VRIJE UNIVERSITEIT AMSTERDAM

MASTER BUSINESS ANALYTICS
MSC THESIS

Utilizing Available Data to Warm Start Online Reinforcement Learning

Acedemic Supervisor:
A. EL HASSOUNI
M. Hoogendoorn

Author:
S.V. DE GEUS

Date:
June 2020

Second Reader:
D. DOBLER

External Supervisors:
V. MUHONEN
D. TIMMERS
R. PRICE

Utilizing Available Data to Warm Start Online Reinforcement Learning

Sofie Valesca de Geus

Master Thesis

June 2020

Vrije Universiteit Amsterdam
Faculty of Science
Business Analytics
De Boelelaan 1081a
1081 HV Amsterdam

Mobiquity Inc.
Data Analytics Team
Tommaso Albinonistraat 9
1083 HM Amsterdam

Abstract

Reinforcement learning (RL) algorithms have shown to achieve significant successes on a large set of simulated environments. However, the application of RL in real-world problems has proven to be more challenging. In most real-world problems, there is no access to an accurate model of the environment. Therefore, a policy has to be learned by interacting with the real system where every action has real consequences. This is very impractical since reinforcement learning algorithms often start with many steps of poor performance before finding a better policy. It is therefore crucial to find a way to minimize poor actions and learn as efficiently as possible. In this thesis, we propose a setup that utilizes an available dataset with logged environment interactions to warm start reinforcement learning (WS-RL). This proposed setup combines both offline and online reinforcement learning techniques. We show WS-RL is able to improve performance for more complex tasks. It reduces the time needed to find a decent policy and therefore improves both initial and overall performance. It however does not seem to improve the convergence of policies in the long run. Additionally, we try to get a better understanding of the effect of exploration when learning on the real system. To do this we compare multiple exploration methods and settings and show which one performs best across different environments.

Preface

The work presented here contains my thesis: "Utilizing Available Data to Warm Start Online Reinforcement Learning". This thesis has been written as part of the graduation requirements of the MSc Business Analytics at the Vrije Universiteit Amsterdam. The research was conducted part-time for eight months at the host organisation Mobiquity.

Mobiquity gave me complete freedom in choosing the research subject and I, therefore, got to explore my interest in reinforcement learning. Throughout my study, I had not yet learned about this and therefore this was a good opportunity. Mobiquity put me in contact with my supervisor Ali el Hassouni who is doing a PhD in reinforcement learning and machine learning in healthcare. He helped me form my research question and guided me further through the process.

Hereby, I would like to thank Mobiquity for giving me this opportunity and providing me with all the necessary tools. I enjoyed my time at Mobiquity and being part of the analytics team. A special thanks to my external supervisors Vesa Muhonen, Dennis Timmers and Richard Price, and my academic supervisor Ali el Hassouni. They provided me with guidance and useful feedback throughout the entire research process.

Sofie de Geus,
May 2020, Amsterdam

Contents

1 Introduction **1**

2 Literature: Applications **3**

3 Background **5**
3.1 Reinforcement Learning 5
3.1.1 Markov Decision Process 6
3.2 Algorithms 8
3.2.1 Model-based vs Model-free 8
3.2.2 Online vs Offline 9
3.2.3 On-policy vs Off-policy 10
3.2.4 Overview 10
3.3 Q-learning 11
3.4 Deep Q-learning 13
3.4.1 Experience Replay 13

4 Methodology **15**
4.1 Warm Start Reinforcement Learning (WS-RL) 15
4.2 Environments 18
4.2.1 CartPole 18
4.2.2 LunarLander 19
4.3 Models 20
4.3.1 Deep Q-Network (DQN) 20
4.3.2 Double Deep Q-Network (Double DQN) 20
4.3.3 Dueling Deep Q-Network (Dueling DQN) 22
4.4 Exploration Methods 23
4.4.1 Greedy 23
4.4.2 ϵ-Greedy 24
4.4.3 ϵ-Greedy Decay 24
4.4.4 Softmax 25

4.5 Evaluation 25

5 Experimental setup **27**
5.1 Setup 28
5.1.1 Offline Reinforcement Learning 28
5.1.2 Online Reinforcement Learning 29
5.2 Experiment Logging 30

6 Results & Conclusion **31**
6.1 Offline Reinforcement Learning 31
6.1.1 Performance of Model Architectures 31
6.1.2 Offline Models 34
6.2 Online Reinforcement Learning 34
6.2.1 Performance of Exploration Methods 34
6.2.2 Performance of Model Architectures 37
6.2.3 Performance of Warm Start 40

7 Discussion **42**
7.1 Future Work 43

Bibliography **44**

Appendix A Implementation Details **49**
A.1 Amazon Web Services (AWS) 49
A.2 Packages & Libraries 49
A.3 Experiment Logging 49

Appendix B Horizon **51**
B.1 Training Scripts 51
B.1.1 Offline Reinforcement Learning 51
B.1.2 Online Reinforcement Learning 52
B.2 Hyperparameters 53

Appendix C Results **54**
C.1 Offline Reinforcement Learning 54
C.2 Online Reinforcement Learning 56

CHAPTER 1

Introduction

Nowadays data is everywhere around us and the world of data-driven technologies is ever-evolving. Machine learning techniques are used to learn from data, identify patterns and provide insights. A key area where machine learning can be used to create business value is in the decision-making process. These techniques can analyze data faster and more accurate allowing businesses to make better decisions with minimal human intervention. Most industries working with large amounts of data have already recognised the value of machine learning technology [19].

A promising machine learning technique that can tackle complex sequential decision making is reinforcement learning (RL). It differs from other machine learning techniques as it dynamically learns by adjusting actions to optimize for future reward. The idea behind reinforcement learning is an algorithm, or agent, will learn from environment interactions by receiving rewards for performing correct actions and penalties for incorrect actions. An environment can be seen as a representation of the problem that needs to be solved. During these environment interactions, the agent can choose to exploit what it has learned or explore other options to get a better understanding of the environment dynamics. The goal of reinforcement learning is to find a suitable action model, also called policy, that would maximize the total cumulative future reward. By using the feedback from its actions the agent is able to make better behavioural decisions without being explicitly told what to do. This process of learning is known as the trial and error method, which imitates the learning of human beings.

Throughout the years reinforcement learning algorithms have shown to achieve significant successes on a large set of simulated environments

[26, 36, 28, 42]. The advantage of having access to an accurate simulator or model of the environment is that the agent is able to safely learn a policy before deploying it on the real system. In this case, data becomes effectively unlimited and consequences for poor actions are practically non-existent. Progress made with real-world applications has been much slower. In most real-world problems, it would be extremely difficult to obtain an accurate simulator, since these problems are complex and have unclear rules. This implies the agent has to learn on the real system with real consequences for its actions. Typically, reinforcement learning agents learn good policies only after many steps of poor performance. This becomes very impractical in problems where data collection is expensive (e.g., in robotics) and risky (e.g., in autonomous driving, or healthcare).

To overcome these challenges we propose a setup that utilizes an available dataset with logged environment interactions, created by a previous controller, to warm start reinforcement learning (WS-RL). This proposed setup combines both offline reinforcement learning and online reinforcement learning. First, an offline agent is trained solely on the available dataset without interacting with the environment. This form of reinforcement learning does not affect the real system and is still able to find an improved policy. Next, this pre-trained model is used as a starting point for the online agent. This agent is able to interact with the environment and create its own experience samples. In this warm start, the agent would no longer have to learn from scratch and it is therefore expected to boost performance. To show the effect of this warm start we compare this setup to reinforcement learning without extra inputs referred to as cold start (CS-RL). By comparing both techniques we would like to answer the question: "Does online reinforcement learning become more efficient by utilizing available data as a warm start?"

The second point of interest is to show the effect of exploration when learning on the real system. Finding a good balance between exploration and exploitation has been a fundamental issue in reinforcement learning all along. This becomes even more prominent when every action taken in the environment during learning and exploring has an immediate impact on the real system. In this research, we look at the performance of multiple exploration methods and show which one performs best across different environments. To find this answer we will consider both the rewards obtained during acting and learning (system rewards) as well as the performance of the policies found during learning (policy rewards).

CHAPTER 2

Literature: Applications

Throughout the years reinforcement learning has shown some great successes in learning policies for sequential decision-making problems and control. Especially games with simulated environments proofed to be excellent test-beds. One of the first applications of reinforcement learning was on the game of checkers in Samuel et al. (1959) where heuristic search methods were combined with temporal-difference learning [33]. Another impressive application of reinforcement learning was on the game of backgammon in Tesauro et al. (1992). Here the program TD-Gammon was introduced which required little knowledge about the game itself and used a model-free reinforcement learning algorithm similar to Q-learning [39].

These techniques still had some trouble scaling to high-dimensional problems due to complexity issues (e.g. memory, computational and sample complexity). Advances in deep learning [20] helped to overcome these problems resulting in applications of reinforcement learning in problems with larger state and action spaces. In 2013, Mnih et al. connected a reinforcement learning algorithm to a deep neural network presenting deep Q-learning (DQN). This framework was able to successfully learn control policies directly from high-dimensional sensory input [26]. In 2016, DeepMind developed AlphaGo, a computer program that combines advanced search tree with deep neural networks, which was the first computer program to defeat a professional human Go player [35]. AlphaGo was trained on a large database of expert moves and many hours of self-play. A later improved version, AlphaGo Zero, was introduced that masters the game of Go without human knowledge [36]. In 2019, both OpenAI Five [28] and AlphaStar [42] were able to achieve good results on the real-time strategy (RTS) game Dota 2 and Star-Craft II, respectively. These games capture

more of the messiness and continuous nature of the real world (e.g. long horizons, imperfect information real-time planning and complex state-action spaces).

The above successes require large amounts of environment data to succeed. In real-world settings, this becomes problematic since data collection is often costly, risky, and time-consuming. To overcome this other papers also focus on leveraging available environment data. In robotics reinforcement learning has been applied to multiple robotic control tasks like learning ball-in-a-cup [16], table tennis [15], locomotion [17] and flight control [1, 27]. Here human-expert demonstrations are often used to effectively reproduce the desired behaviour through imitation learning [34]. It can however fail when dealing with sub-optimal demonstrations. Others leverage batch reinforcement learning techniques [41] that can be used to learn from a fixed batch of data no matter the quality. In Komorowski et al. (2018) batch reinforcement learning was applied to find optimal treatment strategies for sepsis using two datasets [13, 30]. Learning was performed on batches of data and off-policy evaluation was used to evaluate their policy against data from another policy [18]. Other work has been done on recommender systems [23, 47], computer resource management [24], Intelligent Traffic Signal Control [25, 7, 46], real-time bidding [12] and more.

CHAPTER 3

Background

3.1 Reinforcement Learning

Each day people encounter problems that involve decision making. Most real-world problems are not simple enough to be solved by a single decision. Solving them involves sequences of decisions that are dependent on each other. This means a single decision will have both immediate and long term consequences. In these situations feedback is often delayed and sparse, making it difficult to see the direct impact of an action on the outcomes. Reinforcement learning is one of the ways to solve sequential decision-making problems.

In reinforcement learning an algorithm, or agent, learns by interacting with an environment via the trial-and-error method. An environment can be seen as a representation of the problem that needs to be solved. Environments can be expressed in a set of states, which the agent tries to influence via its choice of actions. At each step, the agent observes the environment and performs an action. This will change the state of the environment. The agent receives a positive reward signal for performing correctly and a negative reward signal for performing incorrectly. By using the feedback from its actions and experiences the agent can make better behavioural decisions. Contrary to other machine learning approaches an agent can learn how to maximize its long-run reward without being explicitly told how to perform a task.

A representation of this interaction can be seen in the figure below, here the agent and environment interact at each discrete time step $t = 0, 1, 2, 3....$

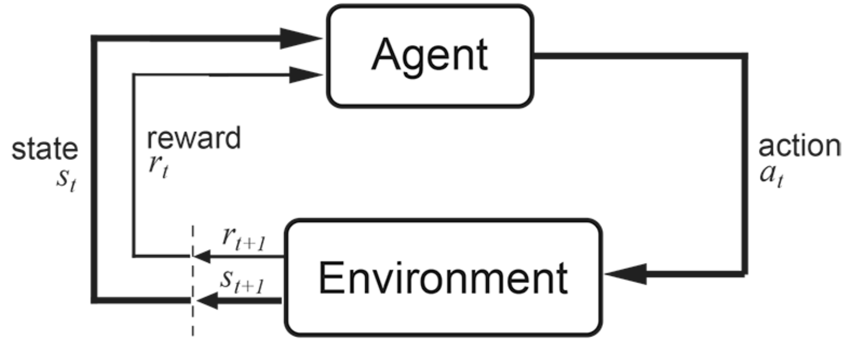


Figure 3.1: The agent–environment interaction in a Markov decision process.

3.1.1 Markov Decision Process

Formally, a reinforcement learning problem can be defined as a Markov decision process (MDP) [41]. An MDP is a 5-tuple $\mathcal{M} =< S, A, T, r, \gamma >$ consisting of:

- S: *the state space*. At each time step, the state of the environment is an element $s \in S$. A state is an observation of all that is important in a state of the problem that is modelled.
- A: *the action set*. At each time step, the agent chooses to perform an action $a \in A$. Actions can be used to control the environment state.
- $T(s, a, s')$: *the transition function*, representing the probability of ending up in state $s' \in S$ after doing action $a \in A$ in state $s \in S$.
  $T : S \times A \times S \rightarrow [0, 1]$
- $r(s, a)$: *the reward function*, representing the (expected) immediate reward obtained after performing action $a \in A$ in state $s \in S$.
  $r : S \times A \rightarrow \mathbb{R}$
- γ: *the discount factor*, controlling the importance of future rewards.
  $0 \leq \gamma \leq 1$

The MDP gives a mathematical framework for sequential decision-making problems. In such a process the transition probabilities to possible next states are only dependent on the current state and the selected action. This means all relevant historical information for determining the next state is present in the current state. A solution of this MDP would describe how the agent should act. The behaviour of an agent is defined by a policy and maps all states to actions.

- $\pi(s)$: *the policy function*, representing the strategy of which action $a \in A$ to take in a state $s \in S$, $\pi : S \rightarrow A$

The goal in reinforcement learning is to learn the optimal policy π^*, such that the cumulative future reward is maximized. The cumulative future reward can also be described as the return, denoted by R. The equation for calculating the return can be seen in equation 3.1.

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{3.1}$$

In problems with a finite horizon, an agent stops when reaching a terminal state. These terminal states are dependent on the definition of the environment. A single run of the environment is called an episode. The return after a terminal state is always zero, such a state can be defined as an absorbing state. A transitional graph, like the one in figure 3.2, can be used to show the dynamics of an MDP.

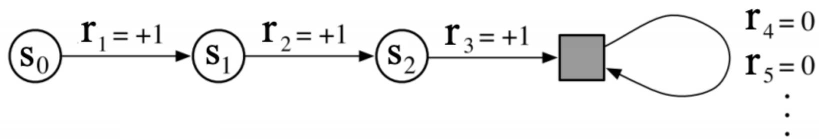


Figure 3.2: Transitional graph

3.2 Algorithms

Multiple distinctions can be made concerning the algorithms of solving reinforcement learning problems. First, there is a distinction between model-based and model-free algorithms, which involves the modelling of the environment. Second, there is a distinction between online and offline algorithms, which involves the way of sampling. Third, there is a distinction between on-policy and off-policy algorithms, which involves the way of performing policy updates. These distinctions will be explained in this section. Additional, an overview is given with some of the more popular algorithms.

3.2.1 Model-based vs Model-free

The distinction between model-based and model-free algorithms concerns the modelling of the environment.

In **model-based algorithms**, the agent attempts to model the environment. First, it tries to capture the transition function T and the reward function R of the MDP by interacting with the environment. Afterwards, based on the functions it learns, the agent tries to estimate the optimal policy. For this, it can use value-iteration or policy-iteration. However, in most real-life problem settings, it is not possible to find the exact reward function or transition function using limited samples due to complexity. If the model is inaccurate, model-based reinforcement learning can fail miserably, because small errors can grow rapidly over a long horizon.

In **model-free algorithms**, on the other hand, the agent tries to estimate the optimal policy without trying to model the environment first. Instead, it uses value functions to estimate the optimal policy. A value function represents an estimate of how good it is to be in a certain state or perform an action in a certain state. Finding these value functions solely relies on interaction with the environment. In Section 3.3 we will explain more about value functions and how they can be used to learn new policies.

3.2.2 Online vs Offline

The distinction between online and offline reinforcement learning concerns the way of sampling.

In **online reinforcement learning**, the agent is allowed to interact with the environment creating its own samples. The optimal policy is found by alternating between the exploration phase and the learning phase multiple times. In the exploration phase, we focus on obtaining new experiences from the environment using a learned policy. In the learning phase, we focus on learning from these previously obtained experiences to find new policies. In this case, the policy performance can be obtained by applying it directly to the environment during the exploration phase.

Figure 3.3: Online reinforcement learning

On the contrary, in **offline reinforcement learning** the agent itself is not allowed to interact with the environment during learning. First, a batch of samples is obtained from the environment, then the optimal policy is learned solely from this earlier obtained batch. This means there is no exploration possible. To estimate the policy performance often a counterfactual policy evaluation (CPE) method is used, which is able to estimate the performance without running it on the real system.

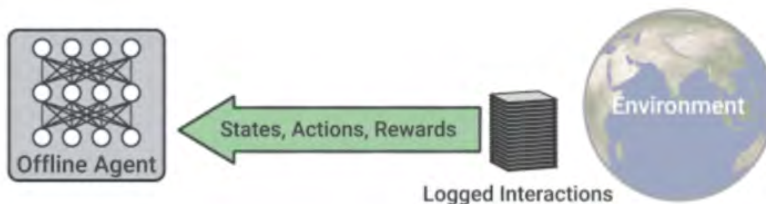


Figure 3.4: Offline reinforcement learning

3.2.3 On-policy vs Off-policy

The distinction between on-policy and off-policy algorithms concerns the way of performing policy updates. A distinction can be made between the behavioural policy which generates the samples, and the target policy which is the learned policy that has to be evaluated.

In **on-policy** algorithms, decisions about the actions are made based on our best policy at that time and it uses the information gathered from taking that action to improve on the best policy. Here the policy you want to evaluate is the same policy that generates the data. On-policy algorithms are only suitable for online learning since the behavioural policy needs to be known.

In **off-policy** algorithms, our behaviour of interacting with the environment is unrelated to the optimal policy at the time of taking the action. Here the policy we want to evaluate is different from the data generating policy. This way it learns an optimal policy no matter which policy it is carrying out. These off-policy algorithms are suitable for both online and offline learning.

3.2.4 Overview

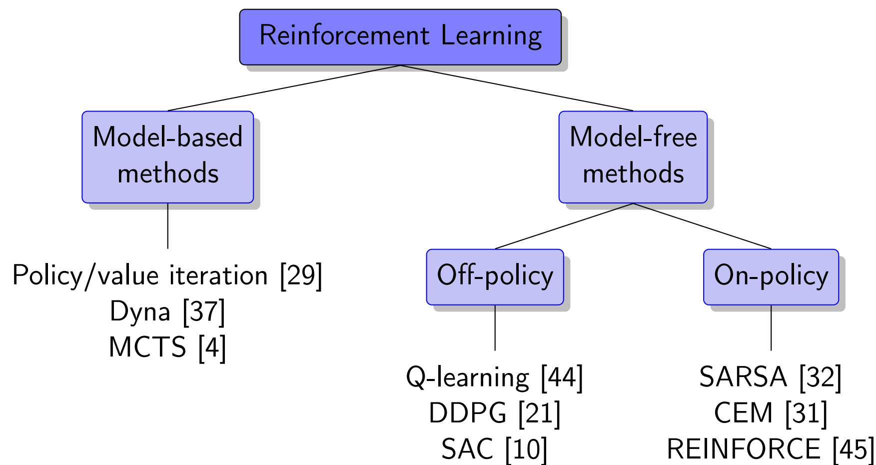


Figure 3.5: Overview of reinforcement learning algorithms

3.3 Q-learning

A popular algorithm used for reinforcement learning is Q-learning. Q-learning is a model-free and off-policy algorithm [44]. The pseudo-code of this algorithm is presented in Algorithm 1. The basic idea of Q-learning is to approximate the state-action value function by observing the interactions with the environment. In Q-learning we define the state-action value function $Q : S \times A \to \mathbb{R}$, denoted by $Q^{\pi}(s,a)$, as the expected long-term return from taking action a in state s and behaving under policy π.

$$Q^{\pi}(s,a) = E_{\pi}\Big\{R_t | s_t = s, a_t = a\Big\} \tag{3.2}$$

Using equation 3.1 it can be written as,

$$Q^{\pi}(s,a) = E_{\pi}\Big\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\Big\} \tag{3.3}$$

This Q-value is used to express the quality of an action in a given state for every state-action pair. The basic form of Q-learning, also called tabular Q-learning, uses a look-up table to represent the Q-function. It is however also possible to use a function approximator to learn the Q-function (i.e. deep learning).

At the start of learning the Q-function estimate will be initialized. Next, an iterative method is used based on the Bellman equation (equation 3.4) to update the Q-function estimate. The Bellman equation defines the expected long-term return for a certain state and action as the immediate reward plus the expected long-term return for the next state. Because rewards in the future are less certain they are discounted by a factor γ, $0 \leq \gamma < 1$. If the discount rate is set to zero, only the immediate rewards are taken into account, while a discount rate approaching one will make it strive for a long-term high reward.

$$Q(s_t, a_t) = r(s_t, a_t) + \gamma \cdot max_a Q(s_{t+1}, a) \tag{3.4}$$

After every time step t the Q-values can be updated by the new information it obtained using the following update rule:

$$Q_{t+1}(s_t, a_t) = (1 - \alpha) \cdot Q_t(s_t, a_t) + \alpha \cdot \big[r(s_t, a_t) + \gamma \cdot max_a Q_t(s_{t+1}, a)\big] \tag{3.5}$$

In this update rule, a weighted average is taken of the old Q-value estimate, $Q_t(s_t, a_t)$, and the new Q-value estimate, $r(s_t, a_t) + \gamma \cdot max_a Q_t(s_{t+1}, a)$. The new learned value consists of the actual obtained reward and the estimate of the expected long-term return of the next state discounted by γ. Here the learning rate α, $0 \leq \alpha < 1$, determines to what extent newly acquired information overrides old information. If the learning rate is set to zero the agent will learn nothing, while with a learning rate of one it will only consider the most recent information. The idea of this method is that the more experience is obtained the more accurate the predictions of the optimal policy will be.

Algorithm 1: Q-learning

1: Set values for learning rate α, discount factor λ
2: Initialize Q
3:
4: **for** all episodes **do**
5: initialize state s_0
6: **for** each time-step t **do**
7: $a_t \leftarrow$ select action based on Q and exploration method
8: take action a_t, and observe reward $r(s_t, a_t)$ and next state s_{t+1}
9: $Q_{t+1}(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q_t(s_t, a_t) + \alpha \cdot \left[r(s_t, a_t) + \gamma \cdot max_a Q_t(s_{t+1}, a) \right]$
10: $s_t \leftarrow s_{t+1}$
11:

3.4 Deep Q-learning

In deep Q-learning, the Q-learning algorithm is combined with an artificial neural network [26]. Here the neural network is used as a function approximator for the value function also referred to as the Q-network. This Q-network maps states to Q-values for all possible actions. Deep Q-learning no longer has to store the Q-values for every state-action pair, but only the neural network with its parameter weights θ. This approach scales much better than Q-learning. Deep Q-learning is often combined with an experience replay mechanism [22]. The pseudo-code for deep Q-learning with experience replay can be found in Algorithm 2.

Again in deep Q-learning the Bellman equation (equation 3.4) is used to iterative update the Q-function estimate. This is done by updating the Q-network weights by minimising the loss function, see equation 3.6. The loss function here is the mean squared error of the predicted Q-value and the target Q-value. The new Q-value, $r(s_t, a_t) + \gamma \cdot max_a Q_t(s_{t+1}, a; \theta)$, represents the target.

$$Loss = (r(s_t, a_t) + \gamma \cdot max_a Q_t(s_{t+1}, a; \theta) - Q_t(s_t, a_t; \theta))^2 \tag{3.6}$$

This is a regression problem, however, the target or actual value is unknown. It can be argued that it is predicting its own value, but since $r(s_t, a_t)$ is the unbiased true reward, the network is going to perform a gradient descent step to update the network weights θ using backpropagation to finally converge.

3.4.1 Experience Replay

Using experience replay [22] enables reinforcement learning agents to memorize and reuse past experiences. Every time step t the agent stores its experience $e_t = (s_t, a_t, r_t, s_{t+1})$ consisting of the state, the action, the reward and the next state. All these experiences are put in a data set $D_t = e_1, ..., e_t$.

During learning, the Q-learning updates can now be applied on samples (mini-batches) taken from D_t. This is done by drawing uniformly at random from the pool of stored samples, $E \sim U(D_t)$. Another name for the samples that are stored is the so-called replay buffer. The size of the replay buffer will define how many experiences will be "remembered". A replay buffer of size N only contains the last N experiences.

The advantage of using experience replay is that it separates the learning phase from gaining experience, which removes correlations in the training samples [26]. Instead of learning from experiences that were obtained directly after each other, with experience replay the chronological relationship between experiences is broken. Another advantage is the efficient use of previous experience, as it can be used for learning multiple times. This is very important when gaining real-world experience is costly.

Algorithm 2: Deep Q-learning with Experience Replay

1: Set values for learning rate α, discount factor λ
2: Initialize empty replay buffer D to capacity N
3: Initialize Q with random weights
4:
5: **for** all episodes **do**
6: initialize state s_0
7: **for** each time-step t **do**
8: $a_t \leftarrow$ select action based on Q and exploration method
9: take action a_t, and observe reward r_t and next state s_{t+1}
10: store experience $e_t = (s_t, a_t, r_t, s_{t+1})$ in D
11:
12: Sample random mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from D
13: Set $y_j = \begin{cases} r_j, & \text{for terminal } s_{j+1} \\ r_j + \gamma max_a Q(s_{j+1}, a_j; \theta), & \text{for non-terminal } s_{j+1} \end{cases}$
14: Calculate loss $(y_j - Q(s_j, a_j; \theta))^2$
15: Perform a gradient descent step to update θ
16:

CHAPTER 4

Methodology

In this section, we discuss the warm start approach (4.1), the environments (4.2), models (4.3), exploration methods (4.4) and evaluation metrics (4.5) used in this research.

4.1 Warm Start Reinforcement Learning (WS-RL)

In many real-world settings of reinforcement learning, there is no accurate simulator of the system. This implies that learning has to be done by interacting with the real system. In this kind of setting, reinforcement learning becomes very impractical, since this technique is exploratory by nature. Typically, a reinforcement learning agent starts with many steps of poor performance before learning a better policy. This is manageable in simulated environments, but poor performance on the real system will have major consequences in most real-world applications (e.g. healthcare, autonomous cars). It is therefore important to utilize knowledge about the environment that is already available, instead of learning from scratch. In a real-world setting, environment knowledge often occurs in the form of available data with logged environment interactions from previous controllers. In our approach, we suggest utilizing this data to warm start reinforcement learning (WS-RL). To do this we create a training pipeline that combines both offline and online reinforcement learning, see Figure 4.1.

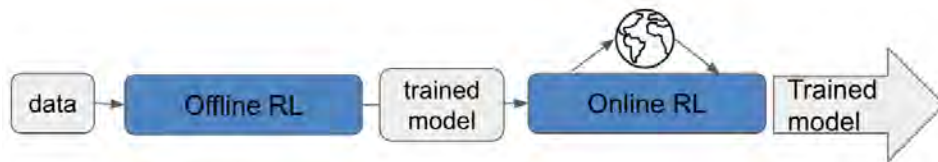


Figure 4.1: Training pipeline of WS-RL

This warm start approach is applied to the popular RL algorithm, deep Q-learning. This technique was earlier described in Section 4.3.1. The pseudo-code of WS-RL can be seen in Algorithm 3. This code can be split up in two parts. Lines 6-10 of Algorithm 3 focus on offline reinforcement learning. In offline reinforcement learning, there is no interaction between the agent and the environment which decouples the data collection from the learning phase. This means the agent has to learn from a fixed batch of data that was earlier obtained from the environment. The experiences from this dataset are now used to approximate the value function. This way of learning does not affect the environment. After learning the agent will have found a better estimate of the value function. Next, lines 12-21 of Algorithm 3 focus on online reinforcement learning. Usually, the agent initializes its value function estimate randomly without any prior knowledge. However, now it uses the value function estimate from offline reinforcement learning as a starting point. Here the agent will be able to interact with the environment and create its own samples. The model alternates between the exploring phase, creating more samples, and the learning phase, learning from these samples, multiple times. Now after every interaction, the new experience will be added to the agent's memory.

Algorithm 3: WS-RL

1: Set values for learning rate α, discount factor λ
2: Initialize D^{data} with available samples
3: Initialize empty replay buffer D^{replay}
4: Initialize Q with random weights θ
5:
6: **for** all epochs **do** ▷ Offline RL
7: **for** all batches in D^{data} **do**
8: Sample random mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from D^{data}
9: Set $y_j = \begin{cases} r_j, & \text{for terminal } s_{j+1} \\ r_j + \gamma max_a Q(s_{j+1}, a; \theta), & \text{for non-terminal } s_{j+1} \end{cases}$
10: Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
11:
12: **for** all episodes **do** ▷ Online RL
13: initialize state s_0
14: **for** each step t in episode **do**
15: $a_t \leftarrow$ select action based on Q and exploration method
16: take action a_t, and observe reward r_t and next state s_{t+1}
17: store experience $e_t = (s_t, a_t, r_t, s_{t+1})$ in D^{replay}
18:
19: Sample random mini-batch of transitions (s_j, a_j, r_j, s_{j+1}) from D^{replay}
20: Set $y_j = \begin{cases} r_j, & \text{for terminal } s_{j+1} \\ r_j + \gamma max_a Q(s_{j+1}, a; \theta), & \text{for non-terminal } s_{j+1} \end{cases}$
21: Perform a gradient descent step on $(y_j - Q(s_j, a_j; \theta))^2$
22:

4.2 Environments

OpenAI Gym provides a collection of test problems, called environments, for developing and comparing reinforcement learning algorithms. It comes with a diverse suite of environments that range from easy to difficult and involve many different kinds of data [3]. In this research, we will be using the CartPole and LunarLander environment. Both environments have a discrete action space and fall under control problems, combining mechanics and reinforcement learning. These environments show similarity to control problems in the real world and show a different level of complexity.

4.2.1 CartPole

In the CartPole-v1 environment, see Figure 4.2, a pole is attached to a cart which moves along a frictionless track. The system can be controlled by applying a force of +1 or -1 to the cart. The pole starts upright, and the goal is to prevent it from falling over. A reward of +1 is provided for every time step the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from centre. The maximum length of an episode is 500 time steps, and therefore also the maximum reward [8].

CartPole-v1 defines "solving" as getting an average reward of 475.0.

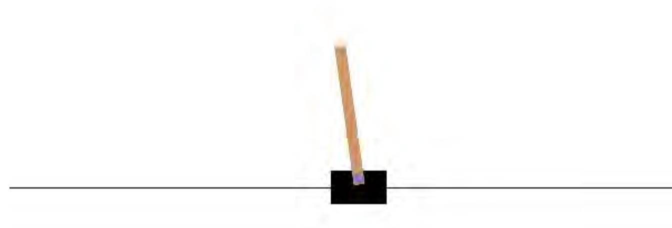


Figure 4.2: CartPole

The state-space has four dimensions of continuous values which represent the position, the velocity, the pole angle and the pole velocity at the tip. The action space has two discrete values which represent the direction of the force applied on the cart.

State Space	
0	Cart Position
1	Cart Velocity
2	Pole Angle
3	Pole Velocity At Tip

Table 4.1: CartPole State Space

Action Space	
0	Force in left direction
1	Force in right direction

Table 4.2: CartPole Action Space

4.2.2 LunarLander

In the LunarLander-v2 environment, see Figure 4.3, a space-ship tries to land between two flags smoothly. The system can be controlled by firing the three different engines or doing nothing. Rewards are given for moving towards the landing pad, not crashing and leg ground contact. Penalties are given for moving away from the landing pad, crashing and firing the main engine. The episode ends when the lander crashes or comes to rest. The maximum length of an episode is 1000 time steps [9].

LunarLander-v2 defines "solving" as getting an average reward of 200.

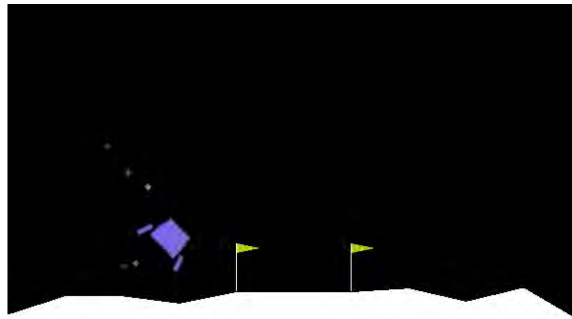


Figure 4.3: LunarLander

The state-space has eight dimensions of continuous values. The action space has four discrete actions, representing doing nothing, firing the left engine, firing the main engine and firing the right engine.

State Space	
0	x-Coordinate
1	y-Coordinate
2	x-Velocity
3	y-Velocity
4	Lander Angle
5	Angular Velocity
6	Right-leg Grounded
7	Left-leg Grounded

Table 4.3: LunarLander State Space

Action Space	
0	Do nothing
1	Fire left engine
2	Fire main engine
3	Fire right engine

Table 4.4: LunarLander Action Space

4.3 Models

In this research, three different models are used. The models used are all based on deep Q-learning which was earlier explained in Section 4.3.1. The first model is identical to this technique and the other two are an extension of this technique, introduced to improve performance.

4.3.1 Deep Q-Network (DQN)

The Deep Q-Network [26] was first used by DeepMind in 2013 and is based on the deep Q-learning technique described in Section 4.3.1. Here Q-learning is combined with a neural network and experience replay. Figure 4.4 shows the architecture of the neural network used in this paper. It is however also possible to create a different architecture. In this research, no convolutional layers are used, but the network consists of fully connected layers. The loss in equation 3.6 is minimized using the Adam optimizer [14].

4.3.2 Double Deep Q-Network (Double DQN)

In 2015, van Hasselt et al. combined deep Q-networks with double Q-learning constructing the Double DQN [40]. Double Q-learning [11] was introduced to solve the overestimation of Q-values in Q-learning. This

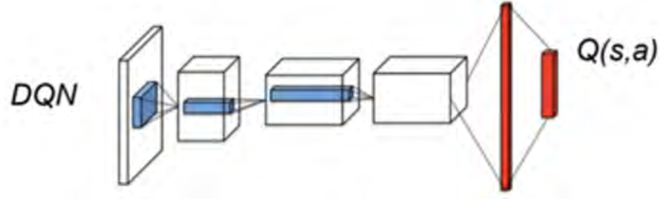


Figure 4.4: architecture Deep Q network [26]

overestimation bias in Q-learning is a result of using the same Q-function estimate for action selection and evaluation. This makes it more likely to select overestimated actions, resulting in overoptimistic value estimates.

To overcome overestimation double Q-learning uses two independent Q-functions Q and $\tilde{Q}$ to decouple the action selection from the evaluation. The update rule is still similar to equation 4.1 in Q-learning, but each Q-function is now updated with a value from the other Q-function for the next state, see equation 4.1 and 4.2. This way when Q is being updated $\tilde{Q}$ is considered to compute the target and vice versa. This is said to improve stability in learning since the target function will stay fixed for a while instead of being a moving target. Both value function Q and $\tilde{Q}$ are also based on different experiences. The pseudo-code of double Q-learning can be found in Algorithm 4. Here it assigns the two different Q-functions Q and $\tilde{Q}$ as either the "update" function or "target" function. It then selects an action based on the update function in line 11 and uses this action in the target function for the update in line 13.

$$\begin{aligned} Q_{t+1}(s_t, a_t) \leftarrow (1-\alpha) \cdot Q_t(s_t, a_t) + \\ \alpha \cdot \left[r(s_t, a_t) + \gamma \cdot \tilde{Q}_t(s_{t+1}, argmax_a Q_t(s_{t+1}, a) \right] \end{aligned} \tag{4.1}$$

$$\begin{aligned} Q_{t+1}(s_t, a_t) \leftarrow (1-\alpha) \cdot Q_t(s_t, a_t) + \\ \alpha \cdot \left[r(s_t, a_t) + \gamma \cdot \tilde{Q}_t(s_{t+1}, argmax_a Q_t(s_{t+1}, a) \right] \end{aligned} \tag{4.2}$$

Again in double Q-learning, a neural network can be used as a function approximator for the two Q-function creating a deep Q-network with double Q-learning. This is done in the same manner as for deep Q-learning explained in Section 4.3.1.

Algorithm 4: Double Q-learning

1: Set values for learning rate α, discount factor λ
2: Initialize Q and $\tilde{Q}$
3:
4: **for** all episodes **do**
5: initialize state s_0
6: **for** each time-step t **do**
7: $a_t \leftarrow$ select action based on Q, $\tilde{Q}$ and exploration method
8: take action a_t, and observe reward $r(s_t, a_t)$ and next state s_{t+1}
9:
10: assign $Q^{update} = Q$ and $Q^{target} = \tilde{Q}$, or $Q^{update} = \tilde{Q}$ and $Q^{target} = Q$
11: $a_{t+1} = argmax_a Q_t^{update}(s_{t+1}, a)$
12: $Q_{t+1}^{update}(s_t, a_t) \leftarrow (1-\alpha) \cdot Q_t^{update}(s_t, a_t) + \alpha \cdot [r(s_t, a_t) + \gamma \cdot Q_t^{target}(s_{t+1}, a_{t+1})]$
13:

4.3.3 Dueling Deep Q-Network (Dueling DQN)

In 2015, Wang et al. combined deep Q-learning with a dueling architecture. The Q-value that is estimated in Q-learning represents the value of a given action a in state s (equation 3.2). The dueling network was introduced to generalize learning across actions without changing the underlying algorithm. In this network, the Q-value is split into two parts. First, $V(s)$ the value of being in state s independent of the action. Second, $A(s,a)$ the advantage of taking action a in state s.

$$Q^{\pi}(s,a) = V^{\pi}(s) + A^{\pi}(s,a) \tag{4.3}$$

$$V^{\pi}(s) = E_{\pi}\{R_t | s_t = s\} \tag{4.4}$$

The idea behind Dueling Q-learning is that in some states the action taken does not have an impact on the environment [43]. For instance, think of a game where the agent has to dodge objects. The action taken will only be important if there is a risk of collision with an object. If there is no risk of collision it is not necessary to calculate the value of each action. The dueling architecture will learn which states are valuable, without learning the effect of each action for each state.

To do this both elements, $V(s)$ and $A(s,a)$, are estimated separately using two streams of fully connected layers. After this, both estimates are combined to produce the Q-function and outputs the Q-values for each action. In Figure A.2 a representation for the Dueling DQN is shown.

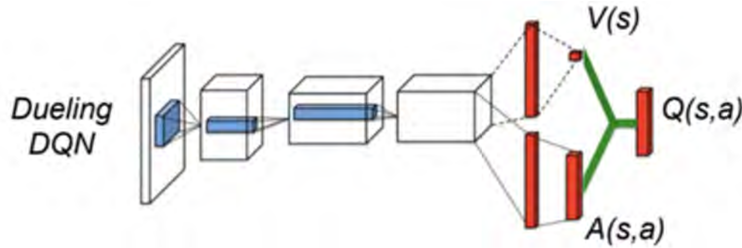


Figure 4.5: architecture Dueling Deep Q network [43]

4.4 Exploration Methods

In Q-learning, the agent learns a Q-function that can be used to determine an optimal action. However, it does not specify how the agent should use these Q-values. Two things are useful for the agent to do. The agent can choose to exploit what it has learned by choosing the action with the highest Q-value estimate. Or the agent can choose to explore other options to create a better estimate of the Q-function and make better decisions in the future. Finding a balance between exploration and exploitation is a fundamental issue in reinforcement learning. To find this balance an exploration method is used, which defines the strategy used to select actions. In this research, multiple exploration methods are used and they are discussed below.

4.4.1 Greedy

The simplest way of selecting actions is via the greedy method. This is an example of pure exploitation since the actions are always selected on the agent's current knowledge. This is done by selecting actions with the highest Q-value.

$$a_t = argmax_{a \in A} Q_\pi(s_t, a), \quad \forall s \in S \tag{4.5}$$

A disadvantage of this method is that the Q-values are used for action selection even though they have not converged yet. This is the case at the beginning of training when the agent knows little about the environment and the estimate of the Q-function is still unreliable. The agent has to explore to gain insight into the environment dynamics. Relying solely on exploitation is also likely to trap the agent in a local optimum. Not exploring unseen parts of the environment can prevent the agent from improving and finding the optimal policy.

4.4.2 ϵ-Greedy

An ϵ-greedy exploration is one of the most used exploration methods [38]. It uses a parameter ϵ, $0 \leq \epsilon \leq 1$, to balance exploration and exploitation. The agent chooses the action with the highest Q-value with a probability of $1 - \epsilon$ and the other times a random action is taken. The ratio between exploration and exploitation is influenced by the value chosen for ϵ. The higher this value, the more exploration is performed. Here it is necessary to choose an appropriate value for epsilon. The greedy strategy can also be defined as an ϵ-greedy strategy with ϵ equal to 0.

$$a_t = \begin{cases} argmax_{a \in A} Q_\pi(s_t, a) & \text{with probability } 1 - \epsilon \\ random & \text{with probability } \epsilon \end{cases} \tag{4.6}$$

A disadvantage of the ϵ-greedy method is that the agent keeps exploring the environment even though it has learned a good policy. When the estimate of the Q-function converges to the true value the focus should be on exploitation. Too much exploration in task-irrelevant parts of the environment is very inefficient.

4.4.3 ϵ-Greedy Decay

The ϵ-greedy decay method is similar to the ϵ-greedy method. It also selects a random action with probability ϵ. The difference is that instead of the value of ϵ staying constant during training it will decay over time. This means that at the beginning of training the exploration rate is higher than later on. This solves the drawback of the ϵ-greedy method, where the agent keeps exploring even though it has found a good policy. Now there will be less exploration as the Q-values converge. Here it is necessary to choose an appropriate decay rate.

4.4.4 Softmax

A drawback of these previous ϵ-greedy methods is that during exploration the agent chooses equally among all actions. This means that it is as likely to choose the worst action as it is to choose the next-best-action. In tasks where taking the worst action has major consequences, this is not recommended. The softmax method uses the Boltzmann distribution to assign probabilities of selecting an action. Now all actions are ranked and weighted based on their value estimates. The greedy action is still given the highest selection probability. At time step t an action a in state s is selected with a probability $p(s_t, a)$.

$$p(s_t, a) = \frac{e^{Q_t(s_t,a)/\tau}}{\sum_{i=1}^{n} e^{Q_t(s_t,a^i)/\tau}} \tag{4.7}$$

The temperature τ, $\tau \geq 0$, controls the randomization of the action selection process. If the temperature is close to zero the agent does not explore, resulting in a greedy policy. If $\tau \to \infty$ there are almost no differences between probabilities resulting in a random policy. Here it is necessary to choose an appropriate value for τ.

4.5 Evaluation

In this research multiple models with different algorithms, exploration methods and hyperparameters are compared. Evaluation of these models is important to get an idea of their performance. The performance can be expressed in the number of rewards obtained from the environment. Since the environments in this research all have an episodic horizon and stop after a finite number of time steps, the rewards can be calculated for a single episode. How the reinforcement model is judged depends on how the agent will be deployed.

If the agent is able to learn safely before being deployed, the amount of rewards obtained by the final policy is most important. To evaluate the performance of a single policy it can be run on the environment without any exploration. This will show how many rewards are obtained by solely following the policy. The *policy rewards* are computed by running multiple episodes and averaging the total rewards. The agent's policy is evaluated at multiple times during training. The performance of the policy shows if it will eventually converge to the optimal policy.

The agent is only able to learn safely if a simulated environment of the real system is available. In real-world problems this is often not the case, meaning the agent has to learn while being deployed. In this situation, not only the learned policies are important, but also the effect of the exploration method. In this situation, the trade-off between exploration and exploitation becomes even more important, since taking bad actions on the real system can have an immediate negative impact. The performance of the model on the real system can be expressed by how much rewards it receives during acting and learning. These so-called *system rewards* are tracked during the whole training period.

CHAPTER 5

Experimental setup

In this section, we will explain the experiments that are performed. The experiments focus on the performance of reinforcement learning in both an offline and online setting. The experiments are done for three different model architectures, namely Deep Q-networks, Deep Q-networks with double Q-learning (Van Hasselt et al., 2016) and Deep Q-networks with dueling architecture (Wang et al., 2015).

For all experiments in this research, we use the reinforcement learning platform Horizon [6]. More information about this platform and the use of it during this research can be found in appendix B. Horizon provides implementations of multiple model architectures based on their original paper and scripts for training these models. These scripts allow for both offline and online reinforcement learning. The pseudo-code and a more extensive explanation of these scripts can also be found in Appendix B.

Additional, the platform provides configuration files with parameter settings for training. These parameter settings are either based on the original papers or recommended by Horizon themselves. Some parameter values are different per environment. An overview of the hyperparameters used during training can be found in Appendix B.2. These settings are used when not specified otherwise.

For all environments, the online experiments are done with 3 different seeds. These seeds affect the initial values of the neural network and can influence learning. The final results are averaged over these different seeds.

5.1 Setup

In our proposed warm start setup we use both offline and online reinforcement learning. In our experiments, we look at both their performance independent of each other as well as combined.

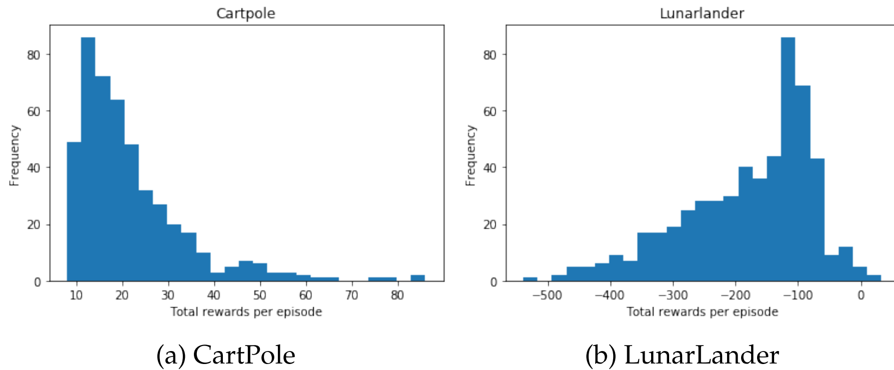
5.1.1 Offline Reinforcement Learning

In offline reinforcement learning, a small parameter search is performed using the *Hyperopt* package. This package makes it possible to run multiple trials, and it chooses its direction within the search space based on the results. Hyperopt uses a method called the Tree-structured Parzen Estimator (TPE). This approach was introduced at NIPS 2011 by Bergstra et. al [2]. In total 25 trials are run for each model architecture. The hyperparameters considered for optimization are the learning rate, gamma and the target update rate. The parameters and search space can be found in Table 5.1.

Parameter	Space
learning rate	LogUniform(-10,0)
gamma	Uniform(0,1)
target update rate	Uniform(0,1)

Table 5.1: Parameter space

During offline training, a model learns from a batch of data without any interaction with the environment. The data used during learning is created by sampling from the OpenAI Gym environment. As already explained in Section 3.1 the agent interacts with the environment every discrete time step and stores its experience. All these experiences obtained during sampling form the dataset. During sampling the agent uses a random behaviour policy, meaning all actions are randomly selected. The number of samples in the datasets are chosen based on the complexity and the episode length of an environment. For the Cartpole and LunarLander environment, the dataset contains 10.000, 50.000 samples, respectively. Training can now be done by looping over this data with batches of size 256. In total, the models are trained for 100 epochs. After every epoch, the performance is evaluated in an online manner. More information about the script used for offline training can be found in the appendix (B.1).



(a) CartPole

(b) LunarLander

Figure 5.1: This figure shows a histogram of the episode rewards in the datasets. The dataset for the CartPole environment contains 460 episodes in total. The policy used to create this dataset achieved 21.77 episode rewards on average. The dataset for the LunarLander environment contains 545 episodes in total. The policy used to create this dataset achieved -178.52 episode rewards on average.

5.1.2 Online Reinforcement Learning

In online reinforcement learning the way of exploration has a major effect on the performance. Finding a good balance between exploration and exploitation is a fundamental issue in reinforcement learning. We, therefore, look at the performance of multiple exploration methods. The three exploration methods used are earlier described in Section 4.4. To get a good idea of their performance multiple exploration settings are tested. For this, we use a grid search for all three exploration methods. An overview of the parameters and their search space can be found below in Table 5.2. In offline reinforcement learning, there is no interaction with the environment and therefore exploration is not applicable.

Method	Parameter	Space
ϵ-greedy	epsilon	{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1}
ϵ-greedy decay	epsilon_decay	{0.25, 0.4, 0.55, 0.7, 0.85, 0.999}
softmax	temperature	{0.2, 0.4, 0.6, 0.8, 1, 1.2, 1.4, 1.6, 1.8, 2}

Table 5.2: Parameter space per exploration method

In online training, the agent interacts with the environment for several time steps. This number is set to 15.000 and 100.000 timesteps for the Cartpole and Lunarlander environment, respectively. Training is done every time step on a batch of size 256. The policy evaluation is not done every time step, but after a constant interval. The interval is set to 100 for the Cartpole and 5.000 for the Lunarlander environment. More information about the script used for online training can be found in the appendix (B.1).

5.2 Experiment Logging

For this research, a lot of different experiments are done. It is important to keep this organized. This is done in Python with the use of Sacred, MongoDB and Omniboard. These packages make it possible to log, store and visualize all information of an experiment. The information consists of the parameter settings, outputs, source files and library versions that are used during training. This allows us to reproduce any experiment with re-runs.

	version
python	3.6.7
numpy	1.17.2
torch	1.3.0
sacred	0.8.0
scikit-learn	0.20.0
pandas	0.23.4
hyperopt	0.2.2
horizon	0.1

Table 5.3: Version of python and the python libraries

CHAPTER 6

Results & Conclusion

In this section, the results obtained for all experiments are reported. In Section 6.1 the results are shown for the offline reinforcement learning experiments. Here we show the performance of the three model architectures. In Section 6.2 the results are shown for the online reinforcement learning experiments. This is divided into three different parts. In the first experiment, we show the performance of the exploration methods (6.2.1). Next, we show the performance of the three model architectures (6.2.2). And at last, we show the performance of the warm start (6.2.3). All experiments are done for two different environments, namely the Cartpole and Lunarlander environment.

6.1 Offline Reinforcement Learning

6.1.1 Performance of Model Architectures

Cartpole environment

Figure 6.1 and Table 6.1 show the results for offline reinforcement learning in the Cartpole environment. The results show the models are able to utilize the provided data with random actions and learn better policies. The Dueling DQN converges the fastest and reaches a policy performance, denoted by the policy rewards, of approximately 280 within the first 5 training epochs (Figure 6.1). The other two models need more time to converge to this performance. The solved reward threshold for this environment is defined by finding a policy that obtains 475 rewards on average. The Dueling DQN is the only model that was not able to find such a policy throughout

the whole training period, as this model shows a maximum policy performance of only 385 (Table 6.1). This maximum policy performance is important in offline learning since the agent can learn safely before deployment. Therefore training can be stopped at any point in time when a good policy is found. The DQN and Double DQN show similar scores for both the mean and maximum policy performance.

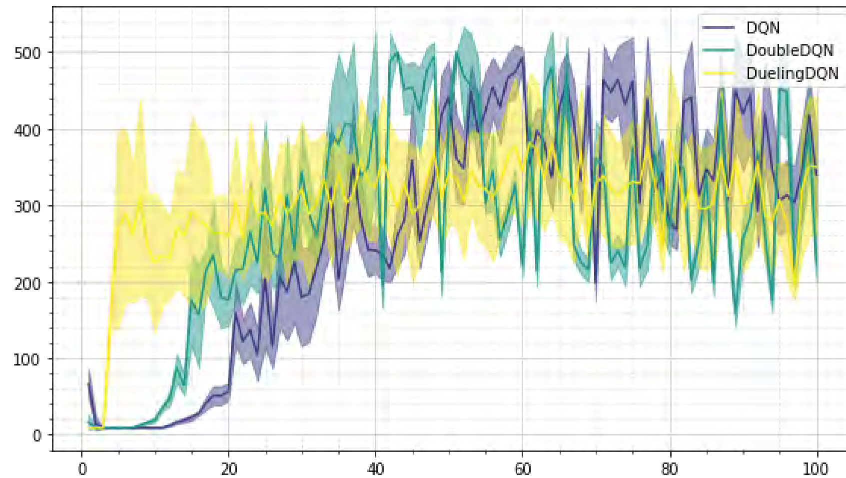


Figure 6.1: This figure shows the policy performance during offline reinforcement learning for the Cartpole environment. The policy performance is represented by the average rewards obtained by running a policy in the environment.

Model	Policy Performance mean	max
DQN	270	492
Double DQN	276	500
Dueling DQN	301	385

Table 6.1: This table shows the mean and maximum policy performance throughout an offline training run for the Cartpole environment.

Lunarlander environment

Figure 6.2 and Table 6.2 show the results for offline reinforcement learning in the Lunarlander environment. The results show the models have more difficulty learning good policies from the provided data with random actions. Not one of the models is able to find a policy that reaches the solved reward threshold of 200. In Figure 6.2 the policy performances of all models can be seen throughout the whole training period. It shows the DQN performs best for most of the training epochs and also reaches the highest mean and maximum policy performance (Table 6.2).

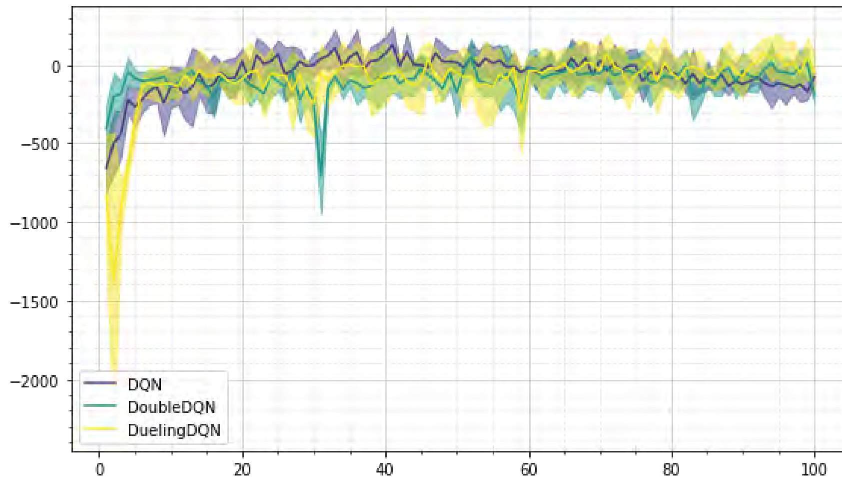


Figure 6.2: This figure shows the policy performance during offline reinforcement learning for the Lunarlander environment. The policy performance is represented by the average rewards obtained by running a policy in the environment.

Model	Policy Performance mean	max
DQN	-58	129
Double DQN	-98	52
Dueling DQN	-96	75

Table 6.2: This table shows the mean and maximum policy performance throughout an offline training run for the Lunarlander environment.

6.1.2 Offline Models

The offline models used for the warm start are chosen based on their performance in the environment. To keep the experiments similar over the different model architectures a performance level is chosen that was reached for all of the models after the training period. For the Cartpole and Lunarlander environment, these models all reach an average reward of approximately 350 and -25 over a thousand episodes, respectively.

6.2 Online Reinforcement Learning

6.2.1 Performance of Exploration Methods

In this section, we show the performance of different exploration methods for the Cartpole and Lunarlander environment. These experiments are only done for online reinforcement learning with a cold start. The results for both environments can be found in Figure 6.3. To allow comparison across the environment the performances in this figure are normalized[1] to deal with the different ranges and distributions of reward in both environments.

Cartpole environment

The performance of the policies, denoted by the normalized policy rewards, found throughout training for the Cartpole environment can be found in Figure 6.3a. It clearly shows the agents have no trouble finding a better policy, since these values lie significantly higher than a random policy (red line). This is the case for all different exploration methods and settings, meaning the actions taken during learning have little effect on this. The Cartpole environment is a relatively easy problem with a small sate and action space which can explain this. Additional, the values lie high enough to state good policies are found. The performance on the system, denoted by the normalized system rewards, seems to be a lot more sensitive to the different exploration settings (Figure 6.3b). The performance on the system decreases a lot when there is too much exploration for both the ϵ-greedy & ϵ-greedy decay methods. This is the case for settings with a high epsilon or epsilon decay rate. It shows the random actions taken during exploration

[1]The normalization uses min-max scaling which scales the values to a fixed range of 0 to 1. It uses the following formula: $\frac{X - X_{min}}{X_{max} - X_{min}}$. For the X_{min} and X_{max} we will use the minimum (random) and maximum (solved reward threshold) possible rewards obtained in the environments.

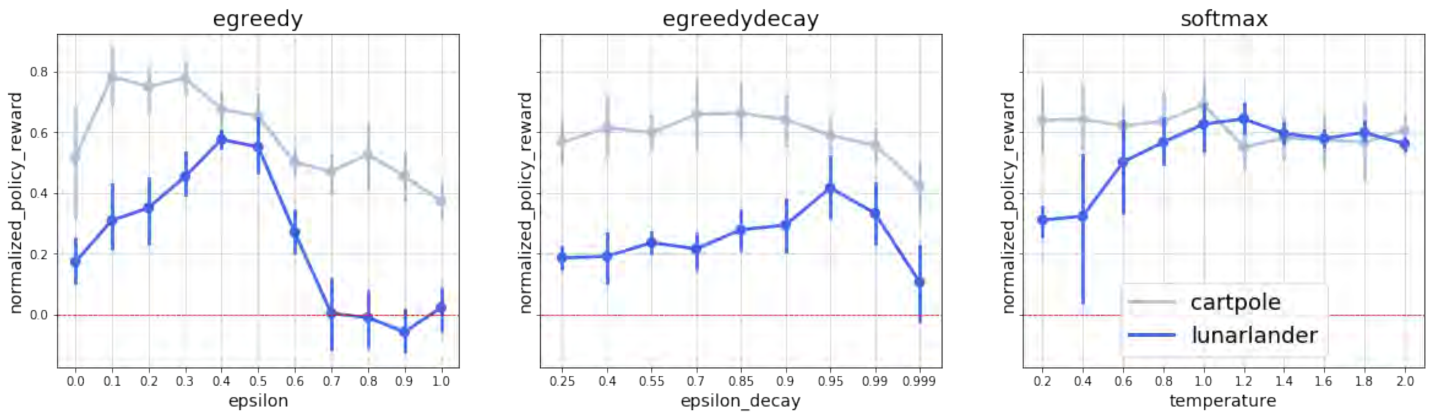
have a drastic negative effect on the rewards obtained from the system. The mean system rewards are highest for an epsilon value of 0.1 or a decay rate of 0.85. The softmax method seems to achieve a good system reward score for all the temperature settings that were tested and is highest at a temperature of 0.2.

Lunarlander environment

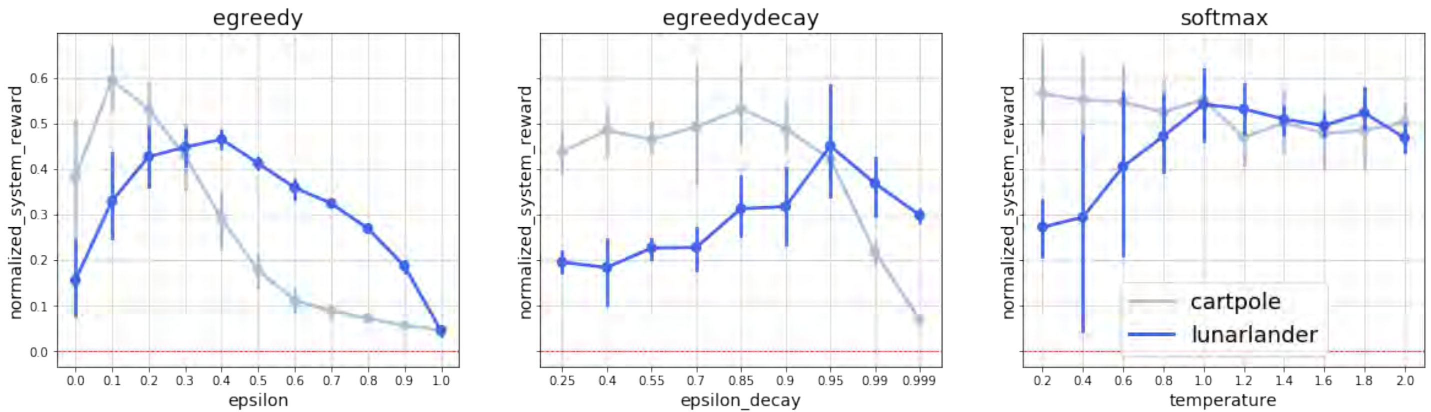
Figure 6.3a also shows the performance of the policies, denoted by the normalized policy rewards, for the Lunarlander environment. It shows the agent has more trouble finding better policies. Here the amount of exploration, defining the actions taken during learning, now has more influence on the policy convergence than for the Cartpole environment. For the ϵ-greedy method with high epsilon values, the agent does not even find a policy that performs better than the random policy (red line). This can, however, be explained by the fact it involves a more complex problem. In this environment, the state space and action space are bigger, which makes it harder to get an understanding of the environment dynamics. Finding a good policy will take more time and effort. The performance on the system in Figure 6.3b, denoted by the normalized system rewards, follows similar behaviour as the policy performance (Figure 6.3a). The performance of the ϵ-greedy & ϵ-greedy decay method on the system is highest for an epsilon of 0.4 or a decay rate of 0.95. This epsilon and decay rate are higher than for the Cartpole environment which again shows this environment needs more exploration. The best scores are reached for the softmax method at a temperature of 1.0.

Across environments

Ideally, we search for an exploration method that performs well for different environments. Therefore we also consider the performance across environments. The results in Figure 6.3b show that choosing an epsilon schema for the ϵ-greedy & ϵ-greedy decay method is very dependent on the environment. Here an epsilon schema defines the value of epsilon throughout training. The softmax method shows not only good performance, but is also more stable across different exploration settings and environments. Therefore, this exploration method will be used for the experiments in the rest of this research.



(a) Policy Rewards



(b) System Rewards

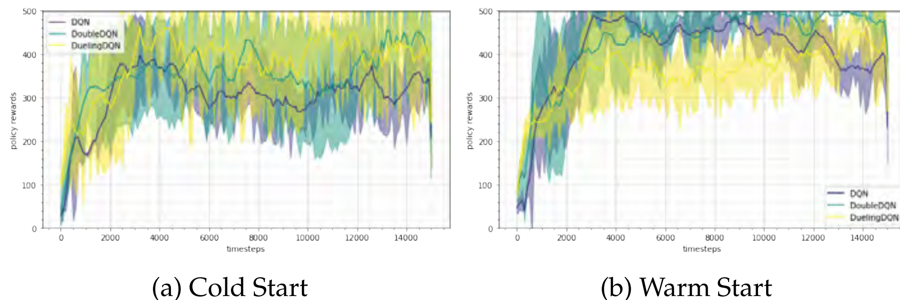
Figure 6.3: This figure shows the results obtained for different exploration methods during online reinforcement learning with a cold start. For each method, the x-axis represents the settings that control the amount of exploration, where a lower value represents less exploration. The policy performance and system performance are both normalized[1]. No distinction is made between the three model architectures (DQN, Double DQN & Dueling DQN). The performance of a random policy is shown with a red line.

6.2.2 Performance of Model Architectures

In this section, we show the performance of different model architectures. The results are shown for online reinforcement with a cold start, where the agent learns from scratch, and a warm start (WS-RL), where an offline model is used as a starting point.

Cartpole environment

The online agent is able to learn a policy that performs well in the Cartpole environment. The solved reward threshold for this environment is defined by finding a policy that obtains 475 rewards on average. Table 6.3 shows that almost all models are able to find a policy that solves the environment. The DQN with a cold start is the only model that does not, but it comes close. At the beginning of training, the models seem to converge at a similar speed (Figure 6.4). However, around 4.000 timesteps the DQN model seems to fall behind for the cold start. Something similar happens for the Dueling DQN at 2.000 timesteps during the warm start. Overall, the training curves are relatively close together. The Dueling DQN and Double DQN seems to perform best for the cold start and warm start, respectively. Additionally, the models stay stable throughout training and do not have sudden drops in performance, meaning they do not lose knowledge from earlier learned policies.



(a) Cold Start

(b) Warm Start

Figure 6.4: This figure shows the policy performance during online reinforcement learning for the Cartpole environment. This plot shows the average over 3 different seeds for each model architecture. A moving average with window size 5 is used for better visibility.

Model	Policy Performance mean	max	System Performance mean
DQN	307	466	240
Double DQN	355	500	311
Dueling DQN	382	500	339

(a) Cold Start

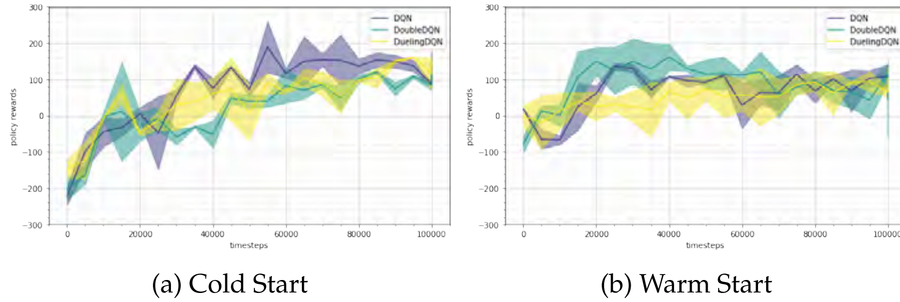
Model	Policy Performance mean	max	System Performance mean
DQN	407	500	296
Double DQN	439	500	341
Dueling DQN	364	500	330

(b) Warm Start

Table 6.3: This table shows the mean and maximum policy performance and mean system performance throughout an online training run for the Cartpole environment. These values show the average over 3 different seeds for each model architecture.

Lunarlander environment

In the Lunarlander environment, the online agent (Table 6.4) is able to find policies that perform significantly better than the offline agent (Table 6.2). Although none of the models are able to solve the environment, which happens at a policy performance of 200, they do show improvement throughout the training period (Figure 6.5). For the cold start, all models come close to this threshold, but for the warm start, the maximum policy performance lies a bit lower (Table 6.4). The Dueling DQN and Double DQN seems to perform best for the cold start and warm start, respectively. In both settings, the performance of the DQN is relatively close to this performance.



(a) Cold Start (b) Warm Start

Figure 6.5: This figure shows the policy performance during online reinforcement learning for the Lunarlander environment. This plot shows the average over 3 different seeds for each model architecture.

	Policy Performance		System Performance
Model	mean	max	mean
DQN	54	179	22
Double DQN	50	180	8
Dueling DQN	68	196	30

(a) Cold Start

	Policy Performance		System Performance
Model	mean	max	mean
DQN	71	143	64
Double DQN	88	178	80
Dueling DQN	52	134	20

(b) Warm Start

Table 6.4: This table shows the mean and maximum policy performance (rewards) and mean system performance (rewards) throughout an online training run for the Lunarlander environment. These values show the average over 3 different seeds for each model architecture.

6.2.3 Performance of Warm Start

In this section, we will discuss the effect of the warm start setup on the performance of an agent. To show the effect of a warm start more clearly we calculate the difference[2] for the policy performance and the system performance. This is done for all the exploration settings of the softmax method. These differences are visualized in Figure 6.6. The performance of cold start and warm start runs throughout training are visualised in Figure C.3 & C.4 in Appendix C.

Cartpole

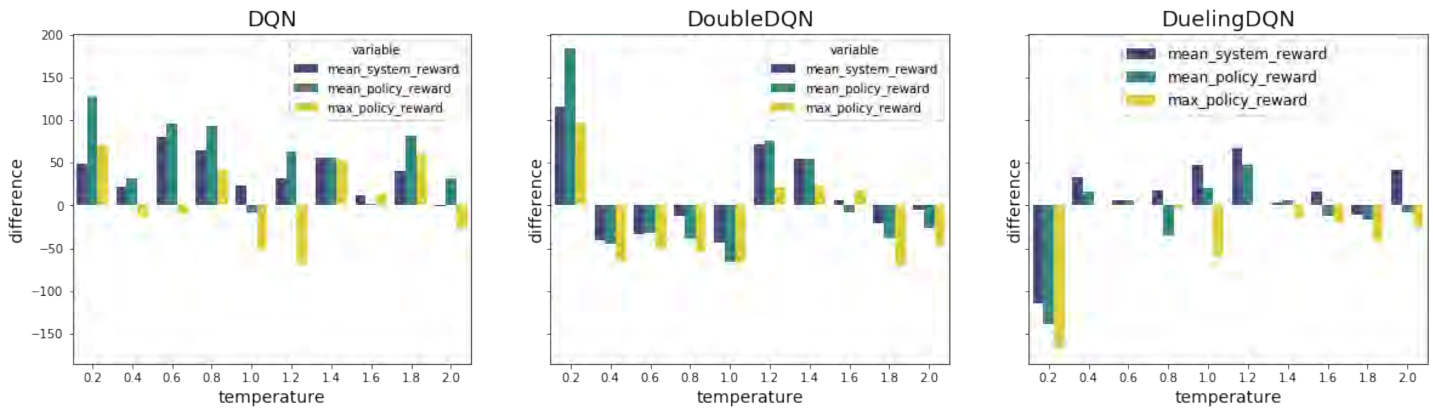
Figure 6.4 and Table 6.3 show the performance of the cold start and warm start for the Cartpole environment with an optimized temperature setting. Here it looks like the warm start improves performance for both the DQN and Double DQN, whereas the Dueling DQN shows comparable results. In Figure 6.6a, we focus on the results per temperature setting. Here we see that for the Cartpole environment the warm start setup is not able to improve results for all models and settings. The DQN seems to benefit most from this setup and generally improves on all three reward metrics. However, this is not the case for the Double DQN, where six out of the ten temperature settings show negative difference meaning a decrease in performance. These results show that in a relatively simple environment, where an agent is already able to quickly find better policies without any help (Figure 6.4a), a warm start does not necessarily improve performance.

Lunarlander

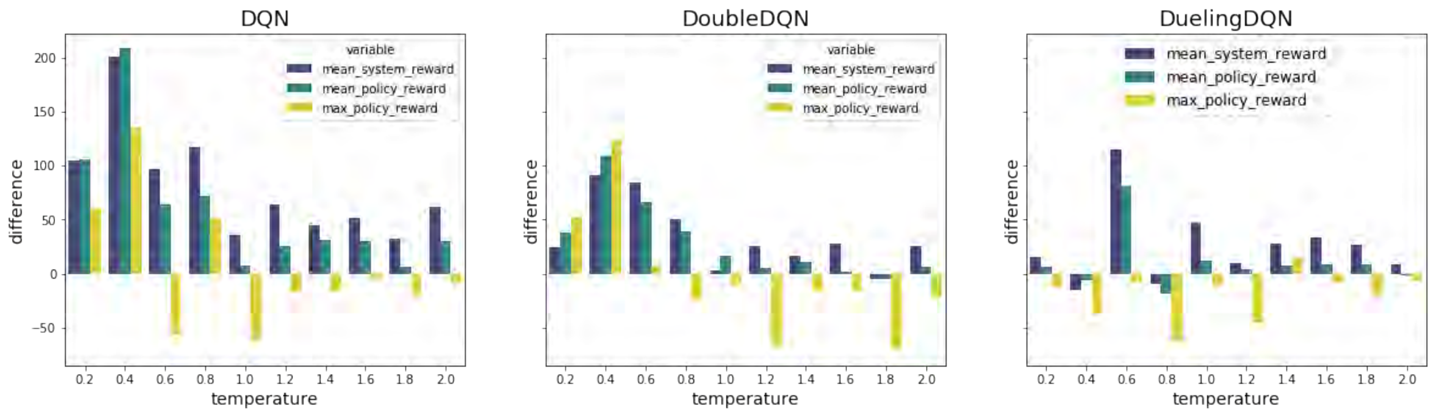
Figure 6.6b shows the warm start for the Lunarlander environment clearly has a positive impact on the mean policy and system reward for the different temperature settings. Again, like the Cartpole environment, the DQN seems to benefit most from this warm start. However, for the maximum policy rewards, this is not the case. In most settings, the warm start seems to have little impact and for some even a negative impact on this score. This can also be seen in Figure C.4. The warm start runs show better performance at the beginning of training, but most of the cold start runs are able to outperform the warm start in the long run. This indicates a harder

[2]The differences are calculated by subtracting the score of a certain metric for a cold start from the warm start score.

environment, where converging to an optimal policy is more difficult, can benefit from a warm start considering the overall performance. However, it does not necessarily converge to a better policy. In some cases, the cold start even converges to better policies than a warm start.



(a) Cartpole



(b) Lunarlander

Figure 6.6: This figure shows the difference[2] in performances of online reinforcement learning with a cold start and warm start for different softmax parameter settings. These values show the average over 3 different seeds for each model architecture.

CHAPTER 7

Discussion

In this research, we investigate the performance of online reinforcement learning performed directly on the system. In this setting, all actions taken during learning have a direct impact. This means exploration of the environment should be done carefully. Therefor multiple exploration methods and parameter settings were compared. We discovered that the amount of exploration needed is dependent on the environment we are dealing with. This is influenced by the state and action space and the complexity of the environment. We found that the softmax method was able to perform best. This is due to the fact it does not take random actions during exploration, but assigns action probabilities based on the expected return learned during training. This way of exploration is sufficient to find a good policy and also has the least negative impact on the system. This method also proves to be stable across the environments and different temperature settings.

We also investigated the possibility of improving online reinforcement learning with a proposed warm start. This warm start uses offline reinforcement learning to utilize an available dataset and provide knowledge for the online agent. Again, the results show this effect is dependent on the environment. It can be concluded that a difficult environment like the Lunarlander environment, where an online reinforcement learning has more difficulty in finding an optimal policy, benefits most from the warm start. It reduces the time needed to find a decent policy and therefore improves the overall performance. It, however, does not necessarily converge to a better policy.

To make stronger claims about the results a more extensive research has to be done. Currently, the experiments are done for only three different

seeds. These different seeds have an effect on the random factors of the algorithms and environments during training. Using more random seeds would give a better understanding of the true distribution. Also, the experiments are only performed for two different environments. To get a better idea of the effect of exploration and warm starts it is important to look at performances across more environments. And at last, for the warm start experiments, only one trained model is selected to be used as a starting point. It might be interesting to compare multiple models with different performances. Due to time limitations, this is not done in this research.

7.1 Future Work

In this research, the focus is more on online reinforcement learning instead of offline reinforcement learning. However, researching the performance of offline reinforcement learning more extensively would be very interesting. Most importantly the offline evaluation. If it would be possible to learn and evaluate offline, it would ensure the models are thoroughly tested before deploying them at scale. Horizon provides these off-policy methods and counterfactual policy evaluation (CPE) to estimate what the reinforcement learning would have done if it were making those past decisions. However, during this research, some challenges occurred to get this working and therefore it is not further discussed. This would, however, be interesting as future work.

It is also important to note that currently the dataset used for offline reinforcement learning only contains random actions. This is not realistic when considering a real-world reinforcement learning problem where existing data is available from the real system. In a real-world problem previous interactions are not taken randomly, but often follow a "human" policy. This policy would certainly be better than taking random actions, especially when bad actions would have a drastic impact on the system. Using a more realistic dataset with limited exploration might negatively affect the results of offline reinforcement learning. It would be interesting to see how this would affect the performance of offline learning.

Bibliography

[1] J Andrew Bagnell and Jeff G Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No. 01CH37164)*, volume 2, pages 1615–1620. IEEE, 2001.

[2] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016.

[4] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *International conference on computers and games*, pages 72–83. Springer, 2006.

[5] Jason Gauci, Edoardo Conti, Yitao Liang, Kittipat Virochsiri, Yuchen He, Zachary Kaden, Vivek Narayanan, Xiaohui Ye, Zhengxing Chen, and Scott Fujimoto. Horizon: Facebook's open source applied reinforcement learning platform. *arXiv preprint arXiv:1811.00260*, 2018.

[6] Jason Gauci, Edoardo Conti, and Kittipat Virochsiri. Horizon: The first open source reinforcement learning platform for large-scale products and services. *AI Research, ML Applications*, 2018.

[7] Wade Genders and Saiedeh Razavi. Using a deep reinforcement learning agent for traffic signal control. *arXiv preprint arXiv:1611.01142*, 2016.

[8] OpenAI Gym. The cartpole-v1 environment. https://gym.openai.com/envs/CartPole-v1/.

[9] OpenAI Gym. The lunarlander-v2 environment. https://gym.openai.com/envs/LunarLander-v2/.

[10] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *arXiv preprint arXiv:1801.01290*, 2018.

[11] Hado V Hasselt. Double q-learning. In *Advances in neural information processing systems*, pages 2613–2621, 2010.

[12] Junqi Jin, Chengru Song, Han Li, Kun Gai, Jun Wang, and Weinan Zhang. Real-time bidding with multi-agent reinforcement learning in display advertising. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, pages 2193–2201, 2018.

[13] Alistair EW Johnson, Tom J Pollard, Lu Shen, H Lehman Li-wei, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G Mark. Mimic-iii, a freely accessible critical care database. *Scientific data*, 3:160035, 2016.

[14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[15] Jens Kober, Erhan Oztop, and Jan Peters. Reinforcement learning to adjust robot movements to new situations. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

[16] Jens Kober and Jan R Peters. Policy search for motor primitives in robotics. In *Advances in neural information processing systems*, pages 849–856, 2009.

[17] Nate Kohl and Peter Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *IEEE International Conference on Robotics and Automation, 2004. Proceedings. ICRA'04. 2004*, volume 3, pages 2619–2624. IEEE, 2004.

[18] Matthieu Komorowski, Leo A Celi, Omar Badawi, Anthony C Gordon, and A Aldo Faisal. The artificial intelligence clinician learns optimal treatment strategies for sepsis in intensive care. *Nature medicine*, 24(11):1716–1720, 2018.

[19] Virendra Singh Kushwah and Aruna Bajpai. Machine learning and its algorithms: A research. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, 8(12S2), 2019.

[20] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *nature*, 521(7553):436–444, 2015.

[21] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[22] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.

[23] Feng Liu, Ruiming Tang, Xutao Li, Weinan Zhang, Yunming Ye, Haokun Chen, Huifeng Guo, and Yuzhou Zhang. Deep reinforcement learning based recommendation with explicit user-item interactions modeling. *arXiv preprint arXiv:1810.12027*, 2018.

[24] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56, 2016.

[25] Sadayoshi Mikami and Yukinori Kakazu. Genetic reinforcement learning for cooperative traffic signal control. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 223–228. IEEE, 1994.

[26] Volodymyr Mnih. Playing atari with deep reinforcement learning.

[27] Andrew Y Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. In *Experimental robotics IX*, pages 363–372. Springer, 2006.

[28] Jakub Pachocki, Greg Brockman, Jonathan Raiman, Susan Zhang, Henrique Pondé, Jie Tang, Filip Wolski, Christy Dennison, Rafal Jozefowicz, Przemyslaw Debiak, et al. Openai five, 2019. *URL https://openai.com/blog/openai-five-defeats-dota-2-world-champions/*.

[29] Elena Pashenkova, Irina Rish, and Rina Dechter. Value iteration and policy iteration algorithms for markov decision problem. In *AAAI'96: Workshop on Structural Issues in Planning and Temporal Reasoning*. Citeseer, 1996.

[30] Tom J Pollard, Alistair EW Johnson, Jesse D Raffa, Leo A Celi, Roger G Mark, and Omar Badawi. The eicu collaborative research database, a freely available multi-center database for critical care research. *Scientific data*, 5:180178, 2018.

[31] Reuven Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability*, 1(2):127–190, 1999.

[32] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[33] Arthur L Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of research and development*, 3(3):210–229, 1959.

[34] Stefan Schaal. Is imitation learning the route to humanoid robots? *Trends in cognitive sciences*, 3(6):233–242, 1999.

[35] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[36] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[37] Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

[38] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2017.

[39] Gerald Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[40] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

[41] Martijn van Otterlo. *Reinforcement learning: State-of-the-Art*. Springer Berlin Heidelberg, 2012.

[42] Oriol Vinyals, Igor Babuschkin, Wojciech M Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H Choi, Richard Powell, Timo Ewalds, Petko Georgiev, et al. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, 2019.

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

[44] Christopher John Cornish Hellaby Watkins. Learning from delayed rewards. 1989.

[45] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

[46] Rusheng Zhang, Akihiro Ishikawa, Wenli Wang, Benjamin Striner, and Ozan Tonguz. Intelligent traffic signal control: Using reinforcement learning with partial detection. *arXiv preprint arXiv:1807.01628*, 2018.

[47] Guanjie Zheng, Fuzheng Zhang, Zihan Zheng, Yang Xiang, Nicholas Jing Yuan, Xing Xie, and Zhenhui Li. Drn: A deep reinforcement learning framework for news recommendation. In *Proceedings of the 2018 World Wide Web Conference*, pages 167–176, 2018.

APPENDIX A

Implementation Details

A.1 Amazon Web Services (AWS)

All experiments are run on an Elastic Compute Cloud (EC2) instance, which is a virtual server that can be used to run applications in AWS. When setting up an EC2 instance, you can custom-configure CPU, storage, memory, and networking resources. Here a p2.xlarge EC2 instance was used. This instance was linked to an S3 bucket to store data and experiment details.

A.2 Packages & Libraries

	version
python	3.6.7
numpy	1.17.2
torch	1.3.0
sacred	0.8.0
scikit-learn	0.20.0
pandas	0.23.4
hyperopt	0.2.2
horizon	0.1

Table A.1: Version of python and the python libraries

A.3 Experiment Logging

The experiment logging was done with the use of Sacred, MongoDB, and Omniboard. These packages make it possible to log, store, and visualize all information of an experiment.

Omniboard (OC_cartpole)

Status: 7 selected Filters: Column Name == Enter Value... Add Filter

75 experiments

Id	Experiment Name	EX NAME	Start Time	Duration	Result ↑	Hostname
74	OC	cartpole_OC_DuelingDQN_softmax	2020-05-16T18:47:21	9m 3.3s	60125.05	AMSsdegeus-C02Z14XMLVCG
36	OC	cartpole_OC_DoubleDQN_softmax	2020-05-16T16:54:16	5m 16.4s	54617	AMSsdegeus-C02Z14XMLVCG
72	OC	cartpole_OC_DuelingDQN_softmax	2020-05-16T18:33:16	8m 11.4s	52143.84999999999	AMSsdegeus-C02Z14XMLVCG

Figure A.1: Overview of the experiment runs in Omniboard.

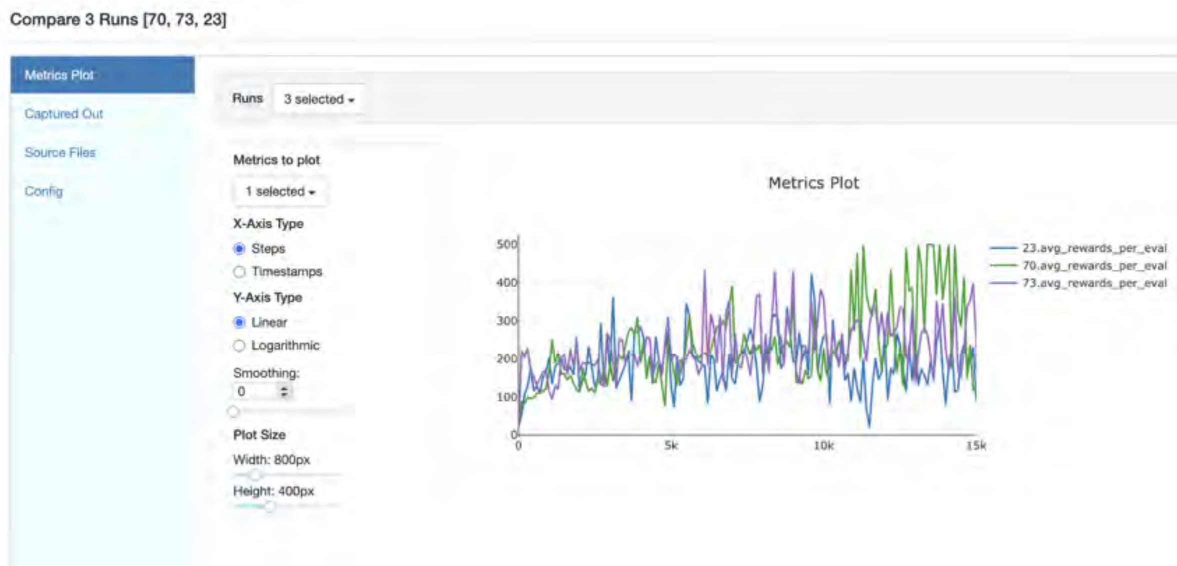


Figure A.2: Comparison of multiple experiment runs with details about the metrics, captured output, source files and configuration.

APPENDIX B

Horizon

In this research, we will use Horizon, an open-source end-to-end applied Reinforcement Learning platform for production, which was introduced by Facebook November last year [5]. Horizon addresses the unique challenges posed by building and deploying RL systems at scale and provides for both offline and online learning. Horizon also takes into account issues specific to production environments, including feature normalisation, distributed training, large-scale deployment and serving, as well as data sets with thousands of varying feature types and distributions and high-dimensional discrete and continuous-action spaces.

B.1 Training Scripts

B.1.1 Offline Reinforcement Learning

In offline reinforcement learning, there is no interaction between the agent and the environment, therefore the data collection and policy update steps are decoupled. The basic idea of offline reinforcement learning can be seen in algorithm 5. First, the replay buffer has to be filled with pre-generated samples. Then training is done by looping over batches of samples from the replay buffer. In a single epoch, all samples are used for training. After a number of epochs, the model will stop training. The model is evaluated after every epoch.

Algorithm 5: Pseudo Code: train_gym_offline_rl

```
1: procedure TRAIN_GYM_OFFLINE_RL(ENV, TRAINER, EPOCHS,...)
2:     initialize replay buffer 𝒟
3:     for all epochs e do
4:         for all batches in replay buffer 𝒟 do
5:             train on batch                                   ▷ Training
6:
7:         calculate policy performance (on-policy or off-policy)▷ Evaluation
```

B.1.2 Online Reinforcement Learning

In online reinforcement learning the agent is able to interact with the environment creating its own samples. The model alternates between the exploring phase and the learning phase multiple times. Every time step a new experience will be added to the replay buffer. The model will train every time step on a batch taken from the replay buffer. The policy evaluation is not done every time step, but after a certain number of time steps. How often is dependent on the environment. It is possible to give the agent two different inputs. One is a replay buffer filled with samples instead of being empty. The other is a previously trained model that will be used as a starting point for the agent. These inputs are both optional. Algorithm 6 shows the pseudo-code of online RL.

Algorithm 6: Pseudo Code: train_gym_online_rl

1: **procedure** TRAIN_GYM_ONLINE_RL(.....)
2: initialize environment *env*, replay buffer $\mathcal{D}^{replay}$, *epsilon*
3:
4: **for** all episodes e **do**
5: initialize state s_0
6:
7: **for** each step t in episode **do**
8: $a_t \leftarrow$ select action based on policy ▷ Exploration policy
9: take action a_t, and observe reward r_t and next state s_{t+1}
10: store experience $e_t = (s_t, a_t, r_t, s_{t+1})$ in $\mathcal{D}^{replay}$ ▷ Experience
11:
12: **if** *total_timesteps* % *train_every_ts* = 0 **then**
13: get *batch* from $\mathcal{D}^{replay}$
14: train on *batch* ▷ Policy Update
15:
16: **if** *total_timesteps* % *test_every_ts* = 0 **then**
17: calculate policy performance ▷ Policy Evaluation
18:
19:
20: decay *epsilon* dependent on exploration method
21:

B.2 Hyperparameters

Hyperparameter	Value		Description
	Cartpole	Lunarlander	
max replay memory size	10000	50000	Updates are sampled from this number of most recent frames (replay buffer size).
replay start size	256	256	The start policy is run for this number of frames before learning starts and the resulting experiences are used to populate the replay buffer.
train every ts	1	1	The number of timesteps between two times of training.
train after ts	1	1	The number of timesteps after which the model will start training.
test every ts	100	5000	The number of timesteps in between two times of evaluation.
test after ts	1	1	The number of timesteps after which the model will start to evaluate.
total ts	15000	100000	The total number of timesteps of learning.
minimum epsilon	0	0	The lowest possible number of epsilon that can be used for action selection. This is used if epsilon is decayed during learning.
gamma	0.99	0.99	Discount factor gamma used in the Q-learning update.
target update rate	0.2	0.001	The frequency with which the target network is updated.
minibatch size	256	256	Number of training cases that are used to compute policy update.
learning rate	0.01	0.001	The learning rate used by the model.
layers	[-1, 128, -1]	[-1, 128, 64, -1]	The layer sizes of the neural network that is used for training.
activations	["relu", "linear"]	["relu", "relu", "linear"]	The activation function that is uses in the neural network layers.

Table B.1: The hyperparameters that were used throughout training with their value and description. These values are chosen based on recommendations by Horizon or other literature.

APPENDIX C

Results

C.1 Offline Reinforcement Learning

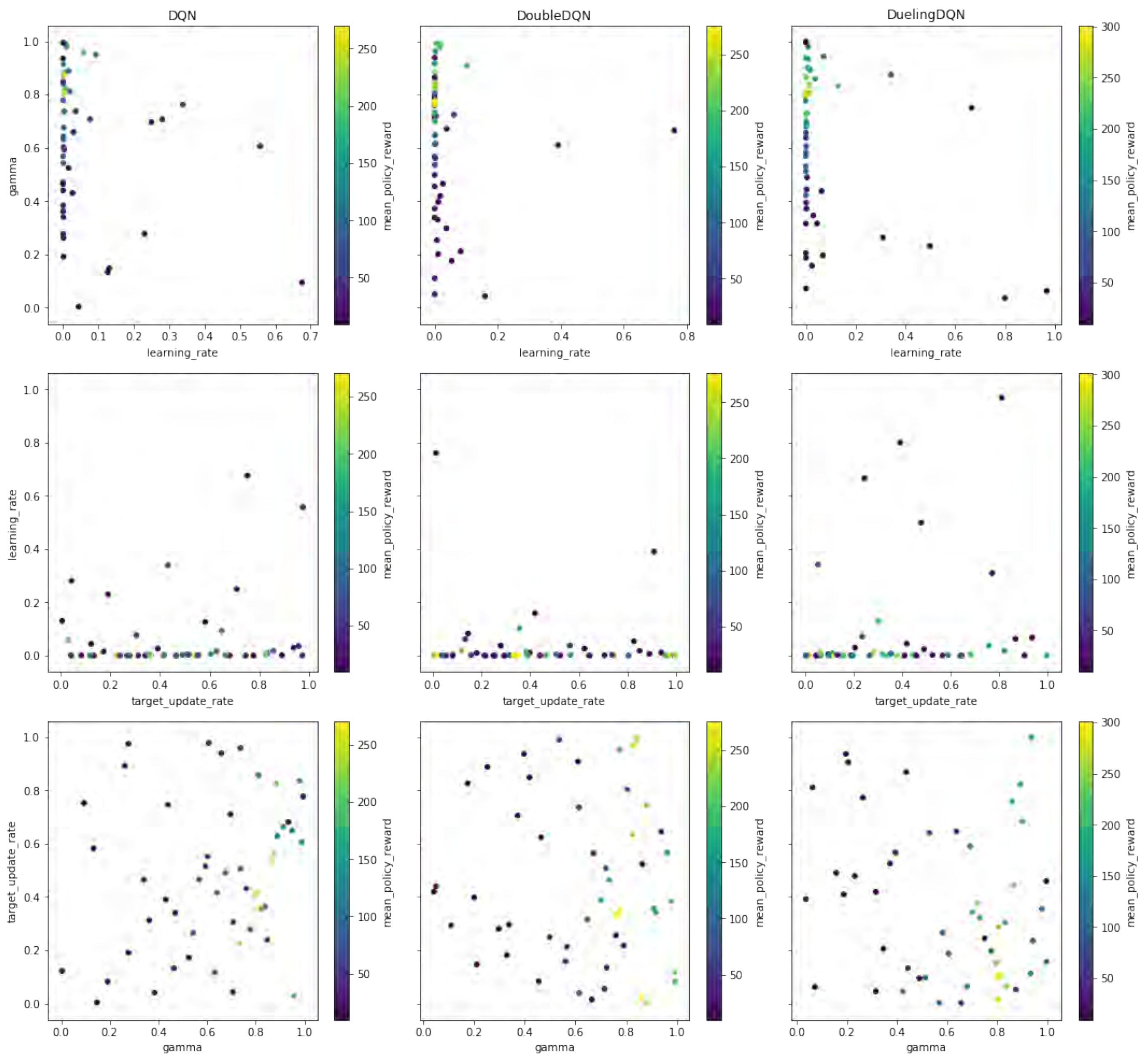


Figure C.1: Hyperparameter optimization for the Cartpole environment during offline reinforcement learning.

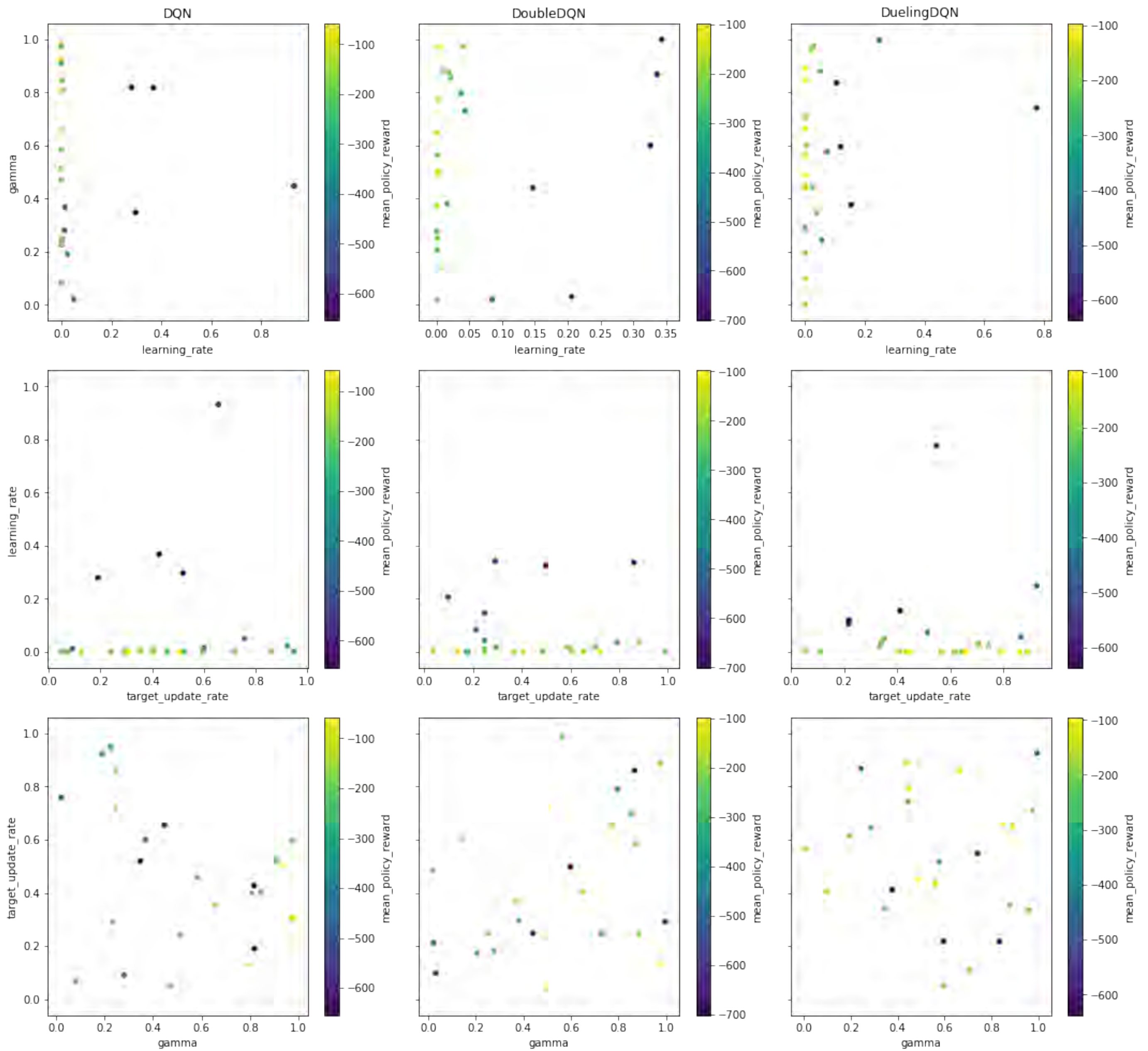


Figure C.2: Hyperparameter optimization for the Lunarlander environment during offline reinforcement learning.

C.2 Online Reinforcement Learning

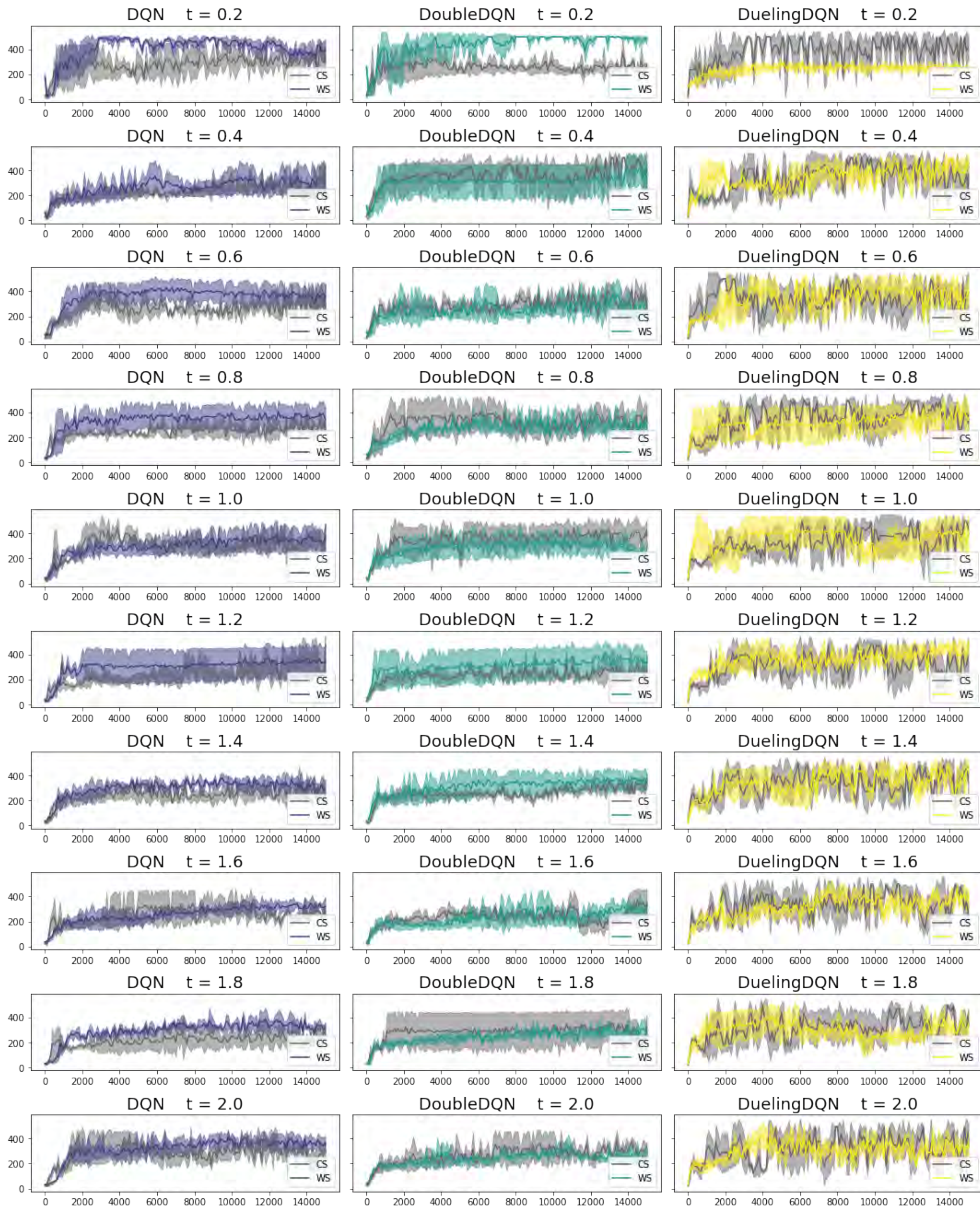


Figure C.3: Policy rewards throughout training runs for Cartpole environment.

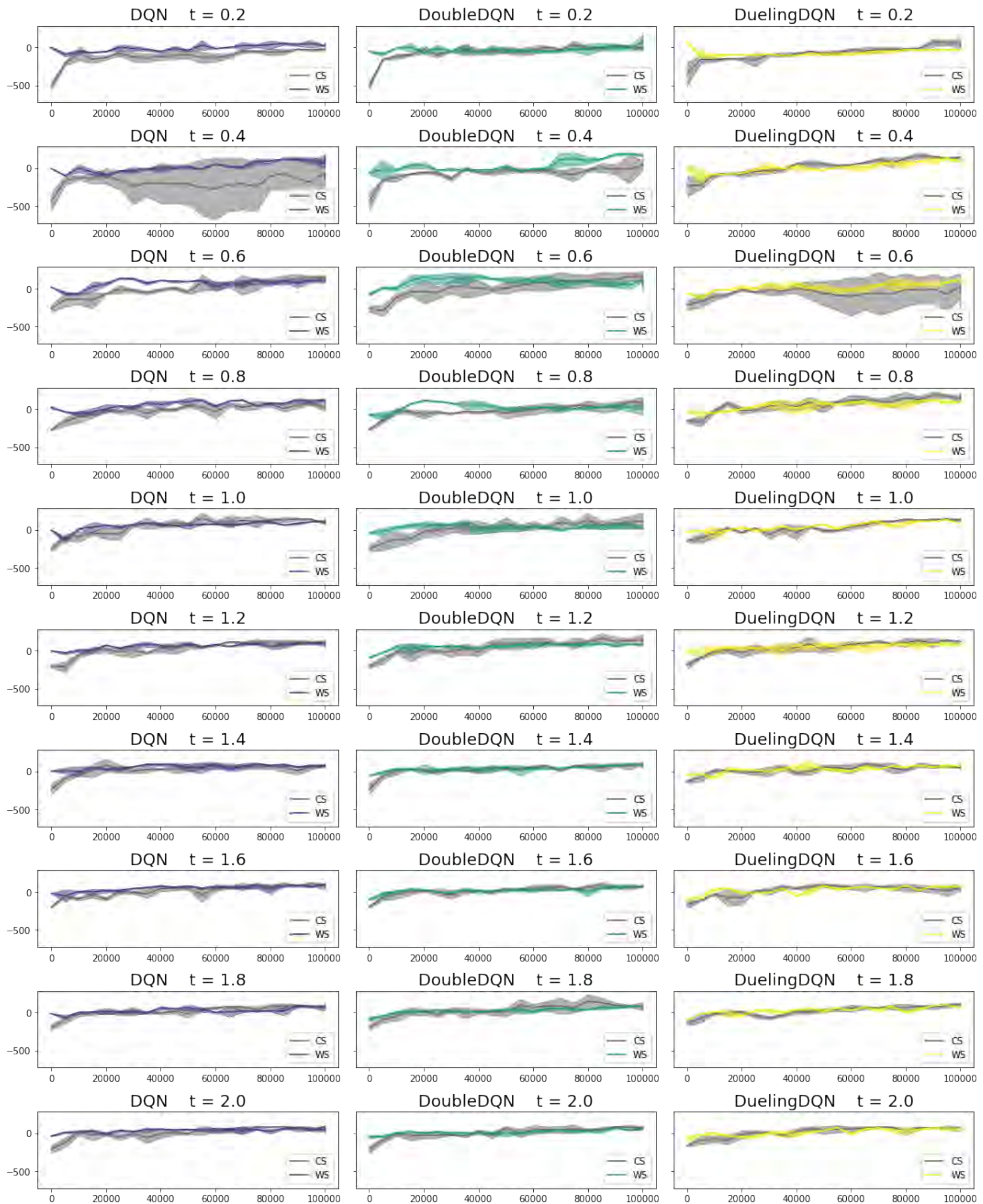


Figure C.4: Policy rewards throughout training runs for Lunarlander environment.