Novel Reinforcement Learning Methods for Robotics

Manuel Luís Pereira de Sousa Esménio

Thesis to obtain the Master of Science Degree in

Aerospace Engineering

Supervisors:  Prof. Pedro Manuel Urbano de Almeida Lima
             Dr. Tiago Santos Veiga

Examination Committee
Chairperson: Prof. Paulo Jorge Coelhoo Ramalho Oliveira
Supervisor: Dr. Tiago Santos Veiga
Member of the Committee: Prof. Manuel Fernando Cabido Peres Lopes

October 2020
Acknowledgements

I would like to thank Instituto Superior Técnico for providing me a challenging environment and a comfortable home for seven years, Pedro Lima for all his patience during this process and Tiago Veiga for his helpful assistance in debugging this work.

Also to express my appreciation towards my parents for putting enough pressure on me when it was time to finish, my sister for the important advice she gave and my brother for his company.

A final thank you note goes to my friends who made me want to come out and play.
Abstract

This work goes over the Markov decision process and how it applies to Reinforcement Learning, using two optimization conditions known as Bellman Equation and Gradient Descent. Those conditions are used to derive historic algorithms such as Q-Learning and Reward Increment non Negative Factor Offset Reinforcement. By combining those algorithms with Function Approximation and Actor-Critic one finds solutions that connect with Deep Learning. A gentle introduction to neural networks is followed by a description of Deep Q-Network, Deep Deterministic Policy Gradient and Advantage Actor Critic algorithms. These are tested on a set of diversified environments which raise some practical challenges faced when implementing agents. Additional layers addressing these issues are itemized along the environment descriptions. Hyper-parameter tables are used to describe the agents implemented on each environment and the experiments they were made to perform. The results obtained will then compared to evaluate the effects of the different hyper-parameter values being applied.

Keywords: Markov decision process, Reinforcement learning, Function approximation, Actor-Critic
Resumo

Este trabalho cobre o processo de decisão de Markov e as aplicações deste em aprendizagem por reforço, usando duas condições de otimização conhecidas como a equação de Bellman e o método do gradiente descendente. Estas condições são utilizadas para derivar algoritmos históricos como Q-Learning e Reward Increment non Negative Factor Offset Reinforcement. Combinar esses algoritmos com aproximação de funções e métodos de ator-critic gera soluções ligadas a aprendizagem profunda. Uma introdução a redes neuronais é seguida por uma descrição dos algoritmos como Deep-Q-Learning, Deep Deterministic Policy Gradient e Advantage Actor Critic. Estes são testados num conjunto diversificado de ambientes, que levantam desafios práticos tipicamente encontrados com a implementação dos algoritmos descritos. Camadas adicionais que permitiram minimizar as condições descritas são listadas com a descrição de cada ambiente. Tabelas de hiperparâmetros são usadas para descrever os agentes implementados em cada ambiente e as experiências efetuadas com os mesmos. Os resultados obtidos são comparados para avaliar o efeito dos valores usados como hiper-parâmetros.

Palavras-chave: Processo de decisão de Markov; Aprendizagem por reforço; Aproximação de funções; Ator-Critic
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acknowledgements</td>
<td>III</td>
</tr>
<tr>
<td>Abstract</td>
<td>V</td>
</tr>
<tr>
<td>Resumo</td>
<td>VII</td>
</tr>
<tr>
<td>List of Tables</td>
<td>XI</td>
</tr>
<tr>
<td>List of Figures</td>
<td>XIII</td>
</tr>
<tr>
<td>Nomenclature</td>
<td>XV</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2 Related work</td>
<td>2</td>
</tr>
<tr>
<td>1.3 Goals and contribution</td>
<td>3</td>
</tr>
<tr>
<td>1.4 Overview</td>
<td>3</td>
</tr>
<tr>
<td>2 Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Markov Decision Process</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Bellman’s equation and Principle of Optimality</td>
<td>6</td>
</tr>
<tr>
<td>2.3 Value based algorithms</td>
<td>7</td>
</tr>
<tr>
<td>2.4 Exploration/Exploitation trade-off</td>
<td>9</td>
</tr>
<tr>
<td>2.5 Gradient Descent</td>
<td>9</td>
</tr>
<tr>
<td>2.6 Policy based algorithms</td>
<td>10</td>
</tr>
<tr>
<td>2.7 Function approximation</td>
<td>12</td>
</tr>
<tr>
<td>2.7.1 Linear features in value based approximations</td>
<td>12</td>
</tr>
<tr>
<td>2.7.2 Linear features in policy based</td>
<td>13</td>
</tr>
<tr>
<td>2.8 Neural Networks</td>
<td>15</td>
</tr>
<tr>
<td>2.9 Deep-Q-Network</td>
<td>17</td>
</tr>
<tr>
<td>2.10 Actor-Critic</td>
<td>18</td>
</tr>
<tr>
<td>2.10.1 Deep Deterministic Policy Gradient</td>
<td>18</td>
</tr>
<tr>
<td>2.10.2 Advantage Actor-Critic</td>
<td>20</td>
</tr>
<tr>
<td>3 Implementation</td>
<td>23</td>
</tr>
<tr>
<td>3.1 Implemented algorithms</td>
<td>23</td>
</tr>
<tr>
<td>3.2 Experimental setup</td>
<td>23</td>
</tr>
</tbody>
</table>
3.3 Case Studies ................................................. 24
  3.3.1 Lunar Lander ........................................... 24
  3.3.2 Space Invaders ......................................... 25
  3.3.3 Bipedal Walker ......................................... 27
  3.3.4 Duckietown ............................................. 28
3.4 Hyper parameter tuning .................................... 30

4 Experiments .................................................. 33
  4.1 Lunar Lander ................................................ 33
    4.1.1 Experimental description ............................. 33
    4.1.2 Results ................................................ 34
  4.2 Space Invaders ............................................. 37
    4.2.1 Experimental description ............................. 37
    4.2.2 Results ................................................ 38
  4.3 Bipedal Walker ............................................. 39
    4.3.1 Experimental description ............................. 39
    4.3.2 Results ................................................ 41
  4.4 Duckietown ................................................ 44
    4.4.1 Experimental description ............................. 44
    4.4.2 Results ................................................ 47
  4.5 Discussion ................................................ 52

5 Conclusions ................................................ 55
  5.1 Future work ............................................... 55

Bibliography .................................................. 57
List of Tables

3.1 Environment selection and criteria ................................. 24
3.2 Environment description for Lunar Lander .......................... 25
3.3 Environment description for Space Invaders ........................ 26
3.4 Environment description for Bipedal Walker ........................ 28
3.5 Environment description for Duckietown .......................... 29
3.6 RL hyper-parameters in implemented agents ........................ 31
3.7 NN hyper-parameters in implemented agents ........................ 31
3.8 Image-based hyper-parameters in implemented agents .............. 32
4.1 Hyper-parameters for agent 1 in Lunar Lander ...................... 33
4.2 Hyper-parameters for agents 2, 3, 4 in Lunar Lander ............... 34
4.3 Hyper-parameters for agent 1 in Space Invaders .................... 38
4.4 Search space for hyper-parameter optimization in Bipedal Walker .... 40
4.5 Hyper-parameters for agents 1 and 2 in Bipedal Walker ........... 40
4.6 Parameter optimization for agents 3 and 4 in Bipedal Walker ........ 41
4.7 Hyper-parameters for agents 1,2,3,4,5 and 6 in Duckietown ........ 45
4.8 Magnitudes for reward function of agents 1,2,3, 4, 5 and 6 in Duckietown .... 46
4.9 Generic properties for agents, environments and their interaction .... 53
List of Figures

2.1 Agent-environment interaction loop .............................................. 5
2.2 Neuron linear model .............................................................. 15
2.3 Learning cycle for Neural Networks ............................................. 16
2.4 DQN block diagram ............................................................... 18
2.5 DDPG block diagram .............................................................. 20

3.1 Screen shot for Lunar Lander ..................................................... 25
3.2 Screen shot for Space Invaders ................................................ 26
3.3 Pre-processed screen shot for Space Invaders ............................. 27
3.4 Screen shot for Bipedal Walker ................................................ 28
3.5 Screen shot for Duckietown ..................................................... 29
3.6 Pre-processed screen shot for Duckietown ................................. 30

4.1 Neural network scheme for agent 1 in Lunar Lander ..................... 34
4.2 Neural network scheme for agents 2, 3, 4 in Lunar Lander ............... 34
4.3 Results for agent 1 in Lunar Lander .......................................... 35
4.4 Results for agent 2 in Lunar Lander .......................................... 36
4.5 Results for agent 3 in Lunar Lander .......................................... 36
4.6 Results for agent 4 in Lunar Lander .......................................... 37
4.7 Neural network scheme for agent 1 in Space Invaders .................. 37
4.8 Results for agent 1 in Space Invaders ....................................... 39
4.9 Neural network scheme for agent 1 and 2 in Bipedal Walker ........... 41
4.10 Results for agent 1 in Bipedal Walker ...................................... 42
4.11 Results for agent 2 in Bipedal Walker ...................................... 42
4.12 Results for agent 3 in Bipedal Walker ...................................... 43
4.13 Results for agent 4 in Bipedal Walker ...................................... 44
4.14 Duckietown map: loop ......................................................... 44
4.15 Neural network scheme for agents 1, 2, 3, 4 and 5 in Duckietown ...... 46
4.16 Neural network scheme for agent 6 in Duckietown ...................... 46
4.17 Results for agent 1 in Duckietown:Loop static .......................... 47
4.18 Results for agent 2 in Duckietown:Loop static .......................... 48
4.19 Results for agent 3 in Duckietown: Loop static ........................................ 48
4.20 Results for agent 4 in Duckietown: Loop static ........................................ 49
4.21 Results for agent 4 in Duckietown: Loop dynamic ................................. 50
4.22 Results for agent 5 in Duckietown: Loop dynamic ................................. 50
4.23 Results for agent 6 in Duckietown: Loop dynamic ................................. 51
4.24 Results for agent 6 in Duckietown: Random map ................................. 51
Nomenclature

$S$ State space
$A$ Action space
$s$ State
$a$ Action
$r$ Scalar reward
$s'$ Next state
$t(s, a, r, s')$ One-step transition
$P(s' | s, a)$ Transition function
$P(r | s, a)$ Reward function
$\pi(s)$ Policy
$\pi_\theta(s)$ Parametrized policy
$V(s)$ State value
$V^\pi(s)$ Policy value
$Q(s, a)$ Q-value
$Q^\pi(s, a)$ Q-value under policy
$\tilde{Q}(s, a)$ Estimated Q-value
$J_\theta$ Gradient descent objective function
$d^\pi(s)$ Policy state distribution
$H_\theta(s, a)$ Score function
$Q_W(s, a)$ Q-value with function approximation
$\delta_W$ Temporal difference error
$A(s, a)$ Advantage function
Chapter 1

Introduction

1.1 Motivation

The motivation for this work is to create a bridge connecting the prevalent sides of theory and practice in Reinforcement Learning (RL), by understanding and addressing the issues at hand, being that some of them have been documented since the formalization of dynamic programming [1] and can be stated as curses [2]:

- **Curse of dimensionality** - As the dimensions required to fully describe the environment as a set of states and actions grow, so does the time and computation power required to solve it. This can make even discrete problems intractable, and was one of the issues addressed by Bellman in [1]. Nowadays this issue is often solved by redefining those sets to reduce their dimensionality. Neural networks (NNs) [3] are currently seen as a solution for this issue and have a vital role in successful RL implementations and algorithms.

- **Curse of goal specification** - In RL optimal behaviours are implicitly described by a scalar signal known as reward, as the agent attempts to maximize the expected cumulative reward obtained from the environment. Designing an efficient reward function can be surprisingly difficult [4] as it requires a detailed translation of the optimal behaviour into a random collection of transitions the agent will learn from, while also preventing the appearance of sub-optimal exploits which RL agents are infamously prepared to find.

- **Curse of real-world sampling** - When agents have to interact with real-world environments, they become susceptible to all the problems associated with these scenarios: robot hardware costs money and requires careful maintenance; local conditions (lighting, temperature, wear, among others) may become non-negligible variations which affect training [5]; most tasks require some form of human supervision and can not be accelerated; physical systems have delays in both sensors and actuators and need to explore safely as to not compromise their integrity; the agent requires a discrete-time set-up with a control frequency that allows its physical body to follow its policy without putting too much strain on its components, a behaviour which might also have to be
modelled into the reward function. These associated problems cause real-world sampling to be considered expensive and to be reduced [6], which raises a demand for sample-efficient agents.

- **Curse of model uncertainty** - One popular way to reduce an agent’s need for real world interaction is to start training in simulated models, transfer the obtained policy to a robot and conclude training in the real environment [7]. Doing so requires designing a sufficiently accurate model of the original environment, which can be very challenging to do or not even feasible [8]. Even if it is possible, certain interactions such as contacts between parts and frictions are susceptible to local conditions and troublesome to model accurately. The underlying model uncertainty generates small errors which can accumulate and cause substantial discrepancies in the agent’s behaviour between simulation and real-world. Nonetheless, these models have an important purpose in RL investigation as they allow training, testing and comparing algorithms and derived policies in simulation before attempting real-world environments.

This work describes a slice of the evolution of RL algorithms under a common nomenclature while also providing the theoretical background behind the representation being made. Additional parts required to fit the resulting algorithms in a diversified set of environments are also described, before testing agents on environments to compare and evaluate the interaction. This approach was useful to provide a more complete description of RL investigation.

### 1.2 Related work

The fundamentals for Markov chains were introduced in 1913 by Andrey Markov [9]. Stochastic gradient descent was developed between 1940-1970 independently by American and Russian investigations [10]. The optimal control theory was formalized in the 1950s, with credit given to the work of Lev Pontryagin and Richard Bellman [11]. In 1989 Christopher Watkins introduces Q-Learning [12] which significantly boosted the practicality of RL algorithms at the time. The investigation for policy gradient methods began in the early 1990s [13]. Gerald Tesauro publishes TD-gammon in 1992 [14], an agent using an artificial NN and temporal difference learning to match the abilities of top human backgammon players. Robert Schapire published a book in 1997 [15] supporting the usage of weakly correlated learners to generate stronger agents. Q-Learning had scalability issues which were reduced by the Deep-Q-Network algorithm published in 2013 [16]. Policy gradient algorithms exhibit significant variance in training, an issue that was addressed by Deep Deterministic Policy Gradient algorithm published in 2015 [17] and further reduced in the Advantage Actor-Critic variants published in 2016 [18].

Nowadays RL investigation has expanded its field of research beyond what was previously possible. Facial recognition and speech recognition have matched human capabilities since the mid 2000s [19] and the recent industry of self-driving vehicles has already deployed several real-world agents [20]. “Smart houses” are becoming more and more common with research being made on how to expand the concept into “smart cities” [21]. Current RL investigation is headed, among other directions, towards making models able to continuously improve from massive sets of training data (big data algorithms)
[22], employing multiple agents on cooperative/competitive tasks [23] and handling environments with imperfect information [24].

1.3 Goals and contribution

Derived from the motivation presented, this work will describe the common ground between some known RL algorithms and use that ground to attempt answering the following research questions:

• What does an agent need to be able to learn from a RL set up?

• How to make an algorithm perform consistently across different environments?

• How to establish valid terms of comparison among different algorithms?

To contribute towards these goals this work will implement a set of known algorithms in a set of pre-built environments and display results obtained from a documented set-up, while also providing theoretical ground for the algorithms used and for some of the decisions made during implementation. From the results will be discussed the possibility of using the presented case studies to extract generic features for both agents and environments.

1.4 Overview

This work will start by defining environments according to the Markov decision process formulation and the generic expressions for two different methods used in RL investigation in Sections 2.1-2.6. Afterwards the approaches used to reduce dimensionality will be considered with the resulting algorithms in Section 2.7, where the prominent part played by NNs will result in a more elaborate explanation in Section 2.8, concluded by the possibility of combining parts provided by different sources to generate "stronger" algorithms in Sections 2.9 and 2.10.

The implementation details will be covered in Sections 3.1 and 3.2. The environments selected for this work will be described in Section 3.3, including an explanation for why given environment was selected and for augments used when solving it. The hyper-parameters used to describe the agents in this work are presented in Section 3.4.

Afterwards, the experiments performed on each environment, the agents employed to do so and results obtained from those set-ups will be shown in Sections 4.1-4.4, along some considerations regarding the nature of the presented results and decisions made when collecting them, followed by a more generic discussion in Section 4.5.

The conclusion in chapter 5 presents considerations about the methodology in this work while also pointing towards potential directions it could follow.
Chapter 2

Background

2.1 Markov Decision Process

A Markov decision process (MDP) was originally designed as a discrete time stochastic control work frame for decision making in environments where the complete dynamics were described into models. This was used to study optimization problems and solve them with dynamic programming, which uses the transition function to decompose the task at hand into a sequence of optimal decisions over time [25]:

![Agent-environment interaction loop](image)

Describing an environment as an MDP requires defining a state space $S$, describing observable states, and an action space $A$, containing available actions [26]. It also requires the environment to approximate the Markov property, which considers that the transition dynamics only depend on the current state and action regardless of the sequence of states and actions (trajectory) leading to those [4]. This property is used to define the probabilities of transitioning between any two states and the scalar return signals received from the environment as functions of state-action pairs:

- $P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, ..., s_0, a_0) = P(s'|s, a)$ - Transition function
This set-up enables describing the environment dynamics using one-step transitions \( t(s, a, r, s') \). The solution for an MDP is a function mapping states into actions, also known as a policy:

- \( \pi(s) = P(a|s) \) - Stochastic policy describing the probabilities of every action in a given state.
- \( \pi(s) = a \) - Deterministic policy assigning a particular action to a given state.

Policies are evaluated by the policy value function \( V^\pi(s) \), or the expected cumulative reward to be obtained when following a given policy starting from a given state. A discount factor \( \gamma \) is also included so it is possible to consider an infinite horizon while keeping the policy value a real number [26]:

\[
V^\pi(s) = E \left[ \sum_{t=0}^{\infty} \gamma^t r_{t+1} | s, \pi \right]
\]

An optimal policy for a given environment maximizes the value of all states:

\[
\pi^*(s) = \max_{\pi} V^\pi(s), \forall s \in S
\]

An MDP is solvable with model-based approaches that expand the expectation term or model-free approaches that estimate it:

- **Model-based approach** - Assumes knowledge of the transition function to build a copy of the model, which can be optimized before ever interacting with the environment.
- **Model-free approach** - Does not assume knowledge of the transition function as the agent builds an internal representation of the environment from a generic model as it learns.

RL focuses on designing model-free algorithms but both these approaches are derived from the same result, formally known as the Bellman equation.

### 2.2 Bellman's equation and Principle of Optimality

The Bellman equation [1] breaks optimization problems into two conditions by separating immediately obtainable rewards from the value of resulting states. As stated in Bellman’s Principle of Optimality "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision":

\[
V^\pi(s) = E[r + \gamma \sum_{t=0}^{\infty} \gamma^t r_{t+2} | s, \pi]
\]

And by writing the expectation term \( E[f(x)] = \sum_x P(x)f(x) \) for the first step using the transition and reward functions [4]:

\[
V^\pi(s) = \sum_a P(a|s) \sum_r P(r|s, a) \sum_{s'} P(s'|s, a)(r + \gamma E[\sum_{t=0}^{\infty} \gamma^t r_{t+1} | s', \pi])
\]
The same decision problem is now delayed by one step, meaning the expression can be written recursively as:

\[ V^\pi(s) = \sum_a P(a|s) \sum_r P(r|s, a) \sum_{s'} P(s'|s, a)(r + \gamma V^\pi(s')) \quad (2.5) \]

Where \( P(a|s) \) is the agent’s policy, \( P(r|s, a) \) and \( P(s'|s, a) \) are respectively the reward and transition functions, and \((r + \gamma V^\pi(s'))\) is a label to be filled using information from transitions an agent collects by training on a given environment.

### 2.3 Value based algorithms

Value based algorithms extend from the notion of policy value function described in Section 2.1 by considering the set of available actions instead of an explicit policy, resulting in the Q-value function:

\[ Q(s, a) = E[\sum_{t=0}^{\infty} \gamma^t r_{t+1}|s, a] \quad (2.6) \]

Inserting this definition in Equation 2.5 results in the expression:

\[ Q(s, a) = \sum_r P(r|s, a) \sum_{s'} P(s'|s, a)(r + \gamma V(s')) \quad (2.7) \]

Where \( V(s) \) is the expected cumulative reward to be obtained from acting optimally from given state, or the state value:

\[ V(s) = \max_{a \in A} Q(s, a) \quad (2.8) \]

The goal for a value based agent is to learn the Q-values of a given environment and find the optimal policy from those. Doing so from a random initialization requires the estimator to eventually forget its initial predictions. As explored in [27], this behaviour can be modelled with a soft-increment estimator:

\[ \hat{Q}(s, a) = (1 - \alpha) \ast \hat{Q}(s, a) + \alpha \ast \text{sample} \quad (2.9) \]

Where \( \alpha \) is the learning rate (LR) for the estimator. Applying a soft-increment estimator with the label from the Bellman equation in 2.7 results in:

\[ \hat{Q}(s, a) = (1 - \alpha) \ast \hat{Q}(s, a) + \alpha \ast (r + \gamma \hat{Q}(s', a')) \quad (2.10) \]

\( V(s') \) is unknown and must be estimated by the agent, with the available options resulting in different value based algorithms. If the agent first chooses its next action and uses the Q-value for that pair as the state value, that results in a State-Action-Reward-State-Action (SARSA) update rule:

\[ \hat{Q}(s, a) = (1 - \alpha) \ast \hat{Q}(s, a) + \alpha \ast (r + \gamma \hat{Q}(s', a')) \quad (2.11) \]
Alternatively, if the agent evaluates all available actions and then uses the highest Q-value as the value estimate, the update rule now resembles Q-learning:

\[
\hat{Q}(s, a) = (1 - \alpha) \cdot \hat{Q}(s, a) + \alpha \cdot (r + \gamma \max_{a' \in A} \hat{Q}(s', a'))
\]  

(2.12)

The practical difference between these two is that with SARSA (Algorithm 1) the agent learns from the policy it’s currently following (on-policy) while Q-learning (Algorithm 2) estimates all Q-values for given state (off-policy), thus requiring a set of discrete actions:

Initialize \( Q(s, a) \) randomly, \( T = 0; \)

for episode do
  Initialize \( a \) randomly, \( s; \)
  while not terminal state do
    \( s', r = step(a); \)
    \( a' = policy(s', T); \)
    \( \hat{Q}(s, a) = (1 - \alpha) \cdot \hat{Q}(s, a) + \alpha \cdot r + \gamma \cdot \hat{Q}(s', a'); \)
    \( s = s', T = T + 1, a = a'; \)
  end
end

Algorithm 1: Pseudo-code for SARSA

Initialize \( Q(s, a) \) randomly, \( T = 0; \)

for episode do
  Initialize \( s; \)
  while not terminal state do
    \( a = policy(s, T); \)
    \( s', r = step(a); \)
    \( \hat{Q}(s, a) = (1 - \alpha) \cdot \hat{Q}(s, a) + \alpha \cdot r + \gamma \cdot \max_{a' \in A} \hat{Q}(s', a'); \)
    \( s = s', T = T + 1; \)
  end
end

Algorithm 2: Pseudo-code for Q-Learning

Using a table to store the Q-function will result in poor scalability for the algorithm as it requires a number of discrete entries growing exponentially with the environment description. The alternative found is to describe the Q-values as a parametrized function. The cost for doing so is adding a potentially biased layer which might have to be hand crafted for the environment [28].

A separate topic is raised by the policy function as it must both explore the environment and collect reward from it, a known multi-disciplinary issue [29] discussed in Section 2.4.
2.4 Exploration/Exploitation trade-off

The end goal in an RL task is to get the agent to collect the most cumulative reward possible from its environment by learning an optimal policy from a random initialization. To do so it needs to explore its environment and use collected information to learn about the respective dynamics. As the agent learns it favours more valuable states over less valuable ones in order to exploit its environment for attainable rewards [30].

Even though exploitation leads the agent towards an optimal policy, exploration is required to diversify the transitions obtained training and prevent the agent from converging on policies with immediate value over better long term solutions [31]. In RL this trade-off can be addressed by combining exploration noise with a greedy decision function:

- **Value based** - Section 2.3 - A common solution is including a decaying probability for the agent to choose a random action instead of the optimal one, known as $\epsilon$-greedy curve and described in Equation 2.13.

- **Policy based** - Section 2.6 - The exploration noise can be generated as variance of a stochastic policy or by adding a zero-mean random signal over a deterministic policy, as described in Equation 2.14

\[
\begin{align*}
\text{action} &= \max_{a \in A} Q(s,a) \quad \mathcal{U}(0,1) > \epsilon \\
\text{action} &= \text{random}(a \in A) \quad \mathcal{U}(0,1) < \epsilon \\
\text{action} &= \pi(s) + \mathcal{N}(0,\sigma^2)
\end{align*}
\] (2.13)

(2.14)

Generating exploration noise requires some hyper-parameters to provide a smooth transition in training from an initial stage favouring exploration to the subsequent stage of exploitation.

2.5 Gradient Descent

Gradient descent is an iterative method which finds the minimum of an objective function $J(x)$ by computing its gradient and taking steps in the negative direction of the obtained values. The same can be done in the positive direction of the gradient, resulting in finding the maximum of the objective function through gradient ascent. This is based upon the idea that as long as the objective function is defined and differentiable in the neighbourhood of a point described by an arbitrary set of parameters $\theta$, then $J(\theta)$ immediately varies the most if one goes from that point in the direction of the gradient of $J(\theta)$ [32]. With $\alpha$ as LR the notation in Equation 2.15 holds for gradient descent:

\[
\Delta \theta = -\alpha \nabla_\theta J_\theta
\] (2.15)
Notice how this expression is in no way related to the MDP description presented so far. All it states is the possibility of finding solutions for objective functions by following the gradient of $\theta$. An important note is that gradient descent does not naturally distinguish between local and global optima, meaning this method only guarantees local convergence and can be susceptible to initial conditions [33].

2.6 Policy based algorithms

In order to turn Equation 2.15 into a RL update rule, it is necessary to connect it to the MDP set up presented in Section 2.1. To do so one can apply parametrization $\theta$ to the policy function:

- $\pi_\theta(s) = P(a|s, \theta)$ - Parametrized stochastic policy
- $\pi_\theta(s) = a$ - Parametrized deterministic policy

And set the objective function to be the policy value:

$$J_\theta = V^\pi(s) = \sum_a P(a|s, \theta) \sum_r P(r|s, a) \sum_{s'} P(s'|s, a)(r + \gamma V^\pi(s'))$$ (2.16)

The gradient of the objective function is then given by:

$$\nabla_\theta J_\theta = \nabla_\theta \sum_a P(a|s, \theta) \sum_r P(r|s, a) \sum_{s'} P(s'|s, a)(r + \gamma V^\pi(s'))$$ (2.17)

From Equation 2.7 one can define:

$$Q^\pi(s, a) = \sum_r P(r|s, a) \sum_{s'} P(s'|s, a)(r + \gamma V^\pi(s'))$$ (2.18)

As the expected value of given action performed on given state under given policy. Equation 2.17 then becomes:

$$\nabla_\theta J_\theta = \nabla_\theta \sum_a P(a|s, \theta)Q^\pi(s, a)$$ (2.19)

Where the gradient is given by:

$$\nabla_\theta J_\theta = \sum_a \left( \delta P(a|s, \theta) \frac{\partial Q^\pi(s, a)}{\partial \theta} + P(a|s, \theta) \frac{\partial Q^\pi(s, a)}{\partial \theta} \right)$$ (2.20)

By defining a policy state distribution as in Equation 2.21:

$$d^\pi(s) = \sum_{t=0}^{\infty} \gamma^t P(s_{t+1}|s_t, \pi)$$ (2.21)

Equation 2.20 can be solved while avoiding the derivative of the policy state distribution in regards to policy parameters, a term without a clear solution [34]. This is performed in [35] under the assumption that a stationary policy state distribution exists and is independent of the original state for given policy.
The result is formally known as the policy gradient theorem, which states that for any differentiable policy its gradient is given by:

$$\nabla_{\theta} J_{\theta} = \sum_s d^1(s) \sum_a \frac{\delta P(a|s, \theta)}{\delta \theta} Q^\pi(s, a)$$  \hspace{1cm} (2.22)

Since $d^\pi(s)$ relates to the transition function and is unknown in RL, it is approximated by having an agent train on the environment and learn from the trajectories performed on each episode. Doing so requires the agent to always follow its policy so the trajectories are sampled from a state distribution that approximates $d^\pi(s)$ [36] and results in Equation 2.23:

$$\nabla_{\theta} J_{\theta} = \frac{\delta \log P(a|s, \theta)}{\delta \theta} Q^\pi(s, a)$$  \hspace{1cm} (2.23)

Where the term $\frac{\delta \log P(a|s, \theta)}{\delta \theta}$ is known in maximum likelihood estimation as the score function $H_\theta(s, a)$ [37]. Common policy parametrizations have documented score functions [38] and NNs compute scores for any given format:

$$\nabla_{\theta} J_{\theta} = H_\theta(s, a)Q^\pi(s, a)$$  \hspace{1cm} (2.24)

$Q^\pi(s, a)$ is unknown and must be estimated by the agent. As a first approach one could use the rewards obtained during an episode and sum them to obtain a return value $v$ for every step, similarly to a Reward Increment non Negative Factor Offset Reinforcement (REINFORCE) algorithm, which generates the update rule of Equation 2.25 and Algorithm 3:

$$\Delta \theta = \alpha H_\theta(s, a)v$$  \hspace{1cm} (2.25)

Initialize $\theta$ randomly, $T = 0$, $t = 0$;

for episode do
  Initialize $s$, $H[t] = 0$, $R[t] = 0$, $v = 0$;
  while not terminal state do
    $a = \text{policy}(s, T)$;
    $s', r = \text{step}(a)$;
    $H[t] = \text{score}(s, a)$;
    $R[t] = r$;
    $s = s'$, $t = t + 1$, $T = T + 1$;
  end
  while $t > 0$ do
    $t = t - 1$;
    $v = R[t] + \gamma \times v$;
    $\theta \leftarrow \alpha \times H[t] \times v$;
  end
end

Algorithm 3: Pseudo-code for REINFORCE
The presented algorithm has some limitations: The conditions of gradient descent described in Section 2.5 are inherited; it is only usable in episodic environments as the terminal state is used to solve the expectation term; variance is generated by having the agent learn an entire episode without accounting possible deviations in its trajectory. Nonetheless, the background for it has made deriving solutions found capable of solving more and more complex environments [39]

2.7 Function approximation

When the environment becomes so massive that requiring a random search to visit every state often is longer feasible, a different representation is needed. This is done by using function approximation (FA) [40], which can conditionally be applied to policy and value based algorithms. Some known solutions are listed in this section:

- **State aggregation** - States and/or actions of similar magnitude are combined with a typically hand-crafted discretization. This allows value based agents to solve continuous environments and more complex discrete environments by reducing the size of the Q-value function. Cannot be done for policy based algorithms, since this solution is not differentiable.

- **Linear combination of features** - A weight matrix is used to apply a linear transformation and reshape the environment description into a set of features. After defining the features to use an appropriate weight matrix can be trained.

- **Neural networks** - A pre-built function approximation model which uses generic linear regression elements behind a highly customizable front-end to achieve non-linear behaviour. They are supported by the universal approximation theorem [41] and generate features from training, but require tuning of hyper-parameters used at the front-end.

The methodology for applying linear features to the value and policy based algorithms described so far will be covered in this section and the application to NNs is covered in Sections 2.8, 2.9 and 2.10.

2.7.1 Linear features in value based approximations

The linear FA $W^T \phi(s,a)$ tries to approximate the Q-function of Equation (2.7) using a parametrized weight matrix $W$ and a custom set of parameters $\phi(s,a)$:

$$Q(s,a) = \sum_r \sum_{s'} P(r|s,a)P(s'|s,a)(r + \gamma V(s'))$$  \hspace{1cm} (2.26)

$$Q_W(s,a) = W^T \phi(s,a)$$  \hspace{1cm} (2.27)

The Q-value estimator of Equation (2.10) now has to update the weight matrix. By writing the same equation as such:
\[
\hat{Q}_W(s,a) = \hat{Q}_W(s,a) + \alpha \ast (r + \gamma V(s') - \hat{Q}_W(s,a))
\] (2.28)

The term \((r + \gamma V(s') - \hat{Q}_W(s,a))\) is also known as the temporal difference error \(\delta_W\) ([42]) between the estimate for the Q-value label derived from transition information \(r + \gamma V(s')\) and the current estimate from the weight matrix \(\hat{Q}_W(s,a)\), an error signal which can be minimized with gradient descent. From Equation (2.15):

\[
\Delta W = -\alpha \nabla_W \delta_W
\] (2.29)

And considering that as explored in [43], the squared error yields better results and is used instead:

\[
\Delta W = -\frac{1}{2} \alpha \nabla_W \delta_W^2
\] (2.30)

The gradient of the squared error is given by:

\[
\Delta W = -\alpha \delta_W \nabla_W \delta_W
\] (2.31)

One solution for approximating \(\nabla_W \delta_W\) is to implement eligibility traces \(z_t\) [44], which track the contribution of each feature to the estimated error according to a discount parameter \(\lambda \in [0, 1]\).

\[
z_t \approx \nabla_W \delta_W
\] (2.32)

\[
z_t = \gamma \lambda z_{t-1} + \phi(s,a)
\] (2.33)

Which results in the \(W\) update rule described in Equation 2.34:

\[
\Delta W = -\alpha (r + \gamma V(s') - \hat{Q}_W(s,a))z_t
\] (2.34)

Estimating \(V(s')\) by using the SARSA criteria in Equation (2.11) results in a SARSA(\(\lambda\)) update rule and Algorithm 4:

\[
\Delta W = -\alpha (r + \gamma \hat{Q}_W(s',a') - \hat{Q}_W(s,a))z_t
\] (2.35)

With \(\lambda = 0\) only the current state-action is considered for the update, resulting in behaviour consistent with a SARSA algorithm; if \(\lambda = 1\) the contribution from passing states-actions only decays by \(\gamma\) on each step, resulting in similarities to an one-step REINFORCE algorithm.

### 2.7.2 Linear features in policy based

For policy based algorithms the goal is to maximize the policy value of a parametrized policy \(\pi_{\theta}(s)\) using gradient ascent. As explored in Section 2.6, Equation (2.36) leads to Equation (2.37):
Define $\phi(s, a)$;
Initialize $W$ randomly, $T = 0$;

for episode do
    Initialize $s, a$ randomly, $z = 0$;
    while not terminal state do
        $s', r = \text{step}(a)$;
        $a = \text{policy}(s, T)$;
        $\delta W = r + W' \ast \phi(s', a') - W' \ast \phi(s, a)$;
        $z = \gamma \ast \lambda \ast z + \phi(s, a)$;
        $W \leftarrow -\alpha \ast \delta W \ast z$;
        $s = s', T = T + 1, a = a'$;
    end
end

Algorithm 4: Pseudo-code for SARSA($\lambda$)

$$J_\theta = V^\pi(s) = \sum_a P(a|s, \theta) \sum_r P(r|s, a) \sum_{s'} P(s'|s, a)(r + \gamma V^\pi(s'))$$

(2.36)

$$\nabla_\theta J_\theta = H_\theta(s, a)Q^\pi(s, a)$$

(2.37)

Where $H_\theta(s, a)$ evaluates the current policy according to $Q^\pi(s, a)$, which is estimated for given transition. Using a Q-value function approximation $Q_W(s, a)$ as an unbiased estimate of $Q^\pi(s, a)$ requires both sets of parameters to meet the conditions for the compatible function approximation theorem [35] in Equations (2.38) and (2.39):

$$\sum_s d^\pi(s) \sum_a P(a|s, \theta)(Q^\pi(s, a) - Q_W(s, a)) \frac{\delta Q_W(s, a)}{\delta W} = 0$$

(2.38)

$$\frac{\delta Q_W(s, a)}{\delta W} = \frac{\delta \log P(a|s, \theta)}{\delta \theta}$$

(2.39)

Under these conditions the algorithm will follow the exact policy gradient in spite of the used parametrizations and the following update rule is valid:

$$\Delta \theta = \alpha H_\theta(s, a)Q_W(s, a)$$

(2.40)

The implication is that the parametrizations used are not independent and should be designed together to ensure compatibility. Algorithms based on training both sets of parameters are referred as Actor-Critic (AC), similar to Algorithm 5:

Having the actor update its parameters according to the Q-values predicted by the critic makes the actor loss not converge towards a zero mean, which adds variance to the algorithm [45]. In spite of this and of longer training steps from training both $W$ and $\theta$, AC algorithms show to be more sample-efficient than value function approximations [46].
Define $\phi(s, a)$;
Initialize $W$ and $\theta$ randomly, $T = 0$;
for episode do
    Initialize $s, a$ randomly, $z = 0, H = 0$;
    while not terminal state do
        $s', r = \text{step}(a)$;
        $a = \text{policy}(s, T)$;
        $\delta_W = r + W' \ast \phi(s', a') - W' \ast \phi(s, a)$;
        $z = \gamma \ast \lambda \ast z + \phi(s, a)$;
        $H = \text{score}(s, a)$;
        $W \leftarrow -\alpha \ast \delta_W \ast z$;
        $\theta \leftarrow \alpha * H * W' * \phi(s, a)$;
        $s = s', T = T + 1, a = a'$;
    end
end

Algorithm 5: Pseudo-code for Q Actor-Critic

2.8 Neural Networks

An NN is an off the shelf black box function approximation consisting on layers of neurons, where each neuron multiplies its inputs by a set of trainable weights before summing the products. A trainable bias is then added to this result, which goes through an activation function before being fed into the next layer, as shown in Figure 2.2:

In Figure 2.2 $n$ is the size of the current layer, $m$ is the size of the next layer and $w_i$ and $b_i$ contain the weights and bias for a given layer $i$. A set-up inspired by models used to describe brain activity [47]. In RL these elements are trained using regression through prediction and back propagation of error signals.

Consider the function to approximate here is the Q-value function. In order to train the sets $w$ and $b$ from a random start, the NN needs to make predictions, find the error between the predicted and the true values for the Q-value function, and use that difference to adjust individual parameters. As discussed in Section 2.7.1, this is attainable by applying gradient descent to minimize the error $\delta_W$ between a soft
increment estimator and the label from Bellman equation:

\[ Q(s, a) = r + \gamma V(s') \]  

(2.41)

\[ \delta_W = Q(s, a) - \hat{Q}_W(s, a) \]  

(2.42)

\[ \Delta W = -\alpha \nabla_W \delta_W \]  

(2.43)

NNs include extra steps in Equation (2.43) by converting the error into a loss function, which is then used by an optimizer to compute \( \Delta W \) and update the NN parameters:

- **Loss function** - Applies a transformation to the error signal before feeding the optimizer. In RL the most common loss is mean-square-error (MSE) in Equation (2.44), with available alternatives such as the Huber loss in Equation (2.45).

- **Optimizer** - The commonly used optimizers for RL employ some variation of gradient descent, where the loss function is used to compute gradient values which discriminate individual parameter contribution for given prediction. They differ in the additional features included to improve performance.

\[ \Delta W = -\frac{1}{2} \sum_{\alpha} \nabla_W \delta_W^2 \]  

(2.44)

\[ \Delta W = -\frac{1}{2} \alpha \nabla_W \delta_W^2 \quad |\delta_W| < K \]  

\[ \Delta W = -\alpha \nabla_W (\delta_W - \frac{K}{2}) \quad |\delta_W| > K \]  

(2.45)

These are included in the description of NN learning cycle in Figure 2.3:

![Figure 2.3: Learning cycle for Neural Networks](image-url)
2.9 Deep-Q-Network

A Deep-Q-Network (DQN) results from implementing Q-Learning using an NN as value function approximation. $V(s')$ is estimated using the criteria of Q-Learning in Equation (2.12), which makes this algorithm require a set of discrete actions:

$$Q(s, a) = r + \gamma \max_{a' \in A} \hat{Q}(s', a')$$

(2.46)

A DQN agent requires at least the following parts:

- **Neural network model** - An effective brain for the agent to be used by the other parts.

- **Acting** - An action selection process based on the current state of the environment and the Q-values predicted by the network. During training some initial noise is required in the action selection as covered in Section 2.4, which in DQN conventionally means an $\epsilon$-greedy curve with linear decay.

- **Training** - Performing an action on a state and observing the reward received and the following state from that decision. These elements are then combined with a terminal state flag $d$ as transitions $t(s, a, r, s', d)$ which will be used in learning. Most algorithms also require preprocessing the raw environment data, often due to array shaping or image manipulation.

- **Learning** - The function which updates the parameters according to data obtained from training. It is possible to learn from pre-existing information (off-line learning) [48] but all agents implemented in this work periodically learn with batches of an available pool as they train (on-line learning).

As stated before learning requires an agent to use its estimated values as labels to train itself. Any fluctuations that appear are repeatedly propagated [49] which adds variance to the algorithm used, an issue that can be partially addressed by also including the following components:

- **Memory buffer** - Having a big pool of transitions to learn from reduces the correlation between the samples used in each batch [50]. The selection process used for batching is random, but can include some quantification of how useful given transition is [51].

- **Target neural network** - A target network is included as a a delayed copy of the network in training and is used for estimating Q-values in learning. The target NN makes it so the parameter changes applied to the NN after each learning operation do not immediately affect the labels used for learning, which stabilizes training. In DQN the target network is periodically updated by copying the training network into it.

These elements are connected as described in Figure 2.4 to generate a DQN agent. The pseudocode is written in Algorithm 6:

In algorithm 6, when a NN is defined as a function of $(s, a)$, that term calls a prediction from the NN. Without it the same term calls the respective NN parameters. This notation holds for Algorithms 7 and 8.
2.10 Actor-Critic

The AC logic presented in Section 2.7.2 can be used with NNs to generate explicit representations of both policy \( \text{ACTOR}(s) = \pi_\theta(s) \) and Q-value \( \text{CRITIC}(s, a) = Q_W(s, a) \) functions.

The critic uses the temporal difference error with the actions predicted by the actor for its estimation:

\[
Q(s, a) = r + \gamma Q_W(s', \pi_\theta(s'))
\]  
(2.47)

The actor uses policy gradient theorem with Q-values predicted by the critic for its estimation:

\[
\Delta \theta = \alpha H_\theta(s, a)Q_W(s, \pi_\theta(s))
\]  
(2.48)

This algorithm estimates both terms required for learning \( Q^\pi(s, a) \) and \( V(s') \) by chaining both parametrizations. To reduce the variance of this algorithm a baseline can be included [45]. One of the most common baselines is derived from temporal difference learning and is known as the advantage function \( A(s, a) = Q(s, a) - V(s) \) [52], or the difference in value between a specific action and the known optimal action in given state. Under the conditions described in Section 2.10.2, the following actor update rule is also valid:

\[
\Delta \theta = \alpha H_\theta(s, a)(\text{CRITIC}(s, \text{ACTOR}(s)) - \text{CRITIC}(s, a))
\]  
(2.49)

2.10.1 Deep Deterministic Policy Gradient

A Deep Deterministic Policy Gradient (DDPG) results from implementing the actor-critic set-up presented in Section 2.10 without including a baseline. A DDPG agent requires the following parts:

- **Actor-Critic network model** - As mentioned before, this algorithm requires two separate NNs: the
Initialize $M$ empty, $W$ randomly, $W^+ = W$, $opt$, $T = 0$;

for episode do
  Initialize $s$, $d = false$;
  while $d = false$ do
    $a = policy(s, T)$;
    $s', r, d = step(a)$;
    $M ← add(s, a, r, s', d)$;
    $f(s, a, s', d) ← batch(M)$;
    $label = f(r) + \gamma\max_{a' ∈ A} W^+(f(s'), a') * f(d)$;
    $loss = (W(f(s), f(a)) - label)^2 / size(f)$;
    $W ← gradients(opt, loss)$;
    $s = s'$, $T = T + 1$;
  if $T/N_copy = integer$ then
    $W^+ = W$
  end
  end
end

Algorithm 6: Pseudo-code for DQN with memory as $M$, optimizer as $opt$, training NN as $W$ and target NN as $W^+$

actor which takes as input a state and predicts an action; the critic which takes as input a state and an action, and predicts the Q-value for that pair.

- **Acting** - The agent’s actions are predicted by the actor and the exploration noise is added to the decision by combining a zero-mean random signal with the actor deterministic prediction. The signal used is commonly a Gaussian curve or an Ornstein–Uhlenbeck process [53], either with exponentially decaying variance.

- **Training** - This component contains all meta-information regarding the agent’s training such as when to act when to learn or when to reset the environment. It also handles the storage of information related to the evaluation metrics to be displayed or plotted.

- **Learning** - A common batch is selected for both networks from the pool of transitions. The actor predicts the action it would greedily choose for the state which is then evaluated by the critic. The resulting gradient from the critic weights is used as a initialization for the actor gradients, which are then computed and applied to the actor without a loss function.

This algorithm also benefits from having a memory buffer and target networks for both actor and critic, due to the same reasons described in Section 2.9. The montage of a DDPG agent is depicted in Figure 2.5 and the associated pseudo-code is written in Algorithm 7:

Updating the target network with a soft increment $\tau$ at each step allows multiple agents with different initializations to train the same targets [54].
2.10.2 Advantage Actor-Critic

To directly apply the advantage function to an update rule, the baseline $V(s)$ used needs to be independent of the parametrization being updated [55]

$$A(s, a) = Q(s, a) - V(s)$$ (2.50)

Modifying the critic to predict an advantage instead of a value requires a value estimate not predicted by the critic. Using a sum of rewards for this would add variance to the algorithm as mentioned in Section 2.6.

Modifying the actor to update its policy towards the advantage requires a value estimate using an action not predicted by the actor. A viable alternative is to use the action performed during training, which will not be the optimal action for most of training but will eventually converge towards it. Doing so results in Equation (2.51):

$$\Delta \theta = \alpha H_\phi(s, a)(CRITIC(s, ACTOR(s)) - CRITIC(s, a))$$ (2.51)

There is an option about using either training or target critic to predict $CRITIC(s, a)$: on the one hand the actor updates are based on the training NNs and using the same NNs for both predictions seems adequate; on the other hand the target critic provides sturdier behaviour as expected of a baseline. The approach taken was to test both alternatives on Section 4.1 and compare their performance, which resulted in the advantage Actor-Critic(A2C) algorithm shown as Algorithm 8:
Initialize $M$ empty, $\theta$ and $W$ randomly, $\theta^+ = \theta$, $W^+ = W$, $T = 0$, $opt1$, $opt2$;

for episode do
  Initialize $s$, $d = false$;
  while $d = false$ do
    $a = policy(s, T)$;
    $s^\prime, r, d = step(a)$;
    $M \leftarrow add(s, a, r, s^\prime, d)$;
    $f(s, a, r, s^\prime, d) \leftarrow batch(M)$;
    $a_1 = \theta^+(f(s^\prime))$
    $label1 = f(r) + \gamma W^+(f(s^\prime), a_1) * f(d)$;
    $loss1 = (W(f(s), f(a)) - label1)^2 / size(f)$;
    $W \leftarrow gradients(opt1, loss1)$;
    $a_2 = \theta(f(s))$;
    $label2 = W(f(s), a_2)$;
    $loss2 = -label2 / size(f)$;
    $\theta \leftarrow gradients(opt2, loss2)$;
    $\theta^+ = \tau * (\theta - \theta^+)$;
    $W^+ = \tau * (W - W^+)$;
    $s = s^\prime$, $T = T + 1$;
  end
end

Algorithm 7: Pseudo-code for DDPG with actor as $\theta$, target actor as $\theta^+$, critic as $W$ and target critic as $W^+$
Initialize $M$ empty, $\theta$ and $W$ randomly, $\theta^+ = \theta$, $W^+ = W$, $T = 0$, $opt1$, $opt2$;

for episode do
  Initialize $s$, $d = false$;
  while $d = false$ do
    $a = policy(s, T)$;
    $s', r, d = step(a)$;
    $M \leftarrow add(s, a, r, s', d)$;
    $f(s, a, r, s', d) \leftarrow batch(M)$;
    $a_1 = \theta^+(feed(s'))$
    $label1 = f(r) + \gamma W^+ (f(s'), a_1) \ast f(d)$;
    $loss1 = (W(f(s), f(a)) - label1)^2 / size(f)$;
    $W \leftarrow gradients(opt1, loss1)$;
    $a_2 = \theta(f(s))$
    $label2 = W(f(s), a_2)$
    $loss2 = -(label2 - W^+(f(s), f(a))) / size(f)$;
    $\theta \leftarrow gradients(opt2, loss2)$;
    $\theta^+ = \tau \ast (\theta - \theta^+)$;
    $W^+ = \tau \ast (W - W^+)$;
    $s = s'$, $T = T + 1$;
  end
end

Algorithm 8: Pseudo-code for A2C with actor as $\theta$, target actor as $\theta^+$, critic as $W$ and target critic as $W^+$
Chapter 3

Implementation

3.1 Implemented algorithms

The algorithms selected for implementation are described in Sections 2.9 and 2.10. These cover a solid portion of the methods described in Chapter 2 while presenting generic solutions. Their main properties can be summarized as:

- **DQN** - Evaluating the entire action space requires a set of discrete actions but stabilizes the resulting policy. The lack of explicit policy iteration generates faster training steps when compared to the other algorithms but lowers sample-efficiency, which can prove detrimental with increasing environment complexity or in real-world applications.

- **DDPG** - Including a policy function increases sample-efficiency and allows the agent to solve continuous action spaces, but also induces variance in resulting policy due to the reasons explored in Section 2.10.

- **A2C** - Modifying the actor update to account for the advantage instead of the value by including an additional prediction in the learning cycle makes the overall training slower and steadier than with DDPG, often resulting in more valuable policies.

The known differences in these algorithms result in different issues associated with RL training. This work will train and observe agents employing these algorithms in different environments and attempt to find appropriate metrics for the resulting behaviour.

3.2 Experimental setup

The implementation of the algorithms listed in Section 3.1 was done using Python (version 3.6.9) [56] and Tensorflow (version 1.14.0) with Keras (version 2.4.3) [57], due to their popularity resulting in extensive documentation available to support the developed algorithms. In the image-based environments (of Sections 3.3.2 and 3.3.4) the mentioned pre-processing operations were written using the OpenCV
library (version 4.1.2) [58], geared towards optimizing diverse image-based operations. The Bayesian optimization described in 4.3 was performed using the Scikit-optimize (version 0.7.4) library [59].

Due to the complications that arise when designing reward functions [2] and to maintain the focus on designing and evaluating agents, the environments listed in Section 3.3 were imported from the OpenAiGym library (version 0.15.4) [60], which contains a sizeable collection of environments varying from classic control problems, old-school Atari games, a 2D physics engine with tasks, among others.

The Duckietown environment described in Section 3.3.4 was a project initiative to build a small-scale platform that maintained some of the challenges associated with real-world autonomous robot tasks [61]. There exists a physical Duckietown to where researchers can send agents to participate in competitions around the year, but this work will use their on-line platform available as an OpenAiGym package (version 2019.0.0) to train and test agents on pre-built Duckietown maps through a customizable simulator.

### 3.3 Case Studies

The environment selection for this work was performed under the premise that each would pose a different challenge and result in different issues featured in RL research. The criteria leading to the collection of case studies presented in this section considered if a given environment had image-based states and if the agent would navigate the environment without significant complications (considered desirable behaviour) or if getting the agent to attempt solving the environment was a challenge in itself (undesirable behaviour). The characteristics for each environment that lead to the selection are presented in Table 3.1:

<table>
<thead>
<tr>
<th></th>
<th>Desirable behaviour</th>
<th>Undesirable behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not image-based</td>
<td>Lunar Lander</td>
<td>Bipedal Walker</td>
</tr>
<tr>
<td>Image-based</td>
<td>Space Invaders</td>
<td>Duckietown</td>
</tr>
</tbody>
</table>

Table 3.1: Environment selection and criteria

The environment being image-based or encouraging undesirable behaviour both can be seen as conditions which require addressing. The solutions implemented are listed and described in the respective sub-sections, along with the description of said environment.

#### 3.3.1 Lunar Lander

The goal in this task is to get an agent to learn how to land on a pad without crashing. Its body consists of an hull with two pads to smooth the ground contact. The environment includes an action wrapper that allows agents to solve it using either discrete or continuous actions.

Since this is an environment with relatively low complexity and a detailed reward function, tuning the parameters of vanilla algorithms is enough to generate agents capable of solving it. Even so the continuous and discrete action wrappers enable all algorithms mentioned in Section 3.1, which is useful for comparing them under similar conditions.
### Lunar Lander

<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>agent’s position and speed in components x and y, angular position and speed, 2 ground contact flags.</td>
</tr>
<tr>
<td>Discrete actions</td>
<td>&quot;do nothing&quot;, &quot;fire left engine&quot;, &quot;fire main engine&quot;, &quot;fire right engine&quot;</td>
</tr>
<tr>
<td>Continuous actions</td>
<td>Input values for the main and side engines, normalized to [-1,1]</td>
</tr>
</tbody>
</table>
| Rewards       | +100 directionally pointing towards the center of the landing pad  
-0.3 for fuel unit on main engine  
-0.03 for fuel unit on side engine  
+10 for ground contact  
+100 for landing  
-100 for crashing |
| Solved condition | Average score of 200 over 100 episodes |

Table 3.2: Environment description for Lunar Lander

Figure 3.1: Screen shot for Lunar Lander

#### 3.3.2 Space Invaders

The goal in this task is to get an agent to learn how to play the old-school Atari game “Space Invaders” using only game information. Its body is the pad at the bottom of the screen. The state space is a $160 \times 210$ image coded with the RGB colour model where each pixel is described by 3 integers between [0,255]. Figure 3.2 is a screen-shot for this environment before pre-processing and Figure 3.3 is a screen-shot for afterwards.

The enormous state space and a reward function which only rewards the agent for destroying an enemy ship will both be responsible for a particularly slow optimization, which raises a need for additional components to accelerate training. Currently image-based environments require the NN model to include convolutional layers [62], both for reducing the amount of NN parameters being trained and for learning dynamic features from states observed while training [63]. Agents training in these en-
<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>Screen pixel values for each frame</td>
</tr>
<tr>
<td>Actions</td>
<td>&quot;do nothing&quot;, &quot;shoot&quot;, &quot;move left&quot;, &quot;move right&quot;, &quot;move left and shoot&quot;, &quot;move right and shoot&quot;.</td>
</tr>
<tr>
<td>Rewards</td>
<td>In-game score.</td>
</tr>
<tr>
<td>Solved condition</td>
<td>Open</td>
</tr>
</tbody>
</table>

Table 3.3: Environment description for Space Invaders

Environments also benefit from converting RGB into a greyscale to reduce environment complexity and from normalizing pixel values as real values between [0,1] to reduce algorithm variance. The following elements were also beneficial:

- **Additional pre-processing** - The original image was downscaled to 80 x 105 and the score at the top of the screen was removed as it proved distracting. Contrast was increased to saturate the values of the dynamic game elements as the original Atari implementation flickered some of them and caused their intensity to vary.

- **Input depth** - Using a single frame as input for the network would not give the agent information about the velocities of movable objects. While it would still learn to play in such conditions, a better solution can be found by feeding the network more frames leading up to the decision point.

- **Frame skipping** - The frames in a video have a strong correlation which can disrupt training [50]. To lower this effect the agent only observes one of every N frames [64] and its action is repeated for the skipped frames.

- **Huber loss** - In this environment there is a mother ship which is harder to hit and gives a sub-

Figure 3.2: Screen shot for Space Invaders
stantially larger reward than other ships. When using a squared loss the highly valuable nature of this ship would disrupt training. To attenuate this effect the NNs are trained with Huber loss, which behaves as a linear loss above a threshold error and as a squared loss below it.

This environment is documented by many open sources and was featured in a popular RL article [50] not long ago, which made it a solid reference and a stepping stone for more complex image-based environments.

### 3.3.3 Bipedal Walker

In this environment the agent learns how to walk through a slightly irregular surface and reach a goal line to the far right. Its body consists of a hull and two legs which the agent can control through joints located on its hips and knees, and a LIDAR system [65] which keeps track of the ground through range measurements.

The challenges in this environment stem from the complexity associated with walk patterns. When this is combined with a reward function that only returns meaningful rewards after the agent starts moving, it can result in agents holding still after countless crashes without significant progression. Addressing this issue requires favouring exploration, which was done with the inclusion of the following parts:

- **Warm start** - In this environment an heuristic function [66] was employed at the start of training as policy function. After a buffer is filled the agent explores according to its policy and learns from its memory, which will temporarily contain this stage of training as a warm start for the agent [67].
<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>States</strong></td>
<td>Angular position and speed of the hull and the 4 joints at both hips/knees, horizontal and vertical speed of the hull, 2 ground contact flags, 10 LiDAR readings.</td>
</tr>
<tr>
<td><strong>Actions</strong></td>
<td>Input values for the torque to be applied at each hip/knee joint, normalized to [-1,1].</td>
</tr>
</tbody>
</table>
| **Rewards**   | +4.333 for unit moved horizontally  
-0.00035 for torque unit  
-5.0 each radian the hull is tilted  
-100 for falling                                |
| **Solved condition** | Average score of 300 over 100 episodes                                               |

Table 3.4: Environment description for Bipedal Walker

![Figure 3.4: Screen shot for Bipedal Walker](image)

- **Reward clipping** - To further prevent the agent from holding still, the penalty for falling was hyper-parametrized and set to smaller values. This approach eases the initial stage of training where the agent is learning a walk pattern, with a potential cost to its balance on later stages.

This environment requires precise hyper-parameter tuning to solve using the described algorithms, which raises a need for efficient tuning options.

### 3.3.4 Duckietown

This environment is a self-driving car simulation task where the agent attempts to navigate different Duckietown maps. Its body consists of an hull with 4 wheels and a frontal camera, from where states are recorded as $640 \times 480$ images. Figure 3.5 is a screen-shot of the environment before pre-processing is applied and Figure 3.6 is a screen-shot after the pre-processing operation:

The difficulty for this environment stems from the fact that it is designed to approximate some of the many real-world issues that increase the task’s complexity: the roads are irregular and lighting varies
<table>
<thead>
<tr>
<th>Element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>States</td>
<td>Screen pixel values for each frame.</td>
</tr>
<tr>
<td>Actions</td>
<td>Input values for the speed and the steering of the hull, normalized to ([-1, 1]).</td>
</tr>
<tr>
<td>Rewards</td>
<td>+1 aligning the agent's velocity with the road.</td>
</tr>
<tr>
<td></td>
<td>-10 for relative position in lane.</td>
</tr>
<tr>
<td></td>
<td>-40 for proximity with road obstacles.</td>
</tr>
<tr>
<td></td>
<td>-1000 for crashing into an obstacle or leaving the road.</td>
</tr>
<tr>
<td>Solved condition</td>
<td>Open</td>
</tr>
</tbody>
</table>

Table 3.5: Environment description for Duckietown

Figure 3.5: Screen shot for Duckietown

along the lanes; there are roadside and moving objects which look similar and a horizon; the states are being observed from a camera on the front of the agent, meaning the agent now has a first person perspective of its surroundings. These issues cause the relevant features for the task to no longer be unique, requiring the agent to account possible variations. The following components were needed to get to the presented results:

- **Additional pre-processing** - The image was downscaled to \(160 \times 120\), contrast was decreased and brightness was reduced due to the presence of colourful landscape elements. Frame skipping and input depth are also applied for the same reasons discussed in Section 3.3.2.

- **Pooling layer** - A pooling layer reduces the amount of parameters being trained by a NN by applying one convolutional filter to aggregate neighbouring values according to a pre-determined operation [68].

- **Reward shaping** - Even though the reward components are adequate, the original magnitudes used caused the agent to favour staying alive over navigating the map. As such all reward components were hyper-parametrized as performed for the crashing penalty in Section 3.3.3. A negative
sign was also added to the velocity component when the agent is moving backwards as the original reward did not account for that behaviour and made it exploitable.

After pre-processing the environment looks like:

![Pre-processed screen shot for Duckietown](image)

Figure 3.6: Pre-processed screen shot for Duckietown

The open ended nature of this environment along its customizable structure and increasing challenge generated by the available maps provide an elaborate sandbox for training, evaluating and comparing agents and learnt policies.

### 3.4 Hyper parameter tuning

To ensure the algorithms of Section 3.1 solve the environments in Section 3.3, their hyper-parameters must be tuned for the given task. Traditionally there is no way of tuning an agent for a specific environment a priori, so many RL implementations had to find configurations using brute force by trial-and-error or random grid searches. As complexity increases training takes longer and the bounds in which the hyper-parameters cause the respective algorithm to learn an optimal solution become narrower, which can turn these methods into a cumbersome exercise.

The tables in this section present the list of hyper-parameters used to describe the agents implemented in Chapter 4, along a generic description and some reference values from this work and other open-sourced implementations using similar algorithms. A training step was arbitrarily defined as one action selection from the agent. Table 3.6 contains the hyper-parameters required by generic RL algorithms, while table 3.7 contains the hyper-parameters associated with NNs and table 3.8 contains additional hyper-parameters for image-based environments:

Some alternative methods for tuning are known: Bayesian optimization [69] suggests a guided random search before training where the learning labels are held as true and used to optimize the set
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Typical values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory size</td>
<td>Maximum length of the memory list filled with transitions</td>
<td>[50000, 1000000]</td>
</tr>
<tr>
<td>Initial buffer</td>
<td>Initial buffer filled with random actions</td>
<td>[5000, 100000]</td>
</tr>
<tr>
<td>Batch size</td>
<td>Number of transitions used in each learning operation</td>
<td>(16, 32, 64, 128)</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>Soft increment applied to the estimator used for policy and/or value function.</td>
<td>( (10^{-5}, 3 \times 10^{-5}, 10^{-4}, 3 \times 10^{-4}, 10^{-3}) )</td>
</tr>
<tr>
<td>( \gamma )</td>
<td>Discount factor used for future rewards.</td>
<td>0.99</td>
</tr>
<tr>
<td>Crash reward</td>
<td>Terminal state reward (when hyper-parametrized)</td>
<td>[-100, 0]</td>
</tr>
<tr>
<td>Copy steps (DQN)</td>
<td>Steps between each copy of training NN into target NN</td>
<td>[500, 50000]</td>
</tr>
<tr>
<td>( \tau ) (DDPG)</td>
<td>Soft increment applied to the target NN update</td>
<td>[0.001, 0.02]</td>
</tr>
<tr>
<td>( \epsilon_{\text{max}} ) (DQN)</td>
<td>The exploration noise added in value-based agents.</td>
<td>[0.8, 1]</td>
</tr>
<tr>
<td>( \epsilon_{\text{min}} ) (DQN)</td>
<td>The exploration noise added in value-based agents.</td>
<td>[0.01, 0.1]</td>
</tr>
<tr>
<td>( \sigma^2_{\text{max}} ) (DDPG)</td>
<td>The exploration noise added in policy-based agents.</td>
<td>[0.5, 1.5]</td>
</tr>
<tr>
<td>( \sigma^2_{\text{min}} ) (DDPG)</td>
<td>The exploration noise added in policy-based agents.</td>
<td>[0.01, 0.05]</td>
</tr>
<tr>
<td>Decay steps</td>
<td>Steps from max to min exploration noise.</td>
<td>[0.5, 1.5]</td>
</tr>
</tbody>
</table>

Table 3.6: RL hyper-parameters in implemented agents

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Typical values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loss</td>
<td>Transformation applied to the error signal</td>
<td>('mse', 'huber loss', 'crossentropy')</td>
</tr>
<tr>
<td>Optimizer</td>
<td>How to apply the loss for backpropagation</td>
<td>('Adam', 'RMSprop', 'SGD')</td>
</tr>
<tr>
<td>Activation</td>
<td>Transformation applied to the output of a layer</td>
<td>('relu', 'linear', 'tanh', 'sigm', 'softmax')</td>
</tr>
<tr>
<td>Min divider</td>
<td>Smallest divisor used in backpropagation</td>
<td>[0.001, 0.1]</td>
</tr>
<tr>
<td>Dense layer n drop</td>
<td>Fully connected layer Number of neurons in layer Dropout fire rate</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[50, 500]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[0.02, 0.5]</td>
</tr>
</tbody>
</table>

Table 3.7: NN hyper-parameters in implemented agents

of hyper-parameters employed; gradient-based optimization [70] suggests including hyper-parameter tuning as part of learning.

An important note is that configurations are only found by balancing these hyper-parameters among themselves and given environment, as its that configuration that manifests in training and learning.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Typical values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Frame-skip</td>
<td>Amount of frames skipped between observed states</td>
<td>[2, 5]</td>
</tr>
<tr>
<td>Learn steps</td>
<td>Steps between learning operations</td>
<td>[1, 4]</td>
</tr>
<tr>
<td>Contrast</td>
<td>Multiplier for the original pixel values</td>
<td>[0.5, 1.5]</td>
</tr>
<tr>
<td>Brightness</td>
<td>Value added to the original pixel values</td>
<td>[-50, 50]</td>
</tr>
<tr>
<td>Conv layer n</td>
<td>2D Convolution layer</td>
<td>-</td>
</tr>
<tr>
<td>size</td>
<td>Number of filters in layer</td>
<td>[10, 100]</td>
</tr>
<tr>
<td>step</td>
<td>Size of each squared filter</td>
<td>[2, 10]</td>
</tr>
<tr>
<td>Pool layer size</td>
<td>2D Pooling layer</td>
<td>-</td>
</tr>
<tr>
<td>step</td>
<td>Stride used by each filter along both axis</td>
<td>[1, 5]</td>
</tr>
<tr>
<td>op</td>
<td>Operation performed</td>
<td>'Max' 'Avg'</td>
</tr>
</tbody>
</table>

Table 3.8: Image-based hyper-parameters in implemented agents
Chapter 4

Experiments

4.1 Lunar Lander

4.1.1 Experimental description

This environment serves as an approach to the stochastic nature of RL environments, which manifests in training and generates different responses even for identical agents. This random signal can be better described by training the same agent multiple times, which gives room to evaluate patterns that may not be visible from a single run. The number of trials displayed $N = 3$ is low but enough to address features displayed by the agents.

Agent 1 uses a DQN algorithm with the hyper-parameters of Table 4.1 and the NN structure displayed in Figure 4.1. Agent 2 uses a DDPG algorithm with the hyper-parameters of Table 4.2 and the NN structure displayed in Figure 4.2:

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory size</td>
<td>50000</td>
</tr>
<tr>
<td>Initial buffer</td>
<td>5000</td>
</tr>
<tr>
<td>Copy steps</td>
<td>5000</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.0001</td>
</tr>
<tr>
<td>$\gamma$</td>
<td>0.99</td>
</tr>
<tr>
<td>Batch size</td>
<td>32</td>
</tr>
<tr>
<td>$\epsilon$ max</td>
<td>1</td>
</tr>
<tr>
<td>$\epsilon$ min</td>
<td>0.05</td>
</tr>
<tr>
<td>$\epsilon$ decay steps</td>
<td>300000</td>
</tr>
</tbody>
</table>

Table 4.1: Hyper-parameters for agent 1 in Lunar Lander

Agents 3 and 4 use the same set-up as agent 2 with an A2C algorithm. Agent 3 uses the training critic to estimate the baseline used in the advantage function of training actor while agent 4 uses the target critic instead. Agents 1 and 2 are employing algorithms with known limitations and their results will be used as a baseline to compare the responses from agents 3 and 4 and determine which of the theoretical options left from Section 2.10.2 provides better results.
Figure 4.1: Neural network scheme for agent 1 in Lunar Lander

Memory size = 50000
Initial buffer = 5000
γ = 0.99
Actor α = 0.00003
Critic α = 0.00001
Batch size = 32
Actor τ = 0.005
Critic τ = 0.005
σ² decay steps = 150000
σ² max = 1
σ² min = 0.01

Table 4.2: Hyper-parameters for agents 2, 3, 4 in Lunar Lander

4.1.2 Results

Generic metrics used for evaluating RL agents are the average score, average network loss and the average episode duration, or the length in steps between environment resets after reaching a terminal state. These are used to describe 3 runs performed by agent 1 in Figure 4.3:

Agent 1 demonstrates an overall good performance as it steadily solves the environment. These
metrics also provide an argument for dividing training in this environment in 3 stages: the agent learns how to stabilize itself; the agent learns how to land with an overly careful approach to not crash; the agent optimizes landing and is able to do so loosely. DQN algorithms are typically limited by their bias and/or sample-inefficiency, but for the relatively low complexity of this environment these issues do not prevent agent 1 from consistently meeting the solved condition.

The results for 3 runs performed by agent 2 using the metrics described so far is given by Figure 4.4:

Agent 2 exhibits more variance in its graphics than agent 1. It is also observable that even though agent 2 is capable of landing, it doesn’t always optimize that behaviour into a steady policy as agent 1 did, or even necessarily learn an optimal policy. These are expected results for a vanilla DDPG agent as explored in Sections 2.7.2 and 2.10.

The results for 3 runs performed by agent 3 is depicted in Figure 4.5 and 3 runs by agent 4 are shown in Figure 4.6:

In the reward and actor loss signals both agents show capability of solving the environment with agent 4 exhibiting less variance, which results in a more stable behaviour than agent 3, as expected from the baselines employed by each agents. Nonetheless, the critic loss and the episode duration signals have similar behaviour for both agents which can make these results inconclusive on their own. Further experimenting done on Bipedal Walker and Duckietown favoured the set-up used by agent 4. As such, the A2C agents described in those environments use the baseline predicted by the target critic.
Figure 4.4: Results for agent 2 in Lunar Lander

Figure 4.5: Results for agent 3 in Lunar Lander
4.2 Space Invaders

4.2.1 Experimental description

As mentioned in Section 3.3.2 this environment takes considerable time to optimize and as shown in section 4.2.2 the agent’s behaviour is mostly unidimensional. For these reasons this environment was mostly used as validation for the additional image-based layers included in the algorithm.

Agent 1 uses a DQN algorithm with the hyper-parameters of Table 4.3 and the NN structure displayed in Figure 4.7:

```
Input
 X/255
  Conv 1
   Conv 2
    Conv 3
     FLAT
      Dense 1
       x
        Mask
         Output
```

Input shape = [105, 80, 4]
Conv 1 n = 32, size = 8, step = 4
Conv 2 n = 64, size = 4, step = 2
Conv 3 n = 64, size = 4, step = 1
Dense 1 n = 512, drop = 0
Output shape = 6
Mask shape = 6
Conv activation = ‘relu’
Dense activation = ‘relu’
Output activation = ‘linear’
Loss = ‘huber loss’
Optimizer = ‘adam’
Min divider = 0.01

Figure 4.7: Neural network scheme for agent 1 in Space Invaders
Memory size = 1000000
Initial buffer = 50000
Copy steps = 40000
\(\alpha = 0.0003\)
\(\gamma = 0.99\)
Batch size = 32
\(\epsilon_{\text{max}} = 1\)
\(\epsilon_{\text{min}} = 0.1\)
\(\epsilon_{\text{decay steps}} = 1000000\)
Frame-skip = 4
Contrast = 100
Brightness = 0
Learn Steps = 4

Table 4.3: Hyper-parameters for agent 1 in Space Invaders

The contrast was greatly increased to saturate the values of the dynamic game elements. The mask layer used is a binary vector describing which actions have the Q-value predicted. This allowing using a one-shot encoding in learning and skip one prediction while also evaluating all actions in the decision function.

4.2.2 Results

The additional metrics evaluated in this environment (average ships destroyed, average episode duration, shooting accuracy, average pixel distance travelled by the agent) all demonstrated a similar evolution throughout training. Due to an increased training time requirement to solve this environment and the following ones when compared to Lunar Lander only 1 run was performed by each agent, which proves an additional constraint on the interpretation being made of those results.

Agent 1 training is displayed in Figure 4.8:

One thing that became obvious both from these graphics and from using different metrics was that the agent’s performance in this environment is notoriously unidimensional. The relevant evaluation metrics for this environment are the rate at which it the agent gets better scores over time and the steady score it eventually hits.

In given results the network loss is increasing throughout the training, an indicator that even though the agent is improving it is not converging towards a solution as its estimated error is getting larger. This is an indicator of the amount of bias latent in the DQN algorithm employed. The other graphics show a relatively steady climb into a peak, followed by a turbulent decay. These graphics indicate the agent hit a performance wall around the 15-20 million training steps and could not continue to improve.

After testing the agent the cause for this issue becomes apparent: it cannot consistently hit the last enemy ship and ends up losing when it lands. Improving past this point would require either additional engineering tricks over the ones implemented, or a different set of hyper-parameters than the one de-
4.3 Bipedal Walker

4.3.1 Experimental description

Decently comparing and commenting on the effect of the hyper-parameters being used on any of these environments would require a grid search varying one parameter at a time with multiple runs for each configuration. Alternative approaches were taken in all environments considered. The experiment performed in this environment was to force a Bayesian optimization solution with the tools available and go over some of the issues which appear when doing so.

A Bayesian optimization solution expects a search space describing the hyper-parameters to be included in the optimization function and the intervals in which to do so, a quick learning session, a validation set or a validation test using a scalar objective function and a set of training data containing the true values for each transition.

The hyper-parameters being evaluated have to be included in a learning session each agent goes through before its validation test. While all hyper-parameters relating to the NN model being used (learning rates, number of hidden layers, size of hidden layers, loss, optimizer, among others) are immediate to include, others like the exploration noise or the target NNs \( \tau \) would require actual training to optimize. As such the parameter search space was defined as in Table 4.4, in accordance to the reference values.
exposed in the tables of Section 3.4:

<table>
<thead>
<tr>
<th>Parameter name</th>
<th>Type</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor $\alpha$</td>
<td>Real</td>
<td>$[10^{-6}, 10^{-3}]$</td>
</tr>
<tr>
<td>Critic $\alpha$</td>
<td>Real</td>
<td>$[10^{-6}, 10^{-3}]$</td>
</tr>
<tr>
<td>Batch size</td>
<td>Integer</td>
<td>$[8, 64]$</td>
</tr>
<tr>
<td>Dense 1 $n$</td>
<td>Integer</td>
<td>$[100, 500]$</td>
</tr>
<tr>
<td>Dense 2 $n$</td>
<td>Integer</td>
<td>$[100, 500]$</td>
</tr>
<tr>
<td>Dense 1,2 drop</td>
<td>Real</td>
<td>$[0.02, 0.5]$</td>
</tr>
</tbody>
</table>

Table 4.4: Search space for hyper-parameter optimization in Bipedal Walker

The learning session has to iterate over $N$ agents over $E$ epochs, which makes using the complete algorithm expensive. The quick version used has to approximate the complete algorithm being evaluated. In doing so the target networks were removed, the value estimate for the actor update was performed by the critic and all other labels used came from the set of training data.

The validation test consisted of having the agent perform $K$ episodes on the environment after learning. The objective function minimized was the negative of the average distance ran by each agent on its test.

Obtaining the set of training data required training agents 1 and 2. After training they were tested in $T$ episodes and the actions and values predicted by these agents were stored with respective states.

Agents 1 and 2 use the hyper-parameters of Table 4.5 and the NN structure displayed in Figure 4.9 with DDPG and A2C algorithms respectively:

<table>
<thead>
<tr>
<th>Hyper-parameter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory size = 100000</td>
</tr>
<tr>
<td>Initial buffer = 20000</td>
</tr>
<tr>
<td>$\gamma = 0.99$</td>
</tr>
<tr>
<td>Actor $\alpha = 0.00003$</td>
</tr>
<tr>
<td>Critic $\alpha = 0.0001$</td>
</tr>
<tr>
<td>Batch size = 32</td>
</tr>
<tr>
<td>Actor $\tau = 0.004$</td>
</tr>
<tr>
<td>Critic $\tau = 0.002$</td>
</tr>
<tr>
<td>$\sigma^2$ decay steps = 300000</td>
</tr>
<tr>
<td>$\sigma^2$ max = 1.5</td>
</tr>
<tr>
<td>$\sigma^2$ min = 0.01</td>
</tr>
<tr>
<td>Crash reward = -20</td>
</tr>
</tbody>
</table>

Table 4.5: Hyper-parameters for agents 1 and 2 in Bipedal Walker

Agent 3 is a DDPG-algorithm that went through the optimization process described using training data from agent 1, while agent 4 is a A2C-algorithm using training data from agent 2. With $N = 100$, $E = 10$, $K = 10$, $T = 50$, the results for the parameter optimization of each agent are described in Table 4.6:
Figure 4.9: Neural network scheme for agent 1 and 2 in Bipedal Walker

<table>
<thead>
<tr>
<th>Hyper-Parameter</th>
<th>Agent 3</th>
<th>Agent 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor $\alpha$</td>
<td>$10^{-3}$</td>
<td>$10^{-3}$</td>
</tr>
<tr>
<td>Critic $\alpha$</td>
<td>$5.253 \times 10^{-4}$</td>
<td>$9.488 \times 10^{-4}$</td>
</tr>
<tr>
<td>Batch size</td>
<td>10</td>
<td>30</td>
</tr>
<tr>
<td>Dense 1 $n$</td>
<td>500</td>
<td>246</td>
</tr>
<tr>
<td>Dense 2 $n$</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Dense 1,2 drop</td>
<td>0.02</td>
<td>0.5</td>
</tr>
<tr>
<td>Objective function</td>
<td>$-80.08$</td>
<td>$-96.74$</td>
</tr>
</tbody>
</table>

Table 4.6: Parameter optimization for agents 3 and 4 in Bipedal Walker

Some of the presented values are on the edge of the search space described in 4.4, an indicator that the described search space may not contain the optimal hyper-parameter configuration for the learning session performed. However the objective function indicates these agents learned some generic walk pattern under given optimization conditions. Agents 3 and 4 result from applying the hyper-parameter values presented in Table 4.6 to the set-up described in Table 4.5 and Figure 4.9 without modifying any other hyper-parameters.

4.3.2 Results

To describe patterns in the agent behaviour that sometimes are not perceptible using the generic metrics, additional evaluation tools may have to be designed for a particular environment. In this environment a custom metric was included relating to the percentage of transitions in the agent's memory that meet a condition derived from human power walking, defined as horizontal speed above 0.23 units and at least one foot touching the ground. This metric is referred in the graphics shown in this section as "Walking".

A run performed by agent 1 yielded Figure 4.10 and the results for agent 2 are shown in Figure 4.11: A trial and error parameter search was conducted around the values used for the graphics shown.
in this section. When tested the agents would either not attempt to walk, or do so with average scores
over 100 episodes in the range [50,200] for DDPG agents and [100,250] for A2C agents. As shown in Section 4.1 even the same configuration makes agents learn different policies, which further turns manual hyper-parameter tuning into a fastidious grind.

A run performed by agent 3 is shown in Figure 4.12 and a run by agent 4 is described by Figure 4.13:

![Figure 4.12: Results for agent 3 in Bipedal Walker](image)

Agent 3 demonstrates a performance similar to agent 1. The comparatively low batch size and high LRs made the average training step for agent 3 faster without compromising the resulting policy.

Agent 4 exhibits an erratic training leading to a policy where it throws itself forward and immediately falls. The relatively small layers combined with the high drop rate made agent 4 unable to learn a walk pattern and immediately converge towards a sub-optimal solution. These hyper-parameters yielded a good result in the parameter optimization described as it did not model the initial stage of training where the agent is influenced by poor value estimates and exploration noise, both conditions which put additional strain in learning.

Although the Bayesian approach being used is still very naive, there already exist options available for implementing better solutions [71] and so continuing to force this path was discouraged. The attempt was left as a thought process and a reminder that manual hyper-parameter tuning can be time consuming.
4.4 Duckietown

4.4.1 Experimental description

The increasing levels of difficulty resulting from a list of different maps featuring different navigation tasks was crucial for the development of all agents presented. A fair amount of work made here was retrofitted to previous environments and presented there. This experiment will evaluate an iterative array of agents on three navigation tasks, two of which are performed on the map shown in Figure 4.14:
The tasks referred in this work are: loop with static obstacles; loop with static and moving obstacles; randomized map from a list of available maps on the end of every episode. The components for the reward function are hyper-parametrized. As shown in Table 3.5 the reward function is in Equation (4.1):

\[
\Delta W = \begin{cases} 
+1 \times F_1 - 10 \times F_2 - 40 \times F_3 & \text{if not done} \\
-1000 & \text{if done}
\end{cases}
\]  

(4.1)

Where \( F_1 \) is proportional to the alignment between the agent velocity and the direction of the road, \( F_2 \) is proportional to its distance to the center of the right lane and \( F_3 \) is proportional to its proximity to any obstacles. The hyper-parametrization applied results in Equation (4.2):

\[
\Delta W = K_1 \times F_1 K_2 \times F_2 K_3 \times F_3 \quad \text{done} = \text{false}
\]

\[
\Delta W = \text{crash} \quad \text{done} = \text{true}
\]

(4.2)

Agents 1, 2, 3, 4 and 5 are A2C algorithms with the hyper-parameters of Table 4.7 and the NN structure of Figure 4.15:

- Memory size = 200000
- Initial buffer = 20000
- \( \gamma = 0.99 \)
- Actor \( \alpha = 0.00003 \)
- Critic \( \alpha = 0.0001 \)
- Batch size = 32
- Actor \( \tau = 0.003 \)
- Critic \( \tau = 0.002 \)
- \( \sigma^2 \) decay steps = 250000
- \( \sigma^2 \) max = 1
- \( \sigma^2 \) min = 0.025
- Frame-skip = 2
- Contrast = 0.9
- Brightness = -10
- Learn Steps = 1

Table 4.7: Hyper-parameters for agents 1,2,3,4,5 and 6 in Duckietown

The reward function employed by each agent is described by Equation (4.2) and Table 4.8:

Agents 1, 2, 3, 4 and 5 are representative of an evaluation on the effects of modifying the reward magnitudes on the configuration described by Table 4.7 and Figure 4.15. The feedback obtained from training and testing each of these agents was part of the process in getting to the final results and is also described in Section 4.4.2.

Agent 6 uses the hyper-parameters of Table 4.7 and the respective reward in Table 4.8 but was modified to only be able to accelerate forward. This alteration was intended to reduce the task complexity.
Figure 4.15: Neural network scheme for agents 1, 2, 3, 4 and 5 in Duckietown

<table>
<thead>
<tr>
<th>Agent</th>
<th>(K_1)</th>
<th>(K_2)</th>
<th>(K_3)</th>
<th>crash</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent 1</td>
<td>0.9</td>
<td>-6</td>
<td>-15</td>
<td>-50</td>
</tr>
<tr>
<td>Agent 2</td>
<td>0.5</td>
<td>-5</td>
<td>-10</td>
<td>-40</td>
</tr>
<tr>
<td>Agent 3</td>
<td>0.5</td>
<td>-5</td>
<td>-10</td>
<td>-35</td>
</tr>
<tr>
<td>Agent 4</td>
<td>0.3</td>
<td>-3</td>
<td>-10</td>
<td>-25</td>
</tr>
<tr>
<td>Agent 5/Agent 6</td>
<td>0.7</td>
<td>-7</td>
<td>-14</td>
<td>-50</td>
</tr>
</tbody>
</table>

Table 4.8: Magnitudes for reward function of agents 1, 2, 3, 4, 5 and 6 in Duckietown

by limiting the actions available for the actor network as shown in Figure 4.16:

Figure 4.16: Neural network scheme for agent 6 in Duckietown
4.4.2 Results

The additional metrics used in this environment were: average reward criteria defined as the percentage of transitions the agent received a reward superior to half the value of its parameter $K_1$, and referred as "Positive"; the percentage of transitions the agent is "Turning", defined as performing an action with $|speed| < 0.8$ and $|steering| > 0.5$. A run performed by agent 1 in the loop with static obstacles resulted in Figure 4.17 and agent 2 presented Figure 4.18 in the same map:

![Figure 4.17: Results for agent 1 in Duckietown:Loop static](image)

Upon testing and observing the agents, the policy for agent 1 had converged to accelerating straight forward until immediately before reaching the end of lane, rotating itself approximately 180 degrees, then repeat until it could not stop in time and cross the end of lane. Upon reaching obstacles the agent would either rotate itself immediately before or crash into the obstacle, being that the obstacle shape did not seem to matter. The policy for agent 2 converged in shorter accelerations and more 180 degree rotations. The agent also moves away from obstacles or rotates before going near them, but crosses the end of lane while rotating.

The results for agent 3 in the static loop are in Figure 4.19 and agent 4 in the same map is shown in Figure 4.20:

The results for agent 3 are interesting in the sense that it is unclear why the agent was able to move past the larger portion of training where its metrics do not exhibit any response, which was eventually broken by an apparent random mutation. Upon testing and observing the agents, agent 3 showed signs of attempting to navigate the map along the right lane and dodging duck shaped obstacles. However an initial position where the agent was not aligned with the road caused it to immediately cross the end.
of lane. Even when navigating properly the agent would often cross the end of lane on some turns or
Figure 4.20: Results for agent 4 in Duckietown: Loop static

Agent 4 also navigates the map favouring the right lane but does so with an hesitant pendular movement. Also while it is really careful with the end of lane, it mostly disregards all obstacles and crashes into them. Another pattern exhibited by agent 4 was that when it faced perpendicular to an end of lane, it would sometimes get stuck in a back to forth or a side to side movement. These results indicate that even though reducing the original reward magnitudes initially proved advantageous, doing so indefinitely did not.

The results for agent 4 on the loop with inclusion of dynamic obstacles are shown in Figure 4.21 and the results for agent 5 on the same task are presented in Figure 4.22. Both these sessions were stopped prematurely as the evolution of the “Turning” and “Episode duration” metrics indicate the policies for both these agents were converging towards accelerating straight forward until crossing the end of lane, which was confirmed upon testing and observing both agents. Even though agent 3 demonstrates it would still be possible for agents 4 and 5 to eventually learn how to navigate the map, the fact these agents converge into such blatantly sub-optimal policies as the exploration noise decays seems symptomatic of instability in the NN models being used, which disrupts training regardless of the policies that are eventually found. The relatively similar responses across the metrics evaluated for both agents is also an indicator that the issue at hand may not be solvable by reshaping the reward functions involved.

The results for agent 6 on the loop with inclusion of dynamic obstacles are shown in Figure 4.23, while the results for the same agent on the task with randomized maps on reset are presented in Figure 4.24.
Figure 4.21: Results for agent 4 in Duckietown: Loop dynamic

Figure 4.22: Results for agent 5 in Duckietown: Loop dynamic

Upon testing and observing the agent in both tasks, the policy for agent 6 in the dynamic loop
converged into decent navigation around the map through the right lane and dodging most of the static
obstacles. However the agent did not show signs of attempting to dodge moving obstacles, as it would move past them regardless of whether collision was imminent or not. Also, it would almost always cross the end of lane when turning on a particular curve, which made it unlikely for this agent to ever get more than one lap around the map. On the task with randomized maps, agent 6 often favours the middle of the road over the right lane when navigating. Even though it attempts to navigate around all maps and is capable of dodging static obstacles, it does not attempt to avoid imminent collisions with moving obstacles. It also exhibited tendencies to getting stuck looping in road intersections and to sometimes, apparently at random, rotate itself 180 degrees and navigate in the opposite direction.

4.5 Discussion

Measuring training length using discrete steps is useful to compare the agent's sample-efficiency but on its own does not provide information about how much time a session took. Displaying the step count along the average frequency of training steps (training frequency) would allow evaluating both sample-efficiency and time elapsed. The training frequency can vary over stages of training and can be derived from the combination of two different effects, which are the time required for the agent to perform a learning operation (learning frequency) and the time required for the agent to interact with its environment (testing frequency). The learning frequency is typically much lower than the testing frequency. In simulated environments the training frequency is mostly an additional metric for benchmarking, but for real-world applications it can become a constraint for algorithms used and require situational suspension of learning.

Tasks have an associated level of complexity, related to the size of both state and action spaces for a given environment and to the underlying reward and transition functions. The reward function indirectly defines the task and can be used to lower its complexity with a detailed description of the optimal behaviour for a given task. This is observable from comparing the results from Space Invaders in Section 4.2.2 to the results from Duckietown in Section 4.4.2, where the intuitively more complex tasks performed in Duckietown are faster to optimize due to multiple components in the respective reward function, which provide diversified and immediate feedback to agents in training. The transition function relates to the environment physics. The results from Duckietown exhibit that the inclusion of additional dynamics such as moving obstacles can affect the task complexity of an otherwise identical navigation task.

The transitions sampled in training contain an unknown amount of noise or uncertainty, which an agent needs to filter in order to learn an optimal policy. Different agents show a different susceptibility to this noise, which generates bias and variance in the policy being trained. Generically, increasing the variance of an estimator lowers its bias an vice-versa, which made this issue known as bias-variance trade-off [72]. During training an agent's susceptibility to noise can cause two known generic conditions: the agent overfits when the variance in learning prevents it from stabilizing on an optimal policy; the agent underfits when the bias in learning prevents it from ever finding an optimal policy. As an example for the terms mentioned in this paragraph: agents 2 and 3 in Lunar Lander exhibit high variance and
overfit, while agent 4 presents a lower variance which softens this effect; agent 1 in Space Invaders begins underfitting when it hits its performance wall, an indicator of the bias latent in the agent; agents 1, 2 and 3 in Bipedal Walker overfit to similar degrees, even though theoretically agent 2 has lower variance which is not clearly discernible in the results presented; agent 4 in Bipedal Walking presents very high bias which makes it underfit near the start of training; the Duckietown environment generates a considerable amount of noise, which results in agents training there often getting stuck in sub-optimal policies.

This discussion leads to Table 4.9, containing the mentioned characteristics:

<table>
<thead>
<tr>
<th>Agent</th>
<th>Environment</th>
<th>Agent-Environment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning frequency</td>
<td>Task complexity</td>
<td>Training/Testing frequency</td>
</tr>
<tr>
<td>Bias/Variance</td>
<td>Noise</td>
<td>Underfitting/Overfitting</td>
</tr>
</tbody>
</table>

Table 4.9: Generic properties for agents, environments and their interaction
Chapter 5

Conclusions

The feedback obtained from training and testing agents on multiple environments is a great assistance when developing RL algorithms, as the different patterns described by similar agents on different environments usually gives more insight into underlying conditions affecting them. Although hyper-parameter tuning often deters the usage of identical agents, evaluating additional components implemented over a set of environments also leads to clearer results and stronger validation than when derived from a single environment.

The algorithms presented in this work are not state-of-the-art, but the basis provided is enough to continue adding components and generate more advanced versions: Asynchronous A2C (A3C) [73] uses multiple agents with different initializations to train the same target networks; Dueling DQN (DDQN) [74] separates the state value from the advantage of a given action in that state; Trust Region Policy Optimization (TRPO) [75] includes the region inside which learning leads to monotonic improvements in the policy value; Soft Actor-Critic (SAC) [76] includes an entropy measurement to have an agent optimize a task while also acting as randomly as possible.

Nevertheless the design of environments and reward functions is an important aspect in RL investigation [77] and opting into pre-made environments left this area mostly untouched. Also all the case studies were simulated environments, meaning there was not a opportune chance to address the curse of real world sampling.

5.1 Future work

Despite this work intending to address some challenges featured in RL investigation, all results and conclusions obtained were derived from experiments performed on simulated environments, which do not convey some of the issues that become more prominent in real world environments [78]. Deploying an agent to the real world requires an additional layer converting actions it predicts into mechanical inputs for its body, which can be implemented with Robot Operating System (ROS) [79]. Nonetheless, this process is hindered by unexpected results which often require re-dimensioning both agents and environments. Documenting these issues to explore potential causes and develop generic solutions
would be an interesting, albeit long and convoluted direction to follow.

An alternate route would be to perform a similar process regarding the employment of multiple agents in cooperative or competitive tasks [15], which raises another plethora of challenges related with agent communication and with the definition of a common goal for multiple agents [80].
Bibliography


