Universiteit Leiden

Master Computer Science

Deep Reinforcement Learning Algorithms
for the Train Unit Shunting Problem

Name: Georgios Kyziridis
Student ID: s2077981

Date: 13/05/2020

Specialisation: Advanced Data Analytics

1st supervisor: Walter Kosters, LIACS
2nd supervisor: Wouter Kool, Ortec
3nd supervisor: Mitra Baratchi, LIACS
4nd supervisor: Wan-Jui Lee, NS

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Master Thesis
Deep Reinforcement Learning Algorithms for the Train Unit Shunting Problem

Georgios Kyziridis

May 25, 2020

Abstract

The focus of this thesis revolves around a challenging optimization problem with many constraints. There are various methods of solving these problems with heuristics, but the goal of this project is to experiment on Artificial Intelligence approaches such as Deep Reinforcement Learning.
The problem at hand is the so-called Train Unit Shunting Problem (TUSP), that depicts a train shunting yard where arriving and departing trains follow a day time schedule. The trains vary in length and material type, and they need several services such as cleaning, maintenance, and more. All the movements and services have to be planned in such a way that the desired train will be serviced and parked efficiently in the correct order at the correct track, ready for its departure, avoiding any delays in the corresponding time schedule. This is a difficult NP-hard optimization problem, which needs sequential decision making. The Dutch Railways NS manages a huge fleet of train units and tackles the problem with heuristic solutions of planing.
The current research focuses on a subset of problems of the whole TUSP, aiming to develop a Deep Reinforcement Learning (DRL) agent which produces solutions and makes plans for multiple scenarios, controlling all the necessary movements in a train shunting yard. We approach the problem by reformulating it as a Markov Decision Process (MDP), where the agent makes decisions using visual-based state observations. The agent interacts with yard simulator software built by NS named TORS, suitable for sequential planning.
The results show that the agent is capable of finding policies that solve scenarios on a small number of trains, satisfying all the constraints. Due to the big state/action space of the problem, and the structure of the environment, the agent faced some obstacles on building a robust policy to solve scenarios with larger number of trains. However, the current research shows that the reinforcement learning models which used, are capable to teach the agent how to handle a limited number of trains, solving successfully three subproblems of the whole TUSP.

1 Introduction

Over the past years the concept of Deep Reinforcement Learning (Deep RL) is of main concern in the Artificial Intelligence research community. More and more optimization problems are tackled with Reinforcement Learning techniques. The combination of neural networks with the Reinforcement Learning framework boosted significantly the performance of RL algorithms and paved the way for new tools in the AI community. Classical tabular methods are not capable of solving difficult problems, cursed with high state/action spaces, therefore, deep learning and neural networks enhanced RL with robust approximations and upgrade the whole Reinforcement Learning framework.

Dutch Railways (NS) handles a huge fleet of trains in a daily base. It has to manipulate all the train movements and schedule the necessary tasks that have to be made for the trains in order to be cleaned, maintained, serviced, and ready to depart without any delays. This is a challenging optimization problem where incoming trains have to be matched with the outgoing ones according to the timetable, without any delays. The physical constraints of a shunting yard, complicate the routing and the parking of the trains due to the fact that many of the tracks are dead-ended, the track lengths have to be taken into consideration, the train lengths as well and so on. Additionally, overflow in a track is impossible. Specific services such as maintenance and cleaning need to take place into specific tracks where the service is

available. All this leads to a challenging optimization problem where many constraints have to be satisfied simultaneously. Figure 1 illustrates the service side of a real train shunting yard called *Kleine Binckhorst* that is used in the current project.

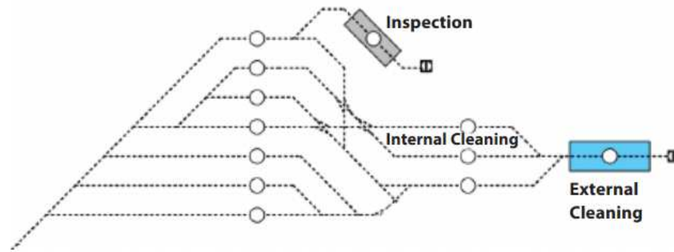

Figure 1: Kleine Binckhorst Shunting Yard. Image taken from http://sporenplan.nl.

NS train station has already started working on research aiming on tackling that problem with Artificial Intelligence techniques such as Reinforcement Learning. Human planners generate plans step-by-step manually using heuristics in order to construct a robust plan for all the trains avoiding conflicts and delays on the departing trains, and satisfy all the aforementioned constraints. There are various classical optimization techniques in order to assist human planners to create robust plans.

In the current research we focus on Reinforcement Learning techniques with the intention of creating an agent that makes decisions sequentially and controls the service side of a shunting yard. More precisely, we aim to create an agent which learns, through trial and error, to manipulate all the necessary movements of the trains, schedule the essential service tasks such as cleaning, and delivers the requested train to depart, according to the current time-table without any delays. The problem of routing, matching (incoming trains with outgoing ones), scheduling service tasks and other operations, consist of the well-known **T**rain **U**nit **S**hunting **P**roblem (TUSP).

The Reinforcement Learning framework consists of an agent and an environment formulated in a Markovian Decision Process (MDP) where the agent is the decision maker who interacts with the environment which operates the agent's actions. In this case the agent is the brain behind the yard which decides sequentially the action that will be exerted. For that purpose, we modeled the TUSP in an MDP formulation where the state observation is indicated by visual representation containing all the significant information of the current time stamp. We trained various agents with different RL algorithms, such as *Policy Gradient* and *Deep Q-Learning*, in order to inspect which fits better in this problem. We trained the agents on real-world scenarios (instances) generated from the NS instance generator and compare their performance with a random agent and a greedy one. The most promising results were came from the deterministic methods such as DQN, which performed better than the Policy Gradient ones. We show that the agent is capable of learn a policy of solving instances, unfortunately only for scenarios with a small number of trains. Deep Reinforcement Learning techniques seem to be a promising approach on this type of challenging optimization problems. These approaches are used for similar NP-hard problems such as the well-known TSP, opened up the way for new techniques and further research on RL algorithms for optimization problems.

The current paper is structured as follows: In Section 2, we provide the preliminaries for the Train Unit Shunting Problem as it has been referred to in the literature. Additionally, initial information about Reinforcement Learning is provided, so the reader can follow the ideas of the current sheet. Section 3, refers to related work that has be done by other scientists. It is split up into two subsections according to the relevance of each paper with the TUSP or similar problems such as TSP. In Section 4, we explain the problem that we focused on, starting from the initial problem of *parking* and *matching* from previous work, and gradually explain how we add extra learnable tasks to the agent. The next section, Section 5, focuses on explaining all the algorithms that we used in the current research. It contains the essential formulas and pseudocodes describing the whole process of the simulations. Afterwards, we describe our findings by stating the results of the correctly solved instances in Section 6, visualizing the performance of the agents, as well. Finally, Section 7, concerns our conclusions about the whole project and explains some of our ideas for further research.

The current paper is a master thesis project at the Leiden Institute of Advanced Computer Science

(LIACS), department of the Leiden University. It was supervised by Dr. W. Kosters and Dr. M. Baratchi from LIACS, PhD W. Kool from Ortec and Wan-Jui Lee from the maintenance development department at NS Dutch Railways.

2 Preliminaries

This section contains the problem description as it is stated in the literature, and continues with basic information about the Reinforcement Learning framework. The main components of the Train Unit Shunting Problem will be described thoroughly, splitting the problem into subproblems in the first subsection. The second one, revolves around the terminology, vocabulary and initial aspects of the Reinforcement Learning context, such as basic formulas and useful information so that the reader can follow the rest of the paper.

2.1 Train Unit Shunting Problem

The Train Unit Shunting Problem (TUSP) depicts a train shunting yard of a station, where trains have to be scheduled according to constraints. The constraints consist of arrival and departure time plans, service tasks that have to be applied to specific train units, e.g., cleaning, and parking operations following the departure order. From previous work [7], the problem involves four subproblems: *matching*, *parking*, *routing*, and *service tasks*. The following paragraphs describe the above four parts of the problem and the setting of the shunting yard. All the information for the problem description is from the master thesis of Van den Broek [2]. The following paragraphs provide information about the TUSP and divide it into subproblems, thus, it consists of smaller optimization problems whose intersection constitutes the whole TUSP as an NP-hard optimization problem.

Infrastructure
The shunting yard infrastructure consists of a finite set of tracks $\mathcal{T}$ classified into two disjoint types $\mathcal{T}_{free}$ and $\mathcal{T}_{LIFO}$, with $\mathcal{T}_{free} \cup \mathcal{T}_{LIFO} = \mathcal{T}$. A track $\tau \in \mathcal{T}$ has length $\ell_{\mathcal{T}}$, and if $\tau \in T_{free}$, it has two entrances, τ_A and τ_B, so the trains can arrive at both sides and move towards both directions; tracks in T_{LIFO} only have one entrance. The tracks are connected via switches. A service site is connected to the rest of the railway network through *gateway*-tracks from where trains get into the shunting yard and a service task will be performed if it is needed. Each service task, $\sigma \in \mathcal{S}$ has a duration, d_σ, and it could be performed on a specific set of tracks, $\tau_\sigma \in \mathcal{T}_\sigma \subset \mathcal{T}$, where the task might take place. There are two categories for service tasks:

- $\mathcal{S}_{spec}$ consists of *track-specific* tasks. Such a task requires a specific facility that is located on a track, such as the train wash or cleaning platform where only a single train can be processed at a time in each facility.
- $\mathcal{S}_{ind}$ consists of *track-independent* tasks. Such a service can be done simultaneously on each possible track and a service crew is needed referred to as *resources*.

The trains consist of bidirectional and self-propelled train units, $tu \in TU$, where TU is the set of possible train units, classified according to type and subtype (e.g., the train is ICM-3 means $type$ = ICM and $subtype = 3$). The subtype of the train indicates the number of train carriages. Generally, the incoming trains at the shunting yard can be split or combined in order to construct a new train composition, depending on the current services. *Crossing* is the situation where the moving path of a train is blocked by another one. If a scheduled plan contains a *crossing* it is considered infeasible.

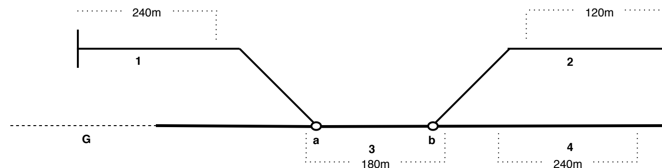

Figure 2: Service site example. Trains enter through gateway G, park on tracks 1 to 4 according to their length. Switches a and b connect the tracks in the service site.

Figure 2 illustrates an example of a service site track of a shunting yard. Imagine that we know the number of trains and the time of arrivals in advance. Some of the tracks (1 to 4) are pledged for specific assigned tasks such as cleaning or maintenance. The planner has to take into consideration if and how many tracks are occupied, and the fact that some service tasks have to be performed on train units on specific platforms.

Matching

In the matching problem we are given a set of arriving trains AT and a set of departing trains DT. An arriving train $at \in AT$ has a unique identifier, and consists of one or more train units represented as sequence $(tu_1^{at}, \ldots, tu_n^{at})$ (where $n = n^{at}$ is the number of its train units), and has an arrival time a^{at}, as well. Similarly a departing train $dt \in DT$, departs at time d^{dt}, and specifies a sequence of one or more train units $(tu_1^{dt}, \ldots, tu_n^{dt})$ (where $n = n^{dt}$ is the number of its train units). The number of train units is the same for AT and DT for all combinations of type and subtype. All arrivals and departures are mixed over time. The objective in the matching problem is to match the ith unit tu_i^{at} from an arriving train at, to the jth unit tu_j^{dt} in the departing train dt following the rules:

- $tu_i^{at} = tu_j^{dt}$, so types and subtypes match.
- $a^{at} \leq d^{dt}$, the train unit enters the service site before departure.

Of course, the matching has to be such that this mapping is injective and surjective. We can enhance the second rule as follows, taking into consideration the duration of the service task that will be performed.

$$a^{at} + \sum_{\sigma \in \mathcal{S}_{tu_i^{at}}} d_\sigma \leq d^{dt}$$

Here $\mathcal{S}_{tu_i^{at}}$ indicates the set of services for the particular unit. It has to be mentioned that parts of matching are already predefined, that is, some train units are already assigned to a specific position of a departing train. The service site has to be empty at the start and at the end of each day. It was shown in [7], that planning the matching which minimizes the number of splits and combines is NP-hard, without even taking the routing and the service tasks schedule, into consideration.

Routing

The objective of the routing problem is to find the optimal unobstructed path, that minimizes the duration of each train movement. Various functions could approximate the duration of a route. Some of the duration estimations are based on the number of tracks, switches and the required *saw*[1] moves on the path, using the following values:

- Traversing a track takes sixty seconds, regardless of train type or length.
- Transitioning from one tack to another requires thirty seconds for each switch along the way. Switches can be operated either automatically or by hand, depending on the service site.
- A *saw* move depends on both the train type and the train length. The process of reversing a shunt train usually takes somewhere between two and four minutes. The time for the driver to walk to the other side of the train requires twenty to thirty seconds per carriage. It has to be mentioned, that a *saw* move, traverses the track used for reversal twice, which has to be taken into consideration during the computation of the duration time.

The above information have been validated according to the study of Van den Broek [2] by the planners at NedTrain to provide an approximation of the actual movement duration observed on the service site. In this study, we do not allow simultaneous movements on the service site, so only a single train movement is happened at any time. In order for a plan to be feasible, it should ensure that no other train moves during an arrival. A shortest path could be found in polynomial time if it is assumed that the track occupation on the service site at the time of the movement would be fixed. The problem of deciding whether there exists a sequence of train movements such that a shunt train reaches its destination closely resembles the decision variant of the sliding block puzzle *Rush Hour*, a problem known to be PSPACE-complete.

[1] A saw move is when the train has to reverse the controls and the driver has to walk to the head or the tail of the train in order to make a manoeuvre.

Parking
The main objective of the parking subproblem is to place all shunt trains on the tracks such that at any moment a train length does not exceed the length of the track, while simultaneously ensuring that each train has an unobstructed path when it has to depart. More precisely, each incoming train moves to a track τ and will be added in front of one of the two sides of the sequence of the already parked trains. The side is fixed if the track τ is a LIFO-track, otherwise it depends on the arrival side of the train to the corresponding track τ. The order of the trains in a free-track depends only on the arrival order and arrival side, as well. A conditional constrain is that in the parking situation, splitting and combining is forbidden at the gateway tracks, which means that all trains of a departure composition should parked at one track in the correct order according to the scenario. Scenario is a day schedule with multiple train arrivals and departures. A scenario contains information about the arriving trains such as the arrival time and the train composition (train-type and number of carriages). It also contains information about the demanding departures such as the departure time and the train composition (train-type and number of carriages).

Scheduling of the Service Tasks
In the scheduling problem, a schedule must be constructed such that all tasks are completed and all trains can depart on the time, given a set of service tasks along with the corresponding duration. Each one of the tasks requires some resources which are limited. We can assume that all trains consist only of a single unit and the service task scheduling is simplified to a generalization of the *Open Shop Scheduling Problem* (OSSP), which is a well-known NP-complete problem. More precisely, each train unit is a job with its service tasks corresponding to operations in the OSSP. The dates and deadlines of each operation are based on the schedule of the arrivals and departures of the trains.

All the above information formulates the problem as it is stated in the literature, using much information from [2]. We perform various heuristics in order to overcome the high complexity of all the subproblems trying to simplify it. In this project we emphasize on following subproblems: *matching*, *parking* and scheduling the *service tasks* as a limited edition of the TUSP.

2.1.1 Complexity

The intersection and the communication of the main four subproblems above, constitutes the TUSP as an NP-hard optimization problem. According to [7], the problem of *matching* and *parking* of trains, referred to as TMP, is NP-complete in its general form. That means that it is in NP and is NP-hard. It is a decision problem to which all other NP problems can be reduced, in polynomial time. Figure 3, illustrates the well-known Euler diagram which depicts the relationships between the complexity classes.

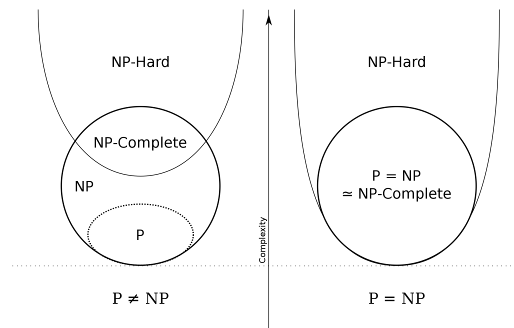

Figure 3: Possible relationships between P and NP. Image taken from Wikipedia.

The following definition of the TMP-0 problem is from [7]:

> Given a timetable of arriving and departing train services, the question of TMP-0 is whether the arriving units can be matched to the departing units in a feasible manner, such that no arriving services are split.

We can check in polynomial time if a given matching of trains is correct, if there are any mismatches on the train types or whether the order of the arriving/departing train units are the same. So, it is not very difficult to observe that the TMP-0 is in NP. It turns out to be NP-complete, as well; this requires a reduction from a known NP-complete problem. The problem which used in [7] for proving the NP-hardness of TMP is the 3 Partition Problem (3PP) which is defined by Garey and Johnson in [3]. Since TMP-0 is just a subproblem of the whole TUSP, that means that the four subproblems together compose a NP-hard problem. This justifies the use of approximating algorithms, such as the ones we will propose.

2.2 Reinforcement Learning

This section contains information about the basic concepts of Reinforcement Learning (RL) and preliminary information so the reader can easily understand the concept of the current paper. Before move on to the description of the RL framework, we first introduce the concept of the *Markovian Decision Process*. It is usually referred as MDP and it is a variation of the standard *Markovian Process* merely including the concept of decision making. So, MDP is a stochastic process following the rules of the *Markovian Process* including decision making in a finite horizon. It depicts a problem of learning through interaction with trial and error. The learner, which in this case is the decision maker, is called *Agent* and interacts with an *Environment* trying to achieve a goal. These two entities are interacting by sending messages to each other. More precisely, the *Environment* is modeled in discrete time steps $t = 0, 1, 2 \dots$, and it is responsible of providing the information of the situation of the current time stamp. All this information is provided to the *Agent* who has to process it and make a decision. Afterwards, the *Agent* communicates with the *Environment* by applying that decision. The *Environment* messages the information back to the *Agent* describing the new situation after that decision is applied. The aforementioned words: information and decision, are called *State* and *Action* respectively, in the RL context.

According to [15], the *State* must include all the significant information regarding all aspects of the past of the *Agent-Environment* interplay that makes a difference for the future. If that happens, then the state is considered to have the *Markov property* and it is referred to as a *Markovian State*, $s \in \mathcal{S}$. The *Environment*, apart from the *State*, it produces a reward which indicates how good was the exerted *Action* in a numerical representation. All in all, the two entities *Agent* and *Environment* interact at a sequence of discrete time steps, $t = 0, 1, 2, \dots, T$. At each time stamp t, the agent receives a representation of the *State*, $s_t \in \mathcal{S}$, and selects an *Action*, $a_t \in \mathcal{A}$. The agent receives the reward, $r_{t+1} \in \mathcal{R} \subset \mathbb{R}$, as a consequence of its exerted action and finds itself in the new state, $s_{t+1} \in \mathcal{S}$. This interactive procedure finishes when the agent reaches the final state, $S_{\text{terminal}} \in \mathcal{S}$, in a finite MDP. The sequence of state-action-reward in the RL framework is called *trajectory* and it represents the data where the agent is trained on, in order to learn doing a specific task, or learn how to achieve a specific goal. A trajectory looks like the following:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots \tag{1}$$

In finite MDPs, the sets $\mathcal{S}$, $\mathcal{A}$ and $\mathcal{R}$, have a finite number of elements. It can easily be observed that in the initial state, $s_0, \in \mathcal{S}$, there is no reward, because the reward reflects numerically the goodness of the performed action $a_t \in \mathcal{A}$, in the current state $s_t \in \mathcal{S}$, thus, the reward is produced after each action. The following schema depicts all the above information graphically, it is called the *Reinforcement Learning Circle* wherefore it represents the interactive process between the agent and the environment.
The total discounted reward is referred to as the return from Equation (2). The goal of the agent is to select actions that maximize the expected discounted return, G_t, where $0 \leq \gamma \leq 1$, is the discount rate which controls the way of how future rewards are weighted. For $\gamma = 0$, the agent maximizes only immediate rewards, whereas greater values of γ also take into account future rewards. Maximizing only immediate rewards leads to a myopic agent which is not an optimal strategy. An agent who cares only about immediate rewards will not explore parts of the state space that will possibly generate a bigger reward in the future. On the other hand, in late stages of learning, exploitation ensures that optimal actions are chosen. Consequently, the agent needs to balance between exploration and exploitation so as

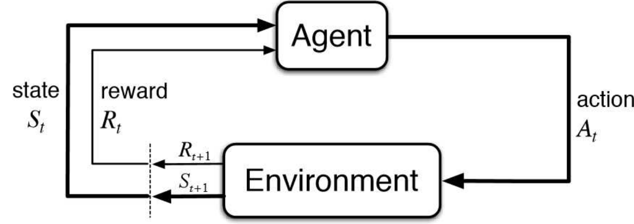

Figure 4: Reinforcement Learning Circle
from Book: *Reinforcement Learning an Introduction*

to converge to an optimal strategy of how to behave in the environment.

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

The agent requires a mapping from states to actions in order to act in the environment. A policy, π, is a function that maps a state $s \in \mathcal{S}$ to an action $a \in \mathcal{A}$, that is, $\pi(s) = a$. A stochastic policy, $\pi(a|s)$, is the probability of taking action a in state s. The optimal way of behaving in the environment, i.e maximizing G_t, is defined by an optimal policy π^*. Optimal decisions require a notion of how good it is to be in a state, or, how good it is to take an action from a state. The value of state s under policy π, $v_\pi(s)$, is the expected return from state s, following the policy π onward, which leads to the formula:

$$v_\pi(s) = \mathbb{E}_\pi[G_t \mid S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s\right], \ \forall s \in \mathcal{S} \tag{3}$$

The value of $v_\pi(s)$, indicates the goodness of being at the state s, following the current policy π until the termination of the episode regarding the discounted rewards. The above equation is called *state value function*. This contains only the information about being on a current state, given the current policy and it does not include any information about the actions. In order to combine the previous information with the goodness of taking action a in state s under the policy π, we use the value of $q_\pi(s, a)$, which represents the expected return starting from state s, taking action a, and following π onward. This is called the *state-action value function* which is formulated as follows:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t \mid S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a\right], \ \forall s \in \mathcal{S}, \forall a \in \mathcal{A} \tag{4}$$

2.2.1 Value-Based Learning

There are various ways that an agent could learn a task by interacting with an environment. One of those is called value-based learning, due to the fact that the agent learns through the values of each state, $V(s)$, or state-action pairs, $Q(s, a)$. More precisely, regarding the information from Section 2.2, the value function estimates how good is to reach certain states. The objective of the agent is to find the optimal policy that maximizes the expected rewards. So, the agent is trained on building the optimal policy thought the value-function formula:

$$V^*(s) = \max_\pi v_\pi(s) \tag{5}$$

The above equation represents the optimal policy which is derived through value learning using only the value function approximation Equation (3), which contains information about the goodness of visiting specific states. If someone wants to enhance the value learning approach, they can include the actions in the value function and transform it into a *state-action value function* approximation using Equation (4). In this way the agent is still trained on the task of building a policy, trying to reach the optimal one π^* through value learning, but this time exploiting information from the actions, $a_t \in \mathcal{A}(s)$, as well. So,

instead of assigning a value, $v_\pi(s)$ for each state, someone could assign a $q_\pi(s,a)$ value for each action, $a_t \in \mathcal{A}$ in each state, $s_t \in \mathcal{S}$. So, we get:

$$Q^*(s,a) = \max_\pi q_\pi(s,a) \tag{6}$$

The above equation indicates the optimal policy under which, the agent selects the action with the highest q value for the current state, $A_{\text{selected}} = \text{argmax}\, q_i(S_t = s, A_t = a_i)$. An optimal policy π^* corresponds to an optimal state-value function v^*, or an optimal action-value function q^*. Having either v^* or q^* is equivalent to know the optimal policy π^*, and thus, the optimal way to behave in the environment.

In many cases we want to reduce the variance of Q, for that reason we introduce a new term called advantage. Advantage function could be considered as another version of Q-value with lower variance by taking the state-value as the baseline. By subtracting the state-value $V(s)$ from $Q(s,a)$, we reduce the variance as we subtract the average value of actions that would be selected in that state. The following formula represents the advantage function, $\hat{A}(s,a)$:

$$\hat{A}(s,a) = Q(s,a) - V(s) \tag{7}$$

Based on TD-residual (Temporal Difference) concept, advantage value could be calculated through the Generalized Advantage Function proposed in [13]. The following formulas describe the estimation of advantage A_t obtained from the discounted advantage function $A(s_t, a_t)$. Let $\delta_t = R_t + \gamma V(s_{t+1}) - V(s_t)$ which represents the TD-residual [15] of $V(s)$, using a discount factor, γ. The parameter δ_t, could be considered as an estimation of the advantage of action A_t as it is described by the formula:

$$\hat{A}_t^k = -V(s_t) + R_t + \gamma R_{t+1} + \ldots + \gamma^{k-1} R_{t+k-1} + \gamma^k V(s_{t+k}) \tag{8}$$

We can rewrite the above equation formulating the Generalized Advantage Function using the parameter δ_t from [14] and an extra parameter $\lambda \in [0,1]$. The following equation could be reduced to Equation (8) for $\lambda = 1$. The generalized advantage estimator GAE could be expressed as the exponentially-weighted average of k-step estimators leading to:

$$\hat{A}_t^{(k)} := \sum_{l=0}^{k-1} (\gamma\lambda)^l \delta_{t+l} = \delta_t + (\gamma\lambda)\delta_{t+1} + \ldots + (\gamma\lambda)^{T-t+1}\delta_{T-1} \tag{9}$$

Two interesting properties of GAE are stated in [13] for two values of λ:

$$\begin{aligned} \lambda = 0 : \hat{A}_t &:= \delta_t = R_t + \gamma V(s_{t+1}) - V(s_t) \\ \lambda = 1 : \hat{A}_t &:= \sum_{l=0}^{k-1} \gamma^l \delta_{t+l} = \sum_{l=0}^{k-1} \gamma^l R_{t+l} - V(s_t) \end{aligned} \tag{10}$$

The Generalized Advantage Estimator for $0 < \lambda < 1$, balances between bias and variance. Both parameters γ and λ contribute to the balance of bias versus variance tradeoff of the advantage estimation.

2.2.2 Policy-Based Learning

Policy based learning is another family of algorithms in reinforcement learning. In these algorithms, the policy π_θ is represented by a model (e.g., a neural network in the deep reinforcement learning framework) which maps each state into probabilities and builds a distribution over the possible actions for the current state. In policy learning the agent tries with various methods to train the parameters θ in order to reach the optimal policy π_θ^*. The basic algorithm behind policy based learning is the *Policy Gradient* which targets at modeling and optimizing the parameterized policy directly using gradient-based methods such as gradient ascent. The reader should keep in mind that the goal of an agent in reinforcement learning, is to find an optimal behavior strategy in order to obtain optimal rewards produced by the environment. Using the policy gradient algorithm, the aforementioned policy is achieved by training the policy-model, π_θ, towards the direction of the gradients. The objective in policy gradient methods is to maximize the expected reward, following the parameterized policy π_θ as it stated in the following formula:

$$J(\theta) = \mathbb{E}_{\pi_\theta}\big[R(\tau)\big] \tag{11}$$

The above equation represents the basic objective, where $R(\tau)$, indicates the rewards collected from the trajectory, τ. It is a differential equation where the model is trained upon, updating the policy weights θ with a proportion of the gradient of Equation (11). The reader should remember that all MDPs have at least one optimal policy, which can maximize the rewards. In the policy gradient framework, the goal is to find those parameters θ^*, which maximize the objective $J(\theta)$. The training and the update rules for this approach are described thoroughly in Section 5.1.

Policy Gradient algorithms could be used to a wider range of problems. There are many occasions where the reward function is too complex to be learned by a simple Q-learning algorithm using the Bellman equation. In those cases Policy Gradients could perform better, they show faster convergence rate than value-based methods which have the tendency to converge to local optima. An other interesting fact about the Policy Gradient family, is that they could be applied on problems with continuous action space because the policy model (policy-network) builds a distribution over actions, so in the continuous domain it builds a distribution over some mean value μ, with some standard deviation σ, from where actions could be sampled. Value based methods have to discretize actions in the continuous domain which is inefficient and undesirable on most of the cases. The basic drawback in the policy-based methods is that they suffer from high variance in the gradient estimation. More precisely, each update is an estimation of the gradient, estimated using samples of the episode. This estimation could be extremely noisy which leads to bad gradient estimations and brings instability in the learning process. So, the basic disadvantage of policy based methods is that they lack in sample efficiency while value-based methods, such as DQN, usually performs better regarding the sample efficiency.

2.2.3 Exploration

The exploration and exploitation ratio is a great challenge in reinforcement learning and has a strong impact on the results. An agent who starts interacting with an unknown environment has no prior knowledge and consequently performs only random actions in the beginning. In this way it collects information by exploring states would be reached by exerting those random actions. Randomness plays a significant role in stochastic processes especially in reinforcement learning. The reason is that we can simulate a human strategy by controlling this randomness and force the agent to learn imitating the way that humans interact in unknown situations. The reader could imagine a situation of a person playing a new video game without prior knowledge. The most common human strategy, when someone is a new player, is first to experiment with the available actions, and learn through trial and error the impact of these actions. In this way, the player is exploring the consecutive states after exerting random actions, following the pattern of a trajectory.

After some experimentation, the player will gain some initial knowledge about good and bad actions or states that have been visited, and this is how the concept of rewards is taking place. The player will avoid actions that lead to bad states (bad reward), and consequently after some exploration, the player should start choosing actions according to the already explored trajectories (states-actions-rewards). This is an initial policy based on the memory built through the exploration. The aforementioned could be expressed using reinforcement learning vocabulary as follows: the agent (player) will start interacting with the environment by selecting random actions in order to explore some of the environment's state space $\mathcal{S}$, and action space $\mathcal{A}$, and store the trajectories in its memory. After some exploration the agent should start building a policy based on the memory, so it has to gradually stop selecting actions at random and choose actions based on the policy. Exploiting uncertain data, prevents the maximization of the long-term rewards because the selected actions are probably not optimal. Consequently, we need balance between the two modes (exploration & exploitation) in order to use them adequately. All the above, are summarized in the well known *Dilemma of exploration and exploitation* from [15].

There are various algorithms and approaches handling this dilemma, one of them is the well known ϵ-greedy. It handles the trade-off between exploration and exploitation by controlling the parameter of randomness. Let $\epsilon \in [0, 1]$, be the probability that the agent selects an action completely at random and not based on the current policy π. The idea of ϵ-greedy is to control the frequency of selecting a random action, by reducing this probability gradually. The following equation describes the binary decision between random and non-random actions.

$$\pi(s) = \begin{cases} \text{random action from} \mathcal{A}(s), & \text{if } \xi < \epsilon \\ \text{argmax}_{a \in \mathcal{A}(s)} \mathcal{Q}(s, a), & \text{otherwise} \end{cases} \tag{12}$$

where ξ is a uniform random number drawn at each time step, t. Each time the agent selects a random action, the ϵ value is decreased by a discount factor $\delta \in [0, 1]$ which is set to a value close to 1, e.g., 0.99. In this way, we slightly decrease the chance of a random action selection according to our requirements. Ideally, exploration mode is preferable in the early stages when the agent is completely uncertain about the action selection due to its zero experience. After the agent collects observations from the environment during the exploration phase, it will exploit its knowledge and select actions derived from the policy.

3 Related Work

Various approaches exist for solving optimization problems such as the Train Unit Shunting Problem (TUSP). In this section we mention some of the papers which are helpful for the current research. Due to the fact that TUSP is a very specific problem, and needs good domain knowledge (train units, yards, shunting/routing problems, etc), some of the papers are not directly related to the TUSP but are related to similar routing problems such as the famous *Traveling Salesperson Problem (TSP)* or the *Vehicle Routing Problem (VRP)*, see, e.g., [1] and [4], respectively. All of them share the same idea of optimization, that is finding optimal solutions. These solutions could be considered as plans, strategies, or policy in the Reinforcement Learning framework.

The current research focuses on Deep Reinforcement Learning approaches due to the fact that we aim to develop methods that produce consistent plans, which are robust to the uncertainty. A method for creating such plans should reason for each decision it takes. So, via Deep Reinforcement Learning methods we can obtain a robust trained policy which find solutions to specific scenarios, and consequently we can gain insight information if we analyze the policy. Furthermore, we can draw conclusions about why specific actions are taken into specific cases and why some scenarios are more difficult to be solved. We split the related work into two categories: papers that focus on the TUSP and papers that focus on similar problems.

3.1 Related work on TUSP

Peer et al. [10] addresses a stable solution for a limited edition of the Train Unit Shunting Problem with Deep Reinforcement Learning. The paper tackles a specific part of the TUSP, the *parking* and the *matching* subproblems. It is a classic optimization problem where trains have to be parked wisely according to a set of constraints following the day schedule (scenario). For this purpose, the paper suggests a Deep Q-Learning (DQN) agent based on the idea of Mnih et al. [8], which learns how to solve these two subproblems through the Q-Learning algorithm, approximating the outcome of the Bellman equation. The agent is trained on scenarios generated from the instance generator developed by NS. The authors represent the problem as a Markovian Decision Process (MDP) in order to use RL methods, and proposed a visual representation of the state observation. The image-based state, includes sufficient information from the scenario and provides to the agent all the information that is needed, in order to learn how to solve such a problem. The DQN approach is a value based Reinforcement Learning method, using neural networks to approximate the Q-value for each action, and it is characterized as an off-policy method. The agent uses two policies: the behavioral policy and the greedy one, both described by the ϵ-greedy algorithm in Section 2.2.3. The agent tries to learn the optimal policy by minimizing the difference between the predicted Q-values and the outcome from the Bellman equation. The reward function was defined according to whether a departure or a parking action is correct, given the current scenario. The results show that the algorithm outperforms the greedy baseline and produces a correct solution on approximately 81% of the $3,000$ test scenarios. We focused on this paper for the current research aiming to upgrade it and add more subproblems of the TUSP, such as the *scheduling of the service tasks*. Peer et al.'s paper [10] is described thoroughly in Section 4.2.

In [18], the authors study the approximation of the Local Search heuristic using Machine Learning models. The paper focuses on the TUSP problem and provides information for the model architecture and the relation between the initial shunting plan for the Local Search (LS). More precisely, the paper describes how a graph convolution neural network (DGCNN) is used to boost the LS method and how the model provides some better initial solutions in order to reduce the time for the search process of an LS run. A shunting plan could be represented as an activity graph, $G = (V, E)$, with V the set of nodes: arrival,

movement, parking, service and departure; and E the set of edges that indicate the precedence relations. The paper focuses on predicting the feasibility of an initial solution represented as an activity graph. Node features contain information about tracks, trains, duration and activities, and each graph has a binary target class $C_{\text{target}} = 0, 1$, indicating the feasibility of the current shunt plan. The problem is considered as a binary classification problem where the accuracy is measured by the model's outcome with the actual label. The DGCNN model addresses useful feature extraction from the graph. To evaluate this approach, the authors measure how much running time could be reduced in determining capacity in the yards. The results of this approach demonstrate that the method can boost optimization algorithms in an industrial application and that combining DGCNN with LS the authors achieve to decrease the computation time by 17.5%.

3.2 Related work on similar problems

The paper of Kool et al. [5] addresses an encoder-decoder attention model for solving the Traveling Salesperson Problem (TSP) using Reinforcement Learning. The model represents a parameterized policy $p_\theta(\pi|s)$ which generates a tour solution π as a possible permutation of the nodes of a graph (note that we use exactly the symbols of the referred paper [5], where π refers to a tour solution and $p_\theta(\pi|s)$ to the policy). The data consist of graph-instances with nodes and edges of the TSP. The paper provides information about four different essential parts of the problem, namely: encoder, decoder, context-embedding and optimization though the Reinforce algorithm. More precisely, the encoder takes as input the node features in a low dimensional space (coordinates in two dimensional Euclidean space) and computes an aggregated embedding which consists of the graph embeddings and the mean of the final embeddings, as well. The encoder uses multi-head attention layers following [17]. The decoder takes as input the aggregated output of the encoder, and yields one node π_t at a time. The output of the decoder is based on the encoder's output embeddings, and on the decoder's output pi'_t as well. The graph is augmented during the decoding procedure with a special context node which represents the decoding-context for specific problems of the TSP. Finally, the optimization made by the Reinforce algorithm with a greedy rollout as baseline. Different approaches based on the policy gradient algorithm, with different baselines such as value-based critic baseline, are described thoroughly. The very detailed paper, regarding the Deep Learning model and the architecture, addresses how the policy gradient theorem among with Deep Learning models, can produce solutions to NP-hard problems. Combining auto-encoders from the Deep Learning framework, multi-head attention techniques including the idea of recurrency, and weights optimization through the well-known Reinforce algorithm, constitute a state of the art implementation and a really promising approach for solving optimization problems. Efficient evaluation for different baselines on different variants of the TSP, promising results that outperforms most of the existing methods, and all the aforementioned, compose a very interesting paper on Deep Reinforcement Learning.

Nazari et al. [9] presents an end-to-end framework for solving the Vehicle Routing Problem (VRP) using Reinforcement Learning. In this specific approach, a policy model is trained towards the direction of finding near-optimal solutions for a broad range of problem instances, observing the rewards and following feasibility rules. Given a set of input data, the paper focuses on finding a stochastic policy which generates sequences that minimize a loss objective satisfying the problem constraints. The parameterized stochastic policy, $\pi(\cdot|Y_t, X_t)$, is trained through the Reinforce algorithm and produces a sequence of consecutive actions in real time without re-training the model for every new problem instance. The paper addresses a model architecture in order to resolve the limitations of Pointer-Networks on problem cases where dynamic elements complicate the forward pass and the back propagation operations. The proposed model is invariant to the input sequence, thus it is not affected from any change in the order of any two inputs. The model also omits the Recurrent Neural Network architecture since the inputs in routing problems are usually represented by a set of unordered customer demands and locations, where the order of the input is not meaningful due to the fact that any random permutation contains the same information as the original ones. The model consists of two components, namely: *Graph Embeddings* and *Decoder*. The former is used to encode structured data inputs, and the latter points to an input at every decoding step using an *Attention Mechanism*. Given a set of inputs $X = \{x^i, i = 1, \dots, M\}$, the model will take a representation of $x^i_t = (s^i, d^i_t)$, where s^i and d^i are the static and the dynamic elements of the sequence, respectively. Note that the static elements are fed into the decoder network and the dynamic elements are used only in the attention layer. The authors combine the embeddings with the Long Short-Term

Memory layer, and plug them into the attention layer which generates the final conditional probabilities. The method is compared with the Google's OR-Tools VRP engine [11], one of the best open-source VRP solvers, and it produced shorter paths in 61% of instances with 50 and 100 customers.

4 Problem Description

The objective of the Train Unit Shunting Problem is to make plans for all the trains in the yard in order to satisfy all the constraints. In this research we want to explore if a part of the TUSP problem could be solved using artificial intelligence approaches. More precisely, to explore if an AI-agent is capable of learning how to solve specific subproblems of the TUSP using Reinforcement Learning algorithms.

4.1 Subproblems

In the current project we focus only on three out of four subproblems which constitutes the whole TUSP: *matching*, *parking* and *scheduling the service tasks*. In [10], the problem was encoded in such a way that the agent learns implicitly how to solve the *matching* and the *parking* subproblem using Deep Reinforcement Learning approaches. Our goal is to upgrade this work, and enhance it by including the *service tasks* in the learning process, to compel the agent to learn this new task of scheduling efficiently the services for the trains. More precisely, with this extra task, the agent has to learn to move the trains into specific service tracks and service the trains. This task contains the concept of time because each service-task has a specific time duration for each train. The agent has to learn to wait until a train is serviced and then move it to the parking tracks in order to be ready for a departure. It is a more complex problem compared with the one addressed in [10], due to the fact that the agent has to do more actions handling the movements of the trains between the tracks, and also learn to service and wait in the intermediate steps between arrivals and departures. The problem needs a bigger action space and consequently a bigger state space. We aim to develop an agent which can learn a robust policy and generate feasible solutions on different scenarios, solving the tasks of *matching*, *parking* and *scheduling the service tasks* by controlling all the movements for the trains. The agent has to make plans for all the actions inside the shunting yard, respecting its physical rules. In the current project the services is the internal cleaning of the trains which can take place only in the *cleaning tracks*, as the cleaning service is a track-specific task. Before move on how we can include the *scheduling of service tasks* as an extra learnable task for the agent, we explain the basic experimental setup used from [10], in the following subsection .

4.2 Matching and Parking

The environment used in [10], depicts a train shunting yard where trains arrive and depart according to day schedule. The referred yard, which is also used in the current research, is illustrated in Figure 1. It is a real shunting yard called *Kleine Binckhorst*, close to the Den Haag area. It has to be mentioned that there are specific tracks for parking and specific tracks for services. Moreover, there are some tracks which were not included in the simulation, for instance the blue platform (External Cleaning) is not included in the simulation. The inspection track (at the top of the image) is used as parking track in [10]. The current problem focuses on solving the *matching* and the *parking* problem (TMP) as a limited edition of the whole TUSP. The infrastructure of the shunting yard used in the simulation, consists of $n = 9$ dead-ended tracks of different capacity where trains can be parked. There are three main constrains in the current problem which have to be satisfied:

i Assign the arriving train to parking tracks wisely in order to avoid track length violation (track overflow).

ii Park the trains in such a way that can easily depart, avoiding delays, according to the scenario and assure that there is always an unobstructed path for the trains to leave.

iii Choose the correct type of train to depart, conforming to the desired one in the scenario.

A scenario consists of a time table of planned arrival trains, including the train type and number of carriages (e.g., SLT-4) and a time table for the departure trains which indicates the desired type of train

to depart. In the departure time, the agent has to choose a train with type that matches to desired one, so it has the freedom choose which exact train of the defined type will depart. For instance if there are 6 trains of SLT type parked in the yard, and the next departure is a SLT train, the agent can choose one of the six trains to depart. All in all, in the departures we only care about the correct matched train type and not for the exact train. Table 1 is an example of a scenario for four trains.

Time	**Type**	**Event**
9:00	SLT-4	Arrival
9:30	VIRM-6	Arrival
10:00	SLT-4	Arrival
10:30	SLT-6	Arrival

Time	**Type**	**Event**
12:30	SLT-4	Departure
12:45	SLT-6	Departure
13:30	SLT-4	Departure
14:15	VIRM-6	Departure

Table 1: Example of scenario for 4 trains.

The authors in [10], address that a Deep Reinforcement Learning agent needs sufficient information in order to learn how to produce a shunting-plan which satisfies all the constraints, so they suggest a visual representation of the state which includes essential information from the scenario. More precisely, the encoding of the state-observation in this problem includes information about the current and future arrivals, the yard occupation for each track and the current and future departures. In this way, they provide to the agent information about the current situation in the yard, the current event of the scenario (arrival or departure), and information about the future arrivals/departures. The state is represented by a 33×33 matrix, where each carriage of a train occupies one pixel. The state contains the following information:

- Current shunting yard occupation.
- Current arrival or departure.
- Lookahead arrival or departure event.
- Number of carriages of arrivals and departures beyond the lookahead events.

An episode consists of a day scenario sampled from a pool of scenarios including different number of trains, all generated by the NS instance generator. Each event of the scenario indicates a step in the episode, where the agent has to select an action. The action space, $\mathcal{A} \in [0, 9]$, is a discrete space with actions, $a_t \in \mathcal{A}$, that indicate the number of a track. So, given each input, the agent has to select a single track for an arriving or a departing train. When a train gets into a track (arrival), if it fits, it pushes the rest of the trains. The agent chooses only the destination track if the event is an arrival, and the current track if the event is a departure, where the first train from the left, will depart from the chosen track. An episode terminates when a violation is occurred. A violation could be a track overflow, a wrong departure (wrong train type) or an action which points an empty track in a departure event. The reward function used in [10] is described by the following formula:

$$r(s, a, s') = \begin{cases} +0.5, & \text{if correct parking} \\ +1.0, & \text{if correct departure} \\ -1.0, & \text{otherwise} \end{cases} \tag{13}$$

The DQN-agent consists of two convolution layers and a fully connected layer before the output which is $n = 9$. The network is trained on batches sampled randomly from the memory buffer after each exerted action. The network outputs the Q-value for each one of the actions and it is trained to predict the outcome of the Bellman equation. The process of training will be described in Section 5. The authors generated $30,000$ problem instances (scenarios) with 14, 15, 16, 17 trains ($7,500$ for each number of trains). They trained the agent for $150,000$ episodes, each one being a random instance drawn from a pool of $30,000$. The performance of the agent was compared with a greedy baseline and a local search optimization model. The agent could solve on average 81% of the instances in the test set. The test set consists of $3,000$ test scenarios (750 for each number of trains).

4.3 The TORS Simulator

The acronym TORS stands for the Dutch phrase *Trein Onderhoud Rangeer Simulator* which means *Train-Shunting Maintenance Simulator.* It is the new software package of NS for sequential planing. The word *shunting* indicates the push or pull (movement) of a train unit, from the main line to a siding one, or from one line of rails to another. TORS takes as input a scenario for a specific yard, produced by the NS instance generator. The software transposes the scenario into multiple events (arrivals-departures) and formulates sequential states together with the corresponding available actions. Moreover, it allows the trains to move from track to track in the intermediate steps between the arrival and departure events. More precisely, TORS plays the role of the environment for the TUSP, thus it generates *States* which include the information of the trains in the service-yard, available *Actions* for each current state, and *Rewards* for the exerted actions, while it updates the state by applying the chosen action. Given a specific problem (e.g., parking and matching), we can decide what information is useful for the state representation, which actions can be used, and what rules are essential to be set in order to solve the problem with Reinforcement Learning. All in all, we design the problem and the simulation by filtering and processing the information provided from TORS, since it provides a functional API for customizing all the above options.

The available actions are in the form of pairs of current track and destination track for each train. TORS provides the action of *wait* and *service* as well. The *wait* action means that all the trains will not move and the situation in the yard will be exactly the same in the next state. The *service* action indicates a service task, e.g., cleaning, and if that action is chosen the environment activates the start of the specific service task. Service action is enabled only if a train is located in a suitable track where the service task is allowed. After the service action is selected, the environment sets the train into the service task for specific duration which is indicated by the action. In the current research the service task is the internal cleaning of the train, and each train needs different time duration for cleaning according to its length and number of carriages. After the end of the service duration, the environment provides a binary information indicating that the train finished the service and it is clean.

The environment has a peculiarity on how generates possible actions on each time stamp. It freezes each time stamp and splits it into many smaller ones. More precisely, in the beginning of each time stamp the environment generates valid actions for all the trains, which means that produces actions for the tracks where trains are located. After the agent chooses the first action on time stamp t_i, the environment moves the proper train from the current track to the destination track according to the physical yard layout (all the movements are made in accordance with the physical constraints of the yard, so the left or the right most train is moved, from track to track, pursuant to the tracks connectivity in the yard). Then, in the same timestamp, the environment produces actions for the rest of the trains excluding the actions which correspond to the train which already moved. This routine happens until the agent selects one action for each train in the yard, one by one, in the current time stamp. Afterwards, the environment moves to the next time stamp, t_{i+1}, and produces actions in the same way. This peculiarity provides to the agent more chances to move the trains, but this could lead to undesired policies where the agent loses much time on unnecessary movements of the trains inside the yard. It would be a better option if the environment produced all the possible actions, for all the trains in each time stamp in order to help the agent to build a distribution over all possible actions on each time stamp. Furthermore, that fact leads the agent to use many times the wait action especially in cases where some of the trains are already serviced and located wisely in a position and they are ready for a departure.

4.4 Matching and Parking on TORS

We wanted to reproduce the results from [10] using the TORS software package as the environment for the RL-simulation. We tried to simulate exactly the set-up of the environment used in the refereed paper and train the agent without providing any prior knowledge in order to test TORS for the first time in a Reinforcement Learning task. TORS provides possible actions for all the trains in a form of pairs (current track and destination track) and provides the action of "wait", as well. We omit the option of wait by using heuristics, so the agent has only option to select a destination track. The scenarios used in this simulation were different from ones in [10] because the instance generator for TORS is different. For time reasons, we test the implementation in TORS only for 14, 15 and 16 trains because we wanted just to test if the implementation using TORS as the environment is working. We generated 3000 scenarios

(1,000 for each number of trains) and trained the agent only for 63,000 episodes. The results from the simulation are illustrated in Table 2.

	Number of Trains			
Simulation	**14**	**15**	**16**	**Overall**
[10]	83.0%	79.8%	81.0%	81.0%
TORS	92.6%	90.2%	91.0%	90.2%

Table 2: Percentages of correctly solved scenarios in the test set.

It is obvious that the results are quite better using TORS. It is not a fair comparison because the instance generator between the two implementations is different, so the data are different. This could lead us to a conclusion that the generator which produces scenarios for TORS is better and generates scenarios which have feasible solutions. This is an interesting finding about the instance generator for TORS, because with less number of instances and less training we achieve better performance. We report the final results using TORS after experimentation on the parameter tuning of the agent. Figure 5 illustrates the average reward per 100 consecutive episodes.

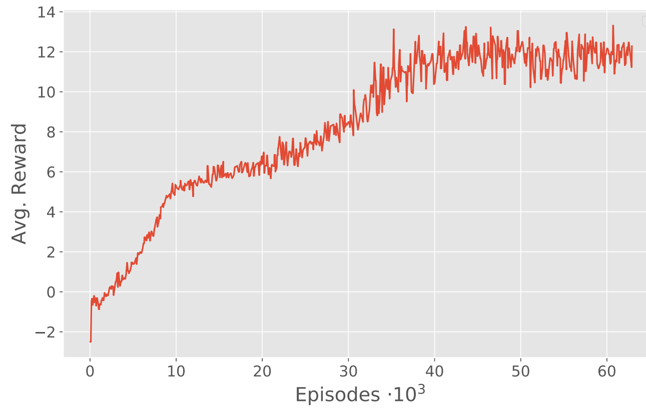

Figure 5: Average reward per 100 episodes.

4.5 Experimental Setup

The following paragraphs describe the experimentation setup of the simulations, using the TORS software package as the environment, including three subproblems of the whole TUSP: *Matching*, *Parking* and *Service tasks*. The current section provides information about the state/action space of the environment, the reward function which used, and some insights for the training procedure.

Action space
Including the *Service tasks* we set a bigger action space $\mathcal{A}^{54}$, which contains 52 actions for *moving* trains between tracks, 1 action for *wait* and 1 for *service* (cleaning). The movements between tracks follow the physical rules of the yard depicted in Figure 1, which means that each track has a specific capacity and that movements have to follow the connections of the tracks. Action *wait* could be considered as the action of ”do nothing” where the agent does not move or service any train at the current state. Action *service* indicates the start of the service task of the specific train on the cleaning platform. The action is available only if a train needs cleaning and it is located on a service platform. The cleaning has a time duration which corresponds to the train type. When the agent chooses the referred action, the service is active until the end of the task, and the agent can move only the rest of the trains. After the end of the service task, actions that involve the serviced train are again available for the agent.

State space
Following the idea from previous work, we experiment with different visual representations of the state. The state has to contain information about the situation in the yard at each time stamp that is the location of each train and its type as well. It has to include also information about the arrivals and the departures from each scenario. The way that the state is encoded in [10], from Section 4.2, is a 2-D matrix containing information for future arrivals and future departures and the current track occupation for each time step, as well. All the information was encoded in a 33×33 matrix. Figure 6, illustrates the state representation following exactly the idea from [10]. The extra information about the service trains was one-hot encoded for each train in another channel. All in all, using two channels (two matrices) we encode all the information we need for the agent.

In order to expand this idea, we separate the pieces of information into channels. Consequently, each piece of information about departures and services was encoded in a separate channel. We reduce the dimensions of the state in 12×33 matrix where each row indicates a shunting track which available capacity for trains is indicated by the non-black pixels. Each train carriage is represented by a single pixel assigned with the number of its type in order to encode the train type, the location, and its length in one channel. The rest of the channels are one-hot encoded information for the service and the departures. The state observation contains the following information:

- Position and train type.
- Information for serviced trains (binary).
- Current departure (binary).
- Future departures in lookahead window (binary).

We wanted to experiment with both ideas of the state representation namely using multiple channels to encode the state information and the idea from [10]. Separating the information of the state using multiple channels and reducing the dimension in a 12×33 matrix, helped the agent to produce better results in the final model. Each channel contained different information from the above list.

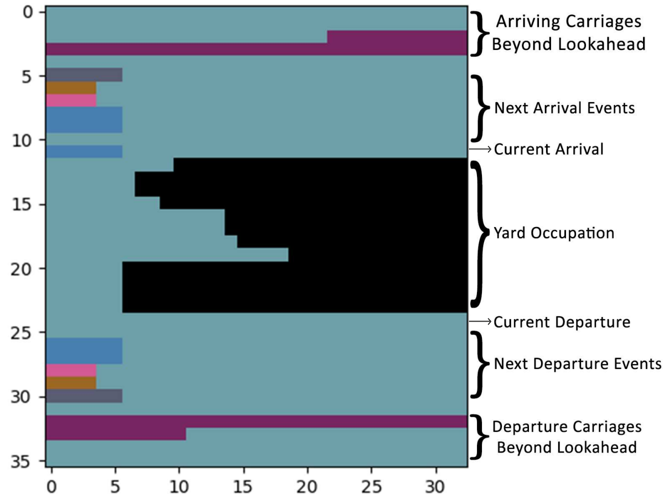

Figure 6: State representation of a 14 trains scenario used in [10].

Rewards
The reward function plays a significant role in the training process of the agent. There are various ways to penalize or praise the actions in RL simulations. Due to the fact that the action space is big, and most of the actions are movements, we experiment with various ways of reward shaping. More precisely, we force the agent to choose only among valid actions so there is no chance to take an action that is wrong or invalid. That fact helps the agent to learn only through valid selections, so there is no reason for penalization. The goal of the agent is to move (shunt) the trains wisely between the tracks, service

them on the service tracks, wait until the termination of the service, park them correctly in the parking tracks, and finally choose the correct track from where the train will depart. For that reason we set a small negative reward, -0.15, to all movements in order to force the agent to select as less *moving* actions as possible. Shunting movements in a shunting yard cost electricity and time, so a trained agent has to use the movement actions with parsimony. For the *service* action we set a reward of $+1$, and for the *wait* action a zero reward. If the agent wins an episode, which means that all the departures are made correctly without any delays, it takes $+10$ reward. At the departure time, the movement action from a parking track to the gate-way track was available and it has three levels of reward described in the following formula:

$$R(a \mid \text{departure}) = \begin{cases} +0.5, & \text{if exit movement is selected} \\ +1.5, & \text{if the exiting train is matched correctly} \\ +3.0, & \text{if the exiting train is serviced} \end{cases} \tag{14}$$

Training
The agent was trained with gradient based algorithms such as gradient descend variations after each action selection during the episode. More precisely, the agent starts the simulation by performing action at random for $40,000$ episodes without updating its weights. This is the first phase of the exploration mode where the agent performs random actions until the memory buffer is full. Afterwards, the agent starts the second phase of its exploration by making decisions following the ϵ-greedy policy described in Section 2.2.3. The discount factor of the referred exploration algorithm is set to a high number in order to slightly decrease the exploration probability starting from 1. After $20,000$ episodes, approximately, the agent stops the exploration having a very small probability of taking a random action and selects greedy actions following the policy of choosing the action with the highest Q-value. During the second phase of the exploration following the ϵ-greedy policy, the agent updates its weights after each action, so at each timestamp the agent makes a gradient step in the loss function and updates the weights of the Q-network. The calculation of the gradient was made from a mini-batch of 32 which is drawn randomly from the whole memory-buffer.

5 Algorithms

In this section the methods which used in the current research are reported. The section consists of two parts, each one describes a different approach on the problem. The first subsection describes the implementation of two variants of the Policy Gradient algorithm. The second subsection provides information about two implementations based on the Q-learning algorithm. The following paragraphs include preliminary information from Section 2.2.

5.1 Policy Gradient Methods

There are various algorithms and approaches based on the Policy Gradient Theorem proposed the recent years. The following subsections introduce some of the basic ones which are used in the current research. We implement the following algorithms on the environment used in [10] which contains only the two sub-problems *Matching* and *Parking* in order to examine how policy-based methods perform on this task and if we can approach such an optimization problem using deep-policy-networks. More precisely, the research question in the next two subsections is weather or not policy-based reinforcement learning algorithms fit to this kind of difficult optimization tasks using visual representation of the state-observation and the reward function described by Equation (13), using exactly the same experimental-setup used in [10].

5.1.1 Reinforce

The Reinforce algorithm is also known as the *Monte-Carlo Policy Gradient* theorem and relies on estimations of the return, G_t from Equation (2), by Monte-Carlo methods. The current algorithm uses episode samples to make updates on the policy. The basic idea behind the algorithm is that the expectation of the sampled gradient is approximately equal to the actual one. More precisely, the agent

follows a parameterized policy π_θ, which tries to optimize by making updates to the parameters θ using gradient-based optimization methods such as gradient ascent. Given the basic reinforcement learning objective from Equation (11) we can write the following formulas:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\pi_\theta}\big[R(\tau)\big] \\ J(\theta) &= \sum_{s\in\mathcal{S}} d^\pi(s)V^\pi(s) \\ J(\theta) &= \sum_{s\in\mathcal{S}} d^\pi(s)\sum_{a\in\mathcal{A}}\pi_\theta(a|s)Q^\pi(s,a) \end{aligned} \tag{15}$$

where d^π indicates a stationary distribution from the Markovian chain which could be considered as the dynamics of the environment. In stochastic environments this distribution moves the agent with a probability to the next state, s_{t+1} given the exerted action, a_t. More accurately, d^π is the distribution of the probability that $s_t = s$, starting from the initial state s_0, and comply the policy π_θ for t number of steps, $\pi_\theta \cdot d^\pi(s) = \lim_{t\to\infty} P(s_t = s|s_0, \pi_\theta)$. In the current project the environment is deterministic, so the probability of moving from one state to the next is fix, $d^\pi = P(s_t = s|s_0, \pi_\theta) = 1$.

Policy Gradient Theorem: *The derivative of the expected reward is the expectation of the product of the reward and the gradient of the log parameterized policy π_θ.*

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \sum_{s\in S} d^\pi \sum_{a\in\mathcal{A}} \pi_\theta(a|s)Q^\pi \\ &\propto \sum_{s\in S} d^\pi \sum_{a\in\mathcal{A}} \nabla_\theta\pi_\theta(a|s)Q^\pi \end{aligned} \tag{16}$$

Equation (16) involves the sum over the actions, $a_t \in \mathcal{A}$ which is derived from the classical Policy Gradient theorem but each term is not weighted by $\pi(a|s)$ needed for the expectation under the policy π_θ. Reinforce algorithm introduces the usage of the action $a_t = A$, which is the action that actually exerted at time t and the state, $s_t = S$, which is the state visited at that time step instead of using the sum. We can re-write Equation (16) by including this weighting without altering the equality, replacing a by the sample $A \sim \pi_\theta$ and use the return G_t, because $\mathbb{E}_\pi\big[G_t|S,A\big] = Q^\pi(S,A)$.

$$\begin{aligned} \nabla_\theta J(\theta) &= \mathbb{E}_\pi\left[\sum_{a\in\mathcal{S}} \pi_\theta(a|S)Q^\pi(S,a)\frac{\nabla_\theta\pi_\theta(a|S)}{\pi_\theta(a|S)}\right] \\ &= \mathbb{E}_\pi\left[Q^\pi(S,A)\frac{\nabla_\theta\pi_\theta(A|S)}{\pi_\theta(A|S)}\right] \\ &= \mathbb{E}_\pi\left[G_t\frac{\nabla_\theta\pi_\theta(A|S)}{\pi_\theta(A|S)}\right] \\ &= \mathbb{E}_\pi\Big[G_t\nabla_\theta \log\pi_\theta(A|S)\Big] \end{aligned} \tag{17}$$

Equation (17) is exactly what is needed for optimizing the parameterized policy π_θ using gradient optimization methods such as gradient ascent. The update formula of the parameters θ is given by the following formula:

$$\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log\pi_\theta(A|S) \tag{18}$$

Each increment of the value θ_t is proportional to the product of the return G_t and the vector of gradients of the probability taking action A divided by the probability of selecting that action from the current policy, $A \sim \pi_\theta$. The vector of gradients indicates the direction in a parameter space that increases the probability of action A, so the update increases θ towards that direction proportionally to the return G_t.

The aforementioned method has good convergence properties in theory but in many cases the Reinforce algorithm may be of high variance so produce slow learning. In general policy gradient methods suffer from the problem of sample efficiency and high variance. For that reason it could be generalized and include a comparison between the action-value and a baseline $b(s)$ in order to reduce the variance of the

gradient estimation, but in the same time to keep unchanged the bias factor. The baseline could be any arbitrary function, even a random variable. We can re-write Equations (17 & 18) including the baseline term $b(s)$.

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\pi\Big[\big(G_t - b(s)\big)\nabla_\theta \log \pi_\theta(A|S)\Big] \\ \theta &\leftarrow \theta + \alpha\big(G_t - b(s)\big)\nabla_\theta \log \pi_\theta(A|S)\end{aligned} \tag{19}$$

In the implementation of the Reinforce algorithm we included a baseline b_i based on [6]. The proposed baseline b_i, is constructed based on other samples $j \neq i : b_i = \frac{1}{k-1}\sum_{j\neq i} G_j$, where k indicates the number of samples (episodes), G_t the discounted reward function $G_t = \sum_{i=t}^{T} \gamma^{i-1} R_i$, and R_i the actual reward obtained from the environment. The following formula describes the gradient of the objective, $J(\theta)$, including the foresaid baseline.

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_\pi\Big[\big(G - b\big)\nabla_\theta \log \pi_\theta(A|S)\Big] \\ \nabla_\theta J(\theta) &\approx \frac{1}{k}\sum_{i}^{k} \nabla_\theta \log \pi_\theta(A|S)\Big(G_i - \frac{1}{k-1}\sum_{j\neq i} G_j\Big) \\ &= \frac{1}{k-1}\sum_{i=1}^{k} \nabla_\theta \log \pi_\theta(A|S)\Big(G_i - \frac{1}{k}\sum_{j}^{k} G_j\Big)\end{aligned} \tag{20}$$

Algorithm 1, describes the process of the Reinforce algorithm using deep-policy-network initialized with parameters θ, for the representation of policy π_θ. The neural network takes as input the state observation s_t at each time-step and constructs a probability distribution over the possible actions using a softmax activation function at the output layer $\sigma(x_i) = \frac{\exp x_i}{\sum_i^K \exp x_i}$. The former function generates a probability distribution over the actions which sums to 1. In this way the agent selects actions from the distribution built by following the current policy.

Algorithm 1 Reinforce

Input: State observation, $S_t \in \mathcal{S}$
Output: Softmax probability over actions, $A_{it} \in \mathcal{A}$
Initialize: Policy Network weights $\theta_i \in \mathbb{R}$
Generate episode by sampling on policy actions from $\pi_\theta : S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
for each episode step $t = 0, 1, \ldots, T-1$ **do**
  Estimate the discounted return $G_t = \sum_{t+1}^{T} \gamma^t R_{t+1}$
  Update policy network parameters: $\theta \leftarrow \theta + \alpha G_t \nabla_\theta \log \pi_\theta(A_t|S_t)$
end for

5.1.2 Proximal Policy Optimization

This subsection refers to the Proximal Policy Optimization algorithm (PPO) as a variant of the Policy Gradient. The authors in [14] proposed the method as an upgrade of the Reinforce algorithm, taking ideas from other algorithms such as the Trust Region Policy Optimization [12]. PPO is an actor-critic off-policy method, which means that combines the idea of learning by value and learning by policy. It is off-policy because it uses samples from old policies together with the current one in the learning process. PPO framework consists of two different models (neural-networks) in order to represent separately the policy, and the state-value. In this case the policy is stochastic and is represented by a network which generates a probability distribution over the possible actions. The value-function yields the expected value of being at the current state following policy π_θ. The policy model is called Actor and the value-function model is called Critic. Both models take as input the current state, $s_t \in \mathcal{S}$.

Actor works exactly as the deep-policy model in Section 5.1.1, which builds a softmax probability distribution over the possible actions $a_{ti} \in \mathcal{A}$ given the current state, s_t. The Critic takes as input the state s_t as well, and generates the expected state-value, $V(s_t)$ for that state. It justifies its name Critic because the model predicts the goodness of being at the current state following the current policy until

the termination of the episode. The referred algorithm optimizes the policy by using samples generated by previous policies. More accurately, the policy network is initialized arbitrarily two times in the beginning of a simulation, in order to create two Actor networks which produce two different softmax distributions: π_θ and $\pi_{\theta_{\text{old}}}$. The whole idea behind PPO is to optimize the target policy by making small updates in a safe region of the difference between the target and the old policy, taking the idea from the TRPO algorithm from [12].

Due to the fact that in PPO policy and value-function are represented separately, there is a need of two loss functions to train both networks. The Actor model (policy network) is trained similarly with the one in Section 5.1.1, based on the policy gradient theorem. The authors in [14] proposed a loss function which includes a new term, $\rho(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$, which denotes the ratio between the current and the old policy. The proposed loss function is a differential function based on the policy gradient algorithm from Equation (17). The following formula is called *Clipped Surrogate Objective*.

$$\mathcal{L}_\theta^{\text{CLIP}} = \mathbb{E}_\pi \left[\min\left(\rho(\theta) \cdot \hat{A}, \text{clip}(\rho(\theta), 1-\epsilon, 1+\epsilon) \cdot \hat{A}\right) \right] \tag{21}$$

Where epsilon is a hyper-parameter, e.g., $\epsilon = 0.2$ from [14], $\hat{A}_t$ the generalized advantage estimator from Equation (9), and $\text{clip}(x, 1-\epsilon, 1+\epsilon)$ indicates the function that sets its input value x, into the boundaries $[1-\epsilon, 1+\epsilon]$. Equation (21) could be split into two pieces: $\rho_t(\theta) \cdot \hat{A}_t$ and $\text{clip}(\rho_t(\theta), 1-\epsilon, 1+\epsilon) \cdot \hat{A}_t$. The objective indicates the expected value of the minimum quantity between the clipped and the unclipped product of $\rho_t(\theta)$ and $\hat{A}_t$. The advantage term, $\hat{A}_t$, is calculated with the Generalized Advantage Estimator (GAE) described in In this formula, $\rho_t(\theta)$ indicates the probability ratio between the likelihood of $\pi_\theta(a_t|s_t)$ and $\pi_{\theta_{old}}(a_t|s_t)$. It has to be mentioned that the division of the two policies is an element-wise division between the probabilities of taking an action a_t, in the state s_t, from distributions $pi_\theta(a_t|s_t)$ and $pi_{\theta_{old}}(a_t|s_t)$.

The current objective function seeks for the minimum change between the two policies and builds a trust region by establishing an interval $[1-\epsilon, 1+\epsilon]$. The idea behind it is that we need slight and smooth changes between the current and the old policy in order to be confident that the policy is trained towards the optimal one, by making small fluctuations with low variance. Succinctly, the final objective could be considered as a lower bound, *pessimistic bound* as it is referred in [14], on the unclipped objective. The changes in the ratio which makes the value $\mathcal{L}_\theta^{\text{CLIP}}$ bigger are ignored, whereas the fluctuations of the ratio that makes it smaller are included. The stability of the fluctuations some times leads to ignore policies which could return high future rewards, but it provides confidence that the changes will be harmless inside the trust region. The update rule of the trainable parameters of the Actor model is given by the following formula which is similar to Equation (18):

$$\theta \leftarrow \theta + \alpha \nabla_\theta \mathcal{L}_\theta^{\text{CLIP}} \tag{22}$$

where parameter α represents the learning rate of the optimization.

The goal of the Critic network is to be trained towards the direction of the gradients in order to produce accurate predictions for the values, V_t, which indicates the goodness of being at a current state and follow the policy π_θ onward. The network is trained with a standard regression loss function such as *mean squared error*:

$$\mathcal{L}^V = \sum_t^T \frac{(V(s_t) - G_t)^2}{T} \tag{23}$$

The final objective function that both networks (actor & critic) trained is a linear relation of $\mathcal{L}^{\text{CLIP}}$ and $\mathcal{L}^V$. In many cases there is an extra term in the final loss function the Shanon's entropy. In our project it was not used.

$$\mathcal{L}^{\text{CLIP}+V} = \mathbb{E}\left[c_1 \mathcal{L}_\theta^{\text{CLIP}} - c_2 \mathcal{L}^V\right] \tag{24}$$

Algorithm 2, describes the proximal policy optimization algorithm in fixed-length trajectory segments as it used in the current project. In [14] the algorithm is reported in a distributed fashion but in this project is used as a single core architecture. In each iteration, each one of the N parallel actors collect T timesteps of the data interacting with the environment. The optimization starts when an adequate amount of trajectory segments is collected.

Algorithm 2 PPO

Input: State observation, $S_t \in \mathcal{S}$
Output: Softmax distribution over actions, $A_{it} \in \mathcal{A}$
Initialize: Actor arbitrarily for π_θ and $\pi_{\theta_{old}}$
Initialize: Critic arbitrarily for V_w
for episode $= 1, 2, \ldots$ **do**
  Create trajectory τ, by sampling on policy Actor actions from π_θ
  Run the old policy $\pi_{\theta_{old}}$ in the same states collected from τ
  Compute $G\hat{A}E$, $\hat{A}_1, ..., \hat{A}_T$
  Optimize objective $\mathcal{L}^{\text{CLIP+}V}$ using gradient ascent
  Update Actor's weights θ of the current policy, π_θ
  Update Critic's weights w of the value-function, V_w
  Update the parameters of the old policy $\theta_{old} \leftarrow \theta$
end for

5.2 Deep Q-Learning Methods

This section contains information about two methods based on Q-Learning. The following methods tackle the problem including three subproblems of the whole TUSP: *Matching*, *Parking* and *Service tasks*. We found that Policy Gradient methods do not perform well on the current project even without including the subproblem of the *Service tasks*, probably due to lack of sample efficiency. In most of the cases, Policy Gradient algorithms are taking advantage of parallel implementations, where many Actors could collect trajectories from different episodes. This way the expectation of the gradient is more robust and the algorithm performs better in respect to convergence. We conclude that deterministic policies learnt by value, such as in Q-Learning, fit better in this kind of problems. The following sub sections describe two different variants of the Deep Q-Learning algorithm using the experimental setup described in Section 4.5.

5.2.1 Double DQN

Q-Learning is the most common reinforcement learning algorithm for learning good policies for sequential decision problems, through the optimization of a cumulative reward signal obtained from the environment. In many cases classic Q-Learning learns extremely high action-values due to the maximization over the estimated q-values. The following paragraphs describes the learning process using classic deep Q-Leaning networks (DQN) and afterwards the idea of how we can upgrade DQN method is described as well.

Deep Q-Networks
A deep Q network (DQN) is a multilayered neural network which takes as input a vector $s \in \mathcal{S}$, namely the state representation and produces values for the available actions, $Q(S, A) \in \mathbb{R}$. It could be considered as a mapping function from a n-dimensional state space to a m-dimensional action space containing m actions. So, it is a trainable function from $\mathbb{R}^n$ to $\mathbb{R}^m$, initialized with parameters θ. DQN algorithm proposed in [8], addresses two essential parts namely the target network and the experience replay memory buffer. Target network is initialized with parameters θ' and is a copy of the behaving network except from its parameters. Parameters of the target network, θ', are updated each t steps from the behaving network, $\theta' \leftarrow \theta$. The target network is used in the training process only for estimating the action-value, $Q_{\theta'}(s_{t+1}, a)$, in the calculation of the Bellman equation which outcome is used in the loss calculation, and consequently at the back propagation of the behaving network. The trajectories are saved in the memory buffer and sampled randomly during the training process where random trajectories selected uniformly in order to calculate the outcome of Equation (25).

$$Y_{\text{target}} = R_{t+1} + \gamma \max_a Q_{\theta'}(S_{t+1}, a) \tag{25}$$

The behaving network is trained towards the direction of the gradients on a regression error function such as *mean squared error*, using gradient descent optimization. Equation (26) describes the update rule of the parameters θ.

$$\theta \leftarrow \theta + \alpha \Big(Y_{\text{target}} - Q_\theta(S_t, A_t) \Big) \nabla_\theta Q_\theta(S_t, A_t) \tag{26}$$

It is obvious that the max operator in Equation (25) uses the same values for selecting and evaluating the action. More precisely, the maximum action-value $\max_a Q_{\theta'}(S_{t+1}, a)$, is obtained by selecting the maximum q-value over all the possible actions $a \in \mathcal{A}$ yield by the the $Q_{\theta'}$-network. This value indicates the action that would exerted using the current policy in the state $S_{t+1} \in \mathcal{S}$. All in all, the target network is used only in the evaluation process from which the $\max_a Q_{\theta'}(S_{t+1}, a)$ quantity is obtained, updating each t steps its weights from the online behaving network.

Using same parameters to select and evaluate the action, makes more likely the selection of overestimated q-values. These overoptimistic estimations could lead to false policies. The proposed algorithm in [16] prevents these overoptimistic value estimations by decoupling the weights of the selection and evaluation of the action values. In the Double Q-Learning algorithm two networks are used, the online (behaving) network and the target one, initialized respectively by weights θ and θ'. The difference between the Deep-Q-Networks approach and the Double Deep-Q-Learning is the calculation of the Y_{target} value in Equation (25). More accurately, the proposed method targets to the separation of the evaluation and the selection of the action, which means that the target model will be responsible of the evaluation of the action, but the selection will be made by the online network. Equation (27) formulates this idea of the calculation of Y_{target} using both networks. The aforementioned algorithm is described in Algorithm 3.

$$Y_{\text{target}}^{\text{DoubleQ}} = R_{t+1} + \gamma Q_{\theta'}\Big(S_{t+1}, \operatorname*{argmax}_a Q_\theta(S_{t+1}, a)\Big) \tag{27}$$

Algorithm 3 Deep Q-Learning with Double DQN

Input: State observation, S_t
Output: Q-value for each action
Initialize: Target and online network with θ' and θ
Collect N trajectories selecting actions at random
Save them in the experience replay memory
for each new episode **do**
  for each step of episode **do**
    Select action, a derived from ϵ-greedy policy using the online network
    Sample b random trajectories from memory
    Calculate $Y_{\text{target}}^{\text{DoubleQ}} = R_{t+1} + \gamma Q_{\theta'}\Big(S_{t+1}, \operatorname{argmax}_a Q_\theta(S_{t+1}, a)\Big)$ for each selected trajectory
    Update the parameters θ of the online network using gradient descent
    $\theta \leftarrow \theta + \alpha\Big(Y_{\text{target}}^{\text{DoubleQ}} - Q_\theta(S_t, A_t)\Big)\nabla_\theta Q_\theta(S_t, A_t)$
  end for
  Update Target network $\theta' \leftarrow \theta$ after n episodes
end for

5.2.2 Dueling DQN

Duelling Deep Q-network is a variation of the Deep Q-Learning algorithm which enhances the algorithm of the Deep Q-Networks from the previous section by including the dueling architecture in the Q-Network. The authors in [19] prposed an alternative neural network architecture which is better for model-free Reinforcement Learning. The current architecture focuses on the separation of the representation of the state value, $V(s)$, and the action advantages, $\hat{A}(s, a)$. The standard Q-Network takes as input the state-observation and predicts the Q-values for the available actions on the current state. The classic architecture of a Q-network from [8], consists of convolutional layers for feature extraction from the image, which outputs a representation in a different feature space. The output vectors of the convolutional layers are fed into fully connected layers which yield the Q-values.

Instead of using the standard architecture, the proposed method in [19] addresses the idea that it is essential to predict the value of the state $V(s)$, and use that information to stabilize the prediction of the Q-values, $Q(s, a)$, for each action choice. The authors claim, that in some states it is of paramount importance to know which action to choose, but in many other states the action choice using the $Q(s, a)$ information has non significant consequences. They found that using the state-value information is of great importance for each state, thus, they proposed a network architecture which encapsulates the

aforementioned idea depicted in Figure 7. It can be observed that two fully-connected layers are used after the convolutional layers, which estimate the state-value $V(s)$, and the advantage $\hat{A}(s,a)$, separately. At the final layer, the output of the fully-connected layers (referred to as *streams* in [19]) is combined in order to produce the Q-value for each action based on the advantage function from Equation (7). The output is obtained from the expression of $Q(s,a) = V(s) + \hat{A}(s,a)$.

The reader could consider a network where a set of weights θ_{conv}, is trained for the feature extraction from the image, another set of weights θ_V is used for the state-value estimation, and finally a θ_A set is used for the advantage estimation. The following formula describes the calculation of the output layer of the network. Note that the state-value, $V(s)$ is scalar while the advantage value, $A(s,a)$ is a vector. The calculation of the Q-values using Equation (28) happens for all state and actions (s,a) so we replicate the scalar $V(s)$ according to the length of the action space (in the current project the action space was 54).

$$Q(s,a;\theta_{\text{conv}},\theta_V,\theta_A) = V(s;\theta_{\text{conv}},\theta_V) + \hat{A}(s,a;\theta_{\text{conv}},\theta_A) \tag{28}$$

The network architecture of the referred paper can be used in various RL algorithms. We keep the same structure of the algorithm of the Double DQN from Section 5.2.1, using the Duelling network architecture. We followed the same training approach described in Algorithm 3, enhanced with the proposed Duelling network architecture from [19].

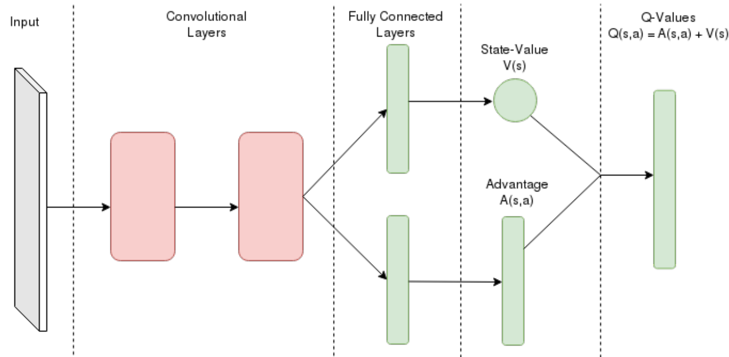

Figure 7: Dueling Network Architecture.

6 Results and Discussion

The current section includes the results of the simulations from Section 5. The policy gradient agents are implemented using the environment from [10] while the Q-Learning agents are implemented using the experimentation setup explained in Section 4.5. The policy gradient implementations were used in the initial problem of *matching* and *parking* (TMP) just to inspect if we can approach the problem with these kind of methods. The results from those agents are not promising, as explained in Section 5.2. Figure 8, visualizes the performance of the policy gradient agents in the small subproblem of TUSP including the *matching* and the *parking* task (TMP).

It is obvious that there is an increasing tendency in both curves in the aforementioned figure. The left-hand side plot illustrates the performance of the Reinforce agent with a baseline described in Section 5.1.1, where can be easily observed a positive spike after episode 80,000. The curve shows signs that the agent is learning, with a stable but slow, increasing trend on the average reward. The right-hand side plot depicts the performance of the PPO agent, a more advanced variation of the policy gradient theorem described in Section 5.1.2. This curve has a positive spike as well, but on different episode and indicates a stronger increasing tendency compared with the left-hand side one. Both algorithms did not perform well on the current problem, they need many more episodes in order to converge. The potential drawback of the algorithms used in the current project is the lack of sample efficiency. In most cases, policy gradient algorithms are taking advantage of parallel/distributed implementations in order to use much more data for the calculation of the gradient. In that way the estimation of the gradient would be more accurate calculated on data from many different instances, thus, the convergence would be faster. The lack of sample efficiency and the curse of high variance are the basic drawbacks of the Policy Gradient algorithms.

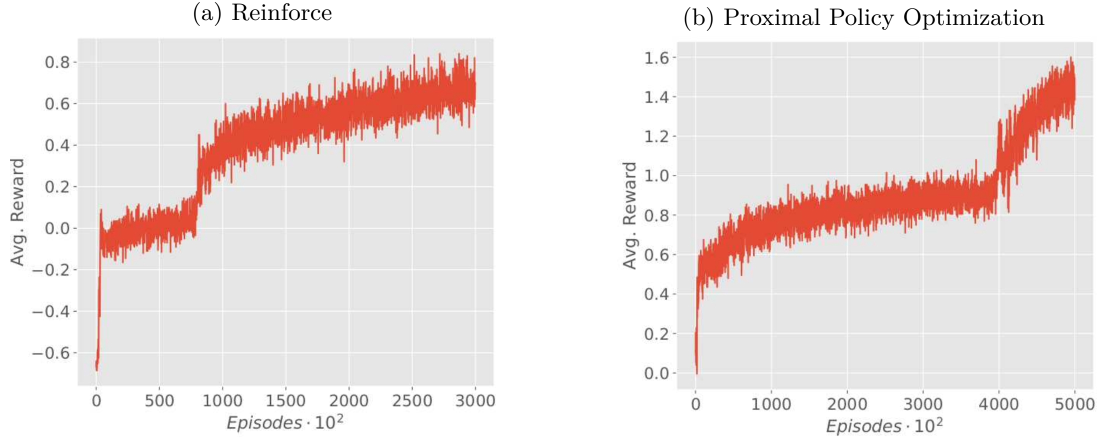

Figure 8: Average reward per 100 episodes for Policy Gradient methods.

Figure 9, illustrates the performance of the Duelling Deep Q-Learning agent described in Section 5.2.2. It visualizes the average reward per 100 consecutive episodes during the training process on 3, 4 and 5 trains scenarios using the yard depicted in Figure 1. First we trained the agent on 3-trains scenarios and then we performed transfer learning for a larger number of trains. More precisely, after the agent's learning convergence on the 3-trains scenarios, we used the trained weights for the next set of 4-trains scenarios, and so forth. In this way the agent uses the trained policy from an easy task (3-trains) and upgrades it through the training on more difficult scenarios (4 and 5 trains) with transfer learning.

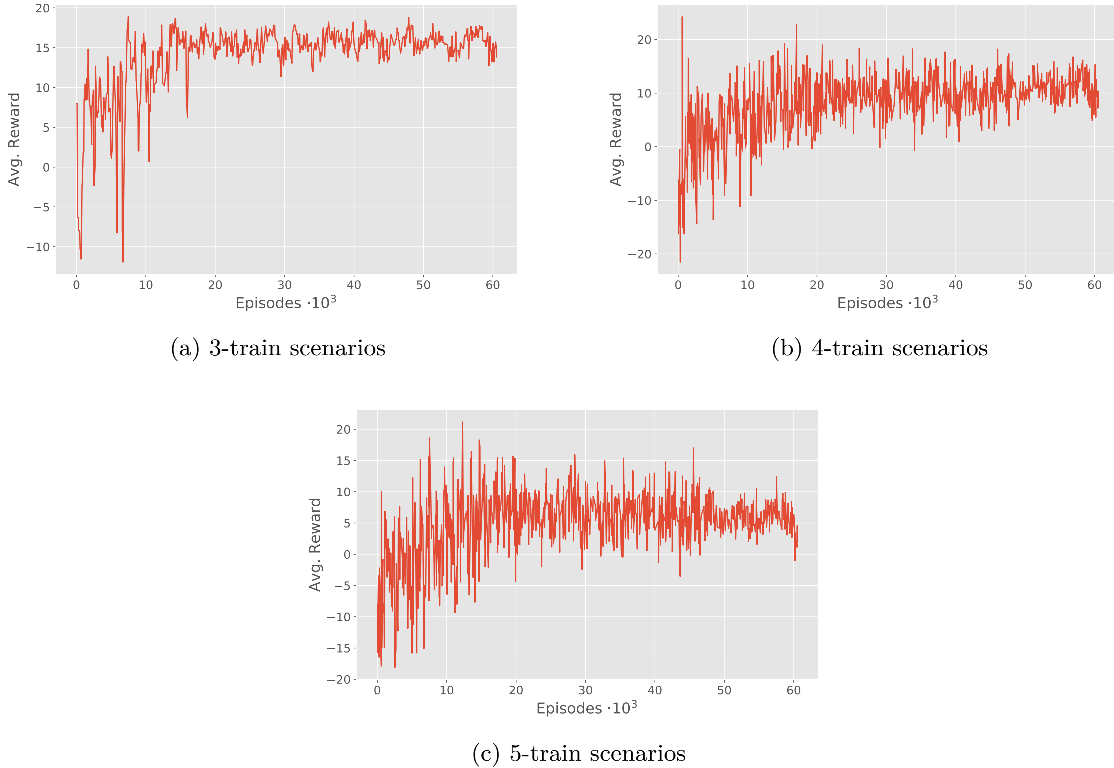

Figure 9: Average performance of the Duelling DQN-Agent during training.

In the left-hand side plot (a) in Figure 9, it can be observed that the agent learns a good policy in the first $20,000$ episodes and the average reward fluctuates around the value of 15 in a range of $[13, 17]$. In

the first episodes the curve is more noisy because the agent is on the exploration phase, conversely after some episodes the variance of the reward is decreased. The agent converges at the average reward value of 15. In the right-hand side plot (b), the curve is more noisy which means that the average reward per 100 episodes was more sparse with higher variance. One could notice that the curve follows almost the same pattern on the three plots but it converges in different values. The (b) one converges on average reward of 11, fluctuating between the range of $[9, 12]$ with a small increasing spike after the episode $40,000$. The third plot (c), illustrates the performance on 5-trains scenarios where the curve is even more sparse with more variance compared to the others. It can be easily observed that the agent's performance on the 5-trains scenarios was not promising because the curve has a stable decreasing tendency after episode $50,000$. The reward curve fluctuates around the average reward value of 7 on a range of $[4, 9]$ after $60,000$ episodes training.

Table 3, illustrates the results on the test scenarios compared with two baseline agents. The random agent selects random actions until a departure option is available, which is chosen immediately. The greedy agent performs random movements, selects always the service and the departure action if they are available. It is obvious that the best results of the Dual DQN agent occur on the 3-trains scenarios since it is an easy task for the agent. The test set contains 100 scenarios with different seeds from the training ones for each number of trains.

The results in Table 3 show that the agent is able to solve scenarios in the test set with a small number of trains. The agent performs well on small number of trains. It straggles to solve scenarios with higher number of trains due to the fact that it selects many movements from track to track and ends up the episode without service or depart some of the trains. It was pointless to train the agent on a larger number of trains since it could not stabilize a robust policy even for a small number of trains.

In order to analyse the learned policy, we measured the frequencies of movements and actions from three random winning and unsuccessful 5-trains scenarios. Figure 10, illustrates the frequency distribution of movement-actions in three successful scenarios, excluding *arrival* and *service* actions. It is obvious that the most selected action is wait (indicated by $[-1, -1]$). This is reasonable according to how the environment is set up. In each timestamp the environment provides to the agent the opportunity to decide the movements of all the trains sequentially by freezing the current timestamp, the agent chooses the actions consecutively. After the agent chooses an action, the environment generates possible actions for the rest of the trains, one by one, in the same timestamp. This leads the agent to learn that the wait action is needed many times in an episode as it explained in Section 4.3. Figure 11, illustrates the action distribution in three random unsuccessful scenarios. It is again obvious that wait is the most picked action. In the left-hand side plot, we can observe that the agent spends its time on moving the trains around the yard which leads him to lose that scenario. The right-hand side graph, shows that the agent uses the wait action many times and probably that was the reason of its failure. The third plot illustrates almost the same case as the right one, where the agent chooses many times the wait action and fails the scenario.

	Number of Trains		
Agents	**3**	**4**	**5**
Random	10%	7%	6%
Greedy	66%	50%	44%
Duelling-DQN	84%	64%	61%

Table 3: Percentages of correctly solved scenarios in the test set.

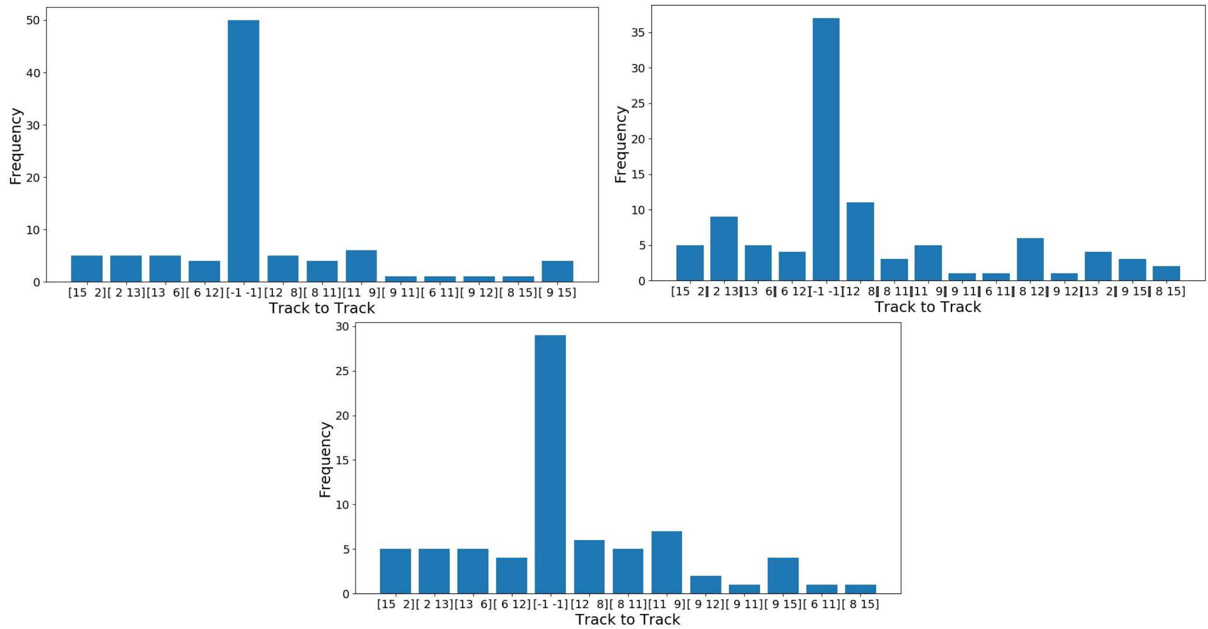

Figure 10: Action frequencies for movements and wait from 3 random winning scenarios.

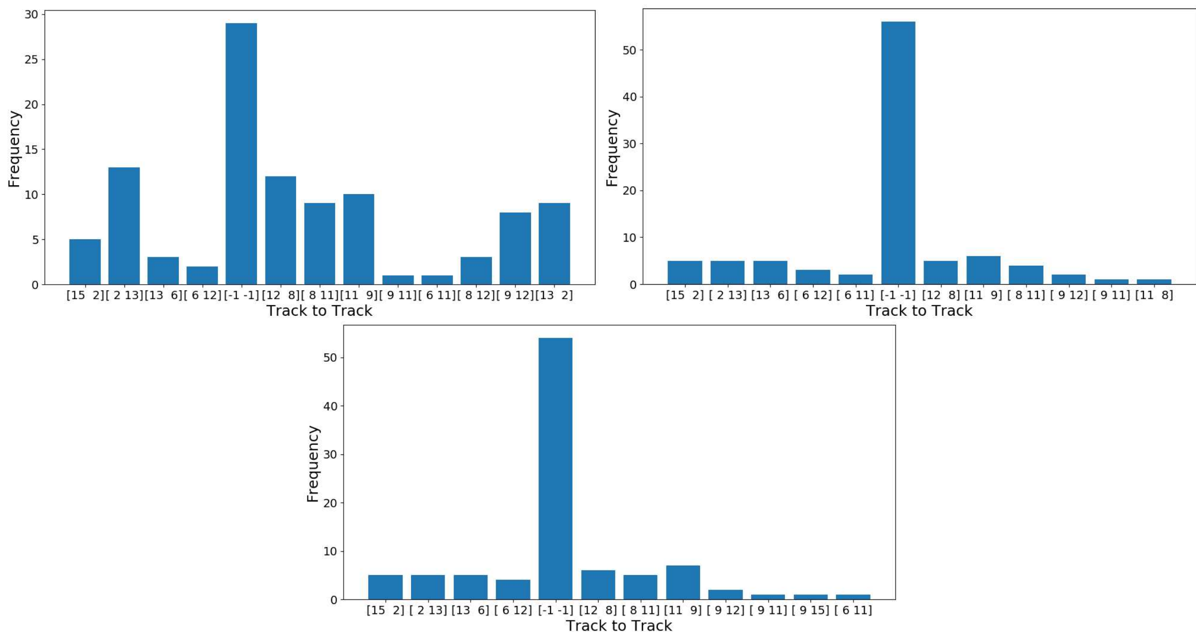

Figure 11: Action frequencies for movements and wait from 3 random unsuccessful scenarios.

7 Conclusion and Future Research

This section contains the final conclusions and interesting findings that we obtained from the simulations on the current project. It includes some suggestions for further research, as well. We show that Deep Reinforcement Learning methods are capable of solving a promising number of scenarios of the Train Unit Shunting Problem, including three basic subproblems namely *matching*, *parking* and *schedulling of the service tasks*. The agent learns the concept of time exerting the *wait* action, and learns how to handle all the necessary movements inside the yard for all the trains.

The whole project includes many different simulations and approaches in order to inspect that fits better to the current problem of the TUSP. We first experiment on how Policy Gradient algorithms perform on the initial problem of *parking* and *matching* (TMP) based on [10], in order to invest if we can follow ideas that vary from the Depp Q-Network approach, such as Policy Gradient methods or attention based Reinforcement Learning from [5]. Since we found that the Policy Gradient framework does not produce promising results on the initial problem of TMP, we tried to implement the problem from [10] on TORS using the same agent, in order to test the environment. The results of the TMP simulation, described in Section 4.4, show that we can use TORS for further research and that we can easily add more learnable tasks for the agent. Afterwards, we set up the environment, including the new task of *scheduling the service tasks* (cleaning) which is the basic goal of the current research.

The initial results using the TORS software package were on the TMP problem, where we tried to implement exactly the same environment set up from [10] and reproduce its results. The first achievement was to build the infrastructure where the agent can communicate correctly with the TORS environment, on a specific yard, using scenarios from the instance generator. The results were promising, which leads to the conclusion that the same agent used in [10], performs better on TORS. Consequently, that fact helps us to continue upgrading the set up of TORS including *scheduling the service tasks* as an extra task, where we trained the final agents explained in Section 5.2.

From the information in Section 6, we can conclude that Policy Gradient methods do not fit well in the current problem. Some of the potential problems are discussed already in the previous sections. It seems that deterministic policies obtained from value-based methods, such as Q-Learning, fit better in the current task. We can see that the agent is not capable to learn a stable policy for solving scenarios, especially with a larger number of trains, e.g., 5-trains scenarios. The reason could be that the action space is big and most of the actions are movements, so the agent spends much time moving the trains from track to track and loses the chances to select the service or the departure action. In most of the cases of an unsolved scenario, the agent spends most of its time moving the trains, thus in the departure time many of the trains are not serviced, which leads to a delay and a termination of the scenario. Other potential problems could be that the state representation is not the optimal one in order to help the agent to distinguish the correct decision. We used a representation of the state based on the idea in [10], which performed well on the task of *matching* and *parking*, but it seems that it is not optimal for the *services* and the set-up of big action spaces. Splitting the information of the state into multiple channels, simulates an image based state like the Atari game from [8].

The environment was implemented in such a way to provide to the agent the opportunity to decide the moves of all the trains inside the yard consecutively in each time stamp, as it explained in Section 4.3. On the one hand, that fact provides to the agent the opportunity to choose the next move of all the trains in a time stamp and make overall decisions for the current state. On the other hand, that fact probably causes a problem to the agent to build a stable policy and to choose the correct action.

7.1 Future Work

There are various ways to tackle the current problem from the AI perspective. Here are some of our ideas for future work on the current problem, which probably provide better solutions and overcome the obstacles which were faced by the agent. The first idea is to change the structure of the environment using the TORS software package. More precisely, the way that TORS produces available actions over the timestamps is peculiar and it probably brings some trouble to the agent to build a robust distribution over the actions in order to learn a stable policy. It would be a good idea if the TORS software could provide all the possible actions for all the trains on each timestamp in order to help the agent to observe the whole likelihood over the actions, as it was already mentioned in previous sections. Then, algorithms such as DQN or PG could probably perform better, keeping the same state-observation representation.

Another idea which could probably produce better results, is provided by Kool et al.'s [5], which formulates the whole problem of TSP into a Markovian Decision Process (MDP) in order to use RL techniques, and paves the way for an alternative way of thinking about solving optimization problems using AI. More accurately, Kool et al. provide an approach on the so-called TSP, a difficult optimization problem, using advanced Deep Learning techniques such as Multihead Attention, inside an encoder-decoder architecture, which is trained through trial and error stochastically, using the Reinforce algorithm from the Policy Gradient family. The state of TUSP could also be represented easily as a graph with V nodes and E edges, since all train shunting yards could be represented as graphs. The proposed idea is to use

Attention techniques and encoder-decoder architectures as a mapping function which learns to represent the state into a different vector space, and use that embeddings in the learning process. Thus, the results of [5] are promising and show that it is a useful approach which could be applied to various routing optimization problems such as the current TUSP.

References

[1] Kanak Chandra Bora. "A Short Survey Report on Application of Traveling Salesman Problem". In: *Binary+, Journal of the School of Engineering Technology, USTM* 1 (2016), pp. 1–7. URL: https://www.ustm.ac.in/wp-content/uploads/2020/02/Ann3tiX2JZH2u7rMqOC7QFHDI46zsGve9KNCooOR.pdf.

[2] Roel W. van den Broek. "Train shunting and service scheduling: An integrated local search approach". MA thesis. Universiteit Utrecht, 2016. URL: https://dspace.library.uu.nl/bitstream/handle/1874/338269/ThesisRoelvandenBroek.pdf.

[3] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. USA: W. H. Freeman Co., 1979. ISBN: 0716710447.

[4] Rajeev Goel and Raman Maini. "Vehicle routing problem and its solution methodologies: a survey". In: *International Journal of Logistics Systems and Management* 28.4 (2017), pp. 419–435.

[5] Wouter Kool, Herke van Hoof, and Max Welling. *Attention, learn to solve routing problems!* 2018. arXiv: 1803.08475 [stat.ML].

[6] Wouter Kool, Herke van Hoof, and Max Welling. "Buy 4 REINFORCE samples, get a baseline for free!" In: *Proceedings ICLR 2019 Workshop: Deep Reinforcement Learning Meets Structured Prediction*. 2019.

[7] Ramon M. Lentink. "Algorithmic decision support for shunt planning". PhD thesis. Erasmus Universiteit Rotterdam, 2006. URL: http://hdl.handle.net/1765/7328.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[9] Mohammadreza Nazari, Afshin Oroojlooy, Lawrence Snyder, and Martin Takác. "Reinforcement learning for solving the vehicle routing problem". In: *Advances in Neural Information Processing Systems (NIPS)*. 2018, pp. 9839–9849.

[10] Evertjan Peer, Vlado Menkovski, Yingqian Zhang, and Wan-Jui Lee. "Shunting trains with deep reinforcement learning". In: *Proceedings 2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. 2018, pp. 3063–3068.

[11] Laurent Perron and Vincent Furnon. *OR-Tools*. Version 7.2. Google. URL: https://developers.google.com/optimization/.

[12] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. "Trust region policy optimization". In: *Proceedings International Conference on Machine Learning (ICML)*. 2015, pp. 1889–1897.

[13] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. *High-dimensional continuous control using generalized advantage estimation*. 2015. arXiv: 1506.02438 [cs.LG].

[14] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. *Proximal policy optimization algorithms*. 2017. arXiv: 1707.06347 [cs.LG].

[15] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2nd ed. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[16] Hado Van Hasselt, Arthur Guez, and David Silver. "Deep reinforcement learning with double Q-learning". In: *Proceedings 30th AAAI Conference on Artificial Intelligence*. 2016, pp. 2094–2100.

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. “Attention is all you need”. In: *Advances in Neural Information Processing Systems (NIPS)*. 2017, pp. 5998–6008.

[18] Arno van de Ven, Yingqian Zhang, Wan-Jui Lee, H Eshuis, and Anna Wilbik. “Determining capacity of shunting yards by combining graph classification with local search”. In: *Proceeedings 11th International Conference on Agents and Artificial Intelligence (ICAART)*. Vol. 2. 2019, pp. 285–293.

[19] Ziyu Wang, Tom Schaul, Matteo Hessel, Hado Van Hasselt, Marc Lanctot, and Nando De Freitas. *Dueling network architectures for deep reinforcement learning*. 2015. arXiv: 1511.06581 [cs.LG].