Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Mathias Graatrud Aronsen

# Path Planning and Obstacle Avoidance for Marine Vessels using the Deep Deterministic Policy Gradient Method

Master's thesis in Engineering Cybernetics
Supervisor: Proffesor Thor I. Fossen
June 2019

NTNU
Norwegian University of
Science and Technology

Mathias Graatrud Aronsen

# Path Planning and Obstacle Avoidance for Marine Vessels using the Deep Deterministic Policy Gradient Method

Master's thesis in Engineering Cybernetics
Supervisor: Proffesor Thor I. Fossen
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Kunnskap for en bedre verden

**NTNU**
**Norwegian University of**
**Science and Technology**

**Faculty of Information Technology**
**and Electrical Engineering**
**Department of Engineering Cybernetics**

# MASTER'S THESIS PROJECT DESCRIPTION

| | |
|---|---|
| **Name:** | Mathias Graatrud Aronsen |
| **Department:** | Engineering Cybernetics |
| **Thesis title:** | Path Planning and Obstacle Avoidance for Marine Vessels using the Deep Deterministic Policy Gradient Method |

## Thesis Description:

The goal of the project is to investigate and test deep reinforcement learning algorithms for ship path planning using a state-of-the-art container ship model in Matlab. In addition, methods for testing and training of the reinforcement learning agent should be addressed.

The following items should be considered in more detail:

1. Literature study and review of existing algorithms.
2. Design and implementation of a Deep Reinforcement Learning control algorithm
3. Porting the container vessel model from the MSS toolbox to Python and integrating it in a simulation environment
4. Training and testing of the reinforcement learning agent
5. Conclude your findings in a report.

| | |
|---|---|
| **Start date:** | 2019-01-07 |
| **Due date:** | 2019-07-17 |

| | |
|---|---|
| **Thesis performed at:** | Department of Engineering Cybernetics, NTNU |
| **Supervisor:** | Professor Thor I. Fossen, Dept. of Eng. Cybernetics, NTNU |

# Preface

This masters thesis is written as the pinnacle of my studies at the Department of Cybernetics and Robotics at the Norwegian University of Science and Technology during the spring of 2019. The work was supervised by professor Thor I. Fossen. The thesis summarizes my research, methods used and findings from applying Deep Reinforcement Learning (DRL) methods to create a control system for path planning and obstacle avoidance for marine surface vessels. The work is meant to serve as a step towards fully automated docking of marine vessels.

To complete my work the Python 3.6 programming language was used. The container vessel model used for simulations was ported from the Maritime Systems Simulator (MSS) toolbox [1]. The automatic differentiation library Tensorflow [2] and the high level API tflearn [3] were used to implement the Deep Deterministic Policy Gradient (DDPG) method [4]. The main contribution of my work is the application of the DDPG method as a control algorithm for the path planning and obstacle avoidance tasks. I also put significant time and effort into implementing a simulation environment integrating the control algorithm and the ported vessel model.

I would like to thank my supervisor professor Thor I. Fossen for for guidance throughout my work and allowing me to work freely with the problems I found interesting. I would also like to thank my fellow classmates Per Johannes and Byung for great collaboration and fun times throughout all the years of our Masters studies. Finally I would like to thank my family and especially beautiful wife Malin for love and support through all the ups and downs of writing my thesis.

17.06.2019

Mathias Graatrud Aronsen

# Summary

With autonomy as one of the main focuses in the development of the maritime industry one aspect of a fully autonomous vessel which has yet to be fully explored is the docking phase. Due to the complexity of the maneuvers it is difficult to compute what is the optimal behaviour when docking. However with the advancement of machine learning in recent years new methods have become available as possible solutions to the problem.

In this thesis the problem of autonomous docking is separated into two separate objectives of path planning and obstacle avoidance. An implementation of the Deep Deterministic Policy Gradient method is then applied as a control algorithm for solving both the separate objectives. Despite limited success in the training and tests performed in this work the algorithm shows promise given certain suggested improvements are made.

# Sammendrag

Med autonomi som et av hovedfokusene innen utviklingen av den maritime industrien er dockingsfasen et aspekt av et fullt autonomt fartøy som ennå ikke er utforsket fullt ut. På grunn av manøvrernes kompleksitet er det vanskelig å beregne hva som er den optimale oppførselen nå r man dokker. Men med fremskritt av maskinlæring de siste å rene har nye metoder blitt tilgjengelige som mulige løsninger på problemet.

I denne oppgaven er problemet med autonom docking delt inn i to separate mål. Disse er baneplanlegging og hinder ungåelse. En implementasjon av Deep Deterministic Policy Gradientmetoden blir deretter brukt som en kontrollalgoritme for å løse begge de separate målene. Til tross for begrenset suksess i trening og tester utført i dette arbeidet, viser algoritmen seg lovende gitt at visse foreslåtte forbedringer blir gjennomført.

# Table of Contents

# List of Tables

# List of Figures

# Abbreviations

| | | |
|---|---|---|
| ANN | = | artificial neural network |
| DDPG | = | Deep Deterministic Policy Gradient |
| DOF | = | Degrees of Freedom |
| DP | = | Dynamic Programming |
| DRL | = | Deep Reinforcement Learning |
| IQL | = | Improved Q Learning |
| MC | = | Monte Carlo |
| MDP | = | Markov decision process |
| ML | = | machine learning |
| RL | = | Reinforcement Learning |
| TD | = | Temporal Difference |

Chapter 1

# Introduction

## 1.1 Background and motivation

Autonomy is growing into one of the main focuses in the development of the shipping industry. One interesting aspect of a fully autonomous ship is the docking phase. Docking a ship requires extreme precision, efficient use of energy and time and has a high risk of damage to the ship itself, cargo and people working both on the ship and the dock. When docking manually the procedure varies greatly depending on the operator. Automating this process has the advantages of making the procedure safer by removing the element of human failure as well as the need for humans in high risk positions. In addition it will be easier to optimize energy and time consumption and make the procedure more predictable.

In recent years a certain field of Artificial Intelligence called Reinforcement Learning(RL) has grown rapidly and has proven to be able to solve problems of optimal behaviour in advanced areas on or above a human level. RL algorithms have been implemented to successfully play Atari games [5] as well as board games such as chess and Go [6]. DeepMind's AlphaGo machine has been able to not only learn to play chess and Go, but was also able to beat the reigning world champion player of Go in 2016. In addition to these discrete state- and action-space tasks performed in simulated environments, RL has been used to solve complex continuous state- and action-space tasks in real life environments such as playing table tennis [7] and flying a helicopter upside down [8].

The problem of autonomous docking is a large one consisting of several smaller problems all needing to be solved at the same time. Two of these problems are the ability to maneuver a vessel to a target position and the ability to avoid contact with obstacles. These two problems are two of the most essential parts of autonomous docking and have therefore been chosen as the focus for this thesis.

### 1.1.1 Previous work

The work in this thesis is a continuation of my research as part of a specialization project thesis[9]. In that thesis four previous works on topics related to the problem at hand were presented and analyzed. In [10] a RL algorithm for path-finding for a robotic fish with discrete state and action space was presented. [11] also proposed an improved Q-learning algorithm for a similar problem. Both these papers solved similar problems as the one at hand, however due to their state and action spaces being discrete rather than continuous their solutions were not applicable to our problem.

[12] and [13] both worked on Deep Reinforcement Learning algorithms to solve the problem of path-following in continuous state and action spaces. Although this problem is different than the one of path-finding, the fact that their algorithms worked with continuous state and action spaces made them relevant for our problem.

Based on these previous works and further research into relevant theory my specialization thesis concluded by proposing the Deep Deterministic Policy Gradient method as a possible solution to the problem of path finding and obstacle avoidance for autonomous docking. This proposal set the starting point for this thesis where the algorithm was further developed and implemented before being tested to analyze its success.

## 1.2 Problem Definition

The problem of autonomous docking is large and consists of several elements. There are several objectives to consider and detailed decisions to be made in all aspects of a solution. To achieve a full optimal solution the autonomous docking problem is unfeasible for a single masters thesis. Therefore I have chosen to break down the problem and focus on two main objectives in this thesis. The first objective is the goal of being able to maneuver from a starting position to a target position in the dock. The second objective is to perform the maneuvering without contacting any obstacles. The goal for this thesis is to attempt to solve these objectives using reinforcement learning, or more specifically the deep deterministic policy gradient method (DDPG). In other words, the thesis seeks to answer the following research questions:

1. Is it possible for a DDPG agent solve the problem of path-finding to a target position?

2. Is it possible for a DDPG agent solve the problem of obstacle avoidance in an unknown environment?

Depending on the results from this thesis it should be possible to continue to improve the solutions and potentially combine them to solve the full problem of autonomous docking in future work.

**Figure 1.1** Traditional control system architecture

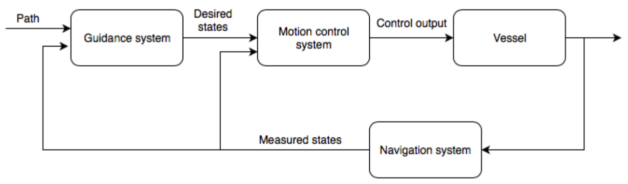
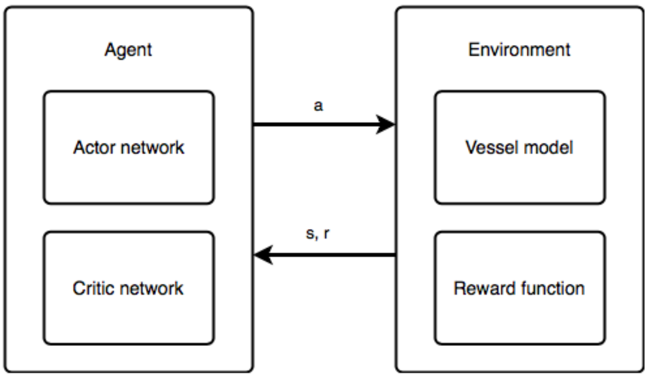


**Figure 1.2** High level overview of the DDPG path finding system



### 1.2.1 System overview

Conventional guidance control systems are usually grouped into a guidance system, motion control system and navigation system such as in figure 1.1. Add system diagram of what both would look like. For my approach since the DRL algorithm in theory can learn any task I wish to attempt to give output directly to the control units from only measuring states, thus combining all conventional modules into one black box control system such as can be seen in figure 1.2

## 1.3 Outline of the Report

This thesis is divided into 6 main chapters. After introducing the problem in chapter 1 we move on to a theoretical review in chapter 2. Here we first look at some basics of machine learning which form the foundation for more advanced methods to be introduced later in the chapter. We then look into reinforcement learning leading up to the most advanced deep reinforcement learning methods which will be used in the proposed solution to the path-finding problem. Most of the theory presented on these topics is a combination of theory from [14; 15; 16]. Chapter 2 will also introduce some basics of vessel dynamics and control for a better understanding of the overall problem. Some additional miscellaneous theory is introduced at the end of the chapter.

In chapter 3 the implementation of the solution is presented along with in depth explanation of certain choices made during the design of the solution. Chapter 4 presents the different tests performed and their results, which are then discussed further in chapter 5. Suggestions for future work in the area are also presented at the end of this chapter. Finally, in chapter 6 the thesis is concluded.

Chapter 2

# Background and theory

## 2.1 Machine Learning

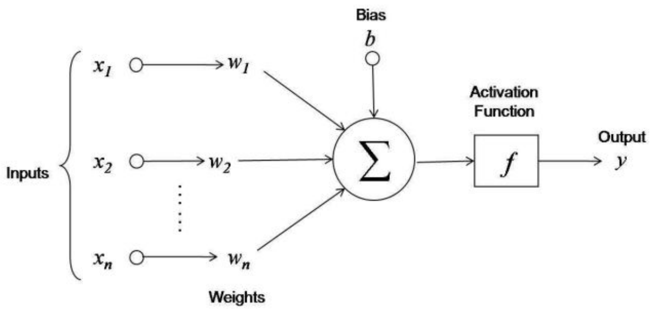
Machine Learning (ML) is the science of creating algorithms which enable a computer to learn to make decisions without being explicitly programmed how to make these decisions. In general ML can be divided into three subcategories. These are supervised learning, unsupervised learning and reinforcement learning. The proposed method in this thesis is a deep reinforcement learning method, which is a combination of RL and supervised learning. Therefore we will first look at some basics of supervised learning before moving on to reinforcement learning in section 2.2 and finally combining the two in section 2.3.

### 2.1.1 Supervised Learning

Supervised learning is a form of machine learning which is based on learning from examples. A data-set consisting of example input data and their corresponding predefined correct outputs must be provided to train a system. The system then learns by applying the example input data to the system and comparing the output to the correct outputs from the data-set. Based on the comparison the system is adjusted until sufficiently correct behaviour is achieved.

Seeing as the system is in need of a data-set with predefined correct outputs for each input we are limited to learning to solve tasks for which we are able to define a single correct behaviour for all possible inputs. A problem such as autonomous docking, where it is difficult or even impossible to calculate a single optimal solution for each possible input, is therefore not suitable to be solved using supervised learning alone. However supervised learning works well for tasks such as image classification, pattern recognition and function approximation, of which the later is of interest to us later when using deep reinforcement learning methods.

**Figure 2.1** A single artificial neuron with its inputs, weights, bias, activation function and output



When it comes to learning function approximators there are several aspects of both the input and the complexity of the desired function that dictate what kind of methods will work. For example the input can be discrete, binary or continuous and the function could be linear, nonlinear or logical. In our case we will have continuous input and wish to approximate a nonlinear function. The best methods to achieve such an approximation are through the use of Artificial Neural Networks and Deep Learning.

### 2.1.2 Artificial Neural Networks

An Artificial Neural Network (ANN) is a mathematical approximation of how a brain works. As first introduced by Frank Rosenblatt in [17], the simplest form of ANN is the perceptron. A perceptron is simply a single neuron as seen in figure 2.1. Given an input vector $\boldsymbol{x}$, the perceptron's weighting vector $\boldsymbol{w}$ and bias bias $b$, the output of a perceptron is given by the equation

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{w}^\top \boldsymbol{x} + b > 0 \\ 0 & \text{else} \end{cases} \tag{2.1}$$

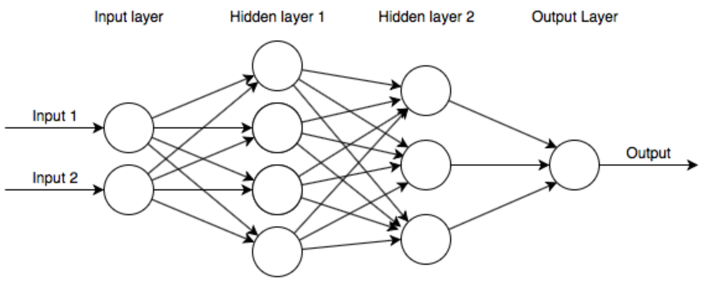
Since this is merely a detection function and the perceptron only has binary output it is not very useful. However, by connecting several such nodes in layers such that the outputs from one layer are the inputs to the next, as seen in figure 2.2, we have what we call an artificial neural network. In addition to connecting multiple neurons we have a number of different activation functions we can use which give continuous outputs and allow us to approximate nonlinear functions.

Given an input vector $\boldsymbol{x}$ we can represent the output $\boldsymbol{y}$ of a layer in an ANN as

$$\boldsymbol{y} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{2.2}$$

where $\boldsymbol{W}$ is the weighting matrix and $\boldsymbol{b}$ is the bias vector, which are our trainable parameters. We also have the activation function $f(\cdot)$ which is applied to each element of the vector resulting from $\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$. Connecting two or more such layers, in addition to an

**Figure 2.2** A fully connected Artificial Neural Network with 2 inputs, 2 hidden layers with 4 and 3 neurons respectively, and a single output



input and a output layer, we get what is commonly called a Deep Neural Network (DNN) which are used in Deep Learning.

### 2.1.3 Deep Neural Networks

As mentioned, an artificial neural network is considered to be deep if it has two or more so called hidden layers between the input and output layers. These networks work by having the output of layer $i$ serve as the input to layer $i+1$ such that $\boldsymbol{x}_{i+1} = \boldsymbol{y}_i$. This is commonly called a feed forward network. Following this principle a full feed forward DNN of $n$ layers is represented by the following equations

$$\begin{aligned} \boldsymbol{y}_1(\boldsymbol{x}) &= f_1(\boldsymbol{W}_1\boldsymbol{x} + \boldsymbol{b}_1) \\ \boldsymbol{y}_2(\boldsymbol{x}) &= f_2(\boldsymbol{W}_2\boldsymbol{y}_1(\boldsymbol{x}) + \boldsymbol{b}_2) \\ &\vdots \\ \boldsymbol{y}_n(\boldsymbol{x}) &= f_n(\boldsymbol{W}_n\boldsymbol{y}_{n-1}(\boldsymbol{x}) + \boldsymbol{b}_n) \end{aligned} \tag{2.3}$$

This formulation of the neural network leads to the property that the output of each layer may be differentiated in respect to not only the input to its own layer but also the input to any of the previous layers through the chain rule. Given a network described by the function $y = f(g(x))$ the chain rule gives us the following property.

$$\frac{\delta y}{\delta x} = \frac{\delta f}{\delta g}\frac{\delta g}{\delta x} \tag{2.4}$$

This rule can be extended in the same way to any number of composite functions. This property will later prove useful when training the network.

### 2.1.4 Deep Learning

When used in relation to neural networks the term learning refers to the process of training the neural network to perform in the desired way by adjusting its trainable parameters.

Deep learning is simply the same process for a deep neural network. This process could essentially be performed as simple trial and error, however this would prove extremely inefficient, especially for large networks. Fortunately, there are several smarter ways to perform this process. Most of these rely on the idea of a cost function which is used to measure the performance of the network by comparing the output of the network to the desired output. Given this cost function $J(\boldsymbol{\theta})$, where we have combined our weight and bias vectors into the trainable parameter vector $\boldsymbol{\theta}$, we can then compute the gradient in respect to the trainable parameters $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. We then have several methods of utilizing this gradient to adjust the trainable parameters which in turn adjust the behaviour of the network.

### Gradient descent

The gradient descent method bases the cost function on a measure of the error between the desired and actual outputs of the network. It then seeks to minimize this cost function by adjusting the parameters based on the gradient of the cost function.

Given a cost function $J(\boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are our trainable parameters and the cost function is formulated in such a way that minimizing it will minimize the output error from our network, we can now use the gradient of this cost function $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ to adjust our parameters by the following update rule

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.5}$$

Here we have introduced $\alpha$ as the learning rate which controls how large each update step is. As we can see this means that we adjust the parameters in the opposite direction of the gradient, or in other words towards the minimum of the cost function, by a factor of the learning rate. The problem now becomes finding the gradient with respect to each of the trainable parameters. Thanks to the formulation of our networks given by equation 2.3 this problem is easily solved using standard differential calculus and the chain rule.

There are several variations of gradient descent which alter 2.5 in different ways. The most straight forward is called batch gradient descent. Here the gradient is averaged over a whole episode, only updating the parameters at the end of an episode. This method leads to stable and less variant gradients, however it also increases training time due to the fact that a full episode must be simulated for a single update of the parameters.

A faster variation is called stochastic gradient descent, where the updates are made at each step based only on the gradient for the current step. This obviously increases the speed of the updates, however this increased speed is accompanied by large variations in the size of the gradients, thus compromising the stability of the training. The advantage of the higher variation of the size of the gradients is that it is more likely for the gradient to push the parameters away from a local minimum.

Mini-batch gradient descent is a combination of the two methods which allows for faster training than batch gradient descent while still having more stable training than stochastic gradient descent. When using mini-batch gradient descent the gradients are computed from a mini-batch of $N$ steps. The gradients are then averaged, just as for batch gradient descent, before being applied to the parameter updates. Given a batch size smaller than the length of a full episode the training will be faster than normal batch gradient descent. Also having batch size larger than 1 will help reduce the variance of the gradients, thus stabilizing training.

Although mini-batch gradient descent combines the advantages of both batch gradient descent and stochastic gradient descent certain difficulties remain. Given the presence of local minima there is always a chance that the gradients will not be able to move the parameters away from these local minima, resulting in sub-optimal behaviour in our network. Also, according to [18] an even more difficult problem arises with the presence of saddle points as these are often followed by plateaus where the gradients tend towards zero. Both of these problems can be minimized by using Adam optimization.

### Adam optimization

Adam optimization [19] is a form of gradient descent optimizer which utilizes the idea of momentum [20]. This idea is, as its name implies, inspired by the physical concept of momentum which causes an object in motion to seek to remain in motion when external forces are applied. For the gradient descent adoption of the term we take previous gradients from earlier steps and add a factor of them to our new gradients. By doing so we keep moving in the same direction as in previous gradients, just as physical momentum would keep a ball rolling even when a force is applied.

Following the notation of [19] we have our gradient $\boldsymbol{g_t} = \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$. Next we have our first and second moment vectors $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$ defined as

$$\begin{aligned} \boldsymbol{m}_t &= \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \\ \boldsymbol{v}_t &= \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2 \end{aligned} \tag{2.6}$$

Here $\beta_1$ and $\beta_2$ are the decay rates of the previous moments. According to [19] these moment terms are biased to the initial values of $\boldsymbol{m}_t$ and $\boldsymbol{v}_t$, therefore a term is added as follows to correct for these biases.

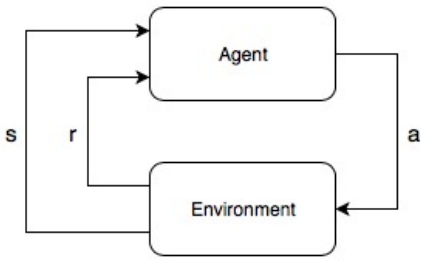
$$\begin{aligned} \hat{\boldsymbol{m}}_t &= \frac{\boldsymbol{m}_t}{1-\beta_1^t} \\ \hat{\boldsymbol{v}}_t &= \frac{\boldsymbol{v}_t}{1-\beta_2^t} \end{aligned} \tag{2.7}$$

Using these bias corrected moments we now have the update rule for our parameters

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon} \hat{\boldsymbol{m}}_t \tag{2.8}$$

where $\alpha$ is our step-size and $\epsilon$ a small constant for ensuring numerical stability by avoiding the possibility of dividing by zero. Utilizing this optimizer we now have a way of overcoming local minima as well as saddle points and plateaus.

**Figure 2.3** Illustration of interactions between a RL agent and environment. A state $s$ and reward $r$ are observed by the agent which applies an action $a$ resulting in a new observation of a state and reward.



## 2.2 Reinforcement Learning

So far we have looked at machine learning methods which are able to learn to perform tasks by learning from a data set of predefined correct behaviour. For some problems, however, it is not always trivial or even possible to provide such a data set. Such problems can still be solved by using Reinforcement Learning (RL). Rather than using example data to know what is the desired behaviour for a system, reinforcement learning has a trial and error based approach. Here the learning agent must interact with its environment before analyzing the result of its interactions to evaluate the quality of its choices. These new methods require a number of different elements and a different way of formulating a problem which we will now look at more in depth.

### 2.2.1 Elements of Reinforcement Learning

In a reinforcement learning problem the system is divided into two main parts, the agent and environment. The agent is the core of the system where the decisions are made, while the environment is everything around the agent which affects the result of each decision the agent makes. We call these decisions actions. For each action the agent makes an observation of the environment. This observation includes the state of the system and a reward. Based on these observations the agent then chooses a new action attempting to maximize the new reward. This process, as illustrated in figure 2.3, is repeated at each time step with the agent continuously learning from its transitions.

The environment and agent in RL systems can be broken down further into four main elements. These are the reward signal, value function, policy and a model. Each of these play important parts in how the overall system works.

### Reward signal

The reward signal is the part of the environment which gives feedback to the agent on the outcome of its action. The value of this signal is calculated by a reward function. The main goal of any RL agent is to maximize the total reward accumulated over time, thus the reward function ultimately defines the agents objective. It is therefore a crucial element of any RL system and must be considered carefully when designing the system. The complexity of the reward function varies depending on the complexity of the task the agent is meant to perform. Often the reward function will include a main component corresponding to the main objective in addition to several additional components corresponding to sub-objectives one might add as a way of defining the objective more detailed.

### Value function

The value function gives us a way of determining the value of being in a certain state. This value is calculated based on the maximum reward which can be accumulated over time after arriving at the given state. Based on this the value function can be used to determine the best action available at any given time. In addition to the normal value function $V(s)$, also called the state value function, we have what is called the action value function $Q(s, a)$ which calculates the value of taking action $a$ from state $s$ then following the given policy. The most common way to calculate the value function is using the Bellman equation which we will introduce later.

### Policy

The policy of a RL system can be seen as a set of rules deciding what actions to take when in a given state. We denote the policy $\pi(s)$. For a policy to be optimal it must have the property of always choosing the action which leads to the highest accumulated future reward. To achieve this the policy is often based on the value or action value function in such a way that it chooses the action which leads to the state with the highest value. In the case of the action value function this simply means choosing the action with highest value.

### Model

The final element of a RL system is a model of the environment in the form of a transition function. This transition function is meant to estimate the behaviour of the environment. This makes it easier for the agent to predict the future rewards, thereby finding the optimal policy. Not all RL systems have a model available, making them pure trial-and-error systems which have no knowledge of the behaviour of the environment until interacting with it. This is in fact most common since many environments are difficult to accurately represent mathematically.

### 2.2.2 Markov Decision Process

A Markov decision process (MDP) is a special case of a sequential decision process. For a MDP we have a fully observable stochastic environment where all states satisfy the Markov property. The Markov property states that given the current state all future states are independent of past states. This property allows us to make decisions solely based on our current state, allowing us to neglect all past states. Ultimately this property greatly simplifies the calculations needed for each decision as well as saving enormous memory space.

We denote a MDP as a tupple $(S, A, R, T)$. Here $S$ is the set of all states $s$, $A$ the set of actions $a$, $R$ the reward function and $T$ the transition function. Since the set of available actions in most cases depends on the current state we often denote $A$ as $A(s)$ where $s \in S$. Similarly the reward is also dependant on the current state and the action taken, thus we denote $R(s, a)$ where $s \in S$ and $a \in A$. The transition function is a probability function which gives the probability of the next state being $s'$ given the current state $s$ and the chosen action $a$.

$$T(s, a, s') = P(s'|s, a) \tag{2.9}$$

It must fulfill the criteria $T(s, a, s') \in [0, 1]$ and $\sum_{s'} T(s, a, s') = 1$. Using this formulation of a decision process we can now use the Bellman equation to calculate our value functions.

### 2.2.3 Bellman equation

As mentioned earlier the Bellman equation is used to calculate the value of a certain state. This equation is based on Richard 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.[21]

For a sequential decision process, such as a MDP, this means that for each decision that must be made we can disregard all previous decisions leading up to this point. It also allows us to reformulate the problem as a recursive process of choosing the optimal action at each time. This is the basis of what we call the Bellman equation.

We express the value of a state as the expected sum of rewards accumulated after starting at the given state. This can be written as

$$V(s_0) = E\Big\{\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\Big\} \tag{2.10}$$

where $s_0$ is the current state and $s_t$ and $a_t$ are the state at time $t$ and the action taken from this state. We have also introduced a discount factor $\gamma \in [0, 1]$ which minimizes the effect of the accumulated rewards far in the future, thereby also introducing a sense of urgency in the decision problem.

Knowing that the transition function $T(s, a, s')$ can be formulated as a probability function $P(s'|s, a)$, returning the probability of ending up in state $s'$ when taking action $a$ from state $s$, we can rewrite the expected sum as

$$\begin{aligned}
V(s_0) &= \sum_{t=0}^{\infty} T(s_t, a_t, s_{t+1})\gamma^t R(s_t, a_t, s_{t+1}) \\
&= T(s_0, a_0, s_1)R(s_0, a_0, s_1) + \sum_{t=1}^{\infty} T(s_t, a_t, s_{t+1})\gamma^t R(s_t, a_t, s_{t+1}) \qquad (2.11) \\
V(s) &= \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V(s'))
\end{aligned}$$

where we have simplified the current state and action $s_0$ and $a_0$ to $s$ and $a$, and the next state $s_{t+1}$ to $s'$. We have also used the principle of optimality to give our value function a recursive property. Now we wish to find the maximum value over time. This can be done by always selecting the action in the current state which results in the highest value in the next state. We then have our Bellman optimality equation in the following form:

$$V^*(s) = \max_{a \in A(s)} \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V^*(s')) \qquad (2.12)$$

Similarly the Bellman optimality equation when using the action value function $Q(s, a)$ is given by

$$Q^*(s, a) = \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma \max_{a'} Q * (s', a')) \qquad (2.13)$$

Using 2.12 or 2.13 we can now find the value function $V(s)$ or action value function $Q(s, a)$ which can be used with dynamic programming to find an approximation of the optimal policy.

### 2.2.4 Dynamic Programming

Dynamic programming (DP) is a way of breaking down a large task into smaller repeatable tasks. In RL this method can be used to find an optimal policy for a MDP. Given a MDP which perfectly models the environment, in other words has full knowledge of the transfer-function $T(s, a, s')$, we can use a method called policy iteration to find an optimal policy.

#### Policy iteration

Policy iteration is an iterative computation divided into two sub-problems, policy evaluation and policy improvement, which themselves are also iterative computations. The process starts by arbitrarily initializing a policy $\pi(s)$ and value function $V(s)$. Next the policy evaluation will determine the value-function for the given policy. This is done by recursively using the Bellman equation to calculate the value at each state when following policy $\pi(s)$. Full sweeps through all states are done until the maximum change in value for any state is lower than a small predefined value. Smaller values will ensure higher accuracy in the estimate of the value-function, but will in turn increase the computational time.

Once the estimate of the value-function is computed we move on to policy improvement. Here we consider if the given policy is in fact optimal in relation to its calculated value-function. This is done by comparing our policy to the greedy policy for our value-function. By greedy policy we mean a policy which always chooses the action leading to the highest value from the current value function. Given that our policy is different than the greedy policy we go back to policy evaluation to repeat the process, however this time we start with the greedy policy computed during policy improvement and the value-function computed in the previous policy evaluation.

$$\pi_0 \xrightarrow{e} v_{\pi_0} \xrightarrow{i} \pi_1 \xrightarrow{e} v_{\pi_1} \xrightarrow{i} \pi_2 \xrightarrow{e} v_{\pi_2} \cdots \xrightarrow{i} \pi_* \xrightarrow{e} v_{\pi_*}$$

This process of repeatedly computing value-functions for our policy and improving our policy based on the value-function guarantees that each policy will be an improvement of the previous until it is in fact optimal. This is illustrated above where $\xrightarrow{e}$ represents a policy evaluation and $\xrightarrow{i}$ represents a policy improvement. Complete pseudo code of the process is shown in algorithm 1.

#### General Policy Iteration

Through policy iteration we see a sequence of policy evaluations and policy improvements interacting. Each of these processes has its own goal which in a way work against each other. The policy evaluation attempts to make the value-function accurate in respect to the given policy, while the policy improvement attempts to make the policy greedy in respect to the given value-function. However since there exists a point where both these goals can be achieved, namely for the optimal value-function and policy, the interaction between the two processes causes them to work together to reach this optimal point. INSERT FIGURE HERE!!!!! This idea of having the policy evaluation and policy iteration interact is called General Policy Iteration (GPI). GPI is a central term in almost all forms of RL, not only DP, since they have both value-functions and policies which are attempting to improve themselves in respect to the other.

### 2.2.5 Monte Carlo methods

Monte Carlo methods differ from DP in two main ways. First of all they base their operation on sample experience, rather than predictions produced from a model. Second they do not use other value estimates as their basis for updating the new value estimates. Both these things come from the fact that in Monte Carlo methods the value function and policy are only updated after the completion of a full episode. The most basic form of MC

**Algorithm 1** Policy Iteration

```
 1: procedure INITIALIZATION
 2:     $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily, commonly $V(s) = 0$
 3: end procedure
 4: procedure POLICY EVALUATION
 5:     repeat
 6:         $\Delta \leftarrow 0$
 7:         for each $s \in S$ do
 8:             $v \leftarrow V(s)$
 9:             $V(s) \leftarrow \sum_{s'} T(s, \pi(s), s')(R(s, a, s') + \gamma V(s'))$
10:             $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
11:         end for
12:     until $\Delta \leq \theta$
13: end procedure
14: procedure POLICY IMPROVEMENT
15:     $policyStable \leftarrow true$
16:     for each $s \in S$ do
17:         $oldaction \leftarrow \pi(s)$
18:         $\pi(s) \leftarrow argmax_a \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V(s'))$
19:         if $\pi \neq oldaction$ then $policyStable \leftarrow false$
20:         end if
21:     end for
22:     if $policyStable$ then return $V \approx v_*$ and $\pi \approx \pi_*$
23:     else
24:         goto Policy Evaluation
25:     end if
26: end procedure
```

method is the constant-alpha method described by the following function:

$$V(s) \leftarrow V(s) + \alpha[G_t - V(s)] \tag{2.14}$$

Here $\alpha$ is the learning rate and $G_t$ is the sum of all returns in the given episode,

$$G_t = \sum_{t=0}^{T} R(s, a, s') \tag{2.15}$$

where $T$ is the total number of time-steps in an episode. We see from equation 2.14 that the value of a given state $s$ will be updated by a factor of the difference in the accumulated reward $G_t$ and the current value for the same state. Therefore we call $G_t$ our target, since the value function will converge as $V(s) \to G_t$. This idea of having a target which we use in our update rule is central in the methods to be described later. Also since our target is based on the actual returns during the episode we say that the learning is based on experience.

MC methods have certain advantages over DP methods. For example basing the operation on sample experience allows for simulation based training. This also means that we do not need to know the transition function $T(s, a, s')$ to compute our value function. However there are also certain disadvantages related to MC methods. The fact that the updates only are made after full episode completions means that the training time necessary for convergence to the optimal value function and policy is dramatically increased. Although this makes MC methods less desirable there are ways to combine elements from MC and DP to take advantage of the positive sides of each of them. These methods are called Temporal Difference methods.

### 2.2.6 TD methods

Temporal Difference methods are similar to MC methods in that they use real experience as the target in the update rule, however where MC methods must wait for an entire episode to complete before having access to this experience, TD methods rather extract the experience over a certain number of time-steps several times throughout an episode. The simplest form of TD methods observes the accumulated reward at each time-step, resulting in the following equation:

$$V(s) \leftarrow V(s) + \alpha[R_{t+1} + \gamma V(s') - V(s)] \tag{2.16}$$

Here we see that the target is the sum of the observed reward $R_{t+1}$ and the estimate $V(s')$. This expression is call TD(0). A more general expression for TD methods is more similar to 2.14

$$V(s) \leftarrow V(s) + \alpha[G_t^n - V(s)] \tag{2.17}$$

where $G_t^n$ is the accumulated reward over n timesteps given by

$$G_t^n = \sum_{t=0}^{n} \gamma^t R(s, a, s') \tag{2.18}$$

By doing this we have successfully combined elements of DP methods and MC methods. Temporal difference methods also introduce us to the term *temporal difference error*. This value is given by

$$\delta \doteq R_{t+1} + \gamma V(s') - V(s) \tag{2.19}$$

This expression can be recognized from the expression for TD(0) and is the difference between the estimated value of state $s$ and the better estimate $R_{t+1} + \gamma V(s')$, which is a better estimate since we have the exact value for the reward accumulated at time-step $t+1$ combined with the value estimate of all future rewards after this time. In other words, the temporal difference error is the difference between our estimated state value and our target. This term is used in several forms of reinforcement learning, as we will see later.

### 2.2.7 Q-learning

Q-learning is a special form of temporal difference method where rather than updating the state value $V(s)$, the action value $Q(s,a)$ is used. This gives us the following update rule for Q-learning

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big[R_{t+1} + \gamma \max_{a'} Q(s',a') - Q(s,a)\big] \tag{2.20}$$

Given sufficient exploration this action value function will converge towards it optimal value. Following the convergence we will have the optimal policy $\pi^*(s)$ by following a greedy policy with respect to the action value function.

One of the main advantages of Q-learning is that it is an off-policy method. This means that it is able to learn a policy without having to follow it. The reason for this is that during the learning phase a Q-learning agent will always choose the action with the highest Q-value, regardless of the policy being used to calculate the Q-values of the next state in the update rule. This makes exploration significantly more achievable since we are not restricted to follow the policy being learned.

### 2.2.8 Multi-Objective Reinforcement Learning

As mentioned earlier, the reward function in reinforcement learning is used to express/comunicate the objective to the agent. It has also been mentioned that it is possible to combine several elements into this reward function each representing a sub-objective. Although this works well in theory recent studies show that this is not always the best way to implement multiple objectives. An alternate approach is multi-objective reinforcement learning [22].

For any system with $n$ objectives, multi-objective reinforcement learning separates the different objectives into $n$ individual reward functions denoted $R_i$. The return from these are then passed as a reward vector $\boldsymbol{R} = [R_1, R_2, ..., R_n]$. These rewards are used to compute a corresponding value function vector $\boldsymbol{Q} = [Q_1, Q_2, ..., Q_n]$ where $Q_i$ is the value function for objective $i$ based on the reward $R_i$. This way we have a measure of the value of following a policy from a state in respect to each of the objectives.

Given these values there are many ways of computing an optimal policy in respect to the separate objectives.The simplest of these is called the weighted sum approach. Given the value function vector $\boldsymbol{Q}$ consisting of value functions for $n$ objectives we combine them into a synthetic value function using a weight vector $\boldsymbol{w}$, with all elements $w_i \in [0,1]$ and $\sum_i^n w_i = 1$, such that

$$TQ(s,a) = \sum_{i=1}^{n} w_i Q_i(s,a) \tag{2.21}$$

This synthetic value function is then used just as a normal value function to generate an optimal policy. Although this may seem to generate very similar value function as would be expected by combining all objectives in a single reward there is a significant difference brought forth through the weighting vector. Adjusting the weights in this vector gives us the power to decide which objectives are more important by assigning them higher weights relative to less important objectives. In this way the corresponding value function of the most important objective will influence the synthetic value function more, leading to the policy being most optimal in respect to this objective. This is especially significant when using value function approximation which will be introduced in the next section.

## 2.3 Deep Reinforcement Learning

The Reinforcement Learning methods discussed to this point all assume we have the ability to represent values and policies for all states and state-action pairs in the form of a lookuptable. This is unfortunately not very realistic in the real world. First of all this assumption ties us to a limited finite set of states and state-action pairs. We also encounter the “curse of dimensionality” [23], which means that for every new state or state-action pair the tables in which the values and policies are stored grow exponentially, resulting in the computational time becoming too long to be implemented in any real world case.

Fortunately there exist methods to approximate values and policies for all states and state-action pairs based on only a subset of these states and state-action pairs. This is called function approximation and can be solved using deep learning. By setting up ANNs to approximate a value function and policy based on a subset of the state-action pairs, then using these approximations in RL methods we have what is called Deep Reinforcement Learning (DRL).

There are three main types of DRL methods. These are called actor-only, critic-only and actor-critic methods. Here the term actor refers to a policy $\pi(s)$ and critic refers to value function $V(s)$ or in some cases the action-value function $Q(s,a)$. Based on this we understand that actor-only methods learn only the policy, critic-only methods learn only the value function and actor-critic methods learn both. Both the actor-only and actor-critic method work for problems with continuous action spaces, while the critic-only method only works for discrete action spaces. This is due to the fact that it only has the value function available and must therefore search through all possible actions to find the one with greatest outcome. Since our problem exists in the continuous action space we will only focus on actor-only and actor-critic methods.

### 2.3.1 Actor-only methods

#### Policy gradient methods

Policy gradient methods [24] are on-policy actor-only methods which seek to approximate the optimal policy $\pi^*(s)$. This is done by using gradient descent to improve the approximated policy with respect to the expected return. We begin by parameterizing the policy by a vector $\boldsymbol{\theta}$ which contains the weights and biases of the policy network. We then assume that the policy $\pi_{\boldsymbol{\theta}}(s)$ can be formulated as a probability distribution giving the probability of taking action $a$ given state $s$ such that $\pi_{\boldsymbol{\theta}}(s) = \pi_{\boldsymbol{\theta}}(a|s)$. Further, since we know that the main objective of any RL agent is to maximize the total future reward, we choose the following cost function

$$J(\boldsymbol{\theta}) = E\Big[\sum_t \sum_{a_t \sim \pi_{\boldsymbol{\theta}(s_t)}} \gamma^t R(s_t, a_t)\Big] \tag{2.22}$$

where $a_t$ is the action sampled at time $t$ following the policy $\pi_{\boldsymbol{\theta}}$. The problem of maximizing this cost function can be reformulated as finding the optimal parameters $\boldsymbol{\theta}^*$ which correspond to the policy giving actions which maximize the expected reward.

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, J(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, E\Big[\sum_t \sum_{a_t \sim \pi_{\boldsymbol{\theta}(s_t)}} \gamma^t R(s_t, a_t)\Big] \tag{2.23}$$

We now use gradient descent, or rather ascent since we are looking to maximize the cost function, to find these optimal parameters with the following formula

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.24}$$

The problem now becomes finding the gradient of the cost function. There are two main ways to do this, finite difference approximation and direct policy differentiation. Of these the later is most commonly used as the development of deep neural networks allows for efficient and accurate calculation of the gradient.

#### Direct policy differentiation

As the name of the method suggests we are going to differentiate the policy directly on our way to finding the gradient of our cost function. Although this might seem complicated there are several factors simplifying the process allowing us to make the differentiation process relatively simple.

To begin we utilize the fact that our policy is represented by a function approximation generated by a neural network. As we know it is quite straight forward to calculate the gradients in a neural network by using the chain rule and backpropagation, thus we can assume that the gradient $\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(s, a)$ is known. Also since we have assumed the policy is a probability distribution we then know that

$$\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(s, a) = \pi_{\boldsymbol{\theta}}(s, a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s, a) \tag{2.25}$$

We can then use standard differential calculus to find the gradient of the objective function to be

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = E\Big[\Big(\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_t, a_t)\Big(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}\Big)\Big)\Big] \tag{2.26}$$

For simplicity we have written $r_{t'}$ for the reward $R(s_{t'}, a_{t'})$. We can also utilize a similar equation in a Monte Carlo approach. Here we will simulate multiple episodes following a certain policy of which we can calculate the gradients. By taking the average of these gradients we have a new approximation of the gradient for the given policy. By denoting the number of episodes as $N$ and number of steps in each episode as $T$ this can be written as

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{N}\sum_{i=1}^{N}\Big(\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_{i,t}, a_{i,t})\Big(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}\Big)\Big) \tag{2.27}$$

From this equation we see that the gradient is given by the product of the total return and the gradient of the probability of taking the actions leading to this return. Intuitively this means that the gradients will be highest in the direction leading to the highest return. We will therefore be moving towards the actions leading to maximum return during gradient ascent, ultimately leading us to the optimal policy.

We now have a gradient we know will lead us to the optimal policy, however since it is calculated using a Monte Carlo approach we might encounter a quite high variance. Using a method known as REINFORCE [25] this can be reduced by subtracting a baseline from the gradients. There are multiple baselines which will suffice, of which the best is usually to use an approximation of the value function $b = V(s_{i,t})$. The gradient is then given by

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{N}\sum_{i=1}^{N}\Big(\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_{i,t}, a_{i,t})\Big(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} - b\Big)\Big) \tag{2.28}$$

### 2.3.2 Actor-Critic methods

For actor-critic methods we now have an added function approximator for our value function. To distinguish between the two we now denote the parameters of the actor network $\boldsymbol{\theta}_a$ and for our critic network $\boldsymbol{\theta}_c$. Having distinguished the two we now wish to find a similar method of using gradient descent on our critic function approximator as we used for our actor function approximator. To do this we first need to define a cost function. A natural starting point is to use the temporal difference error $\delta$ given by

$$\delta = R(s, a, s') + \gamma V_{\boldsymbol{\theta}_c}(s') - V_{\boldsymbol{\theta}_c}(s) \tag{2.29}$$

where $V_{\boldsymbol{\theta}_c}(s)$ is the value function approximation represented by parameters $\boldsymbol{\theta}_c$. As discussed in chapter 2.2.6 this error will tend towards 0 as our approximation approaches the actual optimal value function $V^*(s)$, thus our goal of approximating the optimal value function is equal to the goal of minimizing the square of the temporal difference error. We therefore set our cost function to be

$$J_c(\boldsymbol{\theta}_c) = \frac{1}{2}\delta^2 \tag{2.30}$$

This gives us the following gradient

$$\nabla_{\boldsymbol{\theta}_c} J(\boldsymbol{\theta}_c) = \delta \nabla_{\boldsymbol{\theta}_c} \delta = \delta \nabla_{\boldsymbol{\theta}_c} V_{\boldsymbol{\theta}_c}(s) \tag{2.31}$$

We have here made the assumption that the value function estimate of the next state is independent of the parameters $\boldsymbol{\theta_c}$, thereby allowing us to treat it as a constant when calculating the gradient. Our cost function and its corresponding gradient give us the update law for our critic network as follows

$$\boldsymbol{\theta_c} \leftarrow \boldsymbol{\theta_c} - \alpha_c \delta \nabla_{\boldsymbol{\theta_c}} V_{\boldsymbol{\theta_c}}(s) \tag{2.32}$$

Going back to our actor network we can make a few adjustments to equation 2.28. First of all, by using the value function approximation as our baseline it can be shown that

$$\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} - b = \delta \tag{2.33}$$

Further, if we use a batch size of 1 and a one-step horizon we can simplify the entire gradient to

$$\nabla_{\boldsymbol{\theta}_a} J(\boldsymbol{\theta}_a) \approx \delta \nabla_{\boldsymbol{\theta}_a} \log \pi_{\boldsymbol{\theta}_a}(s) \tag{2.34}$$

resulting in our new update law for our actor network

$$\boldsymbol{\theta}_a \leftarrow \boldsymbol{\theta}_a + \alpha_a \delta \nabla_{\boldsymbol{\theta}_a} \log \pi_{\boldsymbol{\theta}_a}(s) \tag{2.35}$$

### 2.3.3 Deep Deterministic Policy Gradient

The Deep Deterministic Policy Gradients (DDPG) method [4] is an adapted actor-critic method which utilizes elements of Q-Learning. Rather than approximating the value function $V(s)$ as in most actor-critic methods, it approximates the action value function $Q(s, a)$. Just as for Q-Learning this makes the algorithm off-policy. We therefore have the advantage of being able to learn the optimal policy while following any arbitrary policy. This in turn allows us to perform better exploration while still increasing the learning speed.

We start with our action-value function $Q_{\boldsymbol{\theta}_Q}(s, a)$ and our policy $\pi_{\boldsymbol{\theta}_\pi}(s)$ parameterized by respectively $\boldsymbol{\theta}_Q$ and $\boldsymbol{\theta}_\pi$. Similarly to the actor-critic algorithm we use the squared of the temporal difference error $\delta_t$ as our cost function for the critic network.

$$J_Q(\boldsymbol{\theta}_Q) = \frac{1}{2}\delta_t^2 = \frac{1}{2}\big(r + Q_{\boldsymbol{\theta}}(s', \pi_{\boldsymbol{\theta}_\pi}(s')) - Q_{\boldsymbol{\theta}_Q}(s, a)\big)^2 \tag{2.36}$$

Moving on to our actor network we can simplify our cost function to the squared of our action-value function

$$J_\pi(\boldsymbol{\theta}_\pi) = \frac{1}{2} Q_{\boldsymbol{\theta}_Q}(s, \pi_{\boldsymbol{\theta}_\pi})^2 \tag{2.37}$$

This can be done since we know that our action-value function gives the vale of each action in each state. Knowing this we then know that our optimal policy is found by always taking the action which gives the highest action-value. By maximizing the above cost function and using its gradient to update our actor network parameters using gradient ascent we will be adjusting our actor network towards the optimal policy for the given action-value function approximation. At the same time the action-value function approximation network is being adjusted towards the optimal action-value function for the current policy approximation. As described earlier this will eventually lead to finding both our optimal policy and its optimal action-value function.

Using these cost functions we now have the following gradients for our critic and actor networks:

$$\begin{aligned}\nabla_{\boldsymbol{\theta}_Q} J_Q(\boldsymbol{\theta}_Q) &= -\delta \nabla_{\boldsymbol{\theta}_Q} Q_{\boldsymbol{\theta}_Q}(s,a)\\ \nabla_{\boldsymbol{\theta}_\pi} J_\pi(\boldsymbol{\theta}_\pi) &= \nabla_{\boldsymbol{\theta}_\pi} Q_{\boldsymbol{\theta}_Q}(s,\pi_{\boldsymbol{\theta}_\pi}(s))\\ &= \nabla_a Q_{\boldsymbol{\theta}_Q}(s,a) \nabla_{\boldsymbol{\theta}_\pi} \pi_{\boldsymbol{\theta}_\pi}(s)\end{aligned} \tag{2.38}$$

In addition to providing new simplified gradient, the DDPG method also adds several methods of stabilizing training. Two of these are soft parameter updates and experience replay.

### Soft parameter updates

Soft parameter updates [26] are a method of improving the stability of the training of our neural networks $Q_{\boldsymbol{\theta}_Q}$ and $\pi_{\boldsymbol{\theta}_\pi}$. This is done by using an extra set of neural networks which we call the target networks. These networks have parameters $\boldsymbol{\theta}_{Q'}$ and $\boldsymbol{\theta}_{\pi'}$. At the start of training we set these parameters equal to those of our actual action value and policy approximators. When training our networks we use these target networks to calculate our target values to be used in our temporal difference errors. These targets are then used to compute the gradients for training our actual critic network, which in turn is used to compute the gradients for training our actual actor network. Once our actual actor and critic networks are trained we update our target networks by the following update rule:

$$\begin{aligned}\boldsymbol{\theta}_{Q'} &= (1-\tau)\boldsymbol{\theta}_{Q'} + \tau\boldsymbol{\theta}_Q\\ \boldsymbol{\theta}_{\pi'} &= (1-\tau)\boldsymbol{\theta}_{\pi'} + \tau\boldsymbol{\theta}_\pi\end{aligned} \tag{2.39}$$

Here $\tau$ is our target network update rate which dictates how fast the target networks change. By keeping $\tau$ small we will have slowly changing target networks which in turn will lead to more stable target values when computing our gradients for updating our actual actor and critic networks.

This method is made possible by the fact that DDPG is an off-policy method since we are following one policy, the one generated by our actual actor network, while using another, the one generated by our target actor network, to perform our training. This leads to our agent being able to explore by following the faster changing policy while keeping the training stable due to the slowly changing targets.

#### Experience replay

Experience replay was first introduced by [27]. The intent of this method is to speed up what Lin called the “credit/blame propagation” and to give the agent a chance to re-learn behaviour from past experiences. The so-called credit/blame propagation is the process of the reward from a certain action resulting in a change of the behaviour of the agent. Experience replay works in the following way. At each step an action is taken resulting in a new state and observed reward. This set of the starting state, action, next state and reward, commonly called an experience, is then stored to a replay buffer. Next when the network is to be trained, rather than only performing the training based on the most recent experience we sample a number of random experiences from the replay buffer. The training is then based on multiple experiences at each step of training.

The result of this process is that we achieve a larger set of training data for a set number of simulations, thus increasing the relative speed of training in regards to the number of episodes needed for the networks to be sufficiently trained. This also helps generalize the networks since the experiences are used multiple times, thus refreshing what has already been learned.

#### Overall algorithm

Combining the methods of stabilizing training and our new gradients we have the full algorithm for DDPG shown in algorithm 2. We have also added actor noise to improve the exploration of the agent during training. This is again made possible thanks to the algorithm being off-policy.

## 2.4 Marine vessel kinematics

The vessel kinematics are a central part of designing most conventional control systems. In our proposed new method however, the main idea is for the DRL agent to be able to learn an optimal control sequence without prior knowledge of the system. Although this in theory allows us to be less dependant on understanding the vessel dynamics than in more conventional methods we can still improve our system by making better design decisions, for example when it comes to state augmentation, with knowledge of the kinematics. Also, to be able to accurately simulate the vessel dynamics it is necessary to have a full kinematic model of the vessels. We will therefore now look at the kinematics of marine vessels to better understand the overall system.

From [28] we can use the SNAME 1950 notation for a marine vessel with 6 degrees of freedom (DOF) found in table 2.1 to define the pose and velocity vector respectively as $\boldsymbol{\eta} = [x, y, z, \phi, \theta, \psi]'$ and $\boldsymbol{\nu} = [u, v, w, p, q, r]'$. Since we are only interested in the 2-dimensional surface movement of the vessel we can simplify to 3-DOF only using surge,

**Algorithm 2** Deep Deterministic Policy Gradient with experience replay and soft parameter updates

```
 1: Initialize actor $\pi_{\boldsymbol{\theta}_\pi(s)}$ and critic $Q_{\boldsymbol{\theta}_Q}(s,a)$ networks randomly with weights $\boldsymbol{\theta_\pi}$ and $\boldsymbol{\theta}_Q$.
 2: Initialize target networks $\boldsymbol{\theta}_{\pi'} \leftarrow \boldsymbol{\theta}_\pi, \boldsymbol{\theta}_{Q'} \leftarrow \boldsymbol{\theta}_Q$
 3: Initialize replay buffer
 4: for each episode from 1 to $M$ do
 5:     Initialize random process $\mathcal{N}$ for action exploration
 6:     Receive initial observation state $s_1$
 7:     for each $t$ from 1 to $T$ do
 8:         Select action $a_t = \pi_{\boldsymbol{\theta}_\pi}(s_t) + \mathcal{N}_t$
 9:         Perform action $a_t$
10:         Observe $s_{t+1}$ and $r_t$
11:         Save experience/transition $(s_t, a_t, r_t, s_{t+1})$ in replay buffer
12:         Sample N transitions $(s_t, a_t, r_t, s_{t+1})$ from replay buffer
13:         Set $y_i = r_i + \gamma Q_{\boldsymbol{\theta}_{Q'}}(s_{i+1}, \pi_{\boldsymbol{\theta}_{\pi'}}(s_{i+1}))$ for $i \in 1 \cdots N$
14:         Update critic by minimizing loss: $\frac{1}{N}\sum_i \left(y_i - Q_{\boldsymbol{\theta}_Q}(s_i, a_i)\right)^2$
15:         Update policy with: $\frac{1}{N}\sum_i \nabla_{u_i} Q_{\boldsymbol{\theta}_Q}(s_i, a_i) \nabla_{\boldsymbol{\theta}_\pi} \pi_{\boldsymbol{\theta}_\pi}(s_i)$
16:         Update critic target network: $\boldsymbol{\theta}_{Q'} = (1-\tau)\boldsymbol{\theta}_{Q'} + \tau\boldsymbol{\theta}_Q$
17:         Update actor target network: $\boldsymbol{\theta}_{\pi'} = (1-\tau)\boldsymbol{\theta}_{\pi'} + \tau\boldsymbol{\theta}_\pi$
18:     end for
19: end for
```

sway and yaw so that

$$\boldsymbol{\eta} = \begin{bmatrix} x \\ y \\ \psi \end{bmatrix} \boldsymbol{\nu} = \begin{bmatrix} u \\ v \\ r \end{bmatrix} \tag{2.40}$$

We then have from [28] the model of the 3-DOF system on vectorial form as

$$\dot{\boldsymbol{\eta}} = \boldsymbol{R}(\psi)\boldsymbol{\nu} \tag{2.41}$$

$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} = \boldsymbol{\tau} \tag{2.42}$$

Here $\boldsymbol{M} \in \mathbb{R}^{3\times3}$ is the inertial matrix, $\boldsymbol{C}(\boldsymbol{\nu}) \in \mathbb{R}^{3\times3}$ the Corriolis matrix, $\boldsymbol{D}(\boldsymbol{\nu}) \in \mathbb{R}^{3\times3}$ the added mass matrix and $\boldsymbol{\tau}$ the control input vector. $\boldsymbol{R}(\psi) \in SO(3)$ is the rotational matrix given by

$$\boldsymbol{R}(\psi) = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.43}$$

Although this 3-DOF model is sufficient for modeling 2-dimensional surface movement, the addition of a 4th degree of freedom, namely the roll dynamics, can further improve the model due to the fact that the roll dynamics are often strongly coupled to the sway and yaw dynamics. By adding the pitch angle $\phi$ to the state vector we have

$$\boldsymbol{\eta} = \begin{bmatrix} x \\ y \\ \psi \\ \phi \end{bmatrix}, \boldsymbol{\nu} = \begin{bmatrix} u \\ v \\ r \\ p \end{bmatrix} \tag{2.44}$$

We can then write the 4-DOF system follows:

$$\dot{\boldsymbol{\eta}} = \boldsymbol{J}(\boldsymbol{\eta})\boldsymbol{\nu} \tag{2.45}$$

$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} = \boldsymbol{\tau} \tag{2.46}$$

Here $\boldsymbol{J}(\boldsymbol{\eta})$ is the new rotation matrix given by

$$\boldsymbol{J}(\boldsymbol{\eta}) = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & -sin(\phi) \\ 0 & sin(\phi) & cos(\phi) \end{bmatrix} \tag{2.47}$$

**Table 2.1:** SNAME (1950) notation for 6-DOF marine vessels

| DOF | | Forces/ Moments | Velocities | Positions/ Angles |
|---|---|---|---|---|
| 1 | Surge | $X$ | $u$ | $x$ |
| 2 | Sway | $Y$ | $v$ | $y$ |
| 3 | Heave | $Z$ | $w$ | $z$ |
| 4 | Roll | $K$ | $p$ | $\phi$ |
| 5 | Pitch | $M$ | $q$ | $\theta$ |
| 6 | Yaw | $N$ | $r$ | $\psi$ |

## 2.5 Tools and libraries

The Python 3.6 programming language was used to implement all algorithms and models used in the work described in this thesis. The automatic differentiation library Tensorflow [2] was used to implement the function approximators representing the policy and action value functions. Additionally tflearn [3] provided a higher level API to simplify the implementation. The vessel model used in simulations was ported to Python from the Marine Systems Simulator (MSS) toolbox [1]. The 4-DOF container model was ported and integrated into the simulation environment.

Chapter 3

# Design and Implementation

The main contribution of this thesis is a deep reinforcement learning algorithm to solve the problems of path planning and obstacle avoidance. In addition to this a large amount of work went into implementing an environment suitable for simulations of the system in order to train the DRL agent and test its performance. This chapter focuses on the important considerations made when designing the entire system.

## 3.1 DDPG algorithm

The chosen algorithm is a deep deterministic policy gradient algorithm. This algorithm was chosen for its capability to work with a continuous state and action space as well as the fact that it is an off-policy algorithm. The latter fact is crucial since it allows us to have our vessel learn the control policy by observing someone else perform the task. In other words it makes it possible to train the agent on simulations. This method has also been shown promise on the problem of path following [12] which is somewhat related to the problem in this thesis.

The DDPG algorithm consists of two main elements, the policy $\pi(s)$ and the action value function $Q(s, a)$. Each of these are estimated using neural network function approximators. The policy network $\pi_{\theta_\pi}(s)$ was chosen to be a fully connected network with an input layer corresponding to the state vector, two hidden layers and an output layer corresponding to the control output to be given to the vessel. For the output layer all values are scaled between $-1$ and $1$ using a hyperbolic tangent activation function before being linearly transformed to fit the saturation of the corresponding control units. The two hidden layers consisted of 400 and 300 nodes respectively. To introduce nonlinearities to the function approximation we used the relu function

$$relu(x) = \max(x, 0) \tag{3.1}$$

as our activation function between layers. The resulting network can be represented by the following equations:

$$\begin{aligned} h_1(s) &= \text{relu}(W_1 s + b_1) \\ h_2(s) &= \text{relu}(W_2 h_1(s) + b_2) \\ \pi(s) &= \tanh(W_3 h_2(s) + b_3) a_{\text{sat}} \end{aligned} \tag{3.2}$$

Here we have the trainable parameters $W_i$ and $b_i$ as the weight matrix and bias vector for layer $i$. Also $a_{\text{sat}}$ is the saturation vector corresponding to the linear transformation of the output vector.

The action value network has a similar architecture, although slightly more complicated. This is due to the fact that we must pass both the state and the action vectors through the network. To do this we first pass only the state vector through the first layer before passing the action vector through the second layer along with the output from the first layer. Our output layer reduces down to a single output corresponding to the action value for the given state and action. This results in the following equations:

$$\begin{aligned} h_1(s) &= \text{relu}(W_1 s + b_1) \\ h_2(s,a) &= \text{relu}(W_{2,s} h_1(s) + W_{2,a} a + b_2) \\ Q(s,a) &= W_3 h_2(s,a) + b_3 \end{aligned} \tag{3.3}$$

We do not need an activation function for our output layer since we simply want the value from the action value function instead of a scaled output such as with our policy network.

In addition to our policy and value function approximators a copy of each of them was created as target networks to be used for soft parameter updates. These networks were denoted by $\pi'$ and $Q'$ respectively. When training the system our target networks were used to compute our TD-error targets. These TD-errors were then used to compute the gradients used to update our actual actor and critic networks. A mini-batch of 64 transitions was sampled from a replay buffer of up to 300000 transitions for each step of training. The gradients from a mini-batch were then averaged before being applied to the Adam optimizer. A learning rate of $10^{-4}$ was used for the actor network and $10^{-3}$ for the critic network. The target policy and value function networks were then updated by the following update rule:

$$\begin{aligned} \boldsymbol{\theta}_{Q'} &\leftarrow (1-\tau)\boldsymbol{\theta}_{Q'} + \tau\boldsymbol{\theta}_Q \\ \boldsymbol{\theta}_{\pi'} &\leftarrow (1-\tau)\boldsymbol{\theta}_{\pi'} + \tau\boldsymbol{\theta}_\pi \end{aligned} \tag{3.4}$$

Here the target network update rate $\tau$ was chosen to $10^{-3}$.

## 3.2 Environment

As discussed previously the environment in RL includes all parts of the system which are not directly a part of the agent. In our case this means that the environment includes both the dock area around the vessel and also the vessel itself. This is because although the vessel can be seen as the agent, the vessel dynamics are not directly controlled by the agent, thus making them a part of the environment.

To simulate the vessel dynamics the container vessel model from the Marine Systems Simulator (MSS) MatLab toolbox was used. It was ported to Python 3.6 to make it compatible with the rest of the system and to simplify the interaction between the DDPG algorithm and the vessel simulation.

Simulating the docks was slightly more complicated, given that a model of a dock was not available. However for the task at hand, namely learning to maneuver through an unknown environment to a goal, it is not necessary to have a complicated fully accurate model of the docks at this point. Therefore the docks were simply implemented as a set of lines indicating the boundaries which the vessel may not pass through. This lead to the possibility of simulating sensors for obstacle detection by calculating the point along any of the provided lines which is closest to the point of the sensor placement. If this distance is inside the decided range of the simulated sensors the value will be added to the state vector and thus be available when calculating the reward.

Although this is a severe oversimplification, it should be sufficient to determine weather or not the DDPG algorithm is capable of learning to dock a vessel autonomously. Also for more realistic simulations noise could be added to the measurements provided by the sensors. It should be noted, however, that using an accurate mathematical model of a dock environment would be crucial for an attempt to implement a DDPG learning agent intended to learn to control an actual vessel.

## 3.3 Reward function

For any RL system the reward function is one of the most important elements. Reinforcement learning as a whole is centered around maximizing the accumulated reward, hence the design of the reward function dictates the desired behaviour.

Since the reward function ultimately dictates the behaviour of our agent we want to find a way of formulating our objective as a reward function. For the path planning objective the most obvious way to do this is to assign a reward for reaching the goal or target state. For the obstacle avoidance objective we wish to avoid contact with any obstacles, therefore assigning a form of penalty for encountering or getting too close to an obstacle is also advantageous. These two elements are the most important when attempting to solve our chosen problems, therefore they will form the base of our reward functions.

### 3.3.1 Goal reward

The simplest way to assign our goal reward would be to have our reward function give a reward of 1 for being in the goal state and otherwise giving zero. In theory this should work since the ship at some point, given sufficient exploration, will find the goal state and thus learn to find it again. The problem with this reward is that the probability of the agent actually finding the single point where it receives this reward is zero. Because of this the agent will not gain experience and therefor never learn.

An alternative to this reward, which is still quite simple, is to expand the area where our agent receives a reward. This can be implemented as a boundary reward centered around our goal state where the agent receives reward 1 for being inside the boundary and zero for everywhere else. The size of this boundary can then be adjusted to be large enough to ensure the agent is likely to find it while exploring such that sufficient experience is gained over time. A boundary reward of this kind can be given by the following reward function:

$$R_g(d_g) = \begin{cases} 1 & \text{if } d_g < b_g \\ 0 & \text{otherwise} \end{cases} \tag{3.5}$$

where $d_g$ is the distance from the agents position to the goal and $b$ is the radius of the boundary. Although this boundary reward is an improvement on the simple point reward it still has its drawbacks. Since we now give the same reward for the agent being in any point inside the boundary we cannot ensure that the position will converge to the actual goal, only that it will converge to a point we have defined as sufficiently close by being inside the boundary. We therefore want to improve the reward function such that reward is still given for being close to the goal but the maximum reward is only received exactly at the goal. This can be done by using a multivariate Gaussian distribution as a reward function. We have the equation for a multivariate Gaussian distribution

$$f(x) = \frac{1}{2\pi \det \Sigma} e^{-\frac{1}{2}(x-\mu)^\top \Sigma^{-1}(x-\mu)} \tag{3.6}$$

where $\Sigma$ is our co-variance matrix and $\mu$ is our mean vector. For our reward function we would set our mean vector as the goal position and our input vector as the vessels position vector. The co-variance matrix can then be adjusted to change the shape of the distribution.

This improved reward function should allow us to successfully converge to the actual goal while still assigning rewards in a larger area around the goal, such as with the boundary reward.

## 3.3.2 Obstacle penalty

Our second objective is to avoid obstacles. Similarly to the first proposed goal reward this could be done by giving a negative reward when the agent comes in contact with an obstacle. The problem with this type of goal reward was that it gave us too small a chance of actually achieving any experience. This is not necessarily the case for this obstacle penalty, seeing as it is more likely that we at some point will receive a penalty for encountering the dock. The main problem for this obstacle penalty is that we actually have to encounter the obstacle, thus terminating the simulation episode, to receive the experience. This will result in slower learning as it will shorten the amount of experience achieved per simulation episode. Also, for implementation on a real vessel it would be desired to not only avoid direct contact with obstacles but also stay a certain distance away from them. Again a boundary reward, or rather penalty, will be a simple solution to this problem. We therefore propose the following obstacle reward function:

$$R_o(d_o) = \begin{cases} -1 & \text{if } d_o < b_o \\ 0 & \text{otherwise} \end{cases} \tag{3.7}$$

where $d_o$ is the distance to the obstacle and $b_o$ is the width of the obstacle boundary. Unlike the goal reward this boundary penalty does not hinder our agent in converging to the goal. Using a Gaussian style penalty here, however, should prove even more useful. This is due to the fact that it allows us to assign penalty in a large area while still having the penalty grow as the vessel moves closer to the obstacle. This is beneficial in cases where the vessel is forced to move closer to an obstacle in order to achieve another goal.

Implementing this reward simply on the distance from a single point to the closest obstacle will make it difficult for the agent to know which direction it should move to avoid the obstacle. Therefore we implement this reward as four separate elements each calculated based on the distance from a single sensor placed either at the front, back, left or right of the vessel. This way the agent has the opportunity to map the reward to the sensor which is reacting to the obstacle, thus allowing it to know which direction it should move.

## 3.4 State augmentation

When designing our RL system there are some important considerations we need to remember when deciding what states to feed to the agent. These states along with the reward are all the information the agent has available. Based on this information we want the agent to learn to create a mapping from input states to an output action such that the reward is maximized. Therefore it is intuitive that we need to feed the agent the states which affect the reward, or in other words are included in the reward function. Also since we are using function approximators to estimate our policy and value function we do not want to feed the agent states which have no correlation to the reward as this will simply complicate the function approximations.

Once we have selected the states which contain usable information to the agent the next consideration is the complexity of the transformation from the input states to the rewards they affect. For example for the goal reward the most straight forward state to feed the agent is the position of the vessel.

$$s = \begin{bmatrix} x \\ y \end{bmatrix} \tag{3.8}$$

While this should be sufficient we are now relying on the agent to be able to first transform this position into an error, then use the error to estimate a Gaussian function. By simply performing the transformation from position to error beforehand and providing the positional error rather than the position itself we reduce the transformation necessary by the agent. The proposed augmented state vector is then

$$s = \begin{bmatrix} \tilde{x} \\ \tilde{y} \end{bmatrix} = \begin{bmatrix} x_g - x_v \\ y_g - y_v \end{bmatrix} \tag{3.9}$$

where the subscripts $g$ and $v$ denote the goal positions and vessel positions respectively. For the additional directional reward we simply add a third state $\dot{d}$ which is the rate of change in the distance to the goal. The state vector then becomes

$$s = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \dot{d} \end{bmatrix} \tag{3.10}$$

When it comes to the obstacle reward we have based the reward directly on the sensor measurements, thus it is sufficient to provide these measurements. The suggested state vector for this objective then becomes

$$s = \begin{bmatrix} s_f \\ s_b \\ s_r \\ s_l \end{bmatrix} \tag{3.11}$$

where $s_f, s_b, s_r, s_l$ are the sensor measurements for the sensors placed in the front, back, right and left of the vessel respectively.

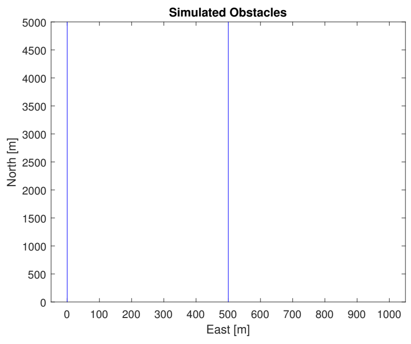
## 3.5 Training

Training the policy and value function networks $\pi_{\boldsymbol{\theta}_\pi}(s)$ and $Q_{\boldsymbol{\theta}_Q}(s)$ for both objectives was performed as described by algorithm 2. At each step a mini-batch of 64 random transitions was sampled from our replay buffer consisting of a maximum of 300000 transitions. The target values were then computed by passing the sampled states and actions through the target policy and value function networks $\pi'_{\boldsymbol{\theta}_{\pi'}}(s)$ and $Q'_{\boldsymbol{\theta}_{Q'}}(s)$. Using these target values the gradients were computed and averaged over the mini-batch before being applied to the networks through Adam optimization. Finally the target networks were updated according to 3.4.

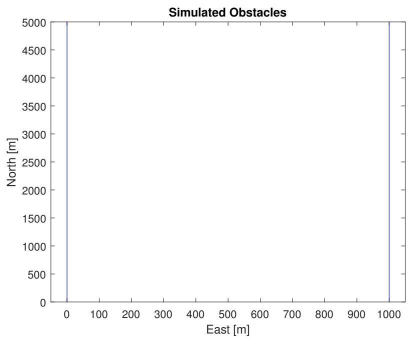
Although the agent was trained similarly for the separate tasks the setup of the environment differed for the two task. For the path finding agent the environment was set up without any simulated obstacles. The goal position for training was set to $[1500, 1500]$. The starting position for the vessel was then chosen randomly varying along the East axis $\pm 1000m$ about $(1500, 0)$. The randomness of the starting position was implemented to induce an extra element of exploration in addition to the actor noise with the intent of improving both the actual learning and the learning speed.

For the task of obstacle avoidance the environment was implemented with obstacles as shown in figure 3.1. These obstacles were represented by the endpoints of the lines. The vessel was then placed between these lines varying the East position by $\pm 100m$ around the center point between the lines.

**Figure 3.1** The simulated environments for the obstacle avoidance objective.



**(a)** Obstacles placed $500m$ apart



**(b)** Obstacles placed $1000m$ apart

Chapter 4

# Testing and Results

Once the entire system consisting of the simulation environment and the reinforcement learning agent were implemented and training was completed a series of tests were performed to asses the performance of the agent. This chapter first presents the training results before presenting the performance tests along with their results. The results from the path planing agent are presented first, followed by the result for the obstacle avoidance agent. Brief discussion into the meaning of the different results will take place, however more in depth discussion of the overall performance is presented in chapter 5. First the tests and results from the single objective agent will be presented, followed by the tests and results from the multi-objective agent.

## 4.1 Path planing

### 4.1.1 Training results

For the path planing objective we trained the agent in four different ways. The first training was set up with the target at $[1500, 1500]$ and the vessel starting at $[1000, 0]$. In figure 4.1 we see the total reward accumulated per episode along with the average reward per episode over time. We see that the agent stabilizes the average reward per episode after about 500 episodes.

For the second training the goal was still at $[1500, 1500]$, but now the vessel started at $[0, 1000]$. Figure 4.2 shows the rewards from this training. As we see the agent now first learns a policy around 200 episodes, before moving on to learning what seems to be a slowly decaying policy for the rest of the training. This suggests that this round of training was unable to find an optimal policy. For the third round of training the addition of the directional reward, which is proportionate to the rate of change in the distance from the

**Figure 4.1** Accumulated reward per episode for training scenario with starting positions around $[1000, 0]$ and goal at $[1500, 1500]$

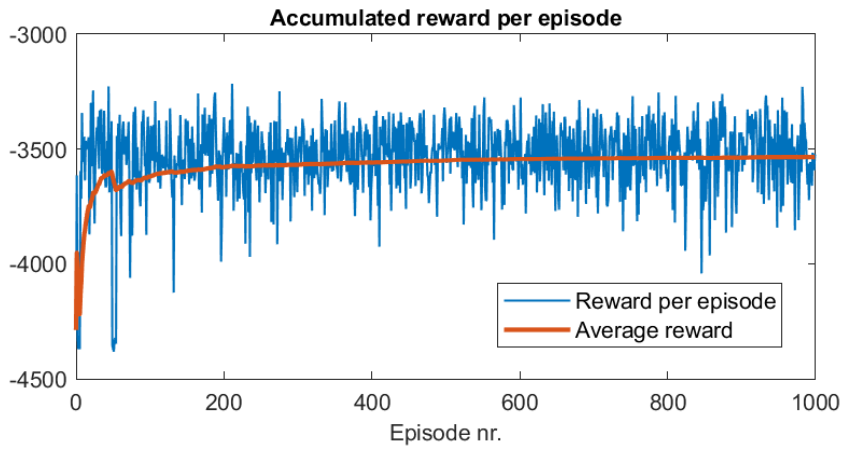


**Figure 4.2** Accumulated reward per episode for training scenario with starting positions around $[0, 1000]$ and goal at $[1500, 1500]$

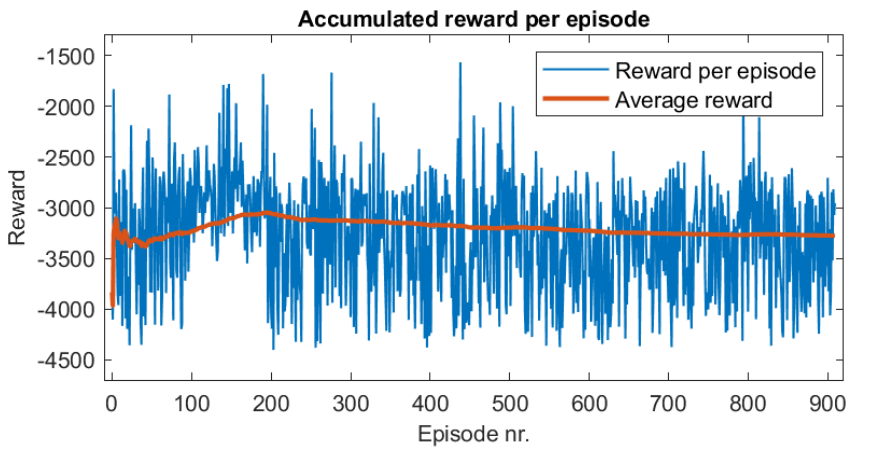
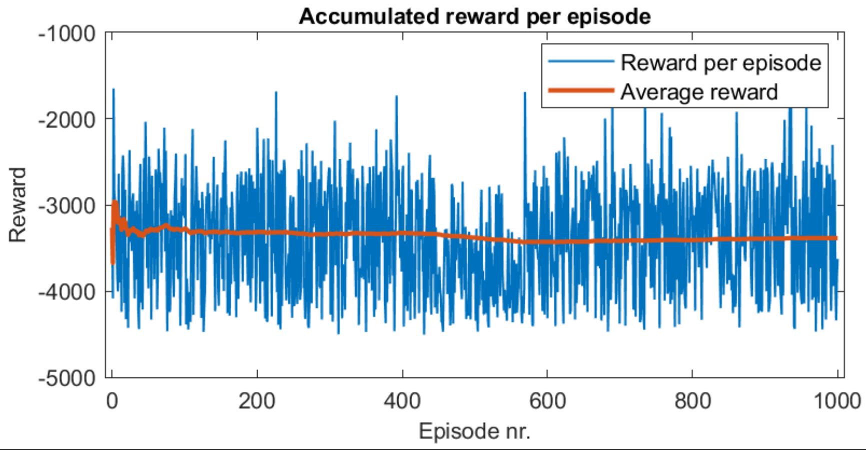


**Figure 4.3** Accumulated reward per episode for training scenario with starting positions around $[0, 1000]$ and goal at $[1500, 1500]$ and the addition of directional reward



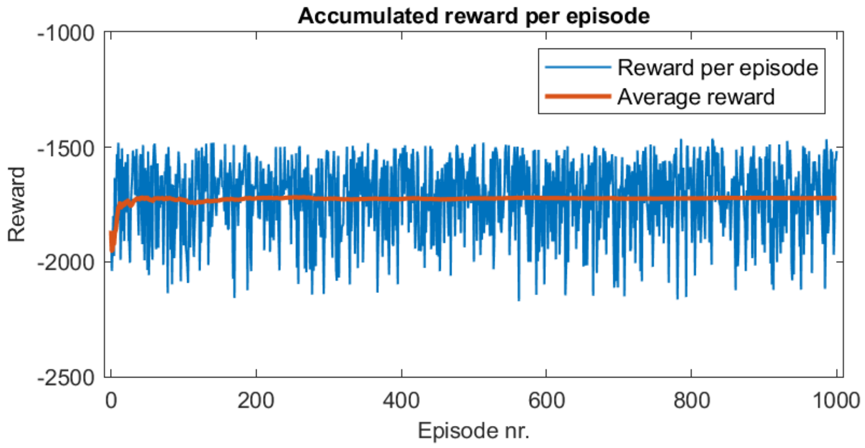
vessel to the goal, resulted in the rewards seen in figure 4.3. Here we see that the average reward stabilizes around 100 episodes before slightly decaying from episode 400 to 600. We also see a greater variance in the total accumulated reward per episode than for the previous training scenario. A final round of training was performed with shorter episodes consisting of 200 simulated seconds rather than 300 as for the rest. Figure 4.4 shows the rewards from this training. This time the average reward stabilizes after around 50 episodes and remains stable for the duration of the training. We also see a reduction in the magnitude of the rewards which is caused by the fact that the episodes were shorter. This lead to the vessel having less time to accumulate reward and never allowed the vessel to reach the goal.

### 4.1.2 Tests

During the training of the agent the parameters of policy network were saved at each 100th episode of training. Once the training had been performed the weights of checkpoint closest to the point of best performance during training were restored so that the learned policy at the given checkpoint could be used to execute our performance tests. By removing the action noise and not updating the network parameters we fed observations into our policy network and simulated tests using the networks output as our control input.

For each variation of agents trained we performed three tests where the vessel starting position was the same but the goal positions varried. The tests started at $(0, 1500)$ and had targets at $(1500, 500)$, $(1500, 1500)$ and $(1500, 2500)$. For each of the tests the vessel paths when following the trained policies are presented.

**Figure 4.4** Accumulated reward per episode for training scenario with starting positions around $[0, 1000]$ and goal at $[1500, 1500]$ and the addition of directional reward. Episode lengths were reduced from 300 seconds to 200 seconds



**Goal-only oriented agent**

The first policy tested was the one trained using only the $x$ and $y$ positional errors as state inputs and having a reward function consisting of the single element corresponding to these inputs. From the training data in figure 4.2 we see that the best policy learned was around 200 episodes. We therefore used the parameters saved after 200 episodes of training to perform our tests.

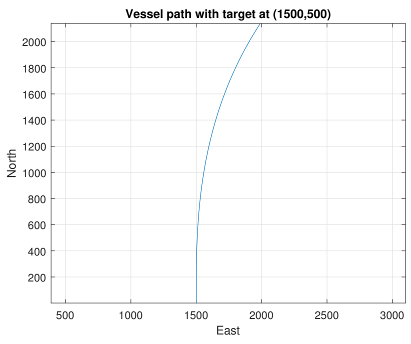
In figure 4.5 we can see the paths resulting from following the trained policy in the three tests. We see that the vessel came close to the goal when it was located straight ahead of the starting position, however it did not reach it exactly. The policy proves to be unable to steer the vessel to the target located west of the starting position, while it was tending towards correct behaviour for the target located to the east, however not steering hard enough resulted in the vessel never coming close to the target.

### 4.1.3 Goal and directional oriented agent

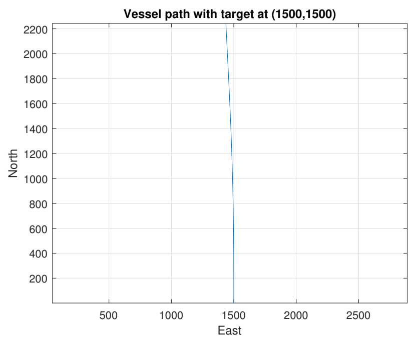
Our next agent was trained with the $x$ and $y$ positional errors and the rate of change in distance to the goal $\dot{d}$ as our state inputs. The reward had the same base as the previous agent with the addition of the term proportionate to $\dot{d}$. For this training we see from figure 4.3 that the policy produces a stable average reward from around 100 episodes to 400 episodes of training. We also see that the highest peaks in this period come at around 200 episodes, which is why we used the weights saved after 200 episodes again when performing our tests.

From the positional plots in figure 4.6 we see that the addition of the directional reward

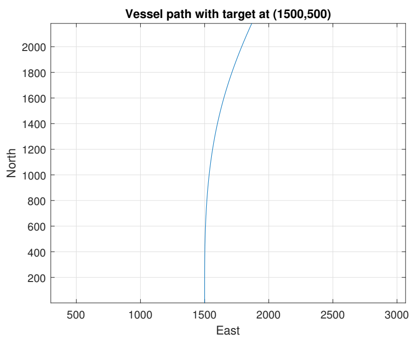
**Figure 4.5** Positional plots of the behaviour of the first trained agent with exclusively positional error oriented reward.



**(a)** Target at $(1500, 500)$

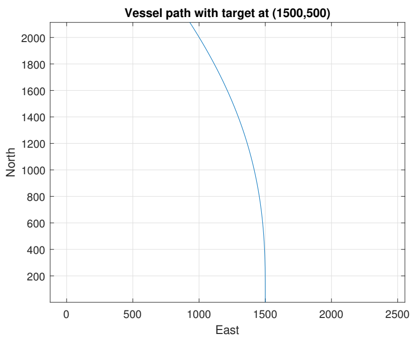


**(b)** Target at $(1500, 1500)$

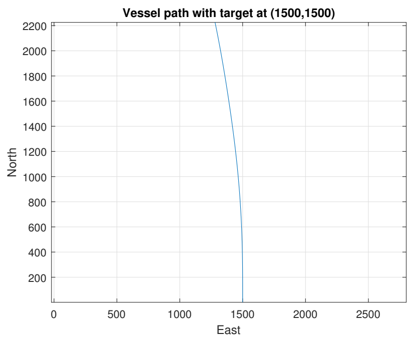


**(c)** Target at $(1500, 2500)$

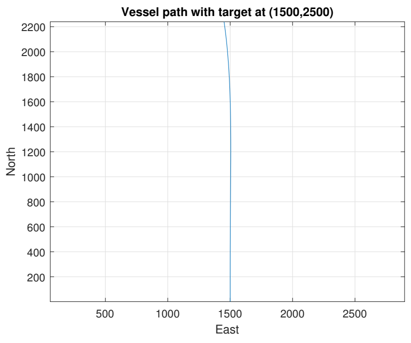
**Figure 4.6** Positional plots of the behaviour of the second trained agent with positional error and directional reward.



**(a)** Target at $(1500, 500)$

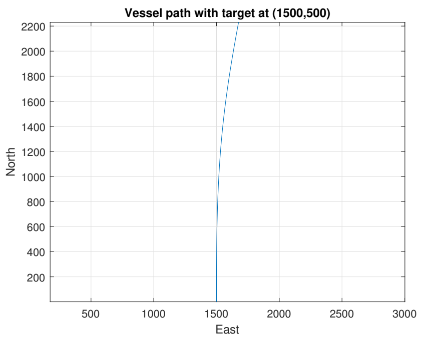


**(b)** Target at $(1500, 1500)$

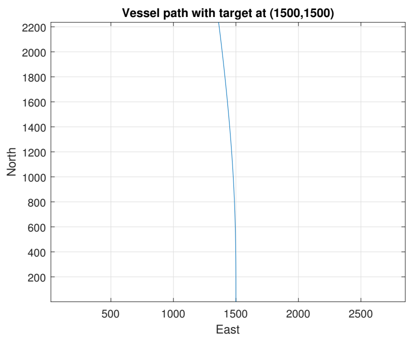


**(c)** Target at $(1500, 2500)$

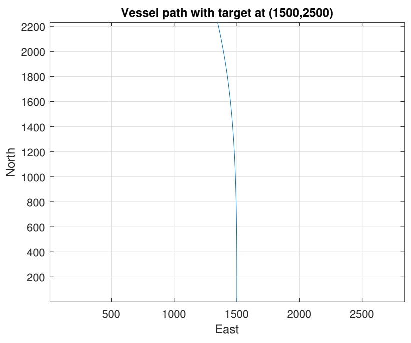
**Figure 4.7** Positional plots of the behaviour of the third trained agent with positional error and directional reward and shortened training episode length.



**(a)** Target at $(1500, 500)$



**(b)** Target at $(1500, 1500)$



**(c)** Target at $(1500, 2500)$

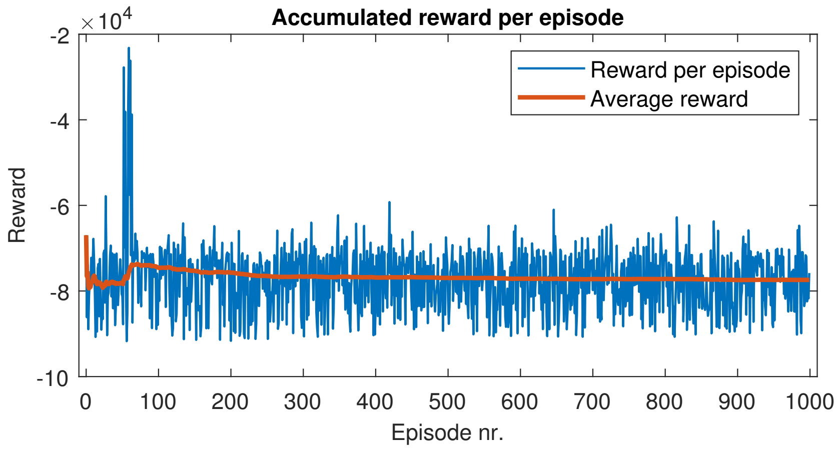
leads to the policy controlling the vessel better towards the target to the west of the starting position. For the straight ahead and east positioned targets the control policy does not turn the vessel sufficiently to the east, resulting in worse performance than the first agent had for these tests.

### 4.1.4 Goal and directional oriented agent trained on shorter episodes

The final agent was trained on similar basis as the previous agent, however the maximum time for a single episode was reduced to 200 second. In this round of training we see that the average reward is quite stable already after 50 episodes and remains at an even level throughout the whole training. From this result any choice of weights following that point should perform quite similarly. To have the most comparable results to the two previous agents the weights after 200 episodes were again chosen to perform our tests. The results are shown in figure 4.7.

Again we see that the vessel is closest to reaching its goal when the goal is directly north

**Figure 4.8** Accumulated reward per episode for training scenario with starting positions around $[0, 250]$ obstacles along $x = 500$ and $x = 0$



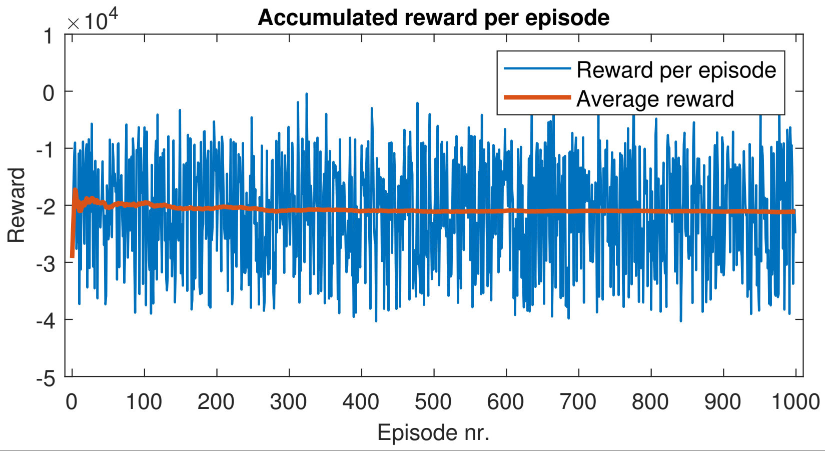
of the starting position. Where the previous agent seemed to be getting closer to reaching the goal to the west as well, this one seems to only be able to come close to the straight goal.

## 4.2 Obstacle avoidance

### 4.2.1 Training results

The obstacle avoidance agent was trained in an environment described in section 3.5. The rewards from this training are shown in figure 4.8. As we see the agent appears to start to learn a good policy around 70 episodes into the training, however the high peak in performance quickly dies out and the policy learned is not able to perform the task of obstacle avoidance. Later the agent was trained in a similar environment but with the obstacles places 1000 meters apart and the sensor range and reward function expanded. The starting position was also adjusted to be varying about $(0, 500)$. This gave the training results shown in figure 4.9. This time we do not see the short peak of performance but rather a relatively stable average reward. Although this might seem as though the agent is actually able to learn a good policy in this case it was apparent through the actions chosen by the agent during training that this agent was not able to learn a good policy either. The lack of learning for this objective was assumed to be caused by oversimplification of the model of the environment. Unfortunately there was not enough time to attempt to implement a better solution, therefore further test were not performed for this objective.

**Figure 4.9** Accumulated reward per episode for training scenario with starting positions around $[0, 500]$ obstacles along $x = 1000$ and $x = 0$



Chapter 5

# Discussion

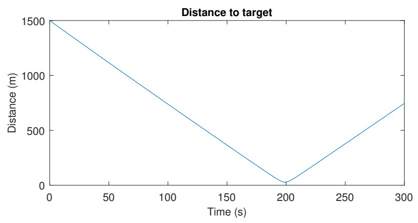
## 5.1 Main results

Based on the training results and the tests performed we see varying performance from the DDPG agents. Figure 5.1 shows the distance from the vessels to the targets throughout the first performed test with targets at $(1500, 1500)$. We see that the first agent trained managed to get the closet to the target position in the first test. This performance was marginally better than the two later agents, however all three performances for this test show that the agents have to a certain degree learned to find the target.

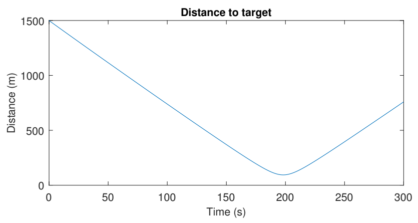
In figure 5.2 we see the control outputs from the three agents in the first test. In all three cases the rudder command angles increase gradually over time. We can also see in all three cases that there is a significant change at around the same time as the distance from the vessel to the target was at its shortest. For the first agent this change comes as a slight peak and dip before continuing to produce rudder command angles following a similar change as before this peak. From the two later agents we see that the command output goes from increasing to decreasing, which indicates that the agent was attempting to turn back towards the target which has been passed. This behaviour is again an indication that despite the agents not being perfectly successful in finding their targets, they were to some degree learning that their goal was to steer toward the targets.

The difference in the control outputs between the first agent and the two others also shows the significance of the additional term added to the reward function and the extended state vector. When the first agent passes by the target the reward it continues to accumulate is equally decaying in all directions away from the target. Therefore the agent does not learn that it should attempt to turn around back towards the target. For the two agents with the additional reward proportionate to the rate of change in the distance from the target there is however a difference in the accumulated reward depending on the path taken once past the target. This has apparently been transferred to the agent such that it has learned to control

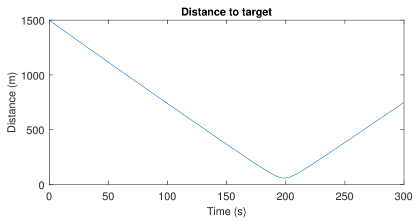
**Figure 5.1** The distance from the vessel to the target in the test with targets at $(1500, 1500)$ for the three separately trained agents



**(a)** First trained agent



**(b)** Second trained agent



**(c)** Third trained agent

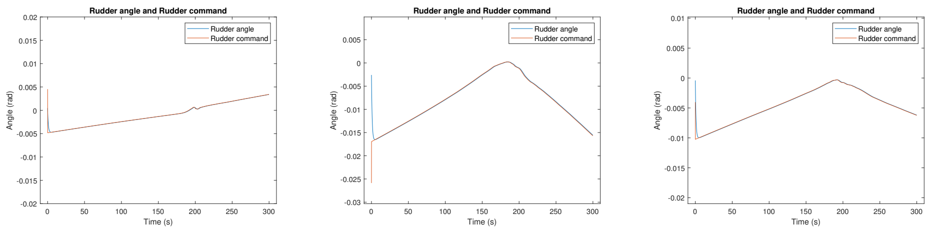
the vessel differently than the first agent. Despite this none of the agents have been able to learn to change the steering early enough that the vessel reaches the target exactly.

Unfortunately, the fact that the vessel has had little exploration in the possible range of $\tilde{y}$ compared to the range of exploration for $\tilde{x}$ limits the performance of all agents. By limiting the starting position we also limited the range of $\tilde{y}$ which we could guarantee would be explored. Although we always give the vessel a starting $x$-position of $0$, the exploration range for $\tilde{x}$ is significantly larger due to the fact that the starting value for $\tilde{x}$ is $1500$. The exploration can still be improved upon, however, by allowing for the possibility of $\tilde{x}$ starting as a negative value, i.e. starting further north than the target.

Ideally for better training of the DRL agents the range of starting positions for training episodes should be increased to a large area surrounding the target. The initial heading of the vessel should also be allowed to vary such that an increased range of transitions can be experienced. This improvement bring with it the need of extended training time both in the form of longer episodes and in the form of a larger number of episodes. The increased length of the simulation episodes is needed so as to allow the vessel to perform large maneuvers if needed, while the increased number of episodes is to ensure that a significant amount of starting states in the enlarged range are in fact experienced. These improvements were unfortunately not implemented due to insufficient time to accommodate for the large increase in training time under improved conditions, but are recommended for any future work on the problem.

Moving on to the tests with targets at $(1500, 500)$ and $(1500, 2500)$ we saw in the previous chapter that the behaviour was quite varying for the different agents. We see again in figure 5.3 that the second agent came the closest to the goal at $(1500, 500)$ while the first

**Figure 5.2** The control output from the DDPG agents in the test with targets at $(1500, 1500)$ for the three separately trained agents



**(a)** First trained agent **(b)** Second trained agent **(c)** Third trained agent

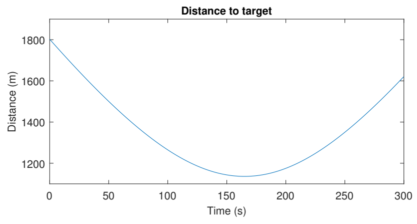
agent came the closes to $(1500, 2500)$, although none of these results are close to actually reaching the targets. Again this poor performance is likely connected to the insufficient exploration . Seeing as all the training was started within the same interval and the targets are now far outside this range it is not surprising that the agents have not learned proper behaviour for such situations. Again increased area for starting positions along with extended episodes and training time will likely improve this performance.

Another improvement which could be made is further the addition of elements to the reward function and state vectors. One possible addition could be based on a desired heading towards the target. By adding the heading error $\tilde{\psi} = \psi_d - \psi$, where $\psi_d$ is the angle from the vessel position pointing directly towards the target, the agent will have the possibility of learning a mapping which induces a desire to point towards the target. Given several such sub-objectives [29] suggests a method of reward shaping and multiobjectivization to further improve a system consisting of multiple sub-objectives.

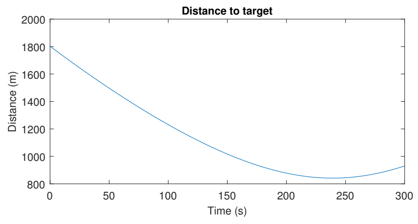
Our final result for discussion is the inability of our DDPG agent to learn the simple obstacle avoidance task of maneuvering in between two obstacles. As mentioned the training results were of such a nature that it was not seen as necessary to test the performance the agent. Despite these poor results, the performance of the DDPG algorithm in other tasks such as in [12] and for our path planing objective still gives reason to believe that the algorithm should be able to perform the task given sufficient improvements on the simulation environment.

Based on the difficulties in the work with our obstacle avoidance objective some suggested areas to focus on for improving the simulation environment are sensors and models of the obstacles. The design of the reward function should also be considered in relation to any changes to the system, however the use of a Gaussian style reward function is thought to be beneficial given it can be integrated with the new system.

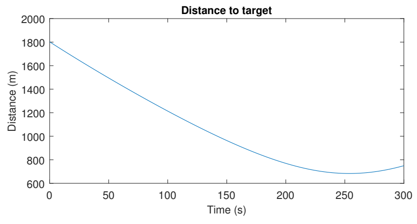
**Figure 5.3** The distance from the vessel to the target in the test with targets at $(1500, 500)$ for the three separately trained agents



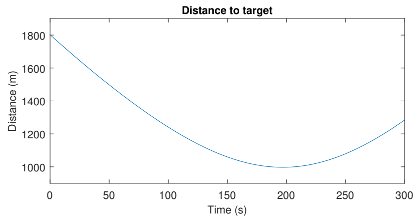
**(a)** First agent $(1500, 500)$



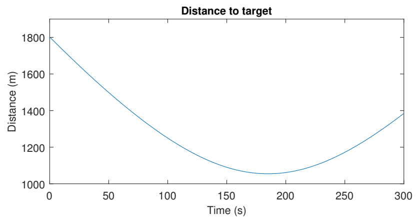
**(b)** First agent $(1500, 2500)$



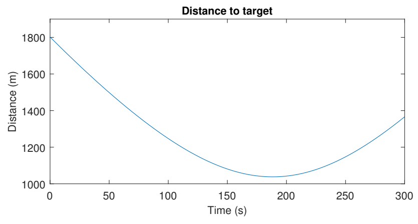
**(c)** Second agent $(1500, 500)$



**(d)** Second agent $(1500, 2500)$



**(e)** Third agent $(1500, 500)$



**(f)** Third agent $(1500, 2500)$

## 5.2 Goals and Method

In chapter 1 the goal for the work in this thesis was defined through 2 research questions. Based on the results presented in chapter 4 we will now discuss to what degree these goals were met.

### 5.2.1 Path planing

The first research question was whether a DDPG agent could learn path planing for a marine vessel. As discussed we see that the DDPG agent was in fact able to achieve this to a certain degree, however not sufficiently to solve the general task of path planing. Both the simplest solution consisting of only a reward based on the augmented states $\tilde{x}$ and $\tilde{y}$ and the two extended with $\dot{d}$ led to the vessel nearly reaching its target in the test scenario closest to the training scenarios. The extended state vector showed signs of slight improvement through the command output after passing the target, however due to lack of exploration the overall performance was not as high as first expected. The insufficient exploration was made even more apparent in the test cases with targets placed far away from the target in the training scenarios. Unfortunately there was not enough time for improvements to be made to the system in this thesis.

Overall these results show that the DDPG agent was able to learn limited path planing and show promising signs for future work, thus somewhat accomplishing our first goal.

### 5.2.2 Obstacle avoidance

The second goal was phrased through the research question on whether the DDPG agent would be able to learn obstacle avoidance. As discussed the training performed was not able to learn this task. This was mainly due to the fact that the obstacle simulator was not accurate enough for the training to work properly. During the training it seemed as though the agent was learning a policy corresponding to a local minimum caused by faulty implementation of the obstacles and sensors. This result gives basis to believe that an agent of similar architecture as presented in this thesis would be capable of learning obstacle avoidance given a better, more accurate simulation of the obstacles and/or a more advanced reward function. Unfortunately the goal of learning obstacle avoidance was not accomplished.

## 5.3 Future Work

As discussed the work in this thesis has revealed areas for improvement for future work. For the task of path planing improved exploration through an expanded set of starting states is suggested as a simple yet time consuming improvement. Also the addition of further sub-objectives such as a desired heading and the use of multiobjectivization based on such sub-objectives is thought to further improve the accuracy and performance of the agent.

For the obstacle avoidance objective there are several areas in need of more focus and improvement. To begin a more accurate model of obstacles and sensors should greatly improve the performance. Also the placement of sensors and what type of sensors to use should be given more thought. The success of DRL using low level input such as cameras and pixel analysis also open the possibility of using image processing as a viable sensor for obstacle detection. Combining such solutions with a DDPG agent should lead to successful obstacle avoidance.

In addition to improving the control algorithm the choice of vessel model should also be considered for future work attempting to combine the objectives of path planing and obstacle avoidance to directly handle the problem of autonomous docking. The container model used in this thesis is presumably not ideal for a full docking solution due to its size and relatively poor maneuverability. Using a model of a smaller or at least more maneuverable fully actuated vessel will be crucial to solve the full complex problem of docking.

Chapter 6

# Conclusion

This thesis was meant to be a contribution to getting a step closer to fully autonomous marine vessels. The main focus was the problem of autonomous docking. The problem was decomposed and constrained to the two objectives of path planing and obstacle avoidance. The goal for the work was to implement a deep reinforcement learning control algorithm to solve each of these objectives using the deep deterministic policy gradient method.

To test the algorithm an environment was first implemented. The Container ship model from the MSS toolbox [1] was ported to Python and all other functionality necessary for combining the DDPG agent and the vessel model was implemented. Once the entire system was functional the method was used to train separate agents to perform the tasks of path planing and obstacle avoidance.

The results from the training showed that the performance of the agents were worse than first expected. For the task of path planing the main cause of the low performance was assumed to come from insufficient exploration during the training phase. For the obstacle avoidance task the poor performance was assumed caused by a combination of oversimplified obstacles and sensors in the implementation of the environment and also here a lack of exploration for the training. Both these problems could have been improved given more time, however complications early on in the work of integrating the vessel model into an implementation of the environment and combining this environment with the DRL method lead to insufficient time to solve said problems.

Despite these complications and poor performance, we still see some sign of promising results for the method. For the path planing agent it was shown that the test most similar to the training scenarios came close to controlling the vessel to the desired target position. Based on this result it is likely to believe that upon improving the training by expanding the exploration area and extending the training time such that a much larger subset of the possible states will be experienced should improve the agents path planing ability.

In addition to proposing these changes to the system, certain aspects of the overall problem are suggested for future work. The problem of obstacle avoidance has much to gain from more work on simulators and environments which better model actual obstacles and make training easier. Also both choice of sensor types and placement are topics which can be explored to great extent. For the path planing objective the method of reward shaping and multiobjectivization presented in [29] is suggested as a possible improvement to the implemented solution. Given solutions to both objectives are found a proposal is made for combining the objectives in a single agent using multi-objective reinforcement learning methods. It is also suggested that a model of a more maneuverable fully actuated vessel be used as the complex maneuvers necessary for actual docking are impossible for vessels such as the one used in this thesis.

# Bibliography

[1] T. I. Fossen and T. Perez, "Marine systems simulator (mss).." `https://github.com/cybergalactic/MSS`, 2004.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

[3] A. Damien *et al.*, "Tflearn." `https://github.com/tflearn/tflearn`, 2016.

[4] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015.

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[6] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *CoRR*, vol. abs/1712.01815, 2017.

[7] J. Kober, K. Mülling, O. Krömer, C. Lampert, B. Schölkopf, and J. Peters, "Movement templates for learning of hitting and batting," (Piscataway, NJ, USA), pp. 853–858, Max-Planck-Gesellschaft, IEEE, May 2010.

[8] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, "Autonomous inverted helicopter flight via reinforcement learning," in *Ex-*

*perimental Robotics IX, The 9th International Symposium on Experimental Robotics [ISER 2004, Singapore, 18.-21. June 2004]*, pp. 363–372, 2004.

[9] M. G. Aronsen, “A study of the possibilities of using reinforcement learning for autonomous docking of marine vessels.” Specialization Project TTK4550 - Engineering Cybernetics, NTNU.

[10] J. Hu, J. Mei, D. Chen, L. Li, and Z. Cheng, “Path planning of robotic fish in unknown environment with improved reinforcement learning algorithm,” in *Internet and Distributed Computing Systems* (Y. Xiang, J. Sun, G. Fortino, A. Guerrieri, and J. J. Jung, eds.), (Cham), pp. 248–257, Springer International Publishing, 2018.

[11] A. Konar, I. G. Chakraborty, S. J. Singh, L. C. Jain, and A. K. Nagar, “A deterministic improved q-learning for path planning of a mobile robot,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 43, no. 5, pp. 1141–1153, 2013.

[12] A. B. Martinsen, “End-to-end training for path following and control of marine vehicles,” 2018.

[13] E. A. Lund, “Path following in simulated environments using the a3c reinforcement learning method,” 2018.

[14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Intorduction. Second edition, in progress*. The MIT Press, 2017.

[15] S. J. Russel and P. Norvig, *Artificial Inteligence: A Modern Approach. Third edition.* Pearson Education, 2010.

[16] L. P. Kaelbling, M. L. Littman, and A. W. Moore, “Reinforcement learning: A survey,” *CoRR*, vol. cs.AI/9605103, 1996.

[17] R. Frank, “The perceptron a perceiving and recognizing automaton,” *Cornell Aeronautical Laboratory, Buffalo, NY, USA, Tech. Rep*, pp. 85–460, 1957.

[18] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization,” in *Advances in neural information processing systems*, pp. 2933–2941, 2014.

[19] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” *arXiv preprint arXiv:1412.6980*, 2014.

[20] N. Qian, “On the momentum term in gradient descent learning algorithms,” *Neural networks*, vol. 12, no. 1, pp. 145–151, 1999.

[21] R. Bellman, *Dynamic programming*. Princeton University Press, Princeton, N.J., 1957.

[22] D. M. Roijers, P. Vamplew, S. Whiteson, and R. Dazeley, “A survey of multi-objective sequential decision-making,” *Journal of Artificial Intelligence Research*, vol. 48, pp. 67–113, 2013.

[23] R. E. Bellman, *Adaptive control processes: a guided tour*. Princeton University Press, Princeton, 1961.

[24] J. Peters, “Policy gradient methods,” *Scholarpedia*, vol. 5, no. 11, p. 3698, 2010. revision #137199.

[25] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine Learning*, vol. 8, pp. 229–256, May 1992.

[26] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, p. 529, 2015.

[27] L.-J. Lin, “Reinforcement learning for robots using neural networks,” tech. rep., Carnegie-Mellon Univ Pittsburgh PA School of Computer Science, 1993.

[28] T. I. Fossen, *Handbook of Marine Craft Hydrodynamics and Motion Control*. John Wiley & Sons, Ltd, 2011.

[29] T. Brys, A. Harutyunyan, P. Vrancx, M. E. Taylor, D. Kudenko, and A. Nowé, “Multi-objectivization of reinforcement learning problems by reward shaping,” in *2014 international joint conference on neural networks (IJCNN)*, pp. 2315–2322, IEEE, 2014.