are dependent on each other and the order is important. So, the elements in the sequence $x_1, x_2, x_3, ..., x_n$ interact in some way, and any permutation of the sequence may not have the same "meaning" as the original one. Designing special neural networks to deal with such data was needed, and RNNs were designed. RNNs take as input sequences in steps where in each step those networks need to have some information about the past of the sequence. That is why they use hidden states. Hidden state vectors offer a way to "remember" information about the past sequence through the process. Let x_t be the input of time step t and h_{t-1}, h_t the hidden states of time steps $t-1$ and t respectively then h_t is given by the

$$h_t = f(h_{t-1}, x_t) \tag{4.17}$$

Since the computation of h_t involves h_{t-1}, the equation above is recurrent. There are many architectures of recurrent neural networks. In this text, we cover the fundamentals.

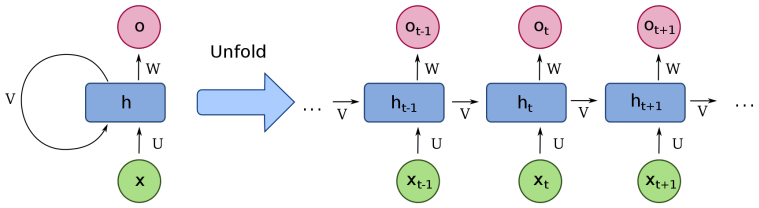

Figure 4.3: An illustration of an RNN unfolding [Wikipedia, 2022c]

The figure below illustrates how RRNs unfold. As one can see there are many parameters above such as U, V and W which are weight matrices.

- **W** is the weight matrix associated with the output of each time step and its hidden state

- **V** is the weight matrix associated with the hidden states
- **U** is the weight matrix associated with the inputs

Let us now give the details of how these matrices are used by RNNs.

$$H_t = g(X_t U + H_{t-1} V + b_h) \tag{4.18}$$

$$O_t = H_t W + b_q \tag{4.19}$$

Where $X_t \in \mathbb{R}^{nxd}$ is the input at time step t, n is the size of the training batch and d is the input size. $H_t \in \mathbb{R}^{nxh}$ is the variable that contains the hidden state of each example in the batch at time step t. With g we denote some activation function (usually tanh is used). The weight matrices are $U \in \mathbb{R}^{dxh}$, $V \in \mathbb{R}^{hxh}$ and $W \in \mathbb{R}^{hXq}$, h is the number of hidden units and q is the length of the outputs. In addition, there are bias parameters: b_h the bias of the hidden layer and b_q the bias of the output layer. In an RNN we have shared parameters, which means we use the same parameters for any time step, so there would be no extra parameterization cost if the number of time steps increased. The training process of these networks is similar to the one used for training multilayer perceptrons. The process starts with the initialization of the parameters (usually the initial hidden state h_0 is a vector with only zeros) then the forward pass follows, in which the outputs are calculated and then comes the loss calculation. Although RNNs can help with tasks that MLPs cannot, in some tasks RNNs can only go so far. For instance, in tasks with long sequences, RNNs appear to fail to catch very long term dependencies. They also suffer from vanishing and exploding gradients problems, which we discussed in section 4.2. In order to deal with those issues, an advancement of recurrent neural networks was designed. This advancement has a new feature, which is gates.

4.4 Gated Recurrent Neural Networks (GRNNs)

GRNNs were introduced in [Cho et al., 2014] and they are used more than simple RNNs. We now discuss two types of gated recurrent neural networks, the gated recurrent units (GRU) and long short term memory networks (LSTMN). The primal difference between GRNNs and RNNs is the first ones use gating mechanisms in the calculation of hidden states. Let us introduce the GRUs first, which are simpler than LSTMNs. Gated recurrent units use an update gate and a reset gate. These gates contain numbers from the interval $(0, 1)$ such that we can perform convex

combinations. The reset gate decides how much of the old hidden state we want to keep. The update gate decides how much of the new state is just a copy of the previous one.

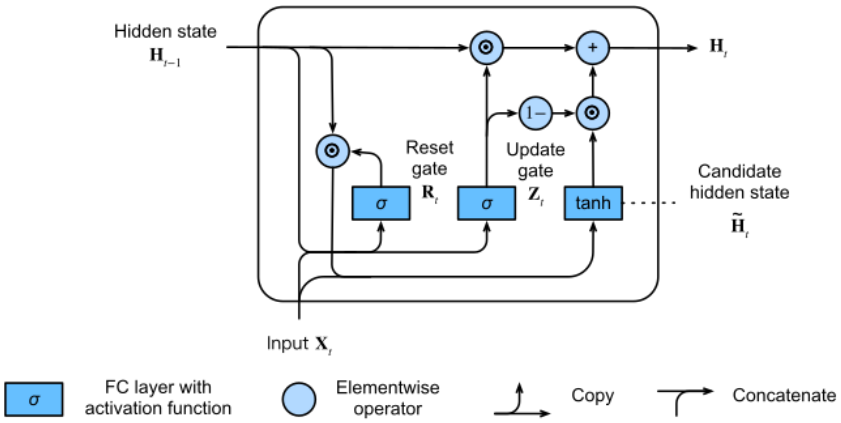

Figure 4.4: GRU illustration as in [Zhang et al., 2021]

We have no better way to describe GRUs than by giving their mathematical details. So as previously let X_t be a batch at time step t and H_{t-1} the hidden state of the previous time step. Then, the Z_t, R_t are given from the equations below.

$$R_t = \sigma(X_t U_r + H_{t-1} V_r + b_r) \tag{4.20}$$

$$Z_t = \sigma(X_t U_z + H_{t-1} V_z + b_z) \tag{4.21}$$

Where σ denotes the sigmoid activation function and b_z, b_r are biases. Another essential component of GRUs is the candidate hidden state $\overline{H}$. Now that we have defined the reset gate in equation 4.20, it is time to use it in calculating the candidate hidden state.

$$\overline{H} = tanh(X_t U + (R_t \odot H_{t-1}) V + b_h) \tag{4.22}$$

Where $\odot$ denotes the Hadamard product operator (multiplication of elements in corresponding cells). It is easy to see that when all the values in R_t are close to one, the calculation of the candidate hidden state coincides with the calculation of a simple RNN's hidden state. When all values in R_t are close to 0, the calculation of the candidate hidden state coincides with a multilayer perceptron output which took X_t as input. To conclude our discussion on GRU's, we should discuss how hidden states are calculated. That calculation involves both the update gate and the candidate hidden state.

$$H_t = Z_t \odot H_{t-1} + (1 - Z_t) \odot \overline{H_t} \tag{4.23}$$

The equation above shows the degree that the new hidden state at the current time step is a caricature of the previous one and how much the new candidate hidden state is used. Similar to before, one can easily see that if all values in Z_t are close to 0 then the new hidden state coincides with the candidate hidden state. On the contrary, when all values in z_t are close to 1 then the new hidden state is equal to the previous one. In [Zhang et al., 2021] one can find more details along with code implementations of such NNs.

4.4.1 Long Short Term Memory

Long Short Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997] networks are neural networks which are designed similarly to GRUs. Although these networks are a bit more sophisticated than GRUs they were invented almost twenty years before GRUs. LSTMs have a cell state which has the same shape as the hidden state and its purpose is to save extra information. In order to manage this cell state, three gates are used. The first gate is called the input gate, the second is called the output gate and the third one is called the forget gate. These gates are responsible for different tasks, which are presented mathematically below.

$$I_t = \sigma(X_t U_i + H_{t-1} V_i + b_i) \tag{4.24}$$

$$F_t = \sigma(X_t U_f + H_{t-1} V_f + b_f) \tag{4.25}$$

$$O_t = \sigma(X_t U_o + H_{t-1} V_o + b_o) \tag{4.26}$$

Where $H_{t-1} \in \mathbb{R}^{nxh}$ is the hidden state variable at the time step $t-1$, $X_t \in \mathbb{R}^{nxd}$ is a batch of n examples of d input size at the time step $t, U_i, U_f, U_o \in \mathbb{R}^{dxh}$, $V_i, V_f, V_o \in \mathbb{R}^{hxh}$ are weight matrices and $b_i, b_f, b_o \in \mathbb{R}^{1Xh}$ are biases. About the newly introduced: $I_t \in \mathbb{R}^{nxh}$ is the input gate, $F_t \in \mathbb{R}^{nxh}$ is the forget gate, $O_t \in \mathbb{R}^{nxh}$ is the output gate.

Candidate Memory Cell is computed pretty much in the same way as the gates are computed. Nonetheless, there is a key distinction: instead of using sigmoid activation function, the candidate memory cell uses tanh. We denote the candidate memory cell at the time step t with $\overline{C_t}$ and we compute it as follows.

$$\overline{C_t} = tanh(X_t U_c + H_{t-1} V_c + b_c) \tag{4.27}$$

Where $U_c \in \mathbb{R}^{dxh}, V_c \in \mathbb{R}^{hxh}$ are weight matrices and $b_c \in \mathbb{R}^{1xh}$ is the bias.

Memory cell uses the two gates. The input I_t, which decides how much new information we will take from the current candidate memory cell $\overline{C_t}$. The forget

gate F_t, which decides how much we will forget from the previous memory cell.

$$C_t = F_t \odot C_{t-1} + I_t \odot \overline{C_t} \tag{4.28}$$

If the values in the forget gate are close to 0 then the information in the previous memory cell is forgotten. The input gate decides what information from the candidate memory cell will be kept. When the values of I_t are close to 1 then the corresponding information from the candidate memory cell is kept when they are close to zero is not kept.

Hidden State uses the output gate and the memory cell in its computation.

$$H_t = O_t \odot tanh(C_t) \tag{4.29}$$

When O_t values are close to 1 then a rich hidden state is passed through to the next time step.

At this point, all the computational aspects of the LSTMs are described. LSTMs between time steps pass through only the hidden state and Memory cell. They "fix" the problem of vanishing or exploding gradients and are capable of catching long time dependencies that simple RNNs cannot. RNNs and especially LSTMNs are widely used in applications of ML in CO. For instance, LSTMNs are used in [Bello et al., 2016] in order to devise a mechanism called attention, which is described below.

4.5 Attention Mechanism

In this section, we briefly discuss attention mechanism [Bahdanau et al., 2014]. This mechanism enhances some parts of the input data and diminishes other parts and shows which parts of the data we should more focus on. Usually, it is used in models that use the following neural network architecture: **encoder decoder architecture**. That architecture consists of two components:

- The **encoder**, which takes an input sequence of any length and creates a context variable of fixed shape. The context variable contains the information of the sequence encoded.
- The **decoder**, which takes the context variable and gives the output sequence of variable length. In the next paragraph we discuss a design that will help us to describe attention.

In the next paragraph we discuss a design that will help us to describe attention.

This design is called **sequence to sequence learning**. In this encoder decoder framework, both the encoder and the decoder are recurrent neural networks. The encoder as an RNN works as follows: given the sequence at the time step

t calculates the hidden state:

$$h_t = f(x_t, h_{t-1})$$

Where f is some function to express how the RNN transforms in the recurrent layer. After the calculation of the hidden state h_T at the final step T the context variable c is calculated

$$c = q(h_1, h_2, ..., h_T) \tag{4.30}$$

where q is some function of the hidden states. Quite often, the context variable c is chosen to be just the hidden state at the final step T , $c = h_T$.

In every time step, the decoder uses the context variable in the calculation of its output and hidden states. For the hidden states, which we denote here with s:

$$s_t = g(y_{t-1}, s_{t-1}, c) \tag{4.31}$$

and, the outputs y_t are conditioned on the previous outputs and the context variable c. So, when we get the hidden state s_t, we apply softmax operation to get the probability distribution of y_t and eventually the output y_t.

The attention mechanism, as mentioned, is a mechanism which helps to make more accurate predictions. This mechanism determines the parts of the sequence to focus on. Moreover, it works similarly to how human attention works. At each time step of decoding, a scoring function is used to compute a score between each hidden state h_j of the encoder and the most recent hidden state of the decoder (the dot product could be the scoring function).

$$\alpha_{tj} = score(s_{t-1}, h_j) \tag{4.32}$$

Next, the scores are normalized in a probability distribution using the softmax operation.

$$a_{tj} = \frac{score(s_{t-1}, h_j)}{\sum_{i=1}^{i=T} score(s_{t-1}, h_i)} \tag{4.33}$$

As soon as we get those normalized values, we use them as weights, and we obtain the context variable of time step t as the weight average of the encoder's hidden

states.

$$c_t = \sum_{j=1}^{j=T} a_{tj} h_j \tag{4.34}$$

Now that the context variable has been computed, the process continues as previously described in sequence to sequence learning.

4.6 Graph Neural Networks

Having covered some NN architectures that we believe are fundamental and a good base to comprehend the more complicated-modern ones, we would like to discuss **Graph Neural Networks (GNNs)** before moving to the next chapter. A specific type of GNNs, **Graph Convolutional Networks (GCNs)** are used in our experiments. These NNs are designed to be used in ML tasks which deal with graph data, and they produce meaningful representations of nodes, edges, or even graphs. They work as follows: they get as input a set of features of nodes and/or edges, and using a mechanism called message passing or neighborhood aggregation, they create representations for each node by aggregating its neighbors' features. At each iteration of the message passing process, an embedding for each node is updated. The number of iterations is determined by the number of GNN layers. The more GNN layers, the more "distant neighbors" are used, stacking L GNN layers results in producing node embeddings based on L-hop neighborhood of each node. GNN layers are based on two functions, the **update** and the **aggregation**. At each iteration, the aggregation function computes a message by aggregating the embeddings of all neighboring nodes of a node u . Then, the update function takes this message and combines it with the previous embedding of u and produces the new embedding of u. These two functions should be differentiable. We proceed with our discussion by giving the mathematical description of a simple GNN layer.

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}_{\text{self}}^{(k)}\,\mathbf{h}_u^{(k-1)} + \mathbf{W}_{\text{neigh}}^{(k)} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k-1)}\right) \tag{4.35}$$

That neural network is a simplification of the one presented in
[Merkwirth and Lengauer, 2005]. Here, h_u^k is the hidden embedding of u node in k_{th} iteration, σ is a function that provides non-linearity, which could be ReLU, tanh, or some other activation function. W_{self}^k, W_{neigh}^k are trainable parameters that can be shared between layers or not. The aggregation function is $m_u = \sum_{v \in \mathcal{N}(u)} \mathbf{h}_v^{(k-1)}$, and the update is $UPDATE(h_u, m_u) = \sigma(W_{self} h_u + W_{neigh} m_u)$.

Now that we have discussed the basics of graph neural networks, we briefly describe how GCNs work as introduced in [Kipf and Welling, 2016]. GCNs work in two stages: first average the node features over the neighborhood of each node, and then pass the averaged feature to a fully connected network. The following equations describe how a GCN works.

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right)$$
$$H^{(0)} = X$$

Where A is the adjacency matrix, $\tilde{A}$ is A with self-loops, $\tilde{D} = \sum_j \tilde{A_{ij}}$ is the degree matrix. X is the node input feature matrix and has a shape of NXD where N is the number of nodes and D the number of features, H^{l+1} is the output of layer $l + 1$ and has a shape of NXF, where F is the dimension of the output feature. GNNS are very popular in applications of ML- graph CO. Examples of applications can be found in [Gasse et al., 2019, Khalil et al., 2017, Barrett et al., 2020, Joshi et al., 2019], which are discussed later.

Chapter 5

REINFORCEMENT LEARNING

Reinforcement Learning (RL) is one of the three types of machine learning. The basic components of RL systems are two entities: environment and agent. The environment is an entity with which the agent interacts. For example, the environment could be a chess game, a backgammon game, or almost any board game. The agent decides which action to take from an action set that contains all possible actions at each time step in the process. Whenever the agent takes an action, it transits from one state to a new state and receives a scalar signal, which is called reward. In RL, there are two types of reward, immediate and accumulative. The goal of RL is to teach the agent to interact well in the environment, and this is done using the maximization of the accumulative rewards. The states, actions and rewards, which are important features of RL, are denoted with the letters S, A and R, respectively.

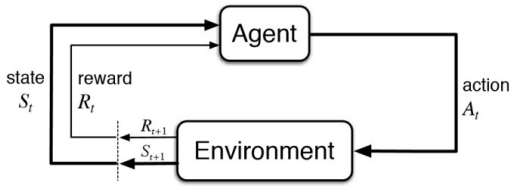

Figure 5.1: Interaction agent environment [1]

As shown in Figure 5.1 and detailed previously in the text, the agent starts from an initial state (usually randomly chosen), takes an action, and receives a reward. Then it transits to a new state. This happens at each time step of the interaction procedure and forms what is called a **trajectory**.

$$\tau = (S_0, A_O, R_1, S_1, A_1, R_2, S_2, ...)$$

The trajectory τ simply means that the agent went from state S_0 to state S_1 taking the action A_0 and received the reward R_1. Then went from S_1 to S_2 taking action A_1 and received the reward R_2.

5.0.1 Markov Decision Process

A Markov Decision Process (MDP) is the way in which an RL environment is mathematically formulated. An MDP consists of a tuple of **(S,A,P,R,**γ**)** where **S** is

a set of states, **A** is a set of actions, and **P** is the probability of transition from state to state taking an action.

$$p(s^{'}|s,a) = p(S_{t+1} = s^{'}|S_t = s, A_t = a) \tag{5.1}$$

As one can see from the equation above, the transition to a state is not only conditioned on the state of the previous time step; it is conditioned on both the previous state and the action taken. **R** represents the immediate reward function.

$$R_t = R(S_t, A_t) \tag{5.2}$$

and γ is a scalar, which is called a discount factor. MDPs are a decent way to formulate sequential decision-making. More about MDPs can be found in [Sutton and Barto, 2018], [Dong et al., 2020].

5.0.2 Rewards and Returns

We have already mentioned that the agent's goal is to maximize the accumulative reward it receives, but this is somewhat vague. For that reason, we introduce a new concept, called return.

$$G_t = R_{t+1} + R_{t+2} ..... + R_T \tag{5.3}$$

So, the return G_t is just the sum of the rewards from the time step t to the final time step T, or the cumulative reward after the time step t. Therefore, the agent's goal is to maximize G_t or ,even better, the expected G_t. In some RL tasks, there is not a final step T, and the task continues without stopping. Thus, the defined return given above is not suitable for such tasks. In such cases, the discount factor is handy.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0} \gamma^k R_{t+k+1} \tag{5.4}$$

G_t and G_{t+1} are related since they satisfy the next equation.

$$G_t = R_{t+1} + \gamma G_{t+1}$$

When γ is equal to zero, the agent is interested only in maximizing the immediate reward and is called "myopic" because it does not care about the future. However, if γ is equal to 1 then (5.4) is exactly the same equation as in (5.3). When $0 < \gamma < 1$, γ acts as a kind of weight for the value of future rewards; the further in the future the reward is received, the less weight is assigned.

5.0.3 Policy and Value Functions

We begin by defining what a policy is. A policy is a correspondence of probabilities from states to actions. When the agent is in a state $S_t = s$ at time step t, the policy gives the probability of taking the action $A_t = a$ for every possible action. In the literature, a policy is denoted by $\pi(a|s)$ and it is clear that it forms a probability distribution over the possible actions in A for each state in the set of states S. We defined what a stochastic policy is. A policy can also be deterministic, which means that each state is mapped to one single action by the policy.

The **value function** of a state s following a policy π is the expected return after being in state s.

$$V_\pi(s) = \mathbb{E}[G_t|S_t = s], \text{for all s in S} \tag{5.5}$$

We call V_π the state value function, and apparently, the terminal states have a value of zero. Now, we proceed to the action value function, which is defined similarly.

$$Q_\pi(s, a) = \mathbb{E}[G_t|S_t = s, A_t = a] \tag{5.6}$$

These functions can be estimated by keeping track of the agent's experience while following policy π. That is, to keep track of the returns that the agent receives starting at each state and then average them. The averages obtained constitute estimates of the real state values and action values. These methods for calculating estimations of value functions are called Monte Carlo methods.

5.0.4 Optimal Value functions and Policies

Using value functions allows us to order policies in some way. A policy π is better than or equal to a policy π' if the value of all states s following the policy π is greater than or equal to the value of the state following π'. More formally, $\pi \geq \pi'$ if and only if $V_\pi(s) \geq V_{\pi'}(s)$. This relation forms a partial ordering. Now we proceed to define the optimal value function. Optimal value functions satisfy the following equation, and we denote them by $V_*(s)$.

$$V_*(s) = \max_\pi V_\pi(s) \tag{5.7}$$

Similarly, an optimal action function satisfies the next equation.

$$Q_*(s, a) = \max_\pi Q_\pi(s, a) \tag{5.8}$$

Given that we follow an optimal policy and using the definition of the action value function Q, which gives the expected return of taking an action α in a state s, we can rewrite Q_* using $V_*(s)$.

$$Q_*(s,a) = \mathbb{E}[R_{t+1} + \gamma V_*(S_{t+1}) | S_t = s, A_t = a] \tag{5.9}$$

5.0.5 Bellman equations

It can be shown that the value of state s in any policy is in relation to the value of the succeeding state and the next relation is satisfied.

$$V_\pi(s) = \mathbb{E}_{\alpha\sim\pi, s'\sim p(|s,a)}[r + \gamma V(s')] \tag{5.10}$$

The above equation is known as the Bellman equation for value functions. In addition, there is a Bellman equation for the action value function.

$$Q_\pi(s,a) = \mathbb{E}_{s'\sim p(|s,a)}[r + \gamma \mathbb{E}_{\alpha\sim\pi(a|s)}[Q(s',a')]] \tag{5.11}$$

Moreover, Bellman's equations also hold for the optimal value functions.

$$V_*(s) = \max_a \mathbb{E}[R_{t+1} + \gamma V_*(S_{t+1}) | S_t = s, A_t = a] \tag{5.12}$$

$$Q_*(s,\alpha) = \mathbb{E}[R_{t+1} + \gamma \max_{\alpha'} Q_*(S_{t+1}, a') | S_t = s, A_t = a] \tag{5.13}$$

5.1 Algorithms

As already mentioned, there is a broad set of algorithms within reinforcement learning. In the next subsection we briefly discuss a few of those algorithms, which we consider very useful for RL-in CO.

5.1.1 Dynamic Programming

We start with algorithms belonging to DP, which we discussed previously, DP can be used in RL in order to compute optimal policies. In order to use DP, some conditions must be met. These algorithms assume a finite state set and action set, as well as complete knowledge of the environment dynamics

Policy Iteration

Policy iteration consists of two distinct processes: **policy evaluation** and **policy improvement**. Policy evaluation is used to determine the state value given an

arbitrary policy π. This is done by expanding (6.5) as follows.

$$V_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1})]|S_t = s] \tag{5.14}$$

$$= \sum_a \pi(\alpha|s) \sum_{s'} p(s'|s,a)[r + \gamma V_\pi(s')] \tag{5.15}$$

At first, an estimation of V_π is picked at random for every non-terminal state. Let us denote it by v_1, then by using equation 5.15 the successor values are calculated.

$$v_{k+1} = \sum_a \pi(\alpha|s) \sum_{s'} p(s'|s,a)[r + \gamma v_k(s')]$$

Policy evaluation stops when two successive estimates have a distance less than θ, a small threshold. That is, when for a natural number m we have:

$$\max_s |v_m(s) - v_{m+1}(s)| < \theta$$

When the policy evaluation is performed, the policy improvement begins. Policy improvement decides whether in a state s the agent should take the action $\pi(s)$ or it is better to choose another action. The agent should choose this action if it leads to a better state value following the policy π. This state action value is $Q_\pi(s,a)$. Policy improvement is based on a theorem called the policy improvement theorem, which is stated below. Whenever for two policies π and π' holds

$$Q_\pi(s, \pi'(s)) \geq V_\pi(s) \text{ for all states } s$$

then

$$V_{\pi'}(s) \geq V_\pi(s)$$

Given a policy π and the state value $V_\pi(s)\ for\ s \in S$ the policy improvement constructs a new policy using the state values following the policy π as described below.

$$\pi'(s) = arg \max_a Q_\pi(s,a) \tag{5.16}$$

The new policy π' satisfies the improvement theorem and therefore is an improvement of π. If the two policies lead to the same state values, Bellman's equation implies that π' is an optimal policy. Now that the policy improvement has been described, we can describe the policy iteration more technically. We start with an initial policy π_1 and evaluate it, then improve it and get π_2. Then we evaluate π_2 and improve it. We go on this way until the termination criterion is satisfied.

$$\pi_1 \longrightarrow^E v_{p_1} \longrightarrow^I \pi_2 \longrightarrow ..... \longrightarrow^I \pi_* \longrightarrow^E v_*$$

5.1.2 Monte Carlo

Another way to evaluate a policy is to use a Monte Carlo (MC) method. These methods work only for episodic tasks, require a large sample of episodes, and do not require any knowledge of the environment dynamics. After generating a large sample, the estimation $v_\pi(s)$ is calculated by averaging the returns. Next, we provide the pseudocode of the **First Visit MC** algorithm, which, unlike every Visit MC, only considers the rewards after the first visit to state s.

Algorithm 5: First visit MC

Input: policy π
Initialize: $v(s)$ for all s and a list $rt(s)$ for each s.

loop
  Generate episode $S_0, A_0, R_0, S_1, A_1, R_1, S_2..., S_{t-1}, A_{T-1}, R_T$
  $G \longleftarrow 0$
  $t \longleftarrow T - 1$
  for t>=0 **do**
    $G \longleftarrow \gamma G + R_{t+1}$
    if $S_0, S_1, .., S_{t-1}$ do not contain S_t **then**
      put G in the list $rt(S_t)$
      $v(S_t) \longleftarrow mean(rt(S_t))$
    end if
    $t \longleftarrow t - 1$
  end for
end loop

So far, we have presented only policy algorithms. These are algorithms that depend on a policy. In some cases, using a method which is not dependent on any policy comes handy; such methods are called off-policy. An off-policy method is the so-called algorithm Q-learning.

5.1.3 Q-learning

To briefly introduce Q-learning, we give the following equation which is how the state-action value function is estimated.

$$Q(S_t, A_t) \longleftarrow Q(S_t, A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)) \tag{5.17}$$

Where α is a step size parameter: a scalar in the interval $(0, 1]$. To be precise, Q-learning bases its choices of actions on a policy derived from the action values Q. Usually, this policy is ε**-greedy** where a greedy action (with respect to the Q values) is selected with probability $1 - \varepsilon$ and with probability ε an action is selected at random. The Q learning algorithm starts by initializing all $Q[s, a]$ randomly for all state action pairs, except for the terminal states, for which it sets $Q(terminal, a) = 0$. Q-learning does not require knowledge of the environment, is not dependent on any specific policy, and is one of the most popular RL algorithms. The pseudocode of the algorithm is given below.

Algorithm 6: Q-learning

Initialize: $Q(s, a)$ randomly for all s, a pairs except for terminal states then Q(terminal,a)=0
  for each episode **do**
    Initialize S
    for each step of the episode **do**
      Select A from S using the policy formed by Q(usually ε-greedy).
      Take action A and keep the reward R and next state S'
      $Q(S, A) \longleftarrow Q(S_t, A_t) + \alpha(R + \gamma \max_a Q(S', a) - Q(S, A))$
      $S \longleftarrow$S'
    until S is terminal
  end for

Knowing the values $Q(s, a)$ for each state action couple, one can easily derive a policy simply by selecting the best action in every state value wise. This policy is obtained by using the following equation.

$$\pi(s) = arg \max_a Q(s, a)$$

5.1.4 Actor Critic Methods (AC)

Actor critic methods are popular methods that are in the middle of on and off policy algorithms. The idea behind AC is to combine estimating a policy and a value function at the same time. A typical AC method works as follows: The actor is said to be the policy, and the critic is said to be the value function. The policy acts and selects an action and then the value function, as the critic criticizes the action taken. Actor-critic algorithms are among the most popular algorithms for applying RL to solve COPs. In [Bello et al., 2016] an actor critic algorithm is used with a technique called function approximation, which is discussed in the next paragraph. There are many AC methods; here, we will give the pseudocode of the method described in [Sutton, 2022], which is fairly simple.

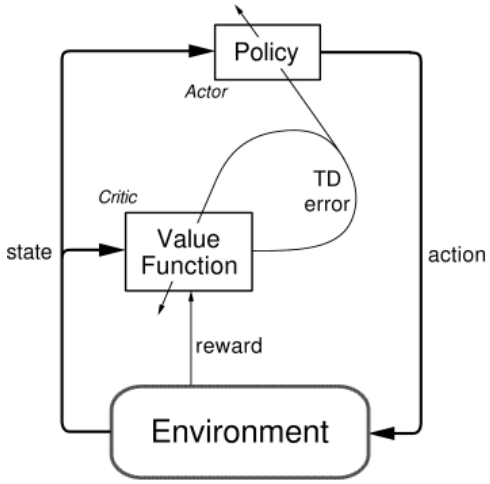

Figure 5.2: picture from [Sutton, 2022]

Algorithm 7: Actor Critic

1 Select action at step t according to current policy
2 Calculate TD error $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$
3 update preference values $p(s_t, a_t) = \pi(s_t, a_t) + \beta\delta_t$
4 generate $\pi_t(s, a) = \Pr\{a_t = a \mid s_t = s\} = \frac{e^{p(s,a)}}{\sum_b e^{p(s,b)}}$
5 update the critic (V) using some rule

The TD error δ_t in 2 is used in 3 to increase or decrease the probability of selecting the action a_t in state s_t. In 3 β is just another positive step size parameter. This method, like all the methods that we have discussed so far, cannot be efficient for problems with huge state spaces.

So far, all of the discussed RL methods estimate state value functions for each state separately and update constantly (one at each step of the algorithm). This is very time-consuming, memory-consuming, and practically infeasible in problems with large state spaces. In these problems, states occur for the first time in most steps. Apparently, it is difficult to make good decisions in such states, and thus basing the decision on previous experience with similar states is recommended. This is done using function approximation methods:

5.1.5 Function Approximation

As mentioned earlier, in the case of problems with massive state action spaces, all algorithms presented so far fall short in solving these problems, and solving this problem requires a function approximation method. RL and dynamic programming, as we have already mentioned, are used to solve such problems. Unfortunately, they

suffer from "the curse of dimensionality". That means that the computational cost increases exponentially with the number of state variables. This term was first used by Bellman in [Bellman, 1957]. To overcome this, function approximation in RL and Approximate Dynamic Programming (ADP) methods were developed. In fact, ADP and RL are not very different from each other, as can be seen in the article [Powell, 2009] and in the book [Bertsekas and Tsitsiklis, 1995]. Whether one calls these methods ADP or RL is pretty much based on which field they belong to. There are many function approximation methods. For example, gradient methods are quite popular. These methods try to minimize the "difference" between the approximate value function and the real value function by using gradient descent methods. As an approximation function, one can use any differentiable function from very simple functions, such as linear functions, to very sophisticated functions, such as state-of-the-art deep neural networks.

Let $\mathbf{w} \in \mathbb{R}^d$ be a vector of d weights and $v(s, \mathbf{w})$ be a function of w for all $s \in S$. The function v must be a differentiable function. Gradient methods need a set of state value examples to adjust the vector $\boldsymbol{w}$ so that the approximate function closely mimics the actual value function. To achieve this, gradient methods calculate gradients and update the vector in a series of steps. Usually, this happens every time an example is seen. The weights are updated by:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \frac{1}{2} a \nabla [V_\pi(S_t) - v(S_t, \mathbf{w}_t)]^2 \tag{5.18}$$

Where α is the learning rate parameter. As an approximate function, one can choose a linear function. A **linear approximate function** is a good choice because it performs well in some problems and it is easy to compute its gradients. In this case, there is a vector called the feature vector **x(s)** for every $s \in S$, where S is the state space. The feature vector and the weight vector are d dimensional. The linear function of our interest is defined with the help of the **x** and **w** vectors. That is, it is defined as their inner product:

$$v(s, w) = w^T x(s) = \sum_{i=1}^{i=d} w_i x_i(s) \tag{5.19}$$

Feature vectors form a linear basis for the space of approximate functions and describe the state. Each component of $x(s) = (x_1(s), ..., x_d(s))^T$ is called a feature of the state $x_i : S \to \mathbb{R}$. As mentioned above, the gradient of the linear function is easily calculated, which facilitates the SGD steps. The gradient of the function we

defined earlier with respect to the weights is:

$$\nabla v(s, \mathbf{w}) = x(s)$$

and therefore the weight update in this case becomes:

$$w_{t+1} = w_t + a[V_\pi(S_t) - v(S_t, w_t)]x(S_t) \tag{5.20}$$

Usually, it is difficult to obtain the value $V_\pi(s)$ of a state s. Therefore, an unbiased estimate is often used instead. V_t is an unbiased estimate of $V_\pi(S_t)$ if and only if

$$\mathbb{E}[U_t|S_t = s] = V(S_t)$$

For Monte Carlo, G_t is an unbiased estimate. Combining all of these, the Monte Carlo algorithm for estimating the approximate values becomes

Algorithm 8: Monte Carlo for estimating V_π (Linear approximate function)

Input: the policy π and
Input: a linear function $v(s, w)$ Initialize: the weight vector $w \in \mathbb{R}^d$ arbitrarily

loop
  Generate episode $S_0, A_0, R_0, ..., A_{t-1}, S_T$
  for t=0,...t=T-1 **do**
    $w_{t+1} = w_t + a[G_t - v(S_t, w_t)]x(S_t)$
  end for
end loop

5.1.6 Q learning with function approximation

Function approximation can also be used with off policy algorithms like Q-learning. We briefly present the next two algorithms with value function approximation, which are called fitted Q-iteration and online Q-iteration.

Algorithm 9: fitted Q-iteration

Input: q(s,a,w) function of w.
Step size parameter α
  for i=1,...,T **do**
    Collect D-samples$[(S_i, A_i, R_i, S'_i)]$
    for t=1,....,K **do**
      $y_i \longleftarrow R_i + \gamma(\max_a Q(S'_i, a, w))$
      $w \longleftarrow arg\min_w \frac{1}{2}\sum_{i=1}^{i=D}(Q(S'_i, A_i, w) - y_i)^2$
    end for
  end for

Algorithm 10: Online Q-iteration

Input: q(s,a,w) differentiable function of w.
Step size parameter α
  for i=1,...,T **do**
    pick a and observe (s,a,r,s')
    $y \leftarrow r + \gamma(\max_a Q(s'_i, a, w))$
    $w \leftarrow w - \alpha(Q(s, a, w) - y)\nabla_w Q(s, a, w)$
  end for

In algorithm 9, the parameter w, as we can see, is updated using samples of size D and the mean square error loss function, while in algorithm 19 the updates are made after each step using only the calculated target y and the observed value Q. This type of algorithm is used in the RL for the COP research line, for example in the following: [Khalil et al., 2017, Barrett et al., 2020]. In the first paper, fitted Q learning is used with the addition of a technique called experience replay, which we discuss below. The second work uses a similar algorithm, which is also discussed below.

5.1.7 Deep Q-learning network (DQN)

DQN was created by a DeepMind's engineering team. It was introduced in [Mnih et al., 2013] where it tackles the atari2600 video game. For such problems before DQN, all algorithms were well-designed by human experts in the game-task. On the contrary, DQN is capable of performing good tasks without human-designed features. For instance, DQN receives only an image and a current score at each step of atari, and using only this information performs well. The algorithm is probably the most complex in this text. Instead of presenting the algorithm in an end-to-end way, we briefly describe the basic and most important components of the algorithm.

Replay Buffer is a list that contains tuples of previous transitions. The buffer has a fixed capacity and stores the previous experience in order to provide samples for the process of experience replay.

Experience Replay is feeding the algorithm randomly sampled mini-batches of transitions from the replay buffer to train it. Experience replay enhances the algorithm's learning ability.

Q-network and target network The two networks are responsible for different functions of DQN. The first is responsible for the prediction of the q values and instantiates the q-learning function, while the latter is responsible for the generation of the target Q values. The parameters of the target network are updated every C steps. That is, the parameters of the updated network are set to be equal to the ones of the Q-network. It is possible not to use the second network and use only the first one for both functions, but it is not recommended since the setting of two networks appears to perform better. In the following, we give the pseudocode of the algorithm as in [Dong et al., 2020].

Algorithm 11: DQN

Hyper Parameters: replay buffer capacity N, discount ε greedy factor.
Input: empty replay buffer D, initial parameter w of Q-network (Q function).
Initialize : $\hat{Q}$ target network with the parameters of Q-network

for episode i=0,1,2,... **do**
  initialize environment and get initial observation O_0
  initialize sequence $s_0 = [0_0]$ and pre-process $\varphi_0 = \varphi(s_0)$
  for t=0,1,2,3... **do**
    select action A_t at random with probability ε and
    $A_t = arg\max_a Q(\varphi(S_t), a, w)$
    Take action A_t and observe O_{t+1} and reward R_t.
    if episode has ended set $D_t = 1$ if it has not set $D_t = 0$
    $s_{t+1} = (s_t, A_t, 0_{t+1})$ and preprocess $\varphi_{t+1} = \varphi(s_{t+1})$
    save the tuple $(\varphi_t, A_t, R_t, D_t, \varphi_{t+1})$ in D
    sample a minibatch of transitions from D $(\varphi_i, A_i, R_i, D_i, \varphi'_i)$
    if $D_i = 0$, assign $Y_i = R_i + \gamma \max_{a'} \hat{Q}(\varphi'_i, a', \hat{w}), else$,$y_i = R_i$
    do a gradient descent step on $(Y_i - Q(\varphi(S_i), A_i, w)^2$
    Update the target network every C steps
    if the episode has ended then break the loop
  end for
end for

In algorithm 11, we assume that we have a partially observable Markov decision

process and that is why we have those Os in the algorithm. Partial observations are not in our interest so they are not presented in this work. However, for more information on the algorithm and its coding implementations, see [Dong et al., 2020] and [Zai and Brown, 2020]. Clearly, algorithms such as DQN, which use experience buffers, require a lot of memory. Moreover, experience replay increases the number of hyperparameters.

Chapter 6

SOLVING COMBINATORIAL OPTIMIZATION PROBLEMS VIA MACHINE LEARNING

As the title of the chapter suggests, the current chapter aims to present applications of machine learning in the field of COP. Recently, this topic has been a very active area of research, and many articles have been published. Most of them use supervised learning or RL in order to devise frameworks capable of solving COPs; unsupervised learning is rarely used. Applications in this field vary. There are some direct applications, such as constructing solutions to COPs, and some indirect, such as improving processes of a COP solver. Regarding their ability to solve multiple instances of problems, models are classified into two categories: **Single Instance learning** models and **Multi Instance Learning** models.

Single instance learning models are the models that are trained to solve a single instance of a COP. These models are ideal when the solver has no reason to want their model to be able to solve other instances with or without the same size. In that setting, for every different instance, a new policy has to be learned, which means that the training process has to restart. multi instance learning models, in contrast with single-instance learning models, can solve multiple instances of a COP. Furthermore, a distinction between direct models and indirect models can be made based on whether a model directly gives a solution to a COP or not. Direct models can be divided into those that are **autoregressive** and those that are non-autoregressive. Apparently, the type of machine learning used to train a model is another dividing characteristic. Usually, SL or RL is used. Although in the end of this chapter we briefly discuss some indirect applications of ML in CO, our focus and our experiments are mainly on direct applications. In [Bengio et al., 2021] more about the spectrum of ML applications in CO can be found. Direct applications can vary greatly. Nevertheless, the following series of steps can provide a rough description of direct ML methods in CO :

- Learn functions for graph representations
- Produce node or edge embeddings
- Using these embeddings learn a policy or a value function
- Use a search algorithm to get a solution

These models can be trained using SL or RL. Obviously, SL requires obtaining a significant number of targets, whereas RL does not. Usually, SL is used in a non-autoregressive way, in contrast to RL. To briefly describe how these models work, we begin with the figures below.

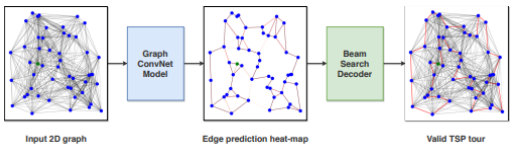

Figure 6.1: [Joshi et al., 2019]

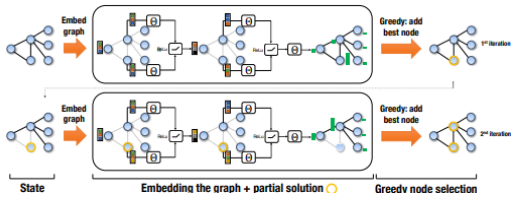

Figure 6.2: Illustration from [Khalil et al., 2017]

Figure 6.1 shows how the [Joshi et al., 2019] framework works. This framework devises a non-autoregressive model trained by supervised learning, which constructs solutions for TSP. As we can see, a two-dimensional graph is input, and a GCN outputs an adjacency matrix containing probabilities of each edge belonging to the tour. Then a solution is given by using beam search. The model is trained by minimizing the cross entropy loss. The ground truth tours for the input graphs are transformed to 0-1 adjacency matrix, so, the cross-entropy loss can be computed.
Figure 6.2 shows how the [Khalil et al., 2017] framework works for the MVC problem. This approach is an autoregressive RL model, which constructs solutions by adding nodes sequentially to a partial solution. A message-passing neural network takes graphs as input and produces graph embeddings and node scores. Then, a node is selected greedily (i.e., the node with the highest score is selected). This happens until all edges are covered. They combine graph embeddings and Q-learning to construct a greedy-learned heuristic.

The latter framework is used as a means of comparison in many other works that follow that line of research. Therefore, it is a fair representative of frameworks belonging to **RL-paradigm**. In fact, most of the work in the line of RL- autoregressive is quite similar to that. Another work worth mentioning is the [Barrett et al., 2020]

framework, which deals with the maximum cut problem. This works similarly to [Khalil et al., 2017], but introduces a new technique. They allow the agent to reverse a previously taken action. Apparently, using this technique is more time-consuming as each episode takes longer to end. Similarly, they use a message-passing neural network and DQN to combine graph embeddings and Q-learning.

In this research line, providing an RL environment for a COP is needed. As an example, one can see how this is done in [Khalil et al., 2017]. The table below contains the definition of states, actions, rewards, and termination criteria.

Components				
Problem	State	Action	Reward	Termination
MVC	the subset of nodes selected so far	append node to subset	-1	all edges are covered
Max Cut	the subset of nodes selected so far	append node to subset	change in cut weight	cut weight cannot be improved
TSP	the partial tour	add a node to partial tour	change in tour cost	tour includes all nodes
MIS	the subset of nodes selected so far	append node to subset	+1	there are no more feasible additions

The first three rows are the same as in [Khalil et al., 2017], while the last row is how we formulate MIS in our experiment. We could have included in the table above other prototypical COPs, but we decided to keep those that we are more interested in. Typically, feature engineering has an impact on the performance of ML models. In this research area, though, we believe feature engineering is not the center of attention. However, some common features are the following: graph statistics in graph COPs like node degree, coordinates (x, y) for euclidean TSP, 0-1 tag depending on whether the node belongs to the partial solution, random features to characterize nodes, and the number of nodes in lesser distance than a fixed number, or some more complicated features. For instance, in [Barrett et al., 2020] they use the features

1. Vertex inclusion, if it is in the current solution
2. Immediate cut change in case the vertex is changed
3. Number of steps since the last state was changed
4. Difference of current cut value from the best observed
5. Distance of the current solution set from the best observed
6. Number of available actions that immediately increase the cut-value

7. Steps remaining in the episode

Only observations 4-7 are global

Training and evaluation data are generated using algorithms that have been designed to generate instances of COPs. For the graph COPs the situation is simple, as the norm is to use synthetic data from random graph generation models such as Erdos-Renyi. In the evaluation stage, real data is also often used. Usually, real data comes from publicly available sources such as TSPLIB [Reinhelt, 2014]. Furthermore, the most common way to measure the performance of a model is to use the approximation ratio averaged over all instances in the evaluation dataset. For the evaluation of a model, producing targets is needed. So, some well-known solver is used to produce the targets. A time limit is often applied, so there is no guarantee of optimality.

Learning to Branch or Cut

Having mentioned some methods for using machine learning to solve COPs directly, we discuss an indirect way in which ML is used in the same context. In chapter two, we discuss solution methods in CO, specifically about branch and bound and the decision problems arising in its implementation. Tackling these decision problems with the assistance of ML systems is another research line in the context of this thesis. There are approaches in which an agent tries to learn policies for adding effective cutting planes in the cutting plane algorithm or branching effectively in the branch and bound algorithm. An example of such an approach can be found in [Gasse et al., 2019], where a graph convolutional neural network model is used for learning variable selection policies. This model is trained using a technique called imitation learning over the strong branching rule. It managed to outperform the previous ML-branching attempts and the default branching mode of SCIP [Achterberg, 2009], which is an open source MIP solver. Apart from the good results, the model performed well on problem instances much larger than those used in the training phase.

In the following, we provide some technical details of this approach. The MILP solver plays the role of the environment, and the branching rule plays the role of the agent. The transitions between two successive states occur as follows:

- The agent selects a fractional variable at the focused node with respect to a policy π

- Then the solver branches on the node, solves the respective LP relaxations, runs its internal heuristics, and prunes the tree in case that is needed
- Picks the next leaf to split and transits to the next state to repeat that process again.

In that framework, the state s_t of the solver compounds of the BB tree, past decisions, and a collection of other properties. The state is encoded as a bipartite graph with node and edge features. The policy, as already mentioned, is parameterized as a graph convolutional neural network and takes as input the state encoded as a graph. Another similar approach is [Sun et al., 2020]. In this article, the authors suggest that imitation learning over the strong branching rule is not the best idea, and they follow a different way.

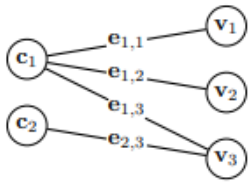

Figure 6.3: Representation of a state [Gasse et al., 2019]

The figure depicts how a state with two constraints and three variables is represented as a bipartite graph. On the left side of the graph, the nodes represent constraints. On the right, they represent variables. An edge joins two nodes on opposite sides (constraint variable) only if the constraint contains the variables.

Learning to cut

We now discuss [Tang et al., 2020], which was published later than "learning to branch". This approach devises an RL agent that learns to add cutting planes effectively in an adaptive way. The devised agent is used as a subroutine in the cutting-plane method, which we have already referred to in chapter 2. The exact cutting plane method used is the Gomory cutting plane; however, its creators suggest that it can be used in branch and bound as well leading to improvements. Another impressive property of that approach is that the agent, apart from having the ability to generalize well to larger problem instances, is also able to generalize from one problem class to another. That work can be considered pioneering, as no previous work formulated the process of selecting cutting planes as an MDP. Therefore, it is worth going deeper into the technical details of this framework.

The state is encoded as $s_t = \{C^{(t)}, c, x^*_{LP}(t), D^{(t)}\}$. Where $C^{(t)}$ is the feasible region of the new LP in the time step t. $x^*_{LP}(t)$ is the optimum solution of LP and D^t is the set of Gomory cuts. The action set consists of all cuts in D^t that are available to be added in the next step. The agent selects an action by selecting a cut; the new state is equal to the union of the previous state and the selected cut. The reward is the gap between the objective values of two consecutive LPs. There are many ways to evaluate the quality of a cutting plane method. In the specified work, a measure called the integral closure gap is used. This measure is the following ratio.

$$\frac{g^0 - g^t}{g^0} \in [0, 1]$$

where $g^t = z^*_{IP} - z^*_{LP}$.

For the RL part, an on-policy algorithm is chosen. The policy is parameterized by an LSTM neural network with attention. Lastly, policy optimization is done using an algorithm called Evolution Strategies (ES) as in [Sun et al., 2020]. ES is a good alternative to the usual RL algorithms. It is easier to use in distributed settings and has fewer hyperparameters.

Chapter 7

EXPERIMENTS

Having discussed the ways of applying ML for CO, we are ready to move on to the last chapter. In the last chapter, we present some of the experiments we ran. These experiments belong to a line of work that we described in the previous chapter. Specifically, they belong to the RL paradigm. We develop a framework which solves the following problems MVC, MIS and TSP. The framework we developed is similar to that developed from [Khalil et al., 2017]. This framework acted as a role model. In algorithmic design, both frameworks construct solutions by sequentially adding nodes to a partial solution and apply greedy search. Algorithm training is similar, too, since fitted Q-learning is used in [Khalil et al., 2017] and DQN in ours. Moreover, graph neural networks for function approximation are used in both cases. That said, our framework is less sophisticated and efficient than that framework. This is due to some differences in training methods, less optimized code, and less computational power. As a means of comparison, we use the framework of [Orrivlin, 2019]. This solves MVC using an actor-critic algorithm. Moreover, we extended it, so it can solve MIS and TSP.

7.1 Training Details

As we already mentioned, we use DQN with a graph neural network in our experiments. Our approach is similar to other approaches, as mentioned previously, but has some differences. For the function approximation part, we use a graph neural network. Specifically, a graph convolutional network for MVC and MIS problems, which is almost identical to [Orrivlin, 2019] with the only difference being the number of convolutional layers. In our approach, we stack 2 layers, whereas in [Orrivlin, 2019] they stack 3. For the TSP, we choose a slightly different architecture. We use the GCN architecture that we use for MVC and MIS, but this time we stack another three gated recurrent layers taken from [Dwivedi et al., 2020]. For the TSP, as we have mentioned, we use a combination of GCN and gated GCN layers. The gated GCN layer that we use works as follows:

$$h_i^{\ell+1} = h_i^{\ell} + \mathrm{ReLU}\left(\mathrm{BN}\left(U^{\ell} h_i^{\ell} + \sum_{j \in \mathcal{N}_i} e_{ij}^{\ell} \odot V^{\ell} h_j^{\ell}\right)\right)$$

where the edge gates e_{ij} are computed by the following equations:

$$e_{ij}^{\ell} = \frac{\sigma\left(\hat{e}_{ij}^{\ell}\right)}{\sum_{j' \in N_i} \sigma\left(\hat{e}_{ij'}^{\ell}\right) + \varepsilon}$$

$$\hat{e}_{ij}^{\ell} = \hat{e}_{ij}^{\ell-1} + \mathrm{ReLU}\left(\mathrm{BN}\left(A^{\ell} h_i^{\ell-1} + B^{\ell} h_j^{\ell-1} + C^{\ell} \hat{e}_{ij}^{\ell-1}\right)\right)$$

Where $U,V,A,B,C\ b \in \mathbb{R}^{dxd}$ and BN stands for batch normalization.

We use DQN with GCN or gatedGCN as follows. We work with ε-greedy policy. Where ε decays exponentially from 1 to 0.05. We use mini-batches of 64 size. Regarding the learning rate, many values performed fine. Usually, a learning rate in the interval $(0.0005 - 0.001)$ would be good. Unlike the original DQN, we update the base network parameters every k steps, where k is a fixed number. This alternation is also used in [Barrett et al., 2020] and appears to improve training and save time. The methodology we follow to train our agents can be summarized in the following algorithm.

Algorithm 12: Algorithm for Training

```
Initialize parameters such as experience replay buffer capacity,
for e in episodes do
    get a COP instance environment.
    Initialize state 𝕊 = ()
    for time step t in e do
        u_t = { random node u in A w.p ε
              { \argmax_u Q(u, S_t, W) w.p 1-ε
        add u_t to the partial solution
        add tuple (S_t, u_t, R_t, S_{t+1}) to memory buffer
        if t mod k = 0 then
            sample a minibatch from memory buffer
            perform an SGD step
    end for
end for
return parameters W
```

Where w.p stands for: with probability. As already mentioned, we perform SGD steps every k steps instead of every step. That is why there is an IF statement in line 9. Every episode ends according to a termination criterion. You can see the termination criteria that we use in table 7.2. The Q function is instantiated by neural networks, as previously described in this section.

7.2 Training and evaluation data

Training and evaluation data is generated using the Erdos-Renyi model. We use graphs of 20 nodes to train agents for MIS and MVC. For TSP, we use graphs of 10 nodes. Our training and evaluation data sets are homogeneous in terms of graph size, in contrast to [Khalil et al., 2017], [Barrett et al., 2020] and other approaches that use graph evaluation sets of various sizes. We use a method to create an RL-environment for each COP task. This method creates a graph for each episode. This graph is accessible as long as it remains in the replay buffer. The performance of our model is evaluated on graph lists of $20, 50$ vertices for MVC, MIS and 10, 30 vertices for TSP.

7.3 Results

In this subsection, we present the results of our experiments. As you will see, we use the following performance measures: approximation ratio and mean return in the evaluation stage. We compare our algorithms with [Orrivlin, 2019] and a classic heuristic for each problem. We also diagnose how the training phase went using visualizations of returns during training or rolling means of returns (with 50 episodes windows). We also present the results of agents trained by the actor critic of [Orrivlin, 2019]. We note again that the framework above works only for MVC, we provide our own environment methods for other problems, and we have made some minor changes in the AC algorithm for MIS. In addition, we describe an experiment we conducted on a single instance of KP using the python OR-gym package [Hubbs et al., 2020].

7.3.1 MVC

Here, we present the results for MVC. As we mentioned:the agents are trained on graphs of 20 vertices that are generated from the Erdos-Renyi model with connection probability 0.15. The episodes terminate when all edges are covered. We can see from the figures below that both agents are able to learn. As you can see from the tables, the AC agent performs better. However, both agents perform better than the human-designed heuristic.

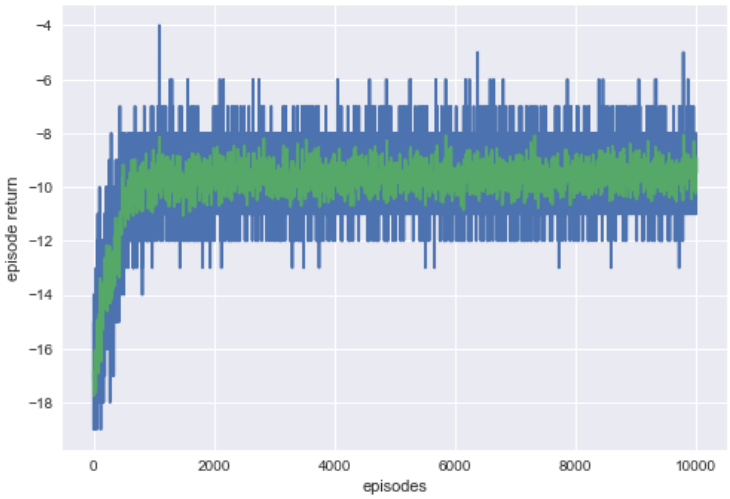

Figure 7.1: Returns during AC agent training

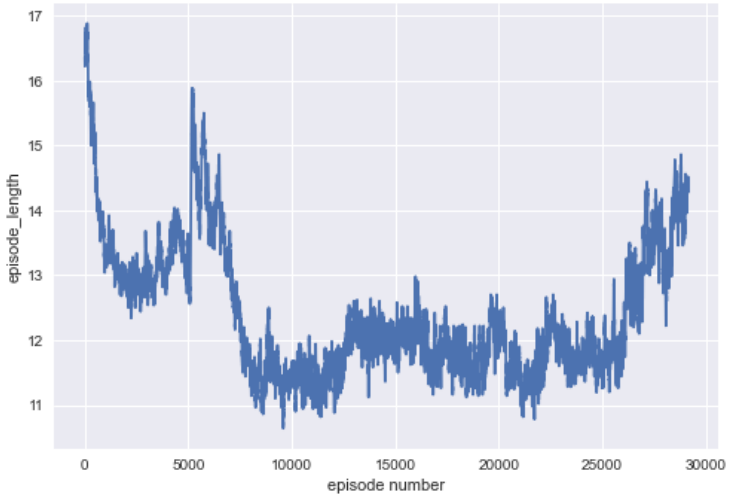

Figure 7.2: Duration of episodes during the training phase of the DQN algorithm for the MVC problem

Method	Approx-ratio	mean-cover-size
CPLEX	1	9.57
AC-3layered-gcn	1.0024	9.59
DQN-2layered-gcn	1.1585	11.08
node-deg	1.38	13.38

Table 7.1: Performance evaluation on 20-vertex graphs

The table above shows the results on 20-node graphs. Whereas, below, you can find the results on 50 node-graphs.

Method	Approx-ratio	mean-cover-size
CPLEX	1	32.69
AC-3layered-gcn	1.0091	32.99
DQN-2layered-gcn	1.1548	37.740
node-deg	1.3415	43.82

Table 7.2: Performance evaluation on 50-vertex graphs

7.3.2 MIS

We move on to the MIS problem, where the two agents perform about equally well. The framework is similar to that of MVC. Here, the termination criterion is: to terminate the episode when there is no node that can join the partial solution without violating the problem's constraints. In order to get the available nodes to join, a more complex function is used compared to the one for the previous problem. As you can see from the visualizations and tables below: 1) the AC agent quickly learns to produce good solutions and then remains stable, whereas the DQN agent takes more time, and its performance fluctuates and eventually declines during the training. 2) Testing them in the evaluation data set, we see that the DQN agent performs slightly better than the AC agent. 3) Both agents are better than the simple human-designed heuristic of min degree.

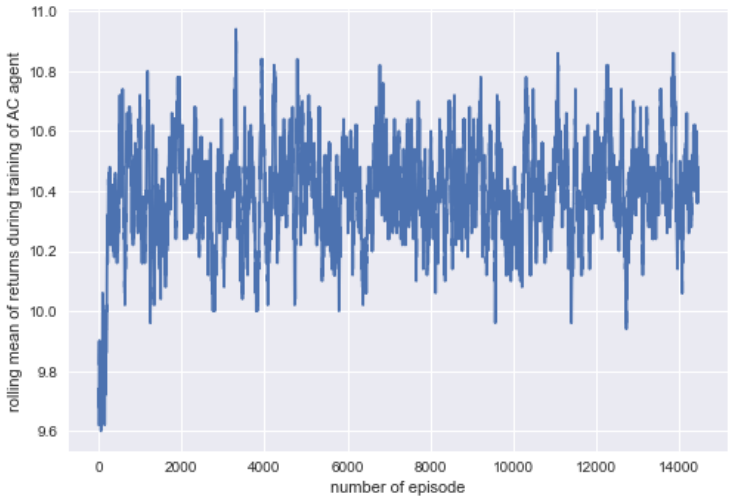

Figure 7.3: Rolling mean of returns during actor critic agent training (time window = 50)

Figure 7.4: Rolling mean of returns during training of DQN agent (time window=50)

Method	Approx-ratio	mean-size-MaxIndSet
CPLEX	1	10.42
AC-gcn	0.987	10.30
DQN-gcn	0.9941	10.364
MinDegHeuristic	0.9652	10.064

Table 7.3: Performance evaluation on 50-vertex graphs-MIS

Method	Approx-ratio	mean-size-MaxIndSet
CPLEX	1	17.51
AC-gcn	0.936	16.39
DQN-gcn	0.9475	16.57
MinDegHeuristic	0.909	15.92

Table 7.4: Performance evaluation on 50-vertex graphs-MIS

7.3.3 TSP

Here we present the results of our TSP experiments. As we did with previous problems, we train 2 agents, one with AC and one with Q-learning (DQN). However, this time the situation is different. Our DQN agent fails to learn a competent heuristic. We believe that this is due to insufficient hyperparameter tuning. Due to the fact that the solutions given by DQN are no better than randomly drawn tours, we decided not to include DQN.

The RL-environment method we created differs slightly from the ones for the previous problems. In every episode, the initial partial solution contains node 0 or the base city. Most of the relevant work on TSP draws distances between cities-nodes from the real number interval *(0,1)*. In contrast, apart from drawing distances

from *(0,1)*, we use the discrete uniform distribution in *{1,2,3,4,5,6,7,8,9,10 }* and therefore two vertices can have a distance with $1 \leq x \leq 10$ where x is an integer.

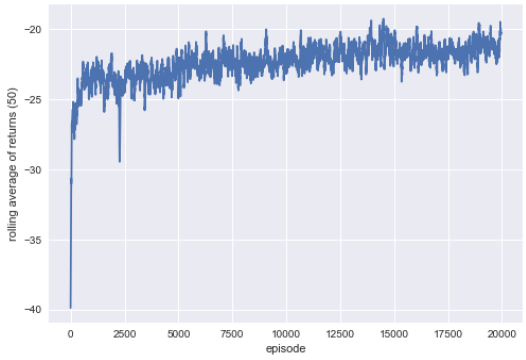

Figure 7.5: rolling mean of returns during training of our model(time window=50)

The last figure shows that the agent improves rapidly in the first 3000 episodes and then very gradually. The best rolling means are slightly above -20, which means that the best tours found during training have a total cost of slightly less than 20. We test the trained model using 250 complete graphs randomly generated. The edges are generated in the same way as those from the training. In order to evaluate the model, we have computed the optimal tours for each of the test graphs. We use multiple metrics to diagnose how the agent performs. In addition, a very popular and simple human-designed heuristic is used as a baseline. The heuristic is the nearest neighbor described in chapter 2. The performance of the agent is shown in the next table.

Method	Approx-ratio	mean-cost-tour
CPLEX	1	24.784
AC-gatedgcn	1.113	27.668
near-neighbour	1.222	30.036

Table 7.5: Performance evaluation on 10-vertex graphs

In table 7. we see that the learned heuristic performs better than the other heuristic.

Then we check how the model generalizes to larger instances. For the evaluation,

Method	Approx-ratio	mean-cost-tour
CPLEX	1	36.94
AC-gatedgcn	1.532	56.48
near-neighbour	1.436	53.02

Table 7.6: Performance evaluation on 30-vertex graphs

we used 50 complete graphs of 30 vertices.

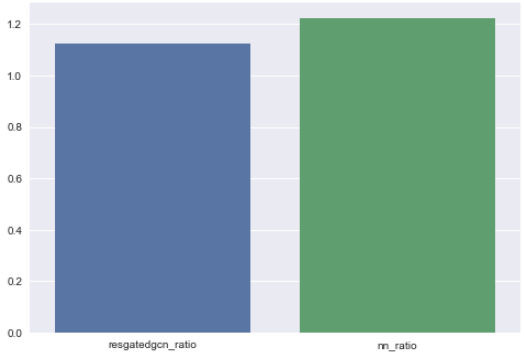

Figure 7.6: approximation ratio

7.3.4 Solving single instances of COPs using OR-gym package

Apart from these three experiments, we experimented with recently developed higher-level packages, which are intended to simplify the procedure of running ML experiments in CO. We used two packages: the OR-gym package [Hubbs et al., 2020], and ECOLE [Prouvost et al., 2020], which stands for Extensible Combinatorial Optimization Learning, and formulates decision problems in MILP solvers as RL-environments, such as variable selection in branch and bound method.

Here, we present a higher-level approach using the OR-gym and ray [Moritz et al., 2018] packages. This experiment was quite different in implementation from the ones described previously. OR-gym is a collection of RL environments for COPs. Some COPs for which they provide environments are KP, TSP, VRP, portfolio optimization, and others. As an example usage of the package, we will show the results of solving a very small KP instance. We chose to use a small instance because when we tried to use the default instance of OR-gym, we ran out of memory. Several variants of KP are provided in the package; we selected the unbounded KP.

we describe the instance in table 7.7.

Number of Items	5
capacity of kp	15
Item values	(2, 4, 2, 1, 10)
item weights	(1, 12, 2, 1, 4)

Table 7.7: Description of the KP instance

We train two agents: 1) using an on-policy algorithm called PPO, which stands for proximal policy optimization and is not covered in our work. Those interested

in the algorithm can check [Dong et al., 2020]. 2) DQN, which we use for the multi-instance experiments, also.

In this experiment, everything that has to do with the agent's properties and training (such as function approximators, algorithm hyperparameters, and optimization of functions) is managed completely by ray without any intervention by us. Next, we visualize the results.

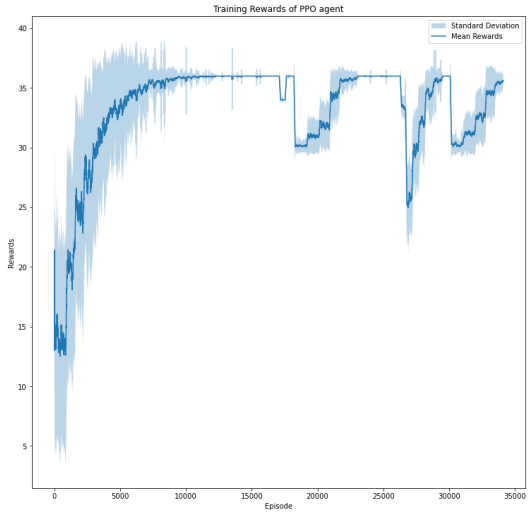

Figure 7.7: Episode rewards of the PPO agent

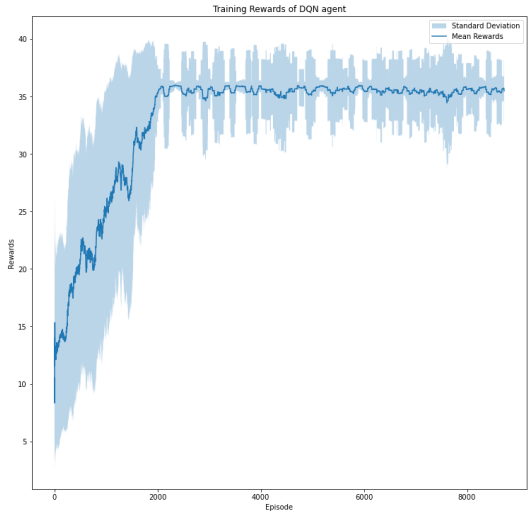

Figure 7.8: Episode rewards of the DQN agent

7.4 Conclusions

We developed a framework that uses the RL- algorithm DQN in order to solve three different graphs COPS, MVC, MIS and TSP. In addition, we extend a framework for MVC to work for MIS and TSP. We benchmark these two frameworks. These frameworks follow the so-called RL-paradigm, and their performance indicates that a machine learning model can learn a competent heuristic for COPs. While, for MVC and MIS, our training was successful and resulted in the agent learning a decent heuristic, for TSP, our training failed. That is, training for TSP resulted in a model that produces tours that are not better than random tours. That probably would not have happened if we had enough computational power to do a competent hyperparameter tuning. Apart from that, we have to note that training for TSP takes much more time than training for MVC and MIS. Therefore, hyperparameter tuning is more difficult when dealing with TSP. From our single-instance experiment on KP, we can conclude that using higher-level libraries saves much time and is more convenient. However, it comes with a cost in flexibility.

7.5 Limitations

In this paragraph, we describe the limitations of our work. First, the running time of the training phase is quite high. This is due to two reasons: our code is far from being optimized and we do not have enough computational power. Another hardware limitation that we faced was insufficient memory, as we ran all the experiments using the computational power of an obsolete laptop. Furthermore, limitations forced us to compromise training and experimentation. Specifically, we trained our models using only fixed-size graphs, which possibly leads to worse results. We tested their capabilities only on two lists of graphs with a fixed size, which is not enough to check the generalization abilities of the models. Insufficient memory hinders us from running experiments freely with or-gym and ray because we run out of memory when we try to solve large instances.

7.6 Future work

In the future, we will try to overcome the limitations that we face. Specifically, we will try to train our models using varying sizes of COPs. This requires reducing the amount of time that training takes. Also, we only applied two distinct RL algorithms (DQN and actor-critic); it will be interesting to use some other algorithms and compare their results and training time. Furthermore, we would like to follow the progress of the high-level packages for integrating machine learning and

combinatorial optimization and use them for experimentation.

BIBLIOGRAPHY

[Achterberg, 2009] Achterberg, T. (2009). Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41.

[Applegate et al., 2006] Applegate, D., Bixby, R., Chvatal, V., and Cook, W. (2006). Concorde tsp solver.

[Applegate et al., 2011] Applegate, D. L., Bixby, R. E., Chvátal, V., and Cook, W. J. (2011). The traveling salesman problem. In *The Traveling Salesman Problem*. Princeton university press.

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[Barrett et al., 2020] Barrett, T., Clements, W., Foerster, J., and Lvovsky, A. (2020). Exploratory combinatorial optimization with reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 3243–3250.

[Bellman, 1957] Bellman, R. E. (1957). *Dynamic Programming*. Princeton University Press.

[Bello et al., 2016] Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2016). Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*.

[Bengio et al., 2021] Bengio, Y., Lodi, A., and Prouvost, A. (2021). Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421.

[Bertsekas and Tsitsiklis, 1995] Bertsekas, D. P. and Tsitsiklis, J. N. (1995). Neuro-dynamic programming: an overview. In *Proceedings of 1995 34th IEEE conference on decision and control*, volume 1, pages 560–564. IEEE.

[Bertsimas and Tsitsiklis, 1997] Bertsimas, D. and Tsitsiklis, J. N. (1997). *Introduction to linear optimization*, volume 6. Athena Scientific Belmont, MA.

[Cho et al., 2014] Cho, K., Van Merriënboer, B., Bahdanau, D., and Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*.

[Cook, 1971] Cook, S. A. (1971). The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158.

[Dantzig et al., 1954] Dantzig, G., Fulkerson, R., and Johnson, S. (1954). Solution of a large-scale traveling-salesman problem. *Journal of the operations research society of America*, 2(4):393–410.

[Dong et al., 2020] Dong, H., Dong, H., Ding, Z., Zhang, S., and Chang (2020). *Deep Reinforcement Learning*. Springer.

[Dwivedi et al., 2020] Dwivedi, V. P., Joshi, C. K., Laurent, T., Bengio, Y., and Bresson, X. (2020). Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*.

[Gasse et al., 2019] Gasse, M., Chételat, D., Ferroni, N., Charlin, L., and Lodi, A. (2019). Exact combinatorial optimization with graph convolutional neural networks. *Advances in Neural Information Processing Systems*, 32.

[Gilbert, 1959] Gilbert, E. N. (1959). Random graphs. *The Annals of Mathematical Statistics*, 30(4):1141–1144.

[Gomory, 1960] Gomory, R. (1960). An algorithm for the mixed integer problem. Technical report, RAND CORP SANTA MONICA CA.

[Hart et al., 1987] Hart, O., Holmstrom, B., and Bewley, T. (1987). Advances in economic theory. In *World Congress*.

[Held and Karp, 1962] Held, M. and Karp, R. M. (1962). A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied mathematics*, 10(1):196–210.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

[Hopfield and Tank, 1985] Hopfield, J. J. and Tank, D. W. (1985). "neural" computation of decisions in optimization problems. *Biological cybernetics*, 52(3):141–152.

[Hubbs et al., 2020] Hubbs, C. D., Perez, H. D., Sarwar, O., Sahinidis, N. V., Grossmann, I. E., and Wassick, J. M. (2020). Or-gym: A reinforcement learning library for operations research problems. *arXiv preprint arXiv:2008.06319*.

[Joshi et al., 2019] Joshi, C. K., Laurent, T., and Bresson, X. (2019). An efficient graph convolutional network technique for the travelling salesman problem. *arXiv preprint arXiv:1906.01227*.

[Karmarkar, 1984] Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311.

[Karp, 1972] Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer.

[Khalil et al., 2017] Khalil, E., Dai, H., Zhang, Y., Dilkina, B., and Song, L. (2017). Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30.

[Kipf and Welling, 2016] Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

[Land and Doig, 1960] Land, A. H. and Doig, A. G. (1960). *Econometrica*, (3):497–520.

[Matai et al., 2010] Matai, R., Singh, S. P., and Mittal, M. L. (2010). Traveling salesman problem: an overview of applications, formulations, and solution approaches. *Traveling salesman problem, theory and applications*, 1.

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

[McClelland and Rumelhart, 1987] McClelland, J. L. and Rumelhart, D. E. (1987). *Schemata and Sequential Thought Processes in PDP Models*, pages 7–57.

[Merkwirth and Lengauer, 2005] Merkwirth, C. and Lengauer, T. (2005). Automatic generation of complementary descriptors with molecular graph networks. *Journal of chemical information and modeling*, 45(5):1159–1168.

[Mnih et al., 2013] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

[Moritz et al., 2018] Moritz, P., Nishihara, R., Wang, S., Tumanov, A., Liaw, R., Liang, E., Elibol, M., Yang, Z., Paul, W., Jordan, M. I., et al. (2018). Ray: A distributed framework for emerging {AI} applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 561–577.

[Orrivlin, 2019] Orrivlin (2019). Minimumvertexcover-drl. `https://github.com/orrivlin/MinimumVertexCover_DRL`.

[Pisinger and Toth, 1998] Pisinger, D. and Toth, P. (1998). Knapsack problems. In *Handbook of combinatorial optimization*, pages 299–428. Springer.

[Powell, 2009] Powell, W. B. (2009). What you should know about approximate dynamic programming. *Naval Research Logistics (NRL)*, 56(3):239–249.

[Prouvost et al., 2020] Prouvost, A., Dumouchelle, J., Scavuzzo, L., Gasse, M., Chételat, D., and Lodi, A. (2020). Ecole: A gym-like library for machine learning in combinatorial optimization solvers. *arXiv preprint arXiv:2011.06069*.

[Reinhelt, 2014] Reinhelt, G. (2014). {TSPLIB}: a library of sample instances for the tsp (and related problems) from various sources and of various types. *URL: http://comopt. ifi. uniheidelberg. de/software/TSPLIB95*.

[RENYI, 1959] RENYI, E. (1959). On random graphs. *PublicationesMathematicate*, 6:290–297.

[Stangor and Walinga, 2014] Stangor, C. and Walinga, J. (2014). *Introduction to psychology*. BCcampus.

[Sun et al., 2020] Sun, H., Chen, W., Li, H., and Song, L. (2020). Improving learning to branch via reinforcement learning.

[Sutton, 2022] Sutton, R. S. (2022). Actor-Critic Methods — incompleteideas.net. http://www.incompleteideas.net/book/ebook/node66.html. [Accessed 01-Aug-2022].

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

[Tang et al., 2020] Tang, Y., Agrawal, S., and Faenza, Y. (2020). Reinforcement learning for integer programming: Learning to cut. In *International conference on machine learning*, pages 9367–9376. PMLR.

[Vesselinova et al., 2020] Vesselinova, N., Steinert, R., Perez-Ramirez, D. F., and Boman, M. (2020). Learning combinatorial optimization on graphs: A survey with applications to networking. *IEEE Access*, 8:120388–120416.

[Waterloo, 2016] Waterloo, M. (2016). A shortest-possible walking tour through the pubs of the united kingdom.

[Wikipedia, 2022a] Wikipedia (2022a). Graph theory — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Graph\%20theory&oldid=1091469158. [Online; accessed 01-August-2022].

[Wikipedia, 2022b] Wikipedia (2022b). Independent set (graph theory) — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Independent\%20set\%20(graph\%20theory)&oldid=1082877025. [Online; accessed 01-August-2022].

[Wikipedia, 2022c] Wikipedia (2022c). Recurrent neural network — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Recurrent\%20neural\%20network&oldid=1106626225. [Online; accessed 06-September-2022].

[Wolsey, 2020] Wolsey, L. A. (2020). *Integer programming*. John Wiley & Sons.

[Zai and Brown, 2020] Zai, A. and Brown, B. (2020). *Deep reinforcement learning in action*. Manning Publications.

[Zhang et al., 2021] Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. (2021). Dive into deep learning. *arXiv preprint arXiv:2106.11342*.

INDEX