IMPLEMENTATION AND EVALUATION OF A DEEP REINFORCEMENT LEARNING MODEL FOR COMBINATORIAL OPTIMIZATION PROBLEMS

Tiago Carvalho Gomes Montalvão

Dissertação de Mestrado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Mestre em Engenharia de Sistemas e Computação.

Orientador: Daniel Ratton Figueiredo

Rio de Janeiro
Dezembro de 2024

IMPLEMENTATION AND EVALUATION OF A DEEP REINFORCEMENT LEARNING MODEL FOR COMBINATORIAL OPTIMIZATION PROBLEMS

Tiago Carvalho Gomes Montalvão

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Orientador: Daniel Ratton Figueiredo

Aprovada por: Prof. Daniel Ratton Figueiredo
Prof. Gerson Zaverucha
Prof. Daniel Sadoc Menasché

RIO DE JANEIRO, RJ – BRASIL
DEZEMBRO DE 2024

Carvalho Gomes Montalvão, Tiago

Implementation and Evaluation of a Deep Reinforcement Learning Model for Combinatorial Optimization Problems/Tiago Carvalho Gomes Montalvão. – Rio de Janeiro: UFRJ/COPPE, 2024.

X, 55 p.: il.; 29, 7cm.

Orientador: Daniel Ratton Figueiredo

Dissertação (mestrado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2024.

Referências Bibliográficas: p. 49 – 55.

1. Deep Reinforcement Learning. 2. Combinatorial Optimization. 3. Graph Embeddings. I. Ratton Figueiredo, Daniel. II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

Agradecimentos

Gostaria de agradecer primeiramente ao meu orientador Daniel Ratton Figueiredo, por ter me acolhido e confiado em mim. Sem o seu direcionamento este trabalho jamais teria sido concluído ou sequer existido.

Agradeço à minha noiva Ingrid, que sempre esteve ao meu lado durante todo este trabalho, me apoiando e dando forças para eu me dedicar e concluir o curso de pós-graduação. Sua ajuda nos bastidores foi essencial, principalmente nos momentos de desespero.

Agradeço aos meus pais Francisco e Marcia, que desde sempre me incentivaram e permitiram que eu tivesse condições suficientes de focar na minha vida acadêmica sem maiores preocupações.

Agradeço a todos os professores, amigos e colegas que tive contato durante os cursos de graduação e pós-graduação na UFRJ, seja durante aulas, estudos, monitorias, iniciações científicas, grupos de extensão ou de pesquisa. Vocês tornaram esses anos especiais para mim e levarei boas recordações e amizades para toda a vida.

De forma mais geral, agradeço a todos que fomentam a ciência e entendem que a educação transforma positivamente uma sociedade, e também a você, que está lendo essa tese neste momento.

Obrigado!

Resumo da Dissertação apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

IMPLEMENTAÇÃO E AVALIAÇÃO DE UM MODELO DE APRENDIZADO POR REFORÇO PROFUNDO EM PROBLEMAS DE OTIMIZAÇÃO COMBINATÓRIA

Tiago Carvalho Gomes Montalvão

Dezembro/2024

Orientador: Daniel Ratton Figueiredo

Programa: Engenharia de Sistemas e Computação

Problemas de Otimização Combinatória (CO) são muito importantes em diversas aplicações do mundo real, mas resolvê-los pode ser frequentemente desafiador. Neste trabalho, exploramos técnicas de Aprendizado por Reforço Profundo (DRL) para treinar agentes para resolver problemas de otimização combinatória, com foco no Structure2Vec Deep Q-Network (S2V-DQN), um dos primeiros modelos que utilizam DRL para aprender representações de vértices e construir boas soluções para esses problemas.

Apresentamos nossa própria implementação, desenvolvida inteiramente do zero, e a avaliamos em dois problemas: o Problema do Caixeiro Viajante (TSP) e o Problema da Cobertura de Vértices Mínima (MVC). Além disso, analisamos como certos hiperparâmetros impactam o aprendizado e discutimos os desafios encontrados durante a fase experimental.

Abstract of Dissertation presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

IMPLEMENTATION AND EVALUATION OF A DEEP REINFORCEMENT LEARNING MODEL FOR COMBINATORIAL OPTIMIZATION PROBLEMS

Tiago Carvalho Gomes Montalvão

December/2024

Advisor: Daniel Ratton Figueiredo

Department: Systems Engineering and Computer Science

Combinatorial Optimization (CO) problems are very important in many real-world applications, but solving them can often be very challenging. In this work, we explore Deep Reinforcement Learning (DRL) techniques to train agents to solve CO problems, focusing on Structure2Vec Deep Q-Network (S2V-DQN), a foundational model that uses DRL to learn node embeddings and construct good solutions for these problems.

We provide our own implementation, developed entirely from scratch, and evaluate it on two CO problems: the Traveling Salesman Problem (TSP) and the Minimum Vertex Cover (MVC) problem. Moreover, we analyze how certain hyperparameters impact learning and discuss the challenges encountered during the experimental phase.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Combinatorial Optimization (CO) consists of finding optimal or approximate solutions for problems with finite or discrete domains, and plays a significant role in solving real-world problems. These problems are of interest not only for their theoretical challenges, but also from a practical point of view. CO models problems from many different areas, such as logistics, transportation, resource allocation, communication, etc., so being able to solve them efficiently is very important, particularly when considering that many CO problems are NP-Hard (BOVET and CRESCENZI (1994)). For this reason, this area has been studied for a long time (SCHRIJVER (2005)). Moreover, due to their computational intractability, many heuristics and approximation algorithms have been proposed to tackle various CO problems.

Two examples of CO problems are the Traveling Salesman Problem (TSP) (APPLEGATE *et al.* (2006); MATAI *et al.* (2010)) and the Minimum Vertex Cover (MVC) (CHEN *et al.* (2001)), which will be explored in more detail in Section 2.4.

The original formulation for TSP consists of finding the shortest route for a salesman, which visits a list of locations and returns to his starting point, visiting each location exactly once. TSP has several variants and applications, such as Selective TSP and Asymmetric TSP for DNA sequencing (BŁAŻEWICZ *et al.* (1999); NIKOLAKOPOULOS and SARIMVEIS (2008)), or the Capacitated Vehicle Routing Problem for delivery route optimization (DANTZIG and RAMSER (1959)).

MVC consists of finding the smallest subset of vertices in a graph that covers all edges. This problem also has many real-world applications, such as in network security (FILIOL *et al.* (2007)) and computational biochemistry (LANCIA *et al.* (2001)).

On the other hand, Reinforcement Learning (RL) (SUTTON and BARTO (2018)) involves training an agent from past experiences to learn how to interact with an environment. This field is strongly inspired by the way that humans learn, by giving positive feedback for good actions and negative feedback for bad actions, in the form of numerical rewards observed for each action performed by the agent.

RL methods have been well explored in areas such as learning how to play games. For example, MNIH *et al.* (2013) developed Deep Q-Networks (DQN) originally to play different Atari 2600 games, and SILVER *et al.* (2017) developed AlphaZero, which is an agent capable of learning how to play different games, like Go and Chess, just by playing against itself without any prior strategic knowledge, and surpassing human-level performance. Other studied areas include autonomous vehicles (KIRAN *et al.* (2022)) and robotics (KOBER *et al.* (2013)).

Combining RL with CO areas has gained traction over the last decade (MAZYAVKINA *et al.* (2021)), and is a very challenging and also promising field of research. The main motivation for this work is to explore this combination by studying one of the most fundamental papers in the field and independently implementing and evaluating their work.

1.1 Learning Techniques to Solve Combinatorial Optimization Problems

Many different techniques have been developed to solve combinatorial optimization problems. These techniques can be divided into three categories: exact, approximate and heuristics (FESTA (2014)).

Exact algorithms, as the name suggests, find the exact optimal solution for an instance, meaning no other solution is better than the output of the algorithm. Although it might seem like the ideal solution, in general exact algorithms are not very efficient in terms of algorithmic complexity, especially for NP-Hard problems, in which the best algorithms have exponential time complexity in the size of the problem instance.

Another category is approximation algorithms, which do not yield the global optimal solutions, but they have a guaranteed worst-case approximation. For example, the TSP can be approximately solved with the Minimum Spanning Tree, which can be used to give a 2-approximation solution, meaning that the output solution is no worse than twice the optimal solution. Another approximate solution for the TSP is given by the Christofides algorithm (CHRISTOFIDES (1976)), which is a $\frac{3}{2}$-approximation algorithm, meaning that the output solution is no worse than 50% above the optimal solution.

The third category contains heuristic algorithms, which aim to find good solutions to CO instances, but which do not give any guarantees on the quality of the final solution, other than empirical evidence that they tend to find good solutions. Inside this third category, there are many different types of heuristics, including Machine Learning (ML) solvers, which are getting increasingly better (BENGIO

et al. (2021); MAZYAVKINA *et al.* (2021)). One advantage of ML techniques is that many of them do not need prior expert knowledge, so they are not limited by human expertise, and sometimes can learn novel heuristics.

1.2 Goals and Contributions

This work aims to study and understand a foundational model for solving combinatorial optimization problems using the deep reinforcement learning framework. The study also includes a review of key techniques used in this field.

To achieve these goals, this work focuses on Structure2Vec Deep Q-Network (S2V-DQN) (KHALIL *et al.* (2017)), which is a deep reinforcement learning model that solves combinatorial optimization problems on graphs, by learning node embeddings and using them to predict the value of selecting new nodes to iteratively construct solutions to these problems.

This work contributes with an independent implementation of the S2V-DQN model, developed entirely from scratch, as well as a step-by-step discussion on the modeling choices made in the process.

The performance of our S2V-DQN implementation is evaluated on the Minimum Vertex Cover and the Traveling Salesman Problem, two widely studied combinatorial optimization problems. We also briefly discuss how hyperparameter tuning affects the model's convergence.

1.3 How This Text is Organized

The remainder of this text is organized with the following structure:

- Chapter 2 discusses the theoretical background needed for both combinatorial optimization and deep reinforcement learning understanding. It also presents and discusses one of the first papers that combined both fields of research, in the S2V-DQN framework.

- Chapter 3 presents the key aspects of our implementation of the framework, discussing some details that are not explicitly covered in the original S2V-DQN paper. The chapter covers the main parts of the framework, detailing how the agent, model, and environment are implemented, how some techniques are used to stabilize the training procedure, and how validation is performed.

- Chapter 4 discusses the experimental results that were obtained by our implementation. This chapter also presents the evaluation metric and lists the actual parameters that were used, both for training and instance generation.

- Finally, Chapter 5 concludes this work, summarizing the key aspects of the framework and the results obtained, and also discussing potential directions for future work that could be explored.

Chapter 2

Background and Related Work

This work strongly depends on many concepts of both Deep Reinforcement Learning and Combinatorial Optimization. As such, this chapter is dedicated at reviewing some of the concepts used in the thesis.

2.1 Reinforcement Learning

Reinforcement Learning is an area of Machine Learning that consists of the interaction of an agent with an environment. This agent observes a state in the environment, takes an action, transitions to a new state, and receives a reward for this transition. A visualization of this dynamic can be found in the Figure 2.1.

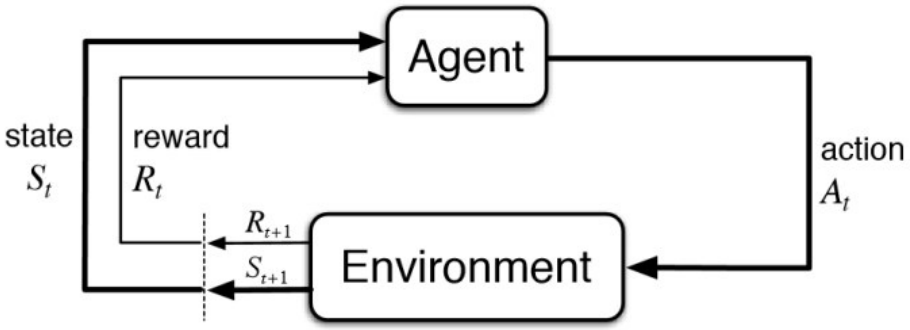


Figure 2.1: Agent observes state S_t and rewards R_t, takes an action A_t and transitions to a new state S_{t+1} receiving a new reward R_{t+1}.

The main objective of training an agent using reinforcement learning techniques is to find a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ that learns which action $a \in \mathcal{A}$ to take from each state $s \in \mathcal{S}$ to maximize an objective function based on the observed rewards.

A policy can be **deterministic**, in which, for each state s, exactly one action $a = \pi(s)$ is associated, or **stochastic**, in which, for each state s, a probability distribution $\pi(a|s)$ is associated for all actions $a \in \mathcal{A}$.

These definitions, as well as techniques for finding such policies, are further discussed in the following sections.

2.1.1 Markov Decision Process (MDP)

The foundation of reinforcement learning modeling is based on Markov models. A model with this characteristic has the Markov property, where the probability of a certain state occurring depends only on the previous state and not on the history of states.

More formally, let $\mathcal{S}$ be a set of states, a discrete-time Markov chain is a representation of a stochastic process in which, given a time step t, the following property holds:

$$P(X_t = s_t | X_{t-1} = s_{t-1}, \ldots, X_2 = s_2, X_1 = s_1) = P(X_t = s_t | X_{t-1} = s_{t-1}) \quad (2.1)$$

where X_t is a random variable representing the observed state at time t, and $s_1, s_2, \ldots, s_t \in \mathcal{S}$.

There are some variations of this definition, such as the continuous-time Markov chain, where the transition time between states is given by a random variable with an exponential distribution, which has the memoryless property, allowing the history of states to not interfere with the current and future transitions.

Markov chains have numerous applications, one of the most famous being the PageRank algorithm (BRIN and PAGE (1998)), which formed the basis of Google's search engine.

A **Markov Decision Process** (MDP) (BELLMAN (1957); SUTTON and BARTO (2018); WEISS (1960)) is a formalization of the notions of agent, state, reward, and environment described previously, also using the Markov property.

In this formulation, an agent interacts with an environment in a sequence of discrete steps. At each step t, the agent observes a state $S_t \in \mathcal{S}$ of the environment and takes an action $A_t \in \mathcal{A}(s)$. In the next step $t+1$, he receives a reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and observes a new state $S_{t+1} \in \mathcal{S}$.

An MDP can then be defined as a tuple $< \mathcal{S}, \mathcal{A}, p >$, where:

- $\mathcal{S}$ is the set of possible states observed by the agent
- $\mathcal{A}$ is the set of possible actions to be taken by the agent
- $p(s', r|s, a)$ is the probability function, also called **transition function**, that defines the dynamics of the MDP. It consists of the probability of observing a new state $s' \in \mathcal{S}$ and a reward $r \in \mathcal{R}$ in the instant following taking the action $a \in \mathcal{A}$ in state $s \in \mathcal{S}$, that is, it is another way to write $Pr(S_t = s', R_t = r | S_{t-1} = s, A_{t-1} = a)$

In some MDP formulations, it is common to define the transition function without dependence on rewards. From this definition above, this new function could be calculated as a marginal distribution:

$$p(s'|s,a) = Pr(S_t = s'|S_{t-1} = s, A_{t-1} = a) = \sum_{r \in \mathcal{R}} p(s', r|s, a) \tag{2.2}$$

Given the MDP transition function, it is possible to calculate many other values, such as the expected value of the reward of a transition between states.

2.1.2 Bellman Functions

Given an MDP, it is necessary to understand which objective function will be maximized. To do this, it is necessary to understand the nature of the problem, whether it is **episodic** (i.e., it has a finite number of steps per episode, such as the CO problems solved in this work), or **infinite** (i.e., the agent's interaction with the environment may potentially have no end, such as a robot learning to walk).

For an episodic problem, the expected return from time t to a terminal time T can be defined as follows:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T \tag{2.3}$$

For an infinite problem, there is no terminal instant T and an infinite sum easily diverges. Therefore, a constant called discount rate is introduced, represented by γ, which can be used to update the Equation 2.3:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \\ &= R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{(t+1)+k+1} = R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{2.4}$$

in which $0 \leq \gamma < 1$. If the rewards are bounded, that is, $R_t < R_{\max}, \forall t \geq 1$, the value of G_t is also bounded, because:

$$\begin{aligned} G_t &= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \leq \sum_{k=0}^{\infty} \gamma^k R_{\max} \\ &= R_{\max} \sum_{k=0}^{\infty} \gamma^k = \frac{R_{\max}}{1-\gamma} \end{aligned} \tag{2.5}$$

It's easy to see that the value of γ influences the weight given to each reward in the future. Values of γ close to 0 make the agent look at closer moments and practically ignore the more distant future, while values of γ closer to 1 make the

agent consider future rewards more.

Starting from the expected reward G_t, it is convenient to define two more functions. Taking into account a policy π to be followed, v_π in Equation 2.6 is defined as **state-value function**, while q_π in Equation 2.7 is defined as **action-value function**.

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t | S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right], \forall s \in \mathcal{S} \tag{2.6}$$

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right], \forall s \in \mathcal{S}, a \in \mathcal{A} \end{aligned} \tag{2.7}$$

It should be noted that the action-value function can be calculated for any pair (s, a), even if a is not an action to be taken based on the chosen policy π.

It is possible to write the state-value function of a state as a function of the following states, as well as the action-value function for the following states and actions. These functions written in this recursive way are called **Bellman functions**. The state-value function can be written as in Equation 2.8.

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[G_t | S_t = s \right] \\ &= \mathbb{E}_\pi \left[R_t + \gamma G_{t+1} | S_t = s \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s'} \sum_{r} p(s', r | s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s' \right] \right] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s', r} p(s', r | s, a) \left[r + \gamma v_\pi(s') \right], \forall s \in \mathcal{S} \end{aligned} \tag{2.8}$$

Similarly, the action-value function can be written as in Equation 2.9.

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a \right] \\ &= \mathbb{E}_\pi \left[R_t + \gamma G_{t+1} | S_t = s, A_t = a \right] \\ &= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s' \right] \right] \\ &= \sum_{s', r} p(s', r | s, a) \left[r + \gamma \sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s', a') \right], \forall s \in \mathcal{S}, a \in \mathcal{A}. \end{aligned} \tag{2.9}$$

An MDP has an optimal state-value function and an optimal action-value function, defined as the function itself evaluated with the best possible policy:

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in \mathcal{S} \tag{2.10}$$

$$q_*(s,a) = \max_{\pi} q_\pi(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{2.11}$$

2.1.3 Q-Learning

A problem with the Bellman equations presented so far is that one needs to know the transition function p of the MDP, which is not the case for most practical problems. To solve this issue, some approaches that do not require an environment model are presented. The first ones are the Monte Carlo methods.

Let G_t be an observed value for the return of time t to the end of an episode, and α be a constant indicating the learning rate. It is possible to calculate an estimate $V(S_t)$ for $v_*(s)$ iteratively from an exponentially weighted moving average for a sufficient number of episodes as:

$$V(S_t) \leftarrow \alpha G_t + (1-\alpha)V(S_t) = V(S_t) + \alpha(G_t - V(S_t)) \tag{2.12}$$

An immediate problem with this method is that it is necessary to wait for an entire episode to finish before making updates to the state-value function estimates. When the episode is very long, this can be very inefficient, and even unfeasible for infinite horizon problems.

Another commonly used approach is a method called Temporal Difference (TD) (SUTTON (1988)). This method allows updating the estimate for the state-value function after each instant t of the interaction between the agent and the environment, and not after each episode, using a technique called bootstrap, in which an estimate is updated based on other estimates. The update equation presented by this method is then:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{2.13}$$

Using this approach, one of the main Reinforcement Learning algorithms was developed, called **Q-Learning** (WATKINS (1989); WATKINS and DAYAN (1992)). It uses an update similar to the one presented in the Equation 2.13, but with the update of the estimate $Q(S_t, A_t)$ to $q_*(s,a)$:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_t + 1, a) - Q(S_t, A_t) \right] \tag{2.14}$$

The complete algorithm can be found in Algorithm 1. It uses a method of exploring the state space known as ε-greedy, in which, with probability ε, a random action is taken and, with probability $(1-\varepsilon)$, the best calculated action is taken at each instant.

Since $Q(s,a)$ is calculated for all states and actions, the policy can be calculated

deterministically as:

$$\pi(s) = \max_a Q(s, a) \tag{2.15}$$

Algorithm 1: Q-Learning

Input: Learning rate $\alpha \in (0, 1]$ and $\varepsilon > 0$

Output: $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}$

$Q(s, a)$ initialized arbitrarily for non-terminal states
$Q(terminal, \cdot) = 0$
for *each episode* **do**
    $S \leftarrow$ initial state
    while S *is not terminal* **do**
        Choose action A from S using an ε-greedy strategy with Q
        Perform action A, observe reward R and next state S'
        $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$
        $S \leftarrow S'$
    end
end

Other methods have been proposed in the literature also using TD. Examples are SARSA (RUMMERY and NIRANJAN (1994); SUTTON and BARTO (2018)) and Expected SARSA (SUTTON and BARTO (2018)), which try to take into account not only the best value of the action-value function for choosing the next action, but also the method of exploration used, but such methods are not as widely used as Q-Learning.

It is also worth mentioning that Q-Learning is a method known as **off-policy**, in the sense that actions can be taken from any policy, even one that does not use the function current $Q(s, a)$. This allows this algorithm to be used for an agent to learn from past experiences and even from other agents.

A problem with all these methods is that the function to be calculated has an extremely large domain. For the Q function, it is necessary to calculate $|\mathcal{S}| \times |\mathcal{A}|$ values. As it is an iterative method, each value must be updated a sufficient number of times to obtain convergence. The number of iterations of this algorithm easily explodes, making it unfeasible for such scenarios. In this context, some techniques must be used, whether to discretize the space of states and actions, such as techniques known as Tile Coding (SUTTON and BARTO (2018)) or Coarse Coding (SUTTON (1996); SUTTON and BARTO (2018)), or to approximate functions calculated from states and functions. This second option is the most used, and it is common to use approximate models based on neural networks.

2.2 Deep Learning

Deep Learning (GOODFELLOW *et al.* (2016)) is an area of Machine Learning that focuses on using multi-layer (deep) neural networks to learn features and patterns from data, and also perform classification. It is an extension of the classic Neural Network model enabled by machines with more powerful hardware and larger datasets.

2.2.1 Neural Networks

A neural network is a parametric model, developed from inspirations in the human brain's structure. This model consists of a set of units, called **neurons**, connected together, forming a computational graph. These connections have parameters called **weights**.

It is common to group neurons into layers, in such a way that one layer receives signals $\mathbf{x}$ weighted by the weights $\mathbf{w}$ of the previous layer and combined linearly in each neuron. After this combination, a nonlinear **activation function** g is applied to this signal. This activation function must be nonlinear, otherwise the combination of several linear functions would result in a single linear function. A visualization of a simple layered neural network can be seen in Figure 2.2.

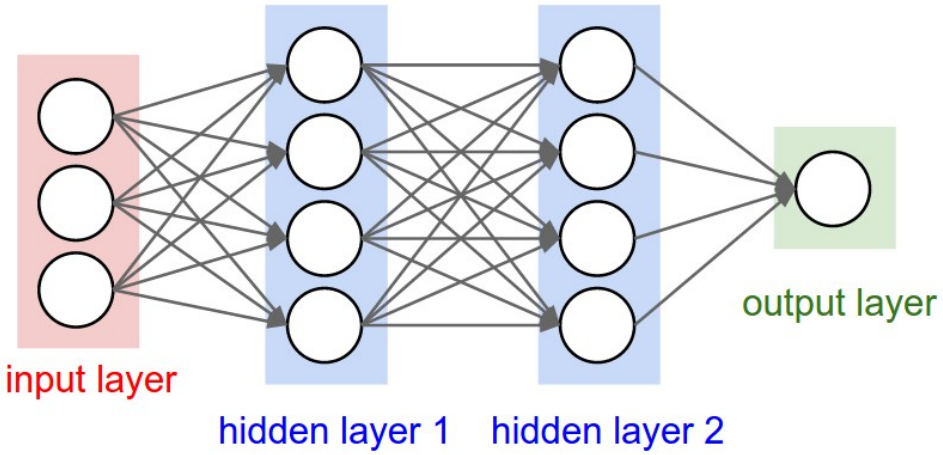


Figure 2.2: Example of a neural network with three input features, two hidden layers and one output

Neural networks have different types of architecture, including feed-forward (BEBIS and GEORGIOPOULOS (1994)), convolutional (LI *et al.* (2022)), recurrent networks (YU *et al.* (2019)), among others. Each network is usually used more for a problem domain. For example, convolutional networks (LI *et al.* (2022)) have become well known with their application in image classification, as well as other tasks in the area of computer vision. Recurrent networks (YU *et al.* (2019)) have properties that make them suitable for modeling sequential inputs and/or outputs.

This characteristic makes them appropriate for the area of natural language processing, expressed by written texts and communicated between human beings. The development of new architectures is a very explored area of research, with the advent of increasingly complex and powerful networks. As an example, we can mention the creation of networks known as Transformers (VASWANI *et al.* (2017)), widely used for modeling sequences, which have several advantages over recurrent networks.

Although neural networks were known for several decades, they remained without much active research for a while, while there was no efficient way to train them, until the development of the so-called **backpropagation** (RUMELHART *et al.* (1986)). This is an algorithm for propagating the loss function with respect to each of the parameters, combining dynamic programming techniques with differential calculus results.

Even with the development of backpropagation, training a neural network has always been considered a costly task. For this reason, this model has become increasingly used in recent years, with increasing computational power, both for processing and storage, thus making massive sets of data increasingly accessible.

An example of a neural network is the one called AlexNet (KRIZHEVSKY *et al.* (2012)), which was the winning model of the ImageNet (DENG *et al.* (2009)) competition in the 2012 edition, showcasing the potential of deep neural networks. This is a competition in which a set of 1.2 million images must be classified into 1000 distinct classes. The network architecture featured convolutional layers and it was trained on a GPU at that time.

2.2.2 Cost Function Optimization

To train a neural network, as well as several other machine learning models, it is necessary to optimize a cost (or loss) function that expresses how well it can represent the distribution of the output as a function of the input. Formally, we want to minimize a cost function $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$, where $\hat{y} = \hat{f}(x; \theta)$ and θ is the set of model parameters that can be adjusted.

One of the most used loss functions, specially for regression tasks, is the mean squared error. For a dataset with M examples, it is calculated as:

$$\mathcal{L}_{\text{MSE}}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{M} \sum_{i=1}^{M} (y_i - \hat{y}_i)^2 \tag{2.16}$$

To optimize the loss function, several techniques can be used, the **gradient descent** technique being the most frequently used.

This is an iterative technique, in which the gradient of the loss function is calculated in relation to each parameter and an update is made to each of these pa-

rameters. This procedure is repeated a maximum number of times or until the loss function converges to some value. Mathematically, this update can be written in vector form as:

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta_t} \mathcal{L}(\theta_t) \tag{2.17}$$

, in which θ_t represents the set of parameters in iteration t, α represents a learning rate, which indicates the size of the update in each iteration, and $\nabla_{\theta_t} \mathcal{L}(\theta_t)$ represents the gradient of $\mathcal{L}$ when evaluated with respect to each of the parameters of θ_t.

In practice, updating the Equation 2.17 can become a computationally heavy problem, because the calculation of $\mathcal{L}$ requires calculations of all points in the data set. To overcome this problem, an alternative is to use the so-called **stochastic gradient descent**, which considers only one point in the data set at a time. This variation creates another problem, which is the great variability of updates, due to the different contributions of each point in the training set.

To consider the advantages of both approaches, a combination of these methods was developed, known as **mini-batch gradient descent**, in which only a fixed size sample is used as an estimate of the loss function for the entire set. This change makes the update much more efficient, but generally requires more iterations for better convergence, in addition to introducing noise into the loss curve throughout the iterations.

However, another problem that arises is adjusting the learning rate. A very small learning rate causes parameter adjustment to occur more slowly, while a very large learning rate causes the optimizer to be unable to converge to good local optima on the loss function curve. With this problem in mind, some techniques were developed to make the learning rate adaptive, including: AdaGrad (DUCHI *et al.* (2011)), RMSProp (HINTON *et al.* (2013)), and Adam (KINGMA and BA (2014)).

The optimizer used in this work is **Adam**. It uses two parameters β_1 and β_2 to maintain exponentially weighted moving averages of the gradient and the square of the gradient, respectively, and uses these averages to automatically adapt the learning rate and to use the idea of **moment**, where updates from previous iterations influence the current update of the optimizer.

2.3 Deep Reinforcement Learning

From the ideas discussed, it is possible to combine neural network modeling with reinforcement learning ideas to calculate value and action-value function approximations, given an input representing a state or a state-action pair.

2.3.1 Deep Q-Learning

The first idea discussed is Deep Q-Learning (MNIH *et al.* (2013)), which directly combines the ideas discussed about Q-Learning with approximations via deep neural networks, in addition to introducing some techniques to stabilize the training process. This idea is used by the authors to build a convolutional neural network to train agents capable of playing various Atari 2600 games without any game rules being encoded, just with the agent's interactions with the game environment. Later, this network was extended and applied to 49 Atari 2600 (MNIH *et al.* (2015)) games, achieving performance comparable to or better than that of a human in 29 of these games.

To recap, to know $Q(s, a)$ for every pair of state and action, it is necessary to calculate $|\mathcal{S}| \times |\mathcal{A}|$ positions. For combinatorial optimization problems, the number of states grows exponentially with the graph size, making it unfeasible to use the Algorithm 1.

The idea known as Deep Q-Learning consists of learning a function $Q(s, a; \theta)$, parameterized by θ, as an approximation to $Q(s, a)$. The proposal presented by the article that introduces this idea is to use a deep neural network, known as Deep Q-Network (DQN), for this approximation. An example of a DQN diagram can be found in Figure 2.3.

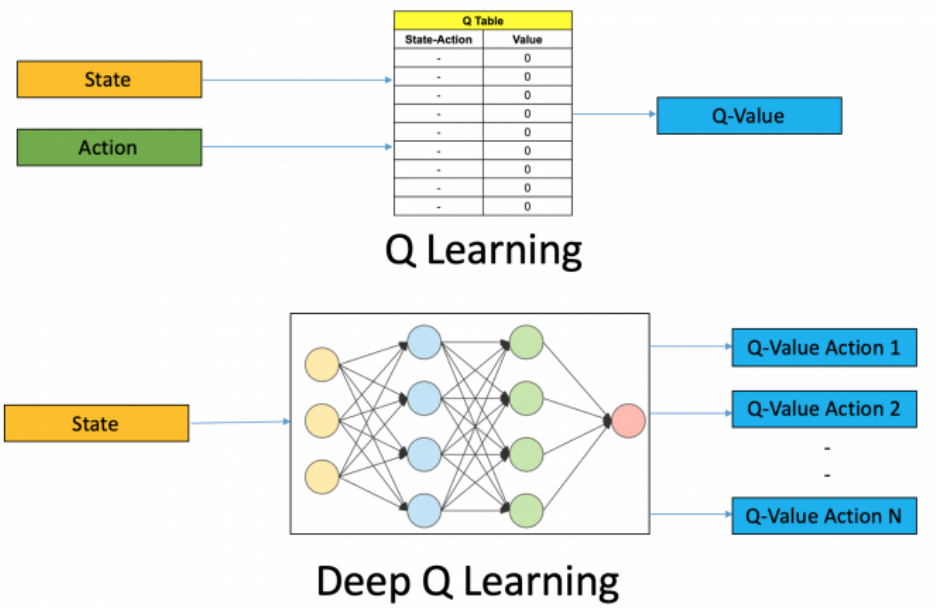


Figure 2.3: Comparison between Q-Learning, in which all pairs (s, a) are calculated, and Deep Q-Learning, in which, for each state s, are calculated the values of $Q(s, a)$ for every possible action a. Source: `https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/`. Accessed on October 18, 2024.

After calculating the approximate values for $Q(s, a; \theta)$, some strategy is used to

choose the action to be taken by the agent. In general, the same strategy mentioned in Section 2.1.3, known as ε-greedy, is used.

An idea used along with the article was **Experience Replay** (LIN (1992)), which consists of a buffer of past experiences used for training the model. An experience consists of a tuple $< s_t, a_t, r_{t+1}, s_{t+1} >$, in which s_t and a_t consist of a state and the action taken in that state at a given instant t, while r_{t+1} and s_{t+1} consist of the reward received and the new state observed after taking the action. This buffer has a certain size N, in order to store the last N tuples of experience. For training, some tuples are sampled to adjust the parameters. This idea is useful for making the model learn from an average of several past experiences, thus avoiding getting stuck in known feedback loops and making learning smoother.

In addition to Experience Replay, the idea of **Target Networks** was also introduced based on an observed problem of instability in model training. The loss function equation can be defined using the mean squared error:

$$\begin{aligned} \mathcal{L}(\theta) &= \mathbb{E}_{s,a,r,s'}\left[y - Q(s,a;\theta)\right]^2 \\ &= \mathbb{E}_{s,a,r,s'}\left[r + \gamma \max_{a'} Q(s',a';\theta) - Q(s,a;\theta)\right]^2 \end{aligned} \tag{2.18}$$

This equation is derived from Equation 2.14, by comparing the current value calculated for $Q(s,a)$ with the target value obtained from the transition to the next state. The target is calculated as $y = r' + \gamma \max_{a'} Q(s',a';\theta)$. This approach has a problem due to the dynamic nature of the training: when an update is made to θ, not only the value of $Q(s_t,a_t;\theta)$ changes, but also the value of $Q(s_{t+1},a;\theta)$, for any action a, and, consequently, the target y. Therefore, a second network $\hat{Q}$ known as **target network** is introduced, with an architecture identical to the original network Q, but keeping its parameters θ^- fixed and being updated only every K iterations as a copy of the original network parameters.

The combination of all these ideas is summarized in Algorithm 2. It is worth mentioning that the optimization algorithm used calculates the gradient of the loss function in relation to θ, not θ^-, therefore the target y_j is considered constant for training.

From the initial version of DQN, several variations emerged trying to address some issues still encountered. For example, the values calculated for $Q(s,a;\theta)$ tend to overestimate the real value of $Q(s,a)$ (THRUN and SCHWARTZ (1993)), a problem that motivated the variation known as Double DQN (VAN HASSELT *et al.* (2015)). Another well-known example is the variation known as Dueling DQN (WANG *et al.* (2016)), in which the decomposition $Q(s,a) = V(s) + A(s,a)$ is made, with $A(s,a)$ a term that indicates the advantage of taking an action a in state s, when compared to the other actions. Each of these terms is learned individually,

resulting in better estimates and better policies in scenarios where different actions have similar values for the action-value function.

Algorithm 2: Deep Q-Learning

Experience replay buffer $\mathcal{D}$ initialized with size N
Neural network Q initialized with random θ parameters
Target neural network $\hat{Q}$ initialized with parameters $\theta^- = \theta$
for *each episode* **do**
  $s_1 \leftarrow$ initial state
  for $t = 1,\ T$ **do**
    Choose action a_t from s_t using an ε-greedy strategy with Q
    Perform action a_t, observe reward r_{t+1} and next state s_{t+1}
    Store the tuple $< s_t, a_t, r_{t+1}, s_{t+1} >$ in $\mathcal{D}$
    Sample experience tuples $< s_j, a_j, r_{j+1}, s_{j+1} >$ from $\mathcal{D}$
    Make $y_j = \begin{cases} r_{j+1} & \text{if episode ends in } j+1 \\ r_{j+1} + \gamma \max_{a'} \hat{Q}\left(s_{j+1}, a'; \theta^-\right) & \text{otherwise} \end{cases}$
    Update θ with an optimizer step to $\left[y_j - Q\left(s_j, a_j; \theta\right)\right]^2$
    Update $\theta^- = \theta$ every K iterations
  end
end

2.4 Combinatorial Optimization Problems

Combinatorial Optimization (CO) is a field of optimization theory focused on finding the best solution from a finite set of discrete viable solutions. Unlike continuous optimization, where variables can take any value within a range, combinatorial optimization deals with problems where variables are restricted to discrete values, often integers or binary numbers.

Solutions are often represented as combinations, permutations, or subsets of a finite set of elements, making the problem space grow exponentially with the size of the input. Many CO problems are computationally intractable to solve, which means that no exact algorithm is known to solve it in polynomial time. These problems belong to the class of NP-hard problems.

There are many examples of NP-hard CO problems. Among them, one can find Traveling Salesman Problem, Minimum Vertex Cover, Maximum Independent Set, Maximum Cut Problem, etc.

2.4.1 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) (APPLEGATE *et al.* (2006); MATAI *et al.* (2010)) is one of the most well-known and studied problems in combinatorial optimization. The objective of the TSP is to find the shortest tour in a graph that visits each node exactly once and returns to the starting node, also known as a Hamiltonian cycle.

Formally, given a set of nodes $V = \{1, 2, \ldots, n\}$ and a matrix $D_{i,j}$ representing the edge weight from node i to node j, the goal is to find a permutation $\sigma : V \to V$ of the nodes in order to minimize the sum of the distances of consecutive nodes $\sum_{i=1}^{n} D_{\sigma(i),\sigma(i+1)}$, with $\sigma(n+1) = \sigma(1)$.

Euclidean version of TSP

A special case of TSP occurs when the nodes represent points in the Euclidean plane, and the distance $D_{i,j}$ represents the Euclidean distance between the points represented by the nodes i and j. This version of the TSP leads to a complete graph. If point k has coordinates (x_k, y_k), then

$$D_{i,j} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{2.19}$$

Figure 2.4 is an example of an Euclidean TSP instance, illustrating the shortest tour visiting each point exactly once.

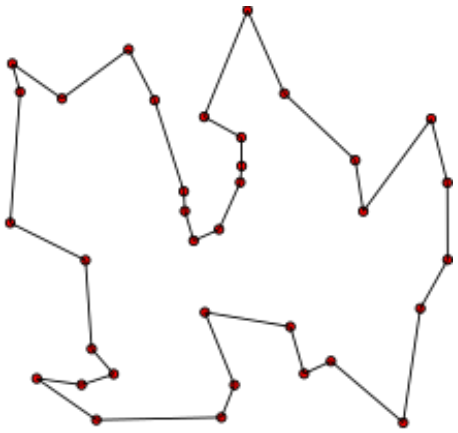


Figure 2.4: Euclidean TSP example

Some problems might be simplified when working with graphs in the plane, but that is not the case for TSP, which is NP-hard for both its original formulation (KARP (1972)) and for the Euclidean variation (PAPADIMITRIOU (1977)).

2.4.2 Minimum Vertex Cover

The Minimum Vertex Cover (MVC) (CHEN *et al.* (2001)) problem is another classical combinatorial optimization problem. Given an undirected graph $G = (V, E)$, the goal is to find the smallest subset of vertices $C \subseteq V$ such that every edge in E is incident to at least one vertex in C. In other words, each edge in the graph must have at least one of its endpoints included in the vertex cover. Just as the TSP, the MVC problem is also NP-Hard.

For example, if we take the two graphs in Figure 2.5, the vertices in red form a minimum vertex cover, because every edge is incident to one of those vertices, and there is no other vertex cover with fewer vertices in those examples.

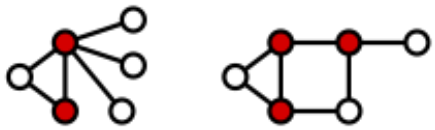


Figure 2.5: Two MVC examples

It can be formulated using a Mixed Integer Linear Programming (MILP) formulation as follows:

$$\begin{aligned} & \min \sum_{v \in V} x_v \\ \text{subject to:} \quad & x_u + x_v \geq 1 \quad \forall (u, v) \in E \\ & x_v \in \{0, 1\} \quad \forall v \in V \end{aligned} \tag{2.20}$$

2.5 Related Work

The application of reinforcement learning to solve combinatorial optimization problems has been an active area of research for many years, leading to a wide variety of models and methods. Over this period, numerous approaches have been developed. To provide a comprehensive understanding of the progress in this field, several surveys have been published, offering in-depth discussions on the different RL frameworks and techniques used to address combinatorial optimization challenges.

One of these surveys was written by BENGIO *et al.* (2021), in which the authors discuss how ML can be used to aid the solving of CO problems in general. For that, they categorize ML methods in two ways. On the one hand, to approximate or speed up some computations. This is done by assuming that expert knowledge is available and enough to solve the instance, and by training a model from real-world demonstrations. On the other hand, this expert knowledge might not be sufficient and ML helps to explore new policies still unexplored.

The authors also discuss how these ML methods can be combined with traditional CO algorithms. For that, they mention three categories. The first one is called end-to-end learning, in which a model is trained to output solutions directly from the input instances. The second category is learning to configure algorithms, in which the model is used to provide additional information, such as some hyperparameters for a solver. The third category is using ML models alongside optimization algorithms, such as using it in a branch-and-bound solver for deciding which variable to use for the branching decisions or for deciding if some expensive heuristic algorithm should be executed.

Another survey was written by MAZYAVKINA *et al.* (2021). The authors discuss many different aspects of the RL approach to solve CO problems. They discuss, for example, different techniques of how to encode a graph as a vector in some latent space, from recurrent neural networks (RNNs) and its variations, such as Long Short-Term Memory (LSTM) unit (HOCHREITER and SCHMIDHUBER (1997)) and Gated Recurrent Unit (GRU) (CHO *et al.* (2014)), up to some custom encodings, such as S2V-DQN (KHALIL *et al.* (2017)), discussed in more details in the next section.

The authors also split RL algorithms into two types. The first one consists of model-based methods, in which the transition probabilities are to be learned. This includes the Monte Carlo Tree Search technique, which is used successfully for AlphaZero (SILVER *et al.* (2016)) and its variations to learn how to play the game of Go and a few other games. The other type consists of model-free methods, in which the trained agent does not try to learn the environment dynamics, but instead try to learn how to act based only on observations and past experience. For model-free methods, there are even two subcategories: policy-based methods, in which the agent directly learns which action to take on each given state, and value-based methods, in which the agent learns a measure of the quality of each action-state pair and then chooses the action to take from that.

2.5.1 S2V-DQN

Given the broad landscape of different methods and models, KHALIL *et al.* (2017) published one of the first works that combined RL with CO. In this paper, they tackle CO problems by combining a greedy framework with graph embeddings and RL.

More specifically, the authors propose a graph embedding that is used in a value-based RL method to parameterize and learn the action-value function $Q(s, a)$. This representation is called Structure2Vec (S2V), a technique proposed in a separate paper to generate embeddings for structured data (DAI *et al.* (2016)).

They then use this parametrization to train an RL agent using n-step Q-Learning, which is called S2V-DQN and is described in more detail in the following sections.

Problem solving formulation

The solution to the CO problem in this framework is constructed incrementally in a greedy manner: whenever an action is taken by the RL agent to include a new vertex in the partial solution, this vertex is not removed anymore. Taking the MVC as an example, if at any iteration a vertex is added to the partial solution constructed by the agent, it will be an element of the final cover. This greedy approach is very intuitive, but has one limitation: it works for problems in which all vertices that are still not added to the partial solution are valid. However, it would require some further refinement when dealing with problems such as the TSP on non-complete graphs. This is not the case in this work, but it is worth highlighting.

More formally, given an instance graph $G = (V, E)$ sampled from a distribution $\mathbb{D}$, the solution is constructed as an ordered list of vertices $S = (v_1, v_2, \ldots, v_{|S|})$ that are one by one included in the solution by the agent. The actual structure representing the solution to the problem is then represented by $h(S)$, which can represent for example some rearrangement of the nodes in S, given their insertion order.

The quality of the solution is given by an objective function $c(h(S), G)$, which is specific to each problem being solved.

While the instance is not terminated, the choice of which node to insert in each iteration is taken greedily, in a way to maximize the action-value function for that node, i.e., the agent should append a node v^* into the partial solution S, such that $v^* = \text{argmax}_{v \in \overline{S}} Q(h(S), v)$.

Structure2Vec

In order for the agent to be able to take steps and incrementally construct the solution to the CO problem, it needs to receive an input that encodes the current state of the instance.

The authors use an encoding called Structure2Vec (S2V) (DAI *et al.* (2016)), which is constructed to capture the local structure of the graph around each node. S2V computes node embeddings using a message-passing mechanism between neighboring nodes. This mechanism works by propagating local information of a node to its neighbors some number T of times. In this way, T defines the depth of the message-passing, meaning that a node embedding will contain aggregated information of all nodes that are up to T edges away.

Formally, the embedding $\mu_v \in \mathbb{R}^p$ for a node v is computed in layers, such that $\mu_v = \mu_v^{(T)}$, $\mu_v^{(0)} = \mathbf{0}$ and, for $t = 1, \ldots, T$:

$$\mu_v^{(t+1)} = F\left(x_v, \left\{\mu_u^{(t)}\right\}_{u \in \mathcal{N}(v)}, \{w(v,u)\}_{u \in \mathcal{N}(v)}; \Theta\right), \tag{2.21}$$

where F is some nonlinear function, x_v represents if a node is part of the partial solution and can even hold other node attributes, $\mathcal{N}(v)$ is the set of neighbors of the node v, $w(v,u)$ represents edge attributes and Θ represents the set of parameters used by S2V.

In the paper, the authors propose the following function F:

$$\mu_v^{(t+1)} = \text{relu}\left(\theta_1 x_v + \theta_2 \sum_{u \in \mathcal{N}(v)} \mu_u^{(t)} + \theta_3 \sum_{u \in \mathcal{N}(v)} \text{relu}\left(\theta_4 w(v,u)\right)\right), \tag{2.22}$$

where $\theta_1, \theta_4 \in \mathbb{R}^p$ and $\theta_2, \theta_3 \in \in \mathbb{R}^{p \times p}$ are model parameters, and relu is the rectified linear unit (NAIR and HINTON (2010)), given by $\text{relu}(z) = \max(0, z)$).

These embeddings are computed at each iteration of the RL agent's interaction with the environment, as it takes into consideration if each node is in the partial solution found so far. They are then used to compute an approximation $\widehat{Q}(h(S), v; \Theta)$ for the action-value function of Q as follows:

$$\widehat{Q}(h(S), v; \Theta) = \theta_5^\top \text{relu}\left(\left[\theta_6 \sum_{u \in V} \mu_u^{(T)}, \theta_7 \mu_v^{(T)}\right]\right), \tag{2.23}$$

where $\theta_5 \in \mathbb{R}^{2p}$ and $\theta_6, \theta_7 \in \in \mathbb{R}^{p \times p}$ are other model parameters and $[\cdot, \cdot]$ is the concatenation operator.

Training

In order to learn the parameters $\theta_i, i = 1, \ldots, 7$, a Reinforcement Learning environment is set up, where all of the above pieces are put together under the Markov Decision Process framework.

The tuple $< \mathcal{S}, \mathcal{A}, p >$ is then defined as follows:

- $\mathcal{S}$ is the set of states, where a state represents the sequence of nodes that compose the encoded partial solution, in the order that they were inserted
- $\mathcal{A}$ is the set of actions, where an action is the set of nodes $v \in \overline{S}$ that are not part of the partial solution S
- $p(s', r|s, a)$ represents the transitions, which are actually deterministic here. When a new node is inserted, it's added to the partial solution encoded by the

state and a deterministic reward is given based on the problem being solved

To train an agent to act in the environment and to find good solutions, an algorithm very similar to Algorithm 2 is used, by adapting it to use n-step Q-Learning and to build the partial solutions incrementally.

The idea for n-step Q-Learning is that the immediate reward is not so important as the total reward for the whole episode, because what matters is the final solution found by the agent, not intermediate steps. The n-step feature modifies the elements that are stored in the experience tuples in the replay buffer, which become the n-th last state S_{t-n}, the action taken in the n-th last state v_{t-n}, the reward accumulated from the n-th last state until the current state $R_{t-n,t} = \sum_{i=0}^{n-1} \gamma^i r(S_{t-n+i}, v_{t-n+i})$, and the current state S_t.

The addition of n-step Q-Learning also modifies the target y used in the MSE loss function calculation. The squared loss reads as:

$$\mathcal{L} = \left(y - \widehat{Q}(h(S_t), v_t; \Theta)\right)^2, \tag{2.24}$$

such that

$$\begin{aligned} y &= R_{t,t+n} + \gamma^n \max_{v'} \widehat{Q}(h(S_{t+n}), v'; \Theta) \\ &= \sum_{i=0}^{n-1} \gamma^i r(S_{t+i}, v_{t+i}) + \gamma^n \max_{v'} \widehat{Q}(h(S_{t+n}), v'; \Theta) \end{aligned} \tag{2.25}$$

Note that $\widehat{Q}(h(S_{t+n}), v'; \Theta) = 0$ in the case that the state represented by S_{t+n} is a terminal state, so the target is left with $y = R_{t,t+n}$.

For the action taken by the agent, it still uses an ε-greedy approach choosing the action that maximizes $\widehat{Q}(h(S), v; \Theta)$ with probability $1 - \varepsilon$ and picking a random action with probability ε.

Putting everything together, the algorithm used for the training of an RL agent to solve CO problems using the S2V-DQN encoding and training is described in

Algorithm 3.

Algorithm 3: Deep Q-Learning

Experience replay buffer $\mathcal{D}$ initialized with size N
Neural network $\widehat{Q}$ initialized with random Θ parameters
Target neural network $\widehat{Q}^{-}$ initialized with parameters $\Theta^{-} = \Theta$
for *each episode* **do**
    Draw graph G from distribution $\mathbb{D}$
    $S_1 \leftarrow$ empty initial state
    for $t = 1$ *to* T **do**
        Choose node $v_t = \begin{cases} \text{random node v} \in \overline{S_t} & \text{, with prob. } \varepsilon \\ \text{argmax}_{v \in \overline{S_t}} \widehat{Q}(h(S_t), v; \Theta) & \text{, otherwise} \end{cases}$
        Append v_t to the partial solution: $S_{t+1} \leftarrow (S_t, v_t)$
        Observe reward R_{t+1}
        if $t \geq n$ **then**
            Store the tuple $< S_{t-n}, v_{t-n}, R_{t-n,t}, S_t >$ to $\mathcal{D}$
            Sample batch B of experience tuples from $\mathcal{D}$
            Update Θ with an optimizer step to (2.24)
            Update $\Theta^{-} = \Theta$ every K iterations
        end
    end
end

It is important to note that an independent implementation of S2V-DQN along with its application to solve two different combinatorial optimization problems (TSP and MVC) is a major contribution of this thesis.

Chapter 3

System Design and Implementation

This chapter aims to give more details on the implementation of the S2V-DQN model and the training procedure in order to learn to solve Combinatorial Optimization (CO) problems.

The design and implementation tried to be as faithful as possible to the original framework presented in the S2V-DQN paper, including details to solve each combinatorial optimization problem.

3.1 System Architecture

There are many building blocks for the implementation. The Figure 3.1 illustrates the main ones and how they interact, which are:

- **Environment:** describes the spaces of observations and actions, together with transitions and rewards;
- **Model:** neural network implementation of the embedding and action-value function approximation $\widehat{Q}(h(S_t), v_t; \Theta)$;
- **Agent:** entity responsible for performing actions on the environment and for triggering model updates;
- **Training and validation loops:** main loop to train the agent's model to perform well in the environment and to assess the quality of the model through validation.

3.2 Implementation

All of the above blocks are implemented in Python, using the PyTorch framework (PASZKE *et al.* (2019)) and following most of the code structure from Open AI

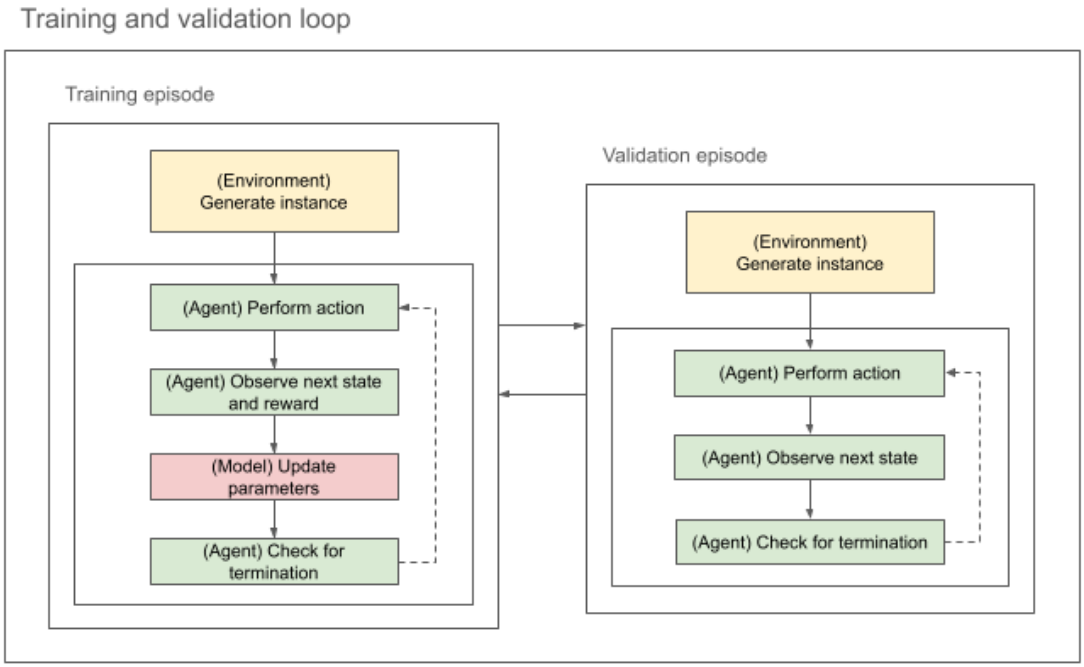


Figure 3.1: Diagram illustrating the interaction between different building blocks of the system. Training episodes are repeated consecutively before a set of validation episodes are executed.

Gym (BROCKMAN *et al.* (2016)), which establishes the API for the interaction between the agent and the environment. The implementation is entirely and publicly available on GitHub [1].

3.2.1 Environment

An environment consists of an instance of a CO problem to be solved, with the definition of its state, and methods to return the state representation, rewards for given actions, and termination criterion.

The environment is the only module that must be implemented in case someone wants to add support for another CO problem.

State representation

Given an environment, a representation must be defined adequately such that an agent can use this representation as input in order to decide its action.

Equation 2.22 is originally defined for a single node attribute x_v, representing if the node belongs to the partial solution, and a single edge attribute $w(u, v)$, representing the weight of the edge (u, v). However, both of these attributes can be

[1] `https://github.com/tiagomontalvao/s2v-dqn`

extended to include more information. When that is the case, the parameters θ_1 and θ_4 must be adjusted accordingly, i.e., increasing their dimensions.

Formally, given a graph $G = (V, E)$, node attributes $\mathbf{X}_v \in \mathbb{R}^{n \times a_v}$, where $n = |V|$ and a_v is the number of attributes for each node (the same for every node), and edge attributes $\mathbf{X}_e \in \mathbb{R}^{m \times a_e}$, where $m = |E|$ and a_e is the number of attributes for each edge (the same for every edge), the state representation is a function $f(G, \mathbf{X}_v, \mathbf{X}_e)$ that encodes all necessary information to replicate the current state of the instance. Parameters θ_1 and θ_4 are adjusted as function of the number of node and edge attributes and become $\theta_1 \in \mathbb{R}^{p \times a_v}$, $\theta_4 \in \mathbb{R}^{p \times a_e}$. Recall that p is a hyperparameter that determines the dimension of the representation.

Note that the set of node and edge attributes is dependent on the problem being solved, as discussed below.

MVC For the MVC problem, only a single node attribute x_v is used for each node $v \in V$, indicating if that node is present on the current partial solution for the problem.

To encode the graph, the adjacency matrix $\mathbf{A}$ is stored, such that $\mathbf{A}_{v,u} = 1$ if there is an edge between nodes v and u, and $\mathbf{A}_{v,u} = 0$ otherwise.

These two pieces of information are concatenated together. The MVC state then has two dimensions with ranges $(n, n + a_v)$, where $a_v = 1$, and no edge attributes are used in this problem.

TSP For TSP, four node attributes are used for each node $v \in V$:

1. $x_v \in \{0, 1\}$ indicating if the node is present on the current partial solution;
2. x and y coordinates for the point represented by node v, normalized by the maximum possible coordinate value; and
3. $l_v \in \{0, 1\}$ indicating if the node is the last one in the tour given by the nodes in the partial solution.

There are also three edge attributes implemented for each edge (u, v):

1. the weight (Euclidean distance) of the edge (u, v), normalized by the maximum possible coordinate value of a point;
2. $x_u \in \{0, 1\}$, indicating if the first endpoint of the edge is present on the current partial solution; and
3. $1_{x_u \neq x_v} \in \{0, 1\}$, indicating if exactly one of the endpoints of the edge is present on the current partial solution.

Just like as in MVC, the adjacency matrix is also added to the state representation. However, as the TSP on the Euclidean space induces complete graphs, the number of edges is $\Theta(n^2)$ and it grows fast for larger values of n.

A k-nearest neighbors (KNN) technique is applied to limit the number of edges processed in the embedding to help scaling the training to larger graph sizes. The adjacency matrix $\mathbf{A}$ is then computed as $\mathbf{A}_{v,u} = 1$ if either v is one of the k closest neighbors of u or u is one of the k closest neighbors of v, and $\mathbf{A}_{v,u} = 0$ otherwise. Note that, given an instance of points (nodes), the matrix $\mathbf{A}$ is defined and does not change during training of this instance

This matrix $\mathbf{A}$ and the node attributes are concatenated, leading to a state with two dimensions with ranges $(n, n + a_v)$, in which $a_v = 4$. On the other hand, edge attributes are represented as a 3-dimensional tensor with ranges (n, n, a_e), in which $a_e = 3$.

Actions

Given the state S as an ordered list of nodes representing the partial solution, an action consists of choosing another node v in the set $\overline{S} = V \setminus S$, and appending v to the end of the list.

However, the actual solution is constructed using the function $h(S)$, as mentioned in Section 2.5.1. This function takes the ordered list of inserted nodes and constructs the combinatorial structure representing the partial solution. Thus, $h(S)$ can rearrange or ignore nodes, for example. The function $h(S)$ is again dependent on the problem being solved, as discussed below.

MVC For MVC, the combinatorial structure for a solution is a set of nodes, so the order they were inserted to construct the solution is not important, and can be ignored.

In this case, $h(S) = \{u \mid u \in S\}$ is just the set of nodes in S.

TSP For TSP, the combinatorial structure for a solution is an ordered list of traversed nodes in the Hamiltonian cycle.

Therefore, the function $h(S)$ must map the ordered list S of inserted nodes to another ordered list for the actual TSP tour, and is therefore a permutation function.

One possible function is $h(S) = S$, i.e., the tour is constructed in the same order as the selection of nodes from the agent, and actions just append new nodes to the end of the tour.

Another possible function $h(S)$ is characterized by actions inserting a new node to the best position in the tour, i.e., the position with the smallest partial tour size.

Formally, let $\pi_i(S)$ denote the i-th node in the permutation constructed by h. In particular, if $h(S) = (\pi_1(S), \ldots, \pi_{|S|}(S))$ and a new node v is taken as an action, the new ordered state will become $S' = (S, v)$ and $h(S') = (\pi_1(S), \ldots, v, \ldots, \pi_{|S|}(S))$, and v is inserted at an index k, such that $w(\pi_{k-1}(S), v) + w(v, \pi_k(S)) - w(\pi_{k-1}(S), \pi_k(S))$ is minimized over all indices k, including $k = 1$, in which case $\pi_0(S) = \pi_{|S|}(S)$ and the new node is inserted at the end.

Rewards

The rewards returned by the environment at each step are completely dependent on the problem being solved, but they all should penalize bad actions more than good actions and direct the agent towards a good solution for the problem.

MVC For the MVC problem, the objective is to minimize the number of nodes in the vertex cover. Therefore, the reward for adding a node to the partial solution is simply -1 for every node. It is negative because MVC is a minimization problem, while the agent always tries to maximize its expected rewards.

TSP For TSP, the objective is to minimize the sum of edge weights in the tour. Therefore, an intuitive reward to consider for a given action v is the negative of the change (increase) in tour size after inserting the node v at the best position in the tour.

However, the original paper argues that empirical evidence shows that designing the reward as the positive change in the tour size, even if counterintuitive, usually leads to better learned policies. This might be explained by the fact that the RL agent typically gives more weight to most recent actions (and rewards), and flipping the reward function might give a hint to the agent to focus on the longer future.

Therefore, both reward functions are implemented and compared in this work.

Instance generation

The environment is also responsible for generating new problem instances. Given the random graph model and its parameters, the environment randomly generates a new graph and resets the partial solution.

For this work, graphs are sampled from three different distributions:

- **Barabási-Albert model** (BARABÁSI and ALBERT (1999)): graphs are generated with the preferential attachment mechanism, creating hubs of central and highly connected nodes. The graph is generated incrementally by adding nodes and the model has a parameter m indicating the number of edges created by each new node added in the graph.

- **Erdős–Rényi $G(n, p)$ model** (ERDÖS and RÉNYI (1959)): graphs are generated with n vertices and an independent probability p of having an edge between each pair of vertices.

- **Euclidean random 2D**: n points are sampled uniformly at random from the square $[0, L] \times [0, L]$, where L is some limit coordinate. These points become nodes of the graph and edge weights are given by the Euclidean distance between the points.

3.2.2 Model

The model consists of the modules that compute node embeddings for a given state and the action-value function approximation. It is parameterized by the set of parameters Θ.

In order to implement a procedure that can learn these parameters, one idea is to create learnable tensors with the correct dimensions described in Equations 2.22 and 2.23. That could work, but it would not be very scalable when considering that learning can be done in batches.

Another possibility is to represent these parameters as fully connected neural network layers, and direct matrix multiplication is replaced by a forward pass of the neural network layer.

Note that the model is evaluated at each iteration to compute the action-value estimates, even if only one node was changed. Because of that, this model must be as efficient as possible, so that computing the embeddings does not become a large bottleneck for the training procedure.

The first optimization is to compute each layer of the embeddings $\mu_v^{(t)}$ for all nodes v at once in one pass. Another optimization is that the terms θ_1 and θ_3 do not change from one layer t to the next layer $t + 1$ when computing $\mu_v^{(t)}$, so these terms can be computed only once for each node embedding.

The diagram for the model implementation is illustrated in the Figure 3.2 for the embedding module, which is stacked on top of itself T times (T is the total number of layers), and in the Figure 3.3 for the module computing $\widehat{Q}$, with the PyTorch layers and functions used in this implementation. These two modules are concatenated, so that a single forward pass and a single backpropagation are computed at each step.

Note that the sum of some function $f(u)$, encoded as a vector z, over neighboring nodes u of a node v, represented by $\sum_{u \in \mathcal{N}(v)} f(u)$ is implemented as a matrix multiplication $\mathbf{A}z$, computing this sum for all nodes at once. That occurs, for example, for the inputs of the θ_2 and θ_3 neural network layers.

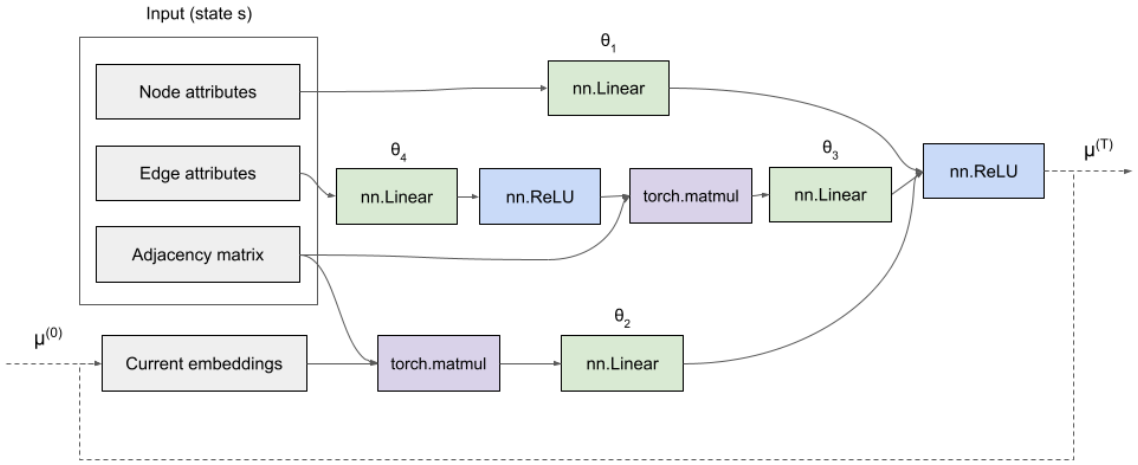


Figure 3.2: Block diagram illustrating the different layers used for the node embeddings. The θ_1 and θ_3 outputs are actually computed only once and used again in subsequent layers.

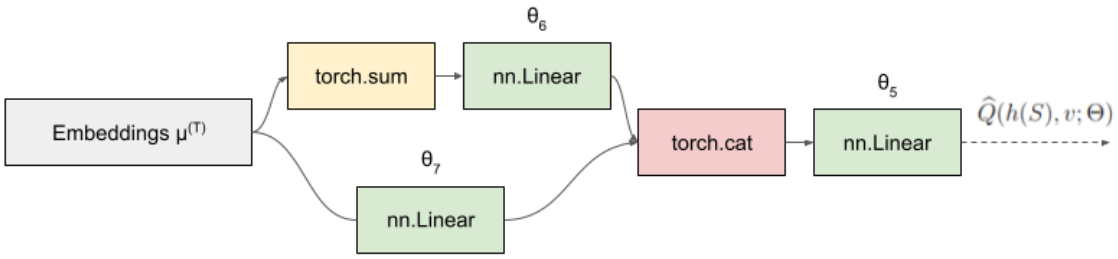


Figure 3.3: Block diagram illustrating the different layers used to compute $\widehat{Q}(h(S), v; \Theta)$

3.2.3 Agent

The agent is the part of the system that interacts with the environment. Thus, the agent is given an observation representing the current state, and the agent should choose a valid action, given the rules of the problem.

The agent is responsible for implementing all of the logic for how to behave in each state, and it is also responsible for handling how to train its underlying model, including the n-step learning procedure.

In this particular case of S2V-DQN, the agent holds two internal neural networks, one which encodes the parameters $\Theta = \{\theta_1, \ldots, \theta_7\}$ and is used to perform actions on the environment, and another neural network of identical dimension called the target network, which is described in more detail below.

Target Network

As discussed previously, a target network is used to help stabilize the training. When computing the MSE loss function in Equation 2.24, the TD target y is also computed using an estimate of the action-value function $\widehat{Q}$, which is parameterized by Θ.

If the same network was used for both calculations, updating the parameter Θ would make the network approach the current target, but then the target itself would change in the next iteration, making the training very unstable.

The target network is then implemented initially as a copy of the original network, but its parameters Θ^- are fixed to stabilize the target and thus the training. However, it should still be updated with the original network parameters Θ (that are changing with training) from time to time in order to give better estimates for the TD target. There are two ways to update the target network parameters: hard update and soft update.

The hard update is already described previously, and consists on copying the whole set of parameters from the original network to the target network every K iterations. In the soft update, the parameters of the target network are updated at every iteration, by updating them according to an exponentially weighted moving average with the parameters of the training network. More specifically, the target parameters Θ^- are updated as:

$$\Theta^- = \tau\Theta + (1-\tau)\Theta^-, \tag{3.1}$$

where $0 < \tau < 1$ and, in general, τ is closer to 0.

Both update types are implemented and supported in this implementation, but the soft update is the one used for all experiments, as empirical evidence showed no significant difference between them, but the soft update leads to smoother curves for the loss function.

Replay Buffer

The agent also has as implementation for the replay buffer, which holds experience tuples and allows for sampling batches of these tuples to update the model. An experience tuple $< S_{t-1}, v_{t-1}, R_t, S_t >$ holds information on a state, the action performed in that stated, the reward obtained by the agent, and the next state that the agent observed next. This tuple can be adjusted to process multiple steps, as described below for n-step learning.

This buffer stores tuples for all of the episodes simultaneously and has a maximum size configured before the training procedure starts. Once it gets full, the oldest experience tuples are removed from the buffer to give space for newer tuples, thus leaving more recent and more accurate tuples in the buffer.

Whenever the agent wants to update its parameters, it first sample a batch of experience tuples, then computes both the loss function on this batch and the gradient of the loss function with respect to its parameters, and then it updates its parameters based on this batch.

The sampling for this batch can be done using different distributions over the tuples. The most common distribution is the uniform distribution. If the buffer has N experience tuples and the agent samples of M tuples, then each tuples would have a probability $\frac{M}{N}$ of being selected for the batch.

There is also another way of sampling called Prioritized Replay (SCHAUL *et al.* (2015)), that gives a higher sampling probability to the samples for which the agent computed a larger error. In this way, the training can focus on the tuples that convey information that is still not learned by the agent. Although Prioritized Replay can help improving the training efficiency, the original S2V-DQN implementation uses a uniform sampling, and so does the implementation for this work.

N-Step Learning

The n-step learning feature is implemented based on the Equation 2.24. From this equation, we need to know the (S_{t-n}, v_{t-n}) state-action pair n steps prior to a given state S_t, as well as the sum of all rewards in that period.

For this, a list with the last n states and n actions is stored for each episode. When a new action is performed by the agent, the tuple $< S_{t-1}, v_{t-1}, R_t, S_t >$ is created, where $r(S_{t-1}, v_{t-1})$ is the reward observed by the agent. The state S_{t-1} and the action v_{t-1} are then inserted in the list, removing the pair (S_{t-n-1}, v_{t-n-1}). The oldest pair of state and action in this list after this operation will then be (S_{t-n}, v_{t-n}).

It is also needed to compute the sum of rewards $R_{t-n,t}$. For this, we keep track of this sum inside each episode and update it with the following recursion:

$$\begin{aligned} R_{t-n,t} &= R_{t-n+1} + \gamma R_{t_n+2} + \cdots + \gamma^{n-1} R_t \\ &= \left(\frac{\gamma R_{t-n+1} + \gamma^2 R_{t_n+2} + \cdots + \gamma^{n-1} R_{t-1}}{\gamma} \right) + \gamma^{n-1} R_t \\ &= \left(\frac{R_{t-n-1,t-1} - R_{t-n}}{\gamma} \right) + \gamma^{n-1} R_t \end{aligned} \tag{3.2}$$

This is valid for all $t \geq n$. In this way, all of the necessary terms are available to create the experience tuple $< S_{t-n}, v_{t-n}, R_{t-n,t}, S_t >$ required by n-step learning.

3.2.4 Training and Validation Loop

Figure 3.1 gives a high-level illustration of how the main components of the system fit together. This section aims to give a more thorough discussion of how the training and validation loop works.

The training and validation loop is implemented so that a batch of training episodes is run, allowing the agent to update its parameters, and then a smaller batch of validation episodes (e.g., 10) are performed, so that the agent can assess its performance. During validation, model parameters are not updated.

A training-validation loop consists of the steps described by Algorithm 4.

Algorithm 4: Training-validation loop

```
Initialize ε used for exploration
Initialize optimizer's learning rate
for episode = 1 to E do
    Run training episode using ε and learning rate
    Report episode's average loss function
    Linearly decay ε
    if run batch of validation episodes every V episodes then
        Run N validation episodes
        Report validation metric for these episodes
    end
    Decay learning rate on certain episodes
end
```

The exploration parameter ε is configured to start at a very high value, so that the agent primarily explores new states in the beginning, but ε continuously decreases, making the agent prefer to make actions based on its learned parameters. For the validation episodes, $\varepsilon = 0$, so that no exploration is performed in this case.

Going further, Algorithm 5 provides the details for the training procedure.

Algorithm 5: Training episode

Environment generates a new instance and notifies the agent with the new observation
Agent resets all of its internal variables, e.g., the list of states and actions and the sum of rewards for the n-step learning procedure
for *t = 1, 2, ...* **do**
    Agent returns an action for the environment using ε-greedy procedure
    Environment updates its internal state representation, returns the new state, the reward and indicates if the episode is finished
    if $t \geq n$ **then**
        The agent adds the n-step experience tuple to its memory buffer $\mathcal{D}$
    end
    if $/\mathcal{D}/ < M$ **then**
        Continue to the next iteration or exit if finished
    end
    Agent samples batch B of experience tuples with size M
    Agent inputs the batch for the model and obtains the estimate $\widehat{Q}(h(S), v; \Theta)$ for all nodes v
    Agent determines the maximum value for all valid nodes
    Agent computes TD target y
    Agent computes MSE loss function from its estimate to the TD target y
    Agent computes gradients of loss function with respect to model parameters
    Agent backpropagates the gradients and updates the model parameters
    Agent hard updates target network every K iterations
    if *episode is finished* **then**
        Exit the episode
    end
end

A validation episode looks just like a subset of a training episode, in which the agent interacts with the environment, but it does not need to store experience tuples or update its parameters from batches of experiences. It is described by Algorithm

6.

Algorithm 6: Validation episode

```
1 Environment generates a new instance and notifies the agent with the new
    observation
2 for t = 1, 2, ... do
3 |   Agent returns an action for the environment using ε
4 |   Environment updates its internal state representation, returns the new
  |     state, the reward and indicates if the episode is finished
5 |   if episode is finished then
6 |   |   Exit the episode
7 |   end
8 |   Computes validation metric using the agent's solution and the optimal
  |     solution obtained by another solver
9 end
```

The validation metric and the solvers chosen for each task (see line 8 in Algorithm 6) are discussed in more detail in Section 4.1.

Chapter 4

Experiments and Results

Several experiments were conducted to evaluate the performance of the trained agent, both for TSP and MVC. In the following sections, we describe the evaluation metric used to validate and test the agents, as well as the parameters used to generate new instances and train the model, along with the results for each problem.

4.1 Evaluation Metrics

For the evaluation of a trained agent (model), we use the Approximation Ratio, defined as

$$\mathcal{R}(S, G) = \frac{c(h(S))}{OPT(G)} \tag{4.1}$$

where $c(h(S))$ is the objective value of the solution $h(S)$ for a sequence S and $OPT(G)$ is the objective value for the optimal solution of instance G. Intuitively, $\mathcal{R}(S, G) \geq 1$ and the closer it is to 1, the better.

To evaluate the optimal solution, we use the following numerical solvers:

- For smaller TSP instances with $n < 20$, we use a dynamic programming solver which implements the Bellman–Held–Karp algorithm (BELLMAN (1962); HELD and KARP (1962)), with a $O(n^2 2^n)$ time complexity, while for larger TSP instances, we use the Concorde [1] solver, with a cutoff of 100 seconds, in which case it reports the best solution found so far. This cutoff is chosen because validation is performed multiple times for each training procedure. In this case, it might be possible that in some cases our solutions is slightly better that the solver's solution, leading to an Approximation Ratio less than 1.

- For MVC, we use an open source mixed integer programming solver library in Python called PuLP [2] to solve the Mixed Integer Linear Programming for-

[1] `https://www.math.uwaterloo.ca/tsp/concorde.html`
[2] `https://github.com/coin-or/pulp`

mulation given by Equation 2.20. We use the default CBC solver (FORREST *et al.* (2024)), which uses a branch-and-cut method.

4.2 Training parameters

For the training procedure, the final hyperparameter values below were chosen trying to replicate the values used in the original paper. However, some of them are not fully documented (e.g., ε-decay schedule) and were chosen after some initial experimentation on independently generated instances. This is the full list of hyperparameters used for training:

- The number of layers in the embedding computation is $T = 4$ for TSP and $T = 5$ for MVC;
- The node and edge embeddings have dimension $p = 64$;
- The discount factor is set as $\gamma = 0.1$ for TSP and $\gamma = 1.0$ for MVC;
- ε begins with 100% and linearly decays on each episode for the first 90% episodes until reaching 1%, and stays at this level for the remainder 10% of the episodes;
- The target network is updated using the soft update rule, described by Equation 3.1, using $\tau = 0.005$;
- The agent samples batches of experience of size 64 for TSP and 128 for MVC;
- Although n-step learning is implemented, $n > 1$ empirically didn't showed any noticeable improvement, so $n = 1$ is used;
- A warmup period of 1000 steps is implemented, so that the agent only starts learning after this number of steps. This helps adding some diversity in the replay buffer before the agent starts adjusting its model parameters;
- The Adam optimizer learning rate is chosen as 10^{-3} for the first 70% of the episodes, and decayed to 10^{-4} for the remainder 30% of the episodes.

4.3 Instances Generation

Instances are generated using the random graph models described in Section 3.2.1. Independent graphs are randomly generated for each episode using the models described below. This ensures that training and validation use independent and identically distributed instances.

For MVC, the Barabási-Albert (BA) model and the Erdős–Rényi (ER) $G(n, p)$ model are used. For the BA model, we use the parameter $m = 4$, just like in the original paper, which means that 4 edges are created for each new node added to the graph during its generation.

For the ER model, we experiment with 2 different values of p, which are $p = 0.15$ (the one used in the original paper) and $p = 0.30$. The reasoning is that the expected degree of a node in a graph generated using this model is $p(n - 1)$, and $p = 0.15$ makes this value very small when n is small (say $n = 10$).

For the TSP problem, we generate instances for the Euclidean version of the problem. The points are drawn at random from the $[0, 10^6] \times [0, 10^6]$ square. The maximum value of 10^6 is chosen, because the Concorde solver used for larger instances only works with integer coordinates, so that they can be rounded to the nearest integer when using this solver.

As mentioned previously, a KNN technique is also used for the Euclidean TSP, and K is chosen to be 10, so that the 10 closest nodes are kept in the neighborhood of a node to compute its embeddings.

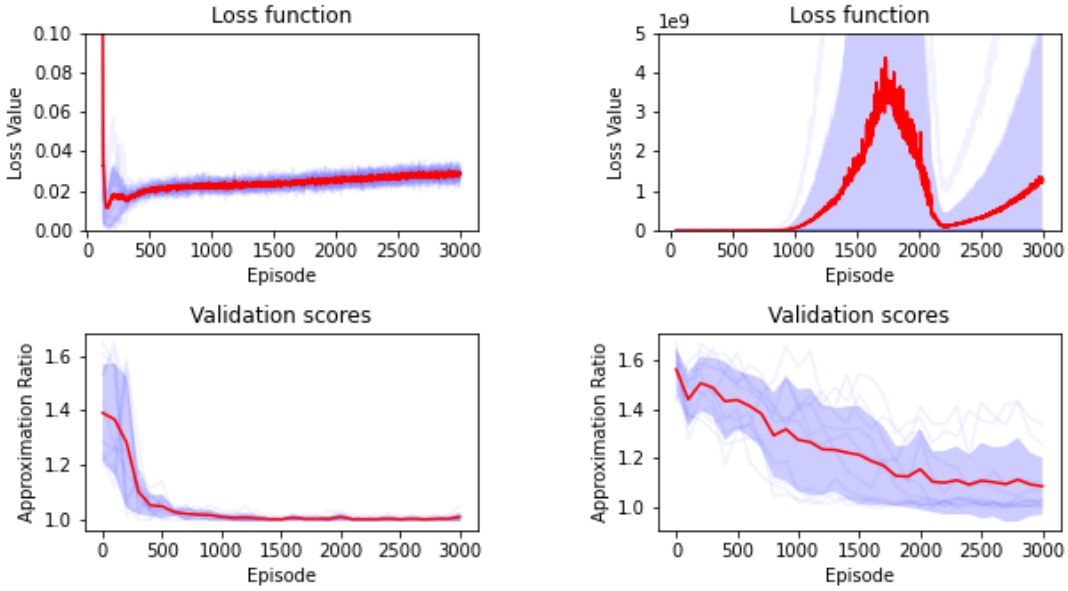
4.4 Results

In this section, we present the results for the experiments performed with the MVC and TSP problems.

For each experiment configuration, we present two graphs indicating the convergence for the training of 10 independent agents, all trained from scratch. The first graph is the average loss function during an episode, for all episodes. The second graph reports the average approximation ratio for each batch of 10 validation episodes. This validation is performed every 100 training episodes.

Both graphs plot the curves for individual runs (in a light blue line), as well as the average value across all 10 different runs at each episode (in a red line), and a blue shaded area indicating 1 standard deviation from the average.

We would like to compare our implementation with the original one, but its code dates back to 2017 and we encountered some compatibility issues to run it. We also tried to run other implementations for S2V-DQN, such as the one provided by BARRETT *et al.* (2019) in their ECO-DQN paper, but they only implement the Maximum Cut problem, and adapting it to MVC and TSP would require making many modifications, defeating the purpose of comparing independent implementations. Therefore, we only trained our implementation and also report the results published in the original S2V-DQN paper (KHALIL *et al.* (2017)).



(a) $n = 10$ (b) $n = 25$

Figure 4.1: Loss function and approximation ratio for MVC, trained and evaluated on graphs generated by the BA model.

4.4.1 MVC

For MVC, we train agents for graph sizes of 10, 15, 20 and 25 for BA and ER graphs, all with 3000 episodes. We present and discuss convergence plots for both graph models, for sizes $n = 10$ and $n = 25$.

Barabási-Albert

For the training with BA graphs with $n = 10$ nodes, the convergence happened quite quickly, and the average approximation ratio stayed very close to 1.0 after the first 1000 episodes, while the loss function stabilized also in the first hundreds of episodes, as shown in Figure 4.1(a).

On the other hand, for the training with $n = 25$ nodes, 1 out of the 10 independent agents actually was not able to converge. It could not recover from there, causing its loss to explode. We can see from the plot in Figure 4.1(b) with the validation metric that a few agents (light blue lines) were capable of converging to find approximation ratios close to 1.0, but a lot of the other agents could not, even after 3000 episodes, causing the average to be closer to 1.1.

Erdős–Rényi

For the ER model, the two probability values of $p = 0.15$ and $p = 0.30$ lead to similar results for $n = 25$, as shown in Figure 4.2, while for $n = 10$ and $p = 0.15$

the approximation ratio is very noisy throughout the episodes, probably because the number of edges is very small.

It is worth noting that, in all cases, the loss function keeps increasing over the episodes, even if only a little. This can partially be explained by the exploration policy, in which the agent is constantly choosing suboptimal actions, just to better explore the state space. In this case, the validation graph is very important to make sure that the agent is actually learning a good policy. This seems to be the case for $n = 10$ and $n = 25$, in which the average approximation ratio decreases largely on the first 1500 episodes.

All Results

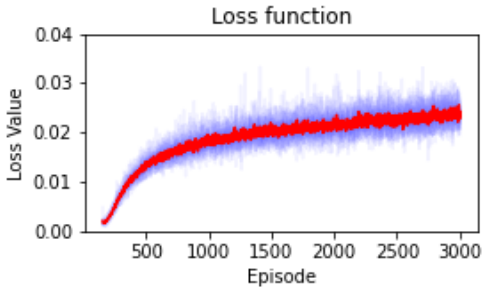
We present all the results for the four graph sizes previously mentioned in Table 4.1. We take the best average approximation ratio for all validation episodes across the 10 runs. This is calculated by computing the average approximation ratio across the 10 runs for each validation episode, and taking the minimum value for all of these averages. We also report the standard deviation of the approximation ratio on those runs. Interestingly, the performance for BA model seems to decrease with n, while for ER model it appears to increase with n.

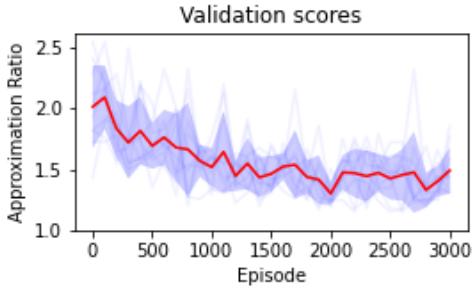
For BA, our agents performed well on smaller graphs, but as the graph size increased, the approximation ratio worsened. Indeed, in Figure 4.1(b) we can see that it started diverging for $n = 25$ and also diverged for larger graph sizes.

For ER, we our agents actually performed better for larger graphs, and also for the larger probability $p = 0.30$ when compared to $p = 0.15$. One hypothesis we raised to explain this difference is that graphs generated using the ER model are often disconnected, specially for lower values of p, and that the learned policies could perform better on connected graphs, when compared to disconnected graphs. To validate this hypothesis, we trained agents on both connected and disconnected $G(n = 20, p = 0.30)$ graphs. For disconnected graphs, the best approximation ratio over 10 runs was 1.0532 ± 0.0191. For connected graphs, the measured ratio was 1.0459 ± 0.0219. Therefore, we cannot conclude that connected graphs lead to better approximation ratio, because these two intervals of 1 standard deviation overlap.

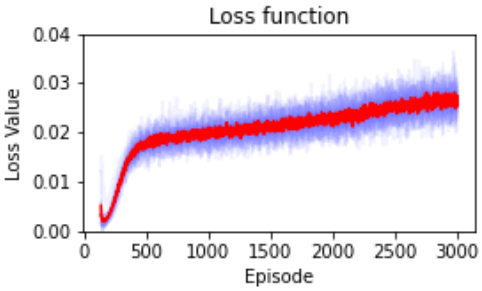
n	BA	ER ($p = 0.15$)	ER ($p = 0.30$)
10	1.0017 ± 0.0053	1.3037 ± 0.0945	1.0823 ± 0.0442
15	1.0079 ± 0.0055	1.1514 ± 0.0600	1.0619 ± 0.0137
20	1.0144 ± 0.0114	1.0903 ± 0.0327	1.0497 ± 0.0173
25	1.0856 ± 0.1161	1.0773 ± 0.0298	1.0424 ± 0.0145

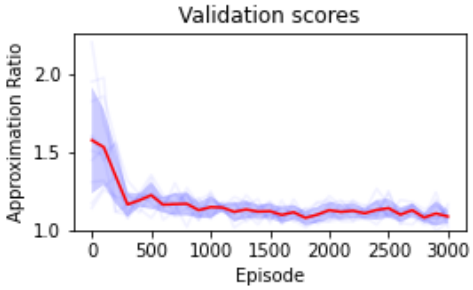
Table 4.1: Best average approximation ratio for each graph size for MVC, also reported with the standard deviation.



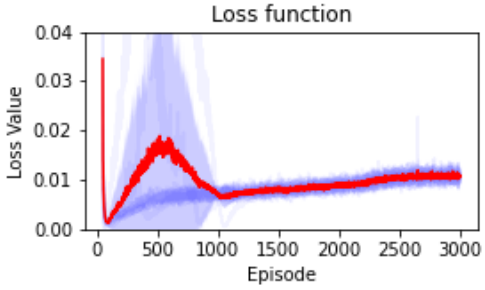


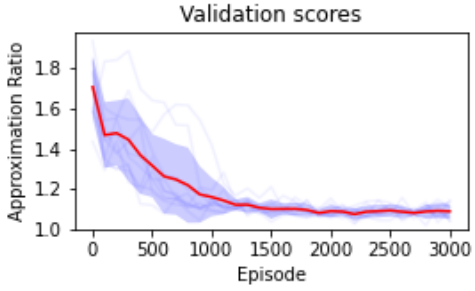
(a) $n = 10,\ p = 0.15$



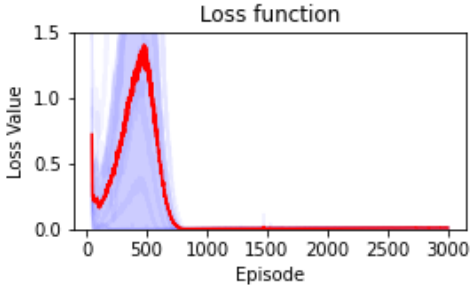


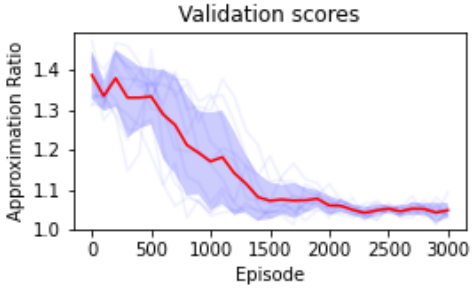
(b) $n = 10,\ p = 0.30$





(c) $n = 25,\ p = 0.15$





(d) $n = 25,\ p = 0.30$

Figure 4.2: Loss function and approximation ratio for MVC, trained and evaluated on graphs generated by the ER model.

The original paper (KHALIL *et al.* (2017)) reports the results for ranges of graph sizes, meaning that it generates graphs with any size in the range for both training and testing. Their approximation ratios for similar graph sizes are in Table 4.2.

n	BA	ER ($p = 0.15$)
15-20	1.0016	1.0032
40-50	1.0027	1.0037

Table 4.2: Results reported in the original S2V-DQN paper (KHALIL *et al.* (2017)) for both BA and ER graphs.

Looking at these results, we can see that the agents were able to learn good policies, but not as good as the agents trained by KHALIL *et al.* (2017). This indicates that the implementation is missing some details used in the original implementation, or that hyperparameters have been tuned differently.

4.4.2 TSP

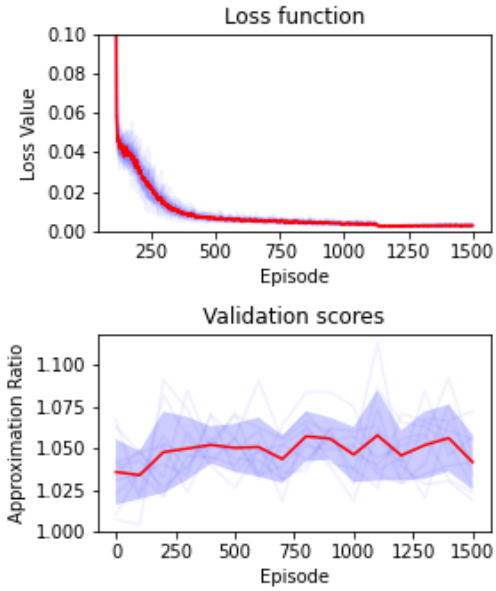
For TSP, training was performed for graphs of sizes 10, 15, 20, 25, 40, 60, all with 1500 episodes. The first 4 graph sizes were trained with both designs for the reward function as described previously: one with a negative reward for each action, indicating the negative of the increase in partial tour size, and one with a positive reward for each action, by flipping the sign of the previous reward function.

The results for $n = 10$ and $n = 15$ are shown in Figure 4.3, which confirms the hypothesis that the positive version of the reward function helps the agent learn better policies when compared to the negative rewards. In the negative reward setup, the average approximation ratio does not decrease over the episodes, which indicates that the model is not learning to generalize, even with the reduction in the loss function. On the other hand, for the positive reward setup, the average approximation ration decreased over the first 200 episodes and then stayed approximately flat, indicating that learning to generalize happened in the beginning of the training.

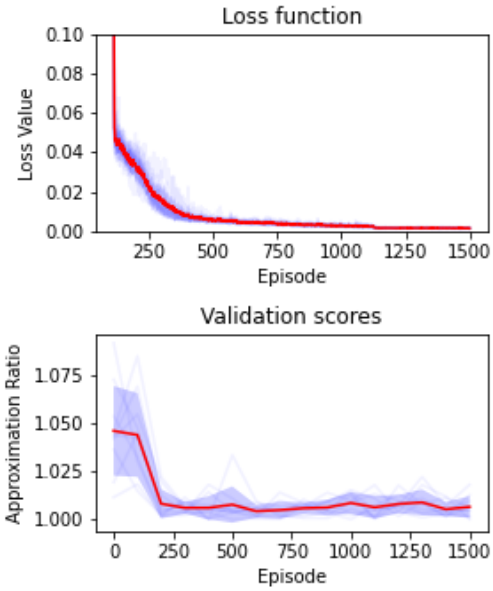
Larger graph sizes take longer to train, so only the positive rewards, which are known to perform better, are used for $n = 40$ and $n = 60$.

Table 4.3 shows the full results for the different graph sizes. We once again take the best average approximation ratio for all validation episodes across the 10 runs. We also report the standard deviation of the approximation ratio on those runs.

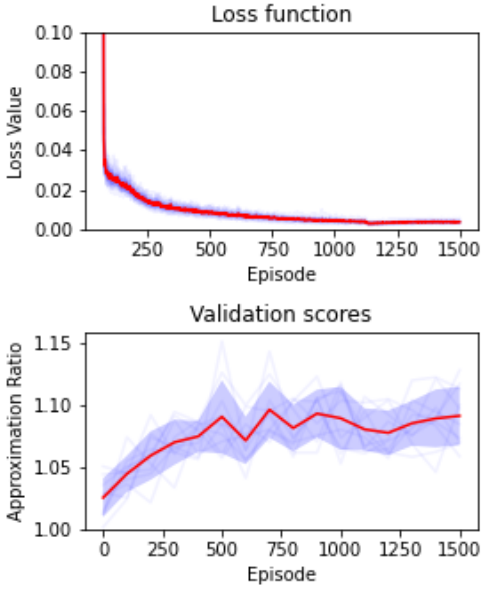
The approximation ratios for similar graph sizes reported in the original paper (KHALIL *et al.* (2017)) are shown in Table 4.4. From these results, we conclude that our trained agents were successful in learning policies comparable to the original implementation of S2V-DQN.



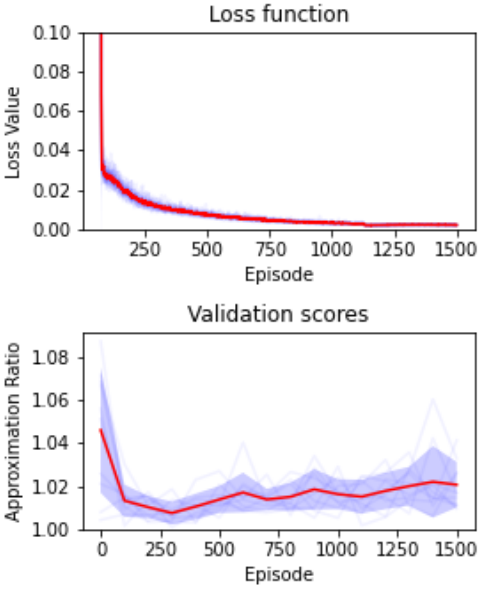
(a) $n = 10$, negative rewards



(b) $n = 10$, positive rewards



(c) $n = 15$, negative rewards



(d) $n = 15$, positive rewards

Figure 4.3: Loss function and approximation ratio for the TSP with $n = 10$ and $n = 15$ nodes, using negative rewards and positive rewards.

n	Negative rewards	Positive rewards
10	1.0339 ± 0.0144	1.0043 ± 0.0033
15	1.0259 ± 0.0155	1.0078 ± 0.0053
20	1.0458 ± 0.0415	1.0061 ± 0.0321
25	1.0772 ± 0.0447	1.0118 ± 0.0452
40	-	1.0443 ± 0.0114
60	-	1.0607 ± 0.0099

Table 4.3: Best average approximation ratio for each graph size for TSP, also reported with the standard deviation.

n	Approx. Ratio
15-20	1.0147
40-50	1.0533
50-100	1.0701

Table 4.4: Results reported in the original S2V-DQN paper (KHALIL *et al.* (2017)) for Euclidean TSP random graphs.

Generalizing to Larger Instances

We also conducted some experiments, in which we took one arbitrary agent out of the 10 independently trained agents for each of the previously mentioned graph sizes and test them on larger graphs, to better understand the generalization capabilities of our S2V-DQN implementation applied to TSP.

Table 4.5 shows the results obtained, where the average approximation ratio and the standard deviation are calculated over 10 different test instances. Recall that the optimal solution on larger graphs is actually calculated using the Concorde TSP solver with a time limit of 100 seconds, and is not truly optimal, but serves as a good benchmark for comparison. Note that, given a test graph size (say 100), training on larger graphs does not necessarily improve performance. Thus, the dependency between train graph size and performance on test graphs of different sizes is far from clear.

Train \Test	40	60	100	500	1000
10	1.0751 ± 0.0282	1.0927 ± 0.0274	1.0885 ± 0.0216	1.1295 ± 0.0089	1.1304 ± 0.0095
15	1.0502 ± 0.0322	1.0851 ± 0.0255	1.0967 ± 0.0238	1.1395 ± 0.0154	1.1434 ± 0.0083
20	1.0522 ± 0.0315	1.0790 ± 0.0354	1.1111 ± 0.0198	1.1675 ± 0.0156	1.1848 ± 0.0126
25	1.0497 ± 0.0271	1.0625 ± 0.0227	1.1027 ± 0.0371	1.1901 ± 0.0131	1.2059 ± 0.0106
40	1.0696 ± 0.0452	1.0743 ± 0.0267	1.0979 ± 0.0220	1.2205 ± 0.0170	1.2331 ± 0.0139
60	1.0632 ± 0.0398	1.0837 ± 0.0302	1.0925 ± 0.0231	1.1501 ± 0.0079	1.1685 ± 0.0115

Table 4.5: Average approximation ratio for each combination of sizes for train and test for TSP, also reported with the standard deviation computed across 10 different instances.

The original paper (KHALIL *et al.* (2017)) also reports their approximation ratio for tests on larger graphs, from which we present the numbers for similar graph

Train \Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0147	1.0511	1.0702	1.0913	1.1022	1.1102	1.1124	1.1156	1.1212
40-50	-	1.0533	1.0701	1.0890	1.0978	1.1051	1.1583	1.1587	1.1609
50-100	-	-	1.0701	1.0871	1.0983	1.1034	1.1071	1.1101	1.1171

Table 4.6: Results reported in the original S2V-DQN paper for Euclidean TSP random graphs.

sizes in Table 4.6. From these results, we can conclude that our implementation is similarly capable of generalizing the training performed on smaller to larger graphs, when compared to the original implementation. Our approximation ratios are just slightly larger than the ones the authors reported in their work.

4.4.3 Detailed Learning Analysis

From the results in the previous sections, we conclude that the agents were able to learn how to find good policies to solve the CO problems, especially for TSP.

Another way we can check that learning is happening throughout the episodes is by observing the gradients of the parameters evolving and getting more and more concentrated around 0. In order to visualize the evolution of these values, we use TensorBoard (ABADI *et al.* (2016)).

For one training loop, we can plot the distribution of the gradients and the values for specific parameters across the episodes. For example, Figures 4.4 and 4.5 show these two distributions for the set of parameters θ_5 in Equation 2.23, when training an agent for 1500 episodes to solve the Euclidean TSP with a graph of size $n = 15$. The distributions are stacked on top of each other for all episodes, with the final episode on the bottom. We can see that the gradients tend to concentrate around 0 and that the distribution of the values for this parameter tend to stabilize over time. However, the values of the parameters move from a normal-like distribution (in episode 1) to a multi-modal distribution (in episode 1500), indicating that learning did happen.

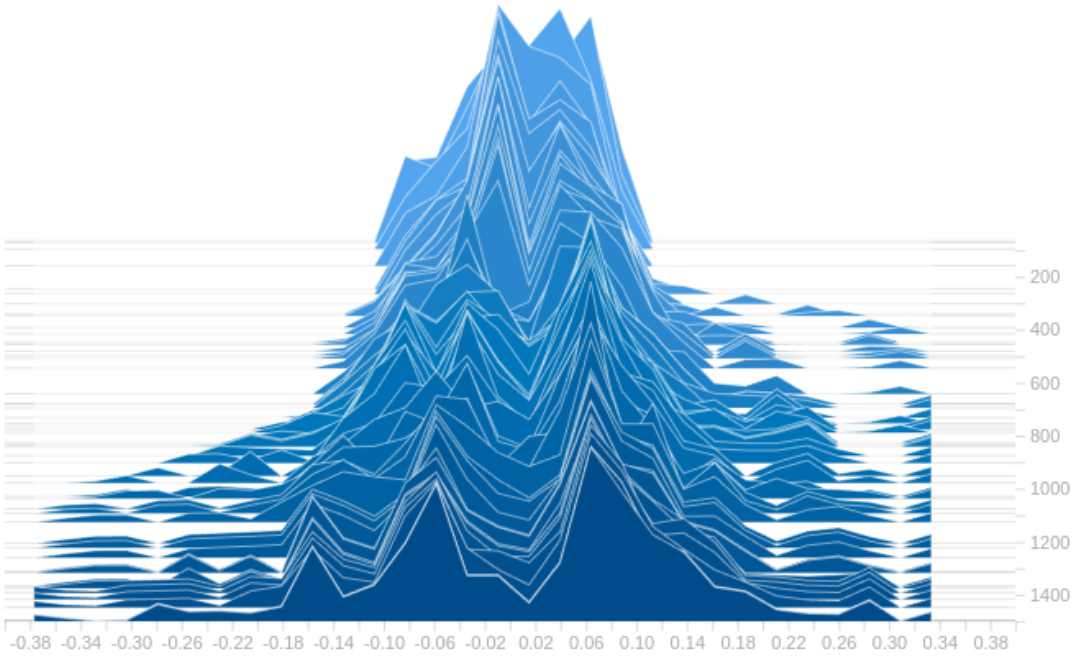


Figure 4.4: Distributions of weights for θ_5 parameters throughout the 1500 episodes.

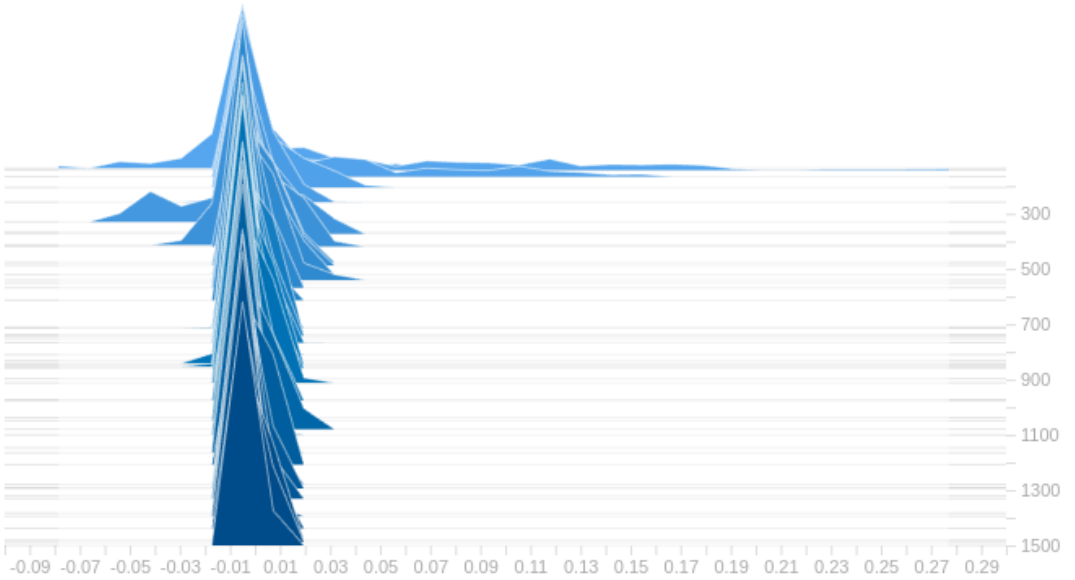


Figure 4.5: Distributions of gradients for θ_5 parameters throughout the 1500 episodes.

Chapter 5

Conclusion

Throughout this work, we were able to explore several aspects of Deep Reinforcement Learning techniques to solve Combinatorial Optimization problems. We started by discussing the theoretical background needed for this work, then reviewed some of the literature on the area, by reading two surveys, and then focused particularly on the S2V-DQN framework.

To better understand and discuss in detail how S2V-DQN works, we contribute with an independent implementation of the model and the greedy framework, and also evaluate it on different scenarios for both the MVC and the Euclidean TSP CO problems.

In the experimental phase of our work, we report only the final values used for each hyperparameter, but a few other different hyperparameter configurations were explored. For example, larger learning rate values caused the training to be very unstable, with large peaks in the loss function curve. All of this experimentation helped us get a better understanding of how to train a DRL agent.

Unfortunately, we could not compare our implementation with the original one, so we compare the numbers reported in the S2V-DQN paper. We conclude that our implementation achieves very good approximation ratio values for the TSP, but not so good for MVC, indicating that some details might be missing (e.g. extra node or edge attributes), or that hyperparameter tuning is still suboptimal.

For the TSP, we also observe a good generalization capacity, in which agents trained on small graphs were able to perform well on larger instance sizes. This indicates that the S2V-DQN embedding and $\widehat{Q}$ calculation are able to capture well the local structure of the nodes, as well as the global graph representation, and use these to scale to larger networks.

5.1 Future Work

Adding comparisons with other algorithms to the results section is left as future work. Several heuristics and approximation algorithms have been used in the past years to solve both MVC (e.g., 2-approximation algorithms from PAPADIMITRIOU and STEIGLITZ (1982)) and TSP (e.g., Christofides algorithm (CHRISTOFIDES (1976)) or adding the closest vertex at each iteration).

Some improvements could also be made on this work and are left as future work. The first improvement is to better scale our implementation for larger graphs. As we store the whole adjacency matrix and use it to perform matrix multiplication, even for sparse graphs (which is the case for Erdős–Rényi graphs with low edge probability or for Euclidean TSP with KNN), we end up with a computationally heavy task. We could improve on that by working with sparse data structures, like sparse matrices or even adjacency lists.

Another improvement is hyperparameter tuning. There are many different hyperparameters used in the training, so an efficient search algorithm can be valuable. The tuning in this work was done manually, by tracking individual contributions of each parameter, without taking into account the combined effect of the different hyperparameters. Other tuning methods, such as grid search, random search, or Bayesian optimization (NGUYEN (2019)) could help us find even better results.

A third improvement could be a change in the architecture of the neural network. This can be a minor change, like replacing the current layers with different ones (e.g., other nonlinear activation functions, adding dropout layers, etc.), or major changes, like completely changing the model architecture, which would mean completely changing the parametrization of the function F in Equation 2.21.

At last, if the goal is just solving CO problems using graph and node embeddings, we could also experiment with other more modern Graph Neural Network (GNN) architectures and its variants, such as Graph Convolutional Network (GCN), which aggregates information from neighboring nodes using convolutional operations, Graph Attention Network (GAT), which uses attention mechanisms to assign importance to different neighbors, or Graph Recurrent Network (GRN), which processes graph data sequentially to capture dynamic dependencies over nodes. All of them also make usage of the message-passing mechanism in order to capture the local structure from the neighborhood of each node.

References

ABADI, M., AGARWAL, A., BARHAM, P., et al., 2016. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". Available at: <https://arxiv.org/abs/1603.04467>.

APPLEGATE, D., BIXBY, R., CHVÁTAL, V., et al., 2006, "The Traveling Salesman Problem: A Computational Study", *The Traveling Salesman Problem: A Computational Study*, (01).

BARABÁSI, A.-L., ALBERT, R., 1999, "Emergence of Scaling in Random Networks", *Science*, v. 286, n. 5439, pp. 509–512. doi: 10.1126/science.286.5439.509. Available at: <https://www.science.org/doi/abs/10.1126/science.286.5439.509>.

BARRETT, T. D., CLEMENTS, W. R., FOERSTER, J. N., et al., 2019, "Exploratory Combinatorial Optimization with Reinforcement Learning", *arXiv preprint arXiv:1909.04063*.

BEBIS, G., GEORGIOPOULOS, M., 1994, "Feed-forward neural networks", *Ieee Potentials*, v. 13, n. 4, pp. 27–31.

BELLMAN, R., 1957, "A Markovian Decision Process", *Indiana Univ. Math. J.*, v. 6, pp. 679–684. ISSN: 0022-2518.

BELLMAN, R., 1962, "Dynamic Programming Treatment of the Travelling Salesman Problem", *J. ACM*, v. 9, n. 1 (jan.), pp. 61–63. ISSN: 0004-5411. doi: 10.1145/321105.321111. Available at: <https://doi.org/10.1145/321105.321111>.

BENGIO, Y., LODI, A., PROUVOST, A., 2021, "Machine learning for combinatorial optimization: A methodological tour d'horizon", *European Journal of Operational Research*, v. 290, n. 2, pp. 405–421. ISSN: 0377-2217. doi: https://doi.org/10.1016/j.ejor.2020.07.063. Available at: <https://www.sciencedirect.com/science/article/pii/S0377221720306895>.

BŁAŻEWICZ, J., FORMANOWICZ, P., KASPRZAK, M., et al., 1999, "DNA sequencing with positive and negative errors", *Journal of Computational Biology*, v. 6, n. 1, pp. 113–123.

BOVET, D., CRESCENZI, P., 1994, *Introduction to the Theory of Complexity*. Prentice-Hall international series in civil engineering and engineering mechanics. Prentice Hall. ISBN: 9780139153808. Available at: <https://books.google.com.br/books?id=xucmAQAAIAAJ>.

BRIN, S., PAGE, L., 1998, "The Anatomy of a Large-Scale Hypertextual Web Search Engine". In: *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, p. 107–117, NLD. Elsevier Science Publishers B. V.

BROCKMAN, G., CHEUNG, V., PETTERSSON, L., et al., 2016. "OpenAI Gym". .

CHEN, J., KANJ, I. A., JIA, W., 2001, "Vertex Cover: Further Observations and Further Improvements", *Journal of Algorithms*, v. 41, n. 2, pp. 280–301. ISSN: 0196-6774. doi: https://doi.org/10.1006/jagm.2001.1186. Available at: <https://www.sciencedirect.com/science/article/pii/S0196677401911861>.

CHO, K., VAN MERRIENBOER, B., ÇAGLAR GÜLÇEHRE, et al., 2014, "Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation". In: *Conference on Empirical Methods in Natural Language Processing*. Available at: <https://api.semanticscholar.org/CorpusID:5590763>.

CHRISTOFIDES, N., 1976, "Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem", *Operations Research Forum*, v. 3. Available at: <https://api.semanticscholar.org/CorpusID:123194397>.

DAI, H., DAI, B., SONG, L., 2016, "Discriminative embeddings of latent variable models for structured data". In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, p. 2702–2711. JMLR.org.

DANTZIG, G. B., RAMSER, J. H., 1959, "The Truck Dispatching Problem", *Management Science*, v. 6, pp. 80–91. Available at: <https://api.semanticscholar.org/CorpusID:154381552>.

DENG, J., DONG, W., SOCHER, R., et al., 2009, "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*.

DUCHI, J., HAZAN, E., SINGER, Y., 2011, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization", *J. Mach. Learn. Res.*, v. 12, n. null (jul.), pp. 2121–2159. ISSN: 1532-4435.

ERDÖS, P., RÉNYI, A., 1959, "On Random Graphs I", *Publicationes Mathematicae Debrecen*, v. 6, pp. 290–297.

FESTA, P., 2014, "A brief introduction to exact, approximation, and heuristic algorithms for solving hard combinatorial optimization problems". In: *2014 16th International Conference on Transparent Optical Networks (ICTON)*, pp. 1–20. doi: 10.1109/ICTON.2014.6876285.

FILIOL, E., FRANC, E., GUBBIOLI, A., et al., 2007, "Combinatorial optimisation of worm propagation on an unknown network", *International Journal of Computer Science*, v. 2, n. 2, pp. 124–130.

FORREST, J., RALPHS, T., VIGERSKE, S., et al., 2024. "Coin-or/CBC: Release releases/2.10.12". .

GOODFELLOW, I., BENGIO, Y., COURVILLE, A., 2016, *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

HELD, M., KARP, R. M., 1962, "A Dynamic Programming Approach to Sequencing Problems", *Journal of the Society for Industrial and Applied Mathematics*, v. 10, n. 1, pp. 196–210. doi: 10.1137/0110015. Available at: <`https://doi.org/10.1137/0110015`>.

HINTON, G., SRIVASTAVA, N., SWERSKY, K., 2013. "RMSProp: Divide the gradient by a running average of its recent magnitude". Available at: <`http://www.cs.toronto.edu/~hinton/coursera/lecture6/lec6.pdf`>.

HOCHREITER, S., SCHMIDHUBER, J., 1997, "Long Short-term Memory", *Neural computation*, v. 9 (12), pp. 1735–80. doi: 10.1162/neco.1997.9.8.1735.

KARP, R. M., 1972, "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations: Proceedings of a symposium on the Complexity of Computer Computations, held March 20–22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, and sponsored by the Office of Naval Research, Mathematics Program, IBM World Trade Corporation, and the IBM Research Mathematical Sciences Department*, pp. 85–103, Boston, MA, Springer US. ISBN: 978-1-4684-2001-2. doi: 10.1007/978-1-4684-2001-2_9. Available at: <`https://doi.org/10.1007/978-1-4684-2001-2_9`>.

KHALIL, E., DAI, H., ZHANG, Y., et al., 2017, "Learning combinatorial optimization algorithms over graphs", *Advances in neural information processing systems*, v. 30.

KINGMA, D. P., BA, J., 2014, "Adam: A Method for Stochastic Optimization", *CoRR*, v. abs/1412.6980. Available at: <https://api.semanticscholar.org/CorpusID:6628106>.

KIRAN, B. R., SOBH, I., TALPAERT, V., et al., 2022, "Deep Reinforcement Learning for Autonomous Driving: A Survey", *IEEE Transactions on Intelligent Transportation Systems*, v. 23, n. 6, pp. 4909–4926. doi: 10.1109/TITS.2021.3054625.

KOBER, J., BAGNELL, J. A., PETERS, J., 2013, "Reinforcement learning in robotics: A survey", *The International Journal of Robotics Research*, v. 32, n. 11, pp. 1238–1274.

KRIZHEVSKY, A., SUTSKEVER, I., HINTON, G. E., 2012, "ImageNet Classification with Deep Convolutional Neural Networks". In: Pereira, F., Burges, C. J. C., Bottou, L., et al. (Eds.), *Advances in Neural Information Processing Systems*, v. 25. Curran Associates, Inc. Available at: <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>.

LANCIA, G., BAFNA, V., ISTRAIL, S., et al., 2001, "SNPs Problems, Complexity, and Algorithms". In: auf der Heide, F. M. (Ed.), *Algorithms — ESA 2001*, pp. 182–193, Berlin, Heidelberg. Springer Berlin Heidelberg. ISBN: 978-3-540-44676-7.

LI, Z., LIU, F., YANG, W., et al., 2022, "A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects", *IEEE Transactions on Neural Networks and Learning Systems*, v. 33, n. 12, pp. 6999–7019. doi: 10.1109/TNNLS.2021.3084827.

LIN, L.-J., 1992, "Self-improving reactive agents based on reinforcement learning, planning and teaching", *Machine Learning*, v. 8, n. 3-4 (maio), pp. 293–321. doi: 10.1007/bf00992699. Available at: <https://doi.org/10.1007/bf00992699>.

MATAI, R., SINGH, S. P., MITTAL, M. L., 2010, "Traveling salesman problem: an overview of applications, formulations, and solution approaches", *Traveling salesman problem, theory and applications*, v. 1, n. 1, pp. 1–25.

MAZYAVKINA, N., SVIRIDOV, S., IVANOV, S., et al., 2021, "Reinforcement learning for combinatorial optimization: A survey", *Computers & Operations Research*, v. 134, pp. 105400. ISSN: 0305-0548. doi: https://doi.org/10.1016/j.cor.2021.105400. Available at: <https://www.sciencedirect.com/science/article/pii/S0305054821001660>.

MNIH, V., KAVUKCUOGLU, K., SILVER, D., et al., 2013. "Playing Atari with Deep Reinforcement Learning". .

MNIH, V., KAVUKCUOGLU, K., SILVER, D., et al., 2015, "Human-level control through deep reinforcement learning", *Nature*, v. 518, n. 7540 (Feb), pp. 529–533. ISSN: 1476-4687. doi: 10.1038/nature14236. Available at: <https://doi.org/10.1038/nature14236>.

NAIR, V., HINTON, G. E., 2010, "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML'10, p. 807–814, Madison, WI, USA. Omnipress. ISBN: 9781605589077.

NGUYEN, V., 2019, "Bayesian Optimization for Accelerating Hyper-Parameter Tuning". In: *2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, pp. 302–305. doi: 10.1109/AIKE.2019.00060.

NIKOLAKOPOULOS, A., SARIMVEIS, H., 2008, "A metaheuristic approach for the sequencing by hybridization problem with positive and negative errors", *Engineering Applications of Artificial Intelligence*, v. 21, n. 2, pp. 247–258.

PAPADIMITRIOU, C., STEIGLITZ, K., 1982. "Combinatorial Optimization: Algorithms and Complexity". 01.

PAPADIMITRIOU, C. H., 1977, "The Euclidean travelling salesman problem is NP-complete", *Theoretical Computer Science*, v. 4, n. 3, pp. 237–244. ISSN: 0304-3975. doi: https://doi.org/10.1016/0304-3975(77)90012-3. Available at: <https://www.sciencedirect.com/science/article/pii/0304397577900123>.

PASZKE, A., GROSS, S., MASSA, F., et al., 2019, "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: Wallach, H., Larochelle, H., Beygelzimer, A., et al. (Eds.), *Advances in Neural Information Processing Systems*, v. 32. Curran Associates, Inc.

Available at: <https://proceedings.neurips.cc/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf>.

RUMELHART, D. E., HINTON, G. E., WILLIAMS, R. J., 1986, "Learning representations by back-propagating errors", *Nature*, v. 323, n. 6088 (Oct), pp. 533–536. ISSN: 1476-4687. doi: 10.1038/323533a0. Available at: <https://doi.org/10.1038/323533a0>.

RUMMERY, G., NIRANJAN, M., 1994, "On-Line Q-Learning Using Connectionist Systems", *Technical Report CUED/F-INFENG/TR 166*, (11).

SCHAUL, T., QUAN, J., ANTONOGLOU, I., et al., 2015. "Prioritized Experience Replay". 11.

SCHRIJVER, A., 2005, "On the History of Combinatorial Optimization (Till 1960)". In: Aardal, K., Nemhauser, G., Weismantel, R. (Eds.), *Discrete Optimization*, v. 12, *Handbooks in Operations Research and Management Science*, Elsevier, pp. 1–68. doi: https://doi.org/10.1016/S0927-0507(05)12001-5. Available at: <https://www.sciencedirect.com/science/article/pii/S0927050705120015>.

SILVER, D., HUANG, A., MADDISON, C. J., et al., 2016, "Mastering the game of Go with deep neural networks and tree search", *Nature*, v. 529, n. 7587 (Jan), pp. 484–489. ISSN: 1476-4687. doi: 10.1038/nature16961. Available at: <https://doi.org/10.1038/nature16961>.

SILVER, D., HUBERT, T., SCHRITTWIESER, J., et al., 2017. "Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm". .

SUTTON, R., 1988, "Learning to Predict by the Method of Temporal Differences", *Machine Learning*, v. 3 (08), pp. 9–44. doi: 10.1007/BF00115009.

SUTTON, R. S., 1996, "Generalization in Reinforcement Learning: Successful Examples Using Sparse Coarse Coding". In: Touretzky, D., Mozer, M. C., Hasselmo, M. (Eds.), *Advances in Neural Information Processing Systems*, v. 8. MIT Press. Available at: <https://proceedings.neurips.cc/paper/1995/file/8f1d43620bc6bb580df6e80b0dc05c48-Paper.pdf>.

SUTTON, R. S., BARTO, A. G., 2018, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA, A Bradford Book. ISBN: 0262039249.

THRUN, S., SCHWARTZ, A., 1993, "Issues in Using Function Approximation for Reinforcement Learning". In: Mozer, M., Smolensky, P., Touretzky,

D., et al. (Eds.), *Proceedings of the 1993 Connectionist Models Summer School*. Erlbaum Associates, June.

VAN HASSELT, H., GUEZ, A., SILVER, D., 2015. "Deep Reinforcement Learning with Double Q-learning". .

VASWANI, A., SHAZEER, N., PARMAR, N., et al., 2017, "Attention is All you Need". In: Guyon, I., Luxburg, U. V., Bengio, S., et al. (Eds.), *Advances in Neural Information Processing Systems*, v. 30. Curran Associates, Inc. Available at: <https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf>.

WANG, Z., SCHAUL, T., HESSEL, M., et al., 2016. "Dueling Network Architectures for Deep Reinforcement Learning". .

WATKINS, C. J. C. H., 1989, *Learning from Delayed Rewards*. PhD Tesis, King's College, Oxford.

WATKINS, C. J. C. H., DAYAN, P., 1992, "Q-learning", *Machine Learning*, v. 8, n. 3 (May), pp. 279–292. ISSN: 1573-0565. doi: 10.1007/BF00992698. Available at: <https://doi.org/10.1007/BF00992698>.

WEISS, G., 1960, "Dynamic Programming and Markov Processes. Ronald A. Howard. Technology Press and Wiley, New York, 1960. viii+ 136 pp. Illus. $5.75." *Science*, v. 132, n. 3428, pp. 667–667.

YU, Y., SI, X., HU, C., et al., 2019, "A review of recurrent neural networks: LSTM cells and network architectures", *Neural computation*, v. 31, n. 7, pp. 1235–1270.