Norwegian University of
Science and Technology

A Study of the Possibilities of Using Reinforcement Learning for Autonomous Docking of Marine Vessels

Author:
Mathias G. ARONSEN

Supervisor:
Thor I. FOSSEN

December 18, 2018

Summary

This paper is written as a preparation for a Master's thesis on the use of reinforcement learning for autonomous docking of marine vessels. Reinforcement learning methods have been growing in popularity in recent years and have been used in implementations of both path-planning and path-following systems. In this paper we first present research on the subject of reinforcement learning. A Deep Deterministic Policy Gradient method is then proposed as a possible solution to the autonomous docking problem. Several aspects of both the problem and method are discussed as well as suggesting areas in need of more focus in the continuation of the solution in future works.

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/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 (1) as well as board games such as chess and Go (2). 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 continuous state- and action-space tasks in real life environments such as playing table tennis (3) and flying a helicopter upside down (4).

The field of autonomous marine vessels has also seen a growing use of RL in recent years. Exploration and path planning in discrete spaces has been implemented with several different RL algorithms such as in (5; 6). Recent studies (7; 8) have also shown success in using RL for continuous space path following of marine vessels. However, the area of path planning in a continuous space remains fairly unexplored. This thesis will look at some of the implementations mentioned to attempt to find possible solutions to this problem.

1.2 Problem Definition

As mentioned, automating the docking process for marine vessels has several benefits. The goal is to accomplish this by breaking down a traditional control system for marine vessels and rebuilding it around the use of Reinforcement Learning. In a traditional control system we usually find separate modules for navigation, guidance and motion control. The guidance segment will take in measurements from the navigation module and a path generated by a separate path-planner. The guidance module then calculates a desired motion which the motion control module turns into the actuator control signals for the vessel.

The proposed solution to the automating of the docking process has an alternative approach. By reformulating the problem as a reinforcement learning problem the entire system is divided into two main segments, the agent and the environment. The environment consists of the vessel we wish to control and a performance measure of how well we are controlling the vessel. The agent is our new control unit which consists of a control policy and a value function. The control policy gives us the desired action for any given state while the value function gives us the value of being in each state. We have thereby combined the tasks of measuring states, generating a desired path, calculating a desired motion to follow this path and finally generating actual motion control commands for the vessel into a single unit which measures the states of the system and directly generates control commands for the vessel.

This thesis is aimed at being a preparation for further work on the problem in a Master's thesis. It therefore consists of general research on reinforcement learning, a review of previous work in similar areas and a proposed method for a solution to be tested as part of the Master's thesis. By doing so we seek to answer the following questions:

1. What is reinforcement learning?
2. What has been done in related fields and how can we take advantage of this?
3. How can the problem of autonomous docking be solved using reinforcement learning methods?

1.3 Outline of the Report

As mentioned, this project thesis has been written as a preparation for a future Master's thesis. Therefore the main focus of the work has been on researching previous works in related fields to look for possible methods to adapt to solve the path planning problem. First in chapter 2 the theory needed to understand and evaluate relevant methods is presented. Most of the theory about general machine learning and reinforcement learning is a combination of theory from (9; 10; 11). Some necessary material on basic marine vessel control is also presented in this chapter.

Next, in chapter 3, a review of some previous works on similar problems is done. Three examples are presented slightly more in depth with focus on the most relevant elements from each of them. In chapter 4 a possible solution to our problem is presented with a basis in theory presented and inspiration from previous works. Several aspects of the problem are discussed to form the basis of a possible solution method to be implemented as a part of the future Master's thesis. Finally the conclusion of the work is presented in 5.

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. Although this thesis only focuses on use of the later, it is important to have a basic understanding also of supervised learning as this forms the basis for deep reinforcement learning.

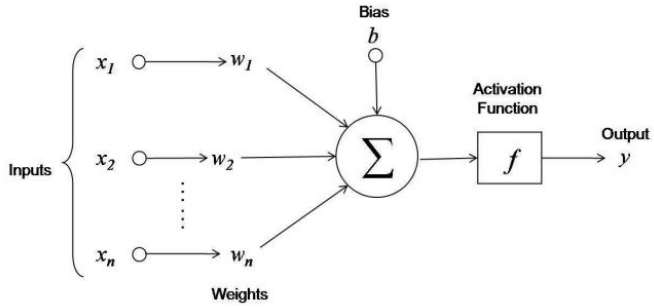
2.1.1 Supervised learning

Supervised learning is a form of machine learning in which one provides a data set consisting of paired inputs and outputs that are meant to show the desired behaviour of the system. For each of the data pairs the system will first be given the input data. It will then attempt to mimic the desired behaviour without actually knowing what the desired behaviour is by approximating a mapping from input to output. The output of the system will then be compared to the ”correct” output from the data set. Based on how different the two outputs are the system will adjust itself so as to come closer to what it now assumes is the correct behavioural pattern before repeating the process with a new set of data pairs. After going through several such loops the system will get closer and closer to an optimal approximation of the desired behaviour.

2.1.2 Artificial Neural Networks

An Artificial Neural Network (ANN) is a mathematical approximation of how a brain works. As first introduced in (12), the simplest form of ANN is the perceptron, which is simply a single neuron as seen in figure 2.1. Given an input vector $\boldsymbol{x}$, the perceptron's

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



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.

For layer i in a ANN we have the weight matix $\boldsymbol{W}_i$, bias vector $\boldsymbol{b}_i$ and input vector $\boldsymbol{x}_i$. Given the activation function $f(\cdot)$ we then have the output vector $\boldsymbol{y}_i$ given as

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

This then becomes the input to the next layer such that $\boldsymbol{x}_{i+1} = \boldsymbol{y}_i$. Connecting multiple such layers gives us what is called a deep neural network.

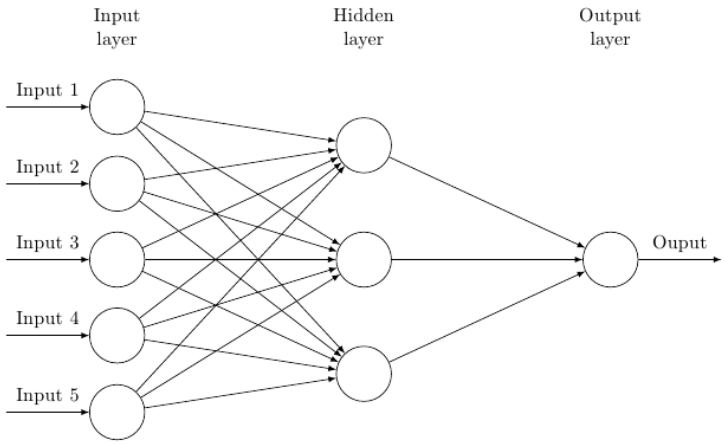
2.1.3 Deep Learning

As established above we can express a deep network as a recursive function of inputs multiplied with weighting matrices. This leads to the property that the output of the network can be differentiated in respect to each of the inputs to the network as well as the inputs to each layer. This property is essential to the way we can train a deep network to learn an approximation of any function. The method mostly used for this is called gradient descent. The algorithm for gradient descent can be written as

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.3}$$

Here $\boldsymbol{\theta}$ collectively contains both the weights and biases. $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ is the gradient of a loss function $J(\boldsymbol{\theta})$ with respect to the parameters $\boldsymbol{\theta}$ and α is the learning rate. It is when calculating the gradient of the loss function that the differentiation property becomes useful

Figure 2.2 An Artificial Neural Network with 5 inputs, 3 neurons in a single hidden layer and a single output



by using the chain rule and backpropagation. Given a network described by the function $y = f(g(x))$ the chain rule give us the following

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

We can see from this that the gradient of the entire network in respect to the network input is simply the product of the individual gradients of each layers output with respect to their individual inputs. This way each layers gradient can be multiplied back through the network from the output layer all the way to the input layer to calculate the total gradient. This is what we call backpropagation.

Despite this apparently simple method for training deep networks there are several other aspects complicating the process. Two of these are over- and under-fitting, which mean that the network is trained so precise that it only works on the training data itself (overfitting) or that it is not trained enough to actually learn the desired approximation (underfitting). There are several different causes which can lead to both these and other complications, and correspondingly several methods for avoiding them. Some examples are data augmentation, early stopping and dropout (13).

2.1.4 Activation functions

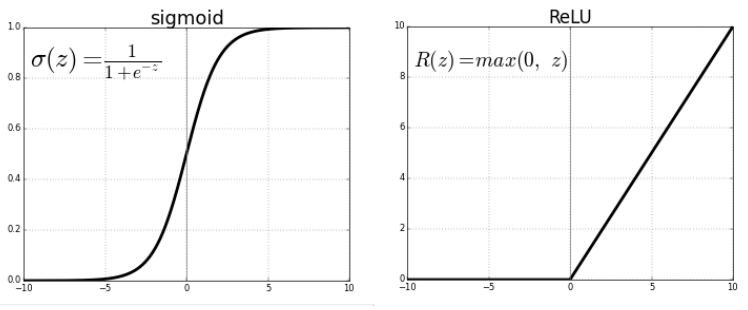
As mentioned above there are several different activation functions commonly used in neural networks. Two of these are the sigmoidal function and the rectified linear function, commonly called the relu function given in respective order as

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

$$R(x) = \max(0, x) \tag{2.6}$$

Each of these have the property of being nonlinear, which can be seen in figure 2.3. This property is essential for a neural network to be able to approximate nonlinear functions.

Figure 2.3 Plots of the sigmoid and relu activation functions



2.1.5 Transfer learning

Transfer learning is a method used when training ANNs to certain tasks or function approximations where a network is first trained to complete a simpler task before using this same network as a starting point for training to complete a more advanced task. In (7) transfer learning is used by first learning path-following for a straight line path before learning curved-path following which is more advanced. For the first task the network is trained from scratch, however for the second task the training rather starts from the pre-trained network already capable of performing the straight line path following.

In general transfer learning allows for faster training of a network than training from scratch. This is due to the fact that pre-trained networks often have learned certain features which can be translated to the new task as well. This obviously then leads to the network not needing to learn these features all over again and thus reduces the need for more training.

2.2 Reinforcement Learning

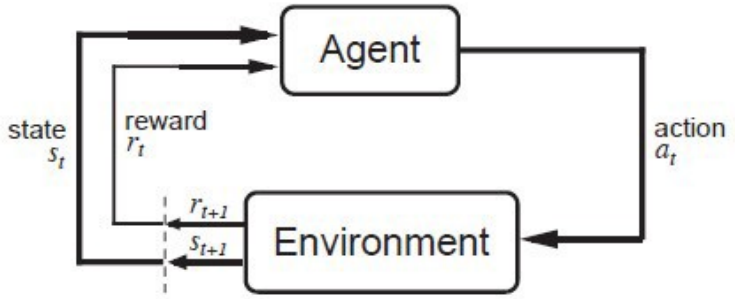
The forms of artificial learning described so far all in a way seek to find certain patterns in sets of training data. However reinforcement learning (RL) is more of a ”learn by doing” approach. A RL agent must explore and interact with its environment to maximize a reward signal. It is therefore required that an agent has some way of sensing its environment to know how it reacts to different actions, however there is no need of a predefined correct solution. This way of learning also brings up the problem of balancing exploration and exploitation which will be discussed later.

2.2.1 Elements of Reinforcement Learning

When using reinforcement learning we separate a system into two main parts, the agent and environment. The agent is the core of the system which is trained to perform a certain

task and later performs it. The environment is everything surrounding the agent which it cannot explicitly control. The agent will perform an action which allows it to encounter with the environment. Next the agent will observe a reward and a new state resulting from its action. These observations will then lead to the agent performing a new action. An illustration of this is shown in figure 2.4

Figure 2.4 Illustration of a reinforcement agent and environment interacting



In RL systems the agent and environment are built up of four main elements. These are reward signal, value function, policy and a model of the environment. We will now look closer at each of these elements to see how they play into solving a RL problem.

Reward signal

At each time-step the agent will receive a signal from the environment called a reward. This is a number which indicates how good (or bad) it is for the agent to be in the current state. Overall the main goal for any RL agent is to maximize total reward over time. Thus the reward signal ultimately indicates the goal of the RL problem. In addition to defining the main objective for the RL agent, the reward signal can include several elements corresponding to any sub-objectives which may be relevant.

Value function

As we just established, the reward signal indicates how good a certain situation is for the agent. In other words this only takes into account the current situation. The value function, however, indicates what is good in the long term. The value of a certain state is defined as the expected amount of reward the agent can accumulate starting at that particular state. This is different from the reward signal in the way that it anticipates rewards from events later in time which are likely to happen when moving on from the current state.

Policy

The policy in a RL system can be described as a set of rules for how the agent should act in a given situation. For any given state, the policy is a mapping of what action to perform

when in that state. An optimal policy will have the property of always choosing the action which leads to the state with the highest value given by the value function. The policy is what defines the behaviour of the agent.

Model

The final element of a RL system is a model of the environment in the form of a transition function. This model is meant to mimic the behaviour of the environment such that the agent will be able to make guesses on how the environment will act in the future, making it easier to choose the most optimal actions. Some RL systems do not have models, making them purely trial-and-error systems which have no knowledge of the behaviour of the environment until interacting with it. Having a model, however, makes it possible for the agent to "plan" rather than simply performing trial-and-error.

2.2.2 Markov decision process

A Markov decision process (MDP) is a special case of a sequential decision process where the environment is fully observable, stochastic, and all states are Markov states. We say that a state is a Markov state if it satisfies the Markov property, which states that "The future is independent of the past given the present". In other words we can say that a MDP is memoryless. The only state affecting the future states is the current state, thus we do not need to store information about previous states.

We denote a MDP as a tupple (S, A, R, T) where S is the set of all states s, A is the set of possible actions a, R is the reward function and T is the transition function. In most cases the current state will effect which actions are possible, therefore we often denote A as $A(s)$ where $s \in S$. In a similar way the reward is dependant on the current state and the action taken from this state, thus we denote R as $R(s, a)$ where again $s \in S$ 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. It must fulfill the criteria $T(s, a, s') \in [0, 1]$ and $\sum_{s'} T(s, a, s') = 1$.

2.2.3 Bellman equation

The Bellman equation is a way of giving a value to a state. This value is based off of immediate reward of arriving at the state and an estimate of the value of the remaining future decisions. By being able to calculate values for our states we will be able to choose an optimal policy from any given state. According to 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 (14)

This principle is quite similar to the Markov property discussed earlier in the sense that the optimal policy from a given state is independent of earlier states and actions. This

fact allows us to divide the problem of finding an optimal policy into several simpler subproblems. This method of breaking down a problem is known as Dynamic Programming, which we will come back to later.

To formulate the Bellman equation we will start with the simplest case where we have a discrete deterministic MDP. We begin with the value of being in state s at a given time t as the expected sum of all future rewards.

$$V(s_0) = E\Big\{\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\Big\} \tag{2.7}$$

Here $\gamma \in [0, 1]$ is the discount factor. Setting the discount factor to a value less than 1 acts as introducing a sence of urgency in the decision making since the contribution of the rewards to the overall sum will mitigate over time. Next we know the transfer function to be $T(s_t, a_t, s_{t+1}) = P(s_{t+1}|s_t, a_t)$ so that we have:

$$\begin{aligned} V(s_0) &= \sum_{t=0}^{\infty} T(s_t, a_t, s_{t+1})\gamma^t R(s_t, a_t, s_{t+1}) \\ &= T(s_0, a_0, s_1)R(s_0, a_0, s_1) + \sum_{t=1}^{\infty} T(s_t, a_t, s_{t+1})\gamma^t R(s_t, a_t, s_{t+1}) \\ V(s) &= \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V(s')) \end{aligned} \tag{2.8}$$

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:

$$V^*(s) = \max_{a \in A(s)} \sum_{s'} T(s, a, s')(R(s, a, s') + V^*(s')) \tag{2.9}$$

This equation only works for discrete deterministic MDP. Fortunateley we have the Hamilton-Jacobi-Bellman equation, which is an extension of the Bellman equation for continuous time systems. Given a continuous time system

$$\dot{s} = f(s, a) \tag{2.10}$$

We have the Hamilton-Jacobi-Bellman equation given as

$$V^*(s, t) = \max_{a \in A(s)} \left\{ \frac{dV(s, t)}{ds} + R(s, a) \right\} \tag{2.11}$$

Using either the Bellman optimality equation or the Hamilton-Jacobi-Bellman equation we can now find the value function $V^(s)$ which can be used with dynamic programming (DP) to find an approximation of the optimal policy.

2.2.4 Dynamic programming

As mentioned earlier, dynamig 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 to a MDP. Given a MDP which perfectly models the environment, in other words full knowledge of the transfer-function $T(s, a, s')$ and reward-function $R(s, a, s')$, we can use a method called policy iteration to find an optimal policy.

Policy iteration

Policy iteration is an iterative computation devided into two subproblems, 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. 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 pseudocode 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 for the given policy, while the policy improvement attempts to make the policy greedy in regards 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. 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

Algorithm 1 Policy Iteration

procedure INITIALIZATION
    $V(s) \in \mathbb{R}$ and $\pi(s) \in A(s)$ arbitrarily, commonly $V(s) = 0$
end procedure
procedure POLICY EVALUATION
    repeat
        $\Delta \leftarrow 0$
        for each $s \in S$ **do**
            $v \leftarrow V(s)$
            $V(s) \leftarrow \sum_{s'} T(s, \pi(s), s')(R(s, a, s') + \gamma V(s'))$
            $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
        end for
    until $\Delta \leq \theta$
end procedure
procedure POLICY IMPROVEMENT
    $policystable \leftarrow true$
    for each $s \in S$ **do**
        $oldaction \leftarrow \pi(s)$
        $\pi(s) \leftarrow argmax_a \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma V(s'))$
        if $\pi \neq oldaction$ **then** $policystable \leftarrow false$
        end if
    end for
    if $policystable$ **then return** $V \approx v_*$ and $\pi \approx \pi_*$
    else
        goto Policy Evaluation
    end if
end procedure

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 (bootstrapping). 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 method is the constant-alpha method described by the following function:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{2.12}$$

Here α is the learning rate and G_t is the sum of all returns in the given episode. In other words

$$G_t = \sum_{t=0}^{T} R(s_t, a_t, s_{t+1}) \tag{2.13}$$

where T is the number of time-steps in an episode. These methods have certain advantages over DP methods. For example basing the operation on sample experience allows for simulation based training. 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 Temporal difference methods

As mentioned earlier, Temporal difference (TD) methods utilize advantageous aspects of both DP and MC methods. For the policy evaluation problem, also known as the prediction problem, we start with an equation similar to 2.12. This approach will need to complete a full episode before estimating the value-function. Seeing as this might be disadvantageous, or even impossible in certain scenarios, we want to incorporate the capability to update the value-function estimate at each time step. Rather than waiting for the episode to terminate such that we know the value of G_t, we will at the next time step $t+1$ look at the observed reward R_{t+1} and the value estimate $V(S_{t+1})$. We then have the following equation:

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{2.14}$$

This is the simplest form of TD methods known as $TD(0)$ or *one-step TD*. This is quite similar to DP in that the value is updated for each time step. Other forms of TD methods such as $TD(\lambda)$, or *n-step TD*, are more of a middle way between DP and MC. Here the observed reward is added over several time steps before updating the value estimate (hence n-step). This is expressed in the following equation:

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{2.15}$$

where G_t^n is given by

$$G_t^n = \sum_{t=0}^{n} \gamma^t R(s_t, u_t, s_{t+1}) \tag{2.16}$$

This differs from the G_t used in MC methods in that it is the sum of discounted rewards over n time steps rather than a full episode. This adds the benefit of reducing the number of iterations necessary for the value estimate to converge by propagating values of later states back to earlier states more quickly. However, we still need to wait for n steps before we can compute an update for the value estimate, which reintroduces some of the difficulties with MC methods.

2.2.7 Q-Learning

So far we have based all our methods on the state values. Q-Learning, however, focuses on the action value rather than the state value. Where the state value $V(s)$ is the sum of all future rewards from following a policy from the state s, the action value $Q(s, a)$ is the sum of all future rewards from starting in state s, performing action a, then following the policy.

$$Q(s, a) = \sum_{s'} T(s, a, s')(R(s, a, s') + \gamma \max_{a'} Q(s', a') \tag{2.17}$$

Similarly to the update rule for TD(0), we have the following update rule for Q-learning:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\big[R_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\big] \tag{2.18}$$

This action value function will converge towards its optimal value given sufficient exploration. At that point it is appropriate to follow a greedy policy, or in other words always taking the action which gives the highest value for $Q(s, a)$. It is however important to maintain a sufficient level of exploration during learning to ensure the values will in fact converge. This can be done in several different ways which we will come back to later.

2.2.8 Deep reinforcement learning

So far all the methods discussed assume we have the ability to represent values and policies for all states and state-action pairs in the form of a table. 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”, 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 functions 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 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 action-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 action-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.

2.2.9 Actor-only methods

Policy gradient methods

Policy gradient methods (15) 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 as described in chapter 2. We then assume that the policy $\pi_{\boldsymbol{\theta}}(s)$ can be written 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 discounted future reward, we choose the following cost function

$$J(\boldsymbol{\theta}) = E\Big[\sum_{t} \gamma^t R(s_t, a_t)\Big] \tag{2.19}$$
$$a_t \sim \pi_{\boldsymbol{\theta}(s_t)}$$

where a_t is the action sampled from policy $\pi_{\boldsymbol{\theta}}$ at time t. 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 this cost function.

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, J(\boldsymbol{\theta}) = \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\, E\Big[\sum_{t} \gamma^t R(s_t, a_t)\Big] \tag{2.20}$$
$$a_t \sim \pi_{\boldsymbol{\theta}(s_t)}$$

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

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \tag{2.21}$$

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

Since we are using a deep neural network to generate an approximation of our policy we can assume that we know the gradient $\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(s, a)$. Also, since we have the policy as a

probability distrobution we have that

$$\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(s,a) = \pi_{\boldsymbol{\theta}}(s,a) \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s,a) \tag{2.22}$$

We can then calculate the gradient of the objective function as

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) = E\Big[\Big(\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_t, a_t)\Big(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}\Big)\Big)\Big] \tag{2.23}$$

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

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^{N} \Big(\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_{i,t}, a_{i,t})\Big(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}\Big)\Big) \tag{2.24}$$

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 highest in the direction leading to the highest return. We will therefore be moving towards the actions leading to maximum return during gradient descent, 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 (16) 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

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \approx \frac{1}{N} \sum_{i=1}^{N} \Big(\sum_{t=1}^{T} \nabla_{\boldsymbol{\theta}} \log \pi_{\boldsymbol{\theta}}(s_{i,t}, a_{i,t})\Big(\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} - b\Big)\Big) \tag{2.25}$$

2.2.10 Actor-critic methods

In actor-critic methods we combine policy approximation methods introduced above with similar methods for approximating the value function. This means that we now have two sets of approximator parameters. We therefore denote the parameters for the value function approximator (critic) as $\boldsymbol{\theta}_c$ and the policy approximator (actor) as $\boldsymbol{\theta}_a$. This gives us the approximated value function and policy $V_{\boldsymbol{\theta}_c}(s)$ and $\pi_{\boldsymbol{\theta}_a}(s)$ respectively.

We know how to approximate the policy from the actor-only methods, thus the remaining task is to approximate the value function. Where we used the maximization of rewards as our basis for the cost function for our policy approximation we need a similar measure to use for our value function. We can define the temporal difference error as the difference in the left and right hand side of the Bellman equation. This gives us

$$\delta = R(s,a,s') + \gamma V_{\boldsymbol{\theta}_c}(s') - V_{\boldsymbol{\theta}_c}(s) \tag{2.26}$$

For the optimal value function $V*(s)$ we would have $\delta = 0$, thus we want to minimize this error. By using the square of this error as our cost function we will achieve this minimization through gradient descent. Therefore we set our cost function as

$$J_c(\boldsymbol{\theta}_c) = \frac{1}{2}\delta^2 \tag{2.27}$$

The gradient of this cost function will then be

$$\nabla_{\boldsymbol{\theta}_c} J(\boldsymbol{\theta}_c) = \delta \nabla_{\boldsymbol{\theta}_c} \delta = \delta \nabla_{\boldsymbol{\theta}_c} V_{\boldsymbol{\theta}_c}(s) \tag{2.28}$$

Here we have made the assumption that the value function estimate of the next state is independent of the parameters $\boldsymbol{\theta}_c$ such that this term is treated as a constant when calculating the gradient of the temporal difference error in respect to $\boldsymbol{\theta}_c$. This gives us the gradient descent update law for the critic parameters

$$\boldsymbol{\theta}_c \leftarrow \boldsymbol{\theta}_c + \alpha_c \delta \nabla_{\boldsymbol{\theta}_c} V_{\boldsymbol{\theta}_c}(s) \tag{2.29}$$

Now that we have our update law for the critic we go back to the actor. With our starting point at equation 2.25 we have a few adjustments we can make. First of all, by using the value function approximation as our baseline it can be shown that

$$\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} - b = \delta \tag{2.30}$$

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

$$\nabla_{\boldsymbol{\theta}_a} J(\boldsymbol{\theta}_a) \approx \delta \nabla_{\boldsymbol{\theta}_a} \log \pi_{\boldsymbol{\theta}_a}(s) \tag{2.31}$$

which then gives us our update law for the actor parameters

$$\boldsymbol{\theta}_a \leftarrow \boldsymbol{\theta}_a + \alpha_a \delta \nabla_{\boldsymbol{\theta}_a} \pi_{\boldsymbol{\theta}_a}(s) \tag{2.32}$$

2.2.11 Deep Deterministic Policy gradients

The Deep Deterministic Policy gradients (DDPG) method (17) is an adaption of actor-critic method which utilizes elements of Q-learning. Rather than approximating the state value function $V(s)$ as in actor-critic methods it approximates the action value function $Q(s, a)$. As discussed in chapter 2.2.7 this makes DDPG an off-policy algorithm, meaning it can learn the optimal policy without following it.

The cost function for the action value function $Q_{\boldsymbol{\theta}_Q}(s, a)$ parameterized by the parameters $\boldsymbol{\theta}_Q$ can be given as the squared of the temporal difference error such as with the state value function. For the policy $\pi_{\boldsymbol{\theta}_\pi}$ we can now use the action value function as the cost function since it gives the value of each action in each state. This gives us

$$\nabla_{\boldsymbol{\theta}_\pi} J_\pi(\boldsymbol{\theta}_\pi) = \nabla_{\boldsymbol{\theta}_\pi} Q_{\boldsymbol{\theta}_Q}(s, \pi_{\boldsymbol{\theta}_\pi}(s)) \tag{2.33}$$

$$= \nabla_a Q_{\boldsymbol{\theta}_Q}(s, a) \nabla_{\boldsymbol{\theta}_\pi} \pi_{\boldsymbol{\theta}_\pi}(s) \tag{2.34}$$

There are also several ways of stabilizing training when using DDPG. One of these methods is called soft parameter updates (17) where target networks which are constrained to change slowly are used to update the parameters. Another method is using experience replay (18) where the training data is randomly sampled from the memory.

2.2.12 Exploration vs exploitation

One aspect that significantly separates RL from other forms of ML is exploration. The fact that the agent must interact with the environment to know if its actions are good or bad provides a new problem, namely balancing exploration and exploitation. By exploitation we mean acting based on previously learned knowledge of what actions are most rewarding, while exploration is trying new actions in spite of having found the seemingly best action pattern. An algorithm focusing only on exploration will potentially never reach the goal state as it will always rather look for new action patterns that continue on promising actions, while focusing purely on exploitation will, unless you are extremely lucky, lead to a sub-optimal solution. The balance between these two is therefor crucial to finding an optimal solution. Luckily there are several ways to implement a balance between the two.

ε-Greedy method

An ε-greedy method is one of the simplest ways to ensure sufficient exploration, while still taking advantage of what we currently believe to be the best policy. We start by following a normal greedy policy. We then select a probability ε to be the probability of the agent not following the greedy policy. Instead in these instances the agent will choose an action randomly from the available set of actions. This means that as the number of episodes goes to the limit we can ensure that all actions will be attempted from all states to a sufficient degree to converge to the optimal action value function. This method can be adjusted to make both the probability ε and the random selection process more advanced to ensure faster or more accurate convergence.

Simulated Annealing

Simulated Annealing is a more advanced way of controlling exploration. This method is based on the simulation of a process of heating and cooling metals in a controlled manner to return the metal to its low energy ground state. The mathematical representation of this process has been adapted to solve large combinatorial optimization problems. For more details on the origin of this method see (19). An example of how simulated annealing can be used to improve a RL algorithm is shown in (5).

2.3 Marine Vessel Kinematics

The kinematics of a marine vessel are central in making both a simulation model and a control system. From a RL point of view it is also beneficial to have knowledge about the kinematics when augmenting our state vector and designing a reward function. From (20) 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,

sway and yaw so that

$$\boldsymbol{\eta} = \begin{bmatrix} x \\ y \\ \psi \end{bmatrix} \boldsymbol{\nu} = \begin{bmatrix} u \\ v \\ r \end{bmatrix} \tag{2.35}$$

We then have from (20) the model of the 3-DOF system on vectorial form as

$$\dot{\boldsymbol{\eta}} = \boldsymbol{R}(\psi)\boldsymbol{\nu} \tag{2.36}$$

$$\boldsymbol{M}\dot{\boldsymbol{\nu}} + \boldsymbol{C}(\boldsymbol{\nu})\boldsymbol{\nu} + \boldsymbol{D}(\boldsymbol{\nu})\boldsymbol{\nu} = \boldsymbol{\tau} \tag{2.37}$$

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

$$\boldsymbol{R}(\psi) = \begin{bmatrix} cos(\psi) & -sin(\psi) & 0 \\ sin(\psi) & cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.38}$$

These pose and velocity vectors are therefore ideal as starting points for our state vector when implementing a RL algorithm.

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	ϕ
5	Pitch	M	q	θ
6	Yaw	N	r	ψ

Chapter 3

Previous work

As mentioned earlier, there have been several studies in recent years on subjects related to path planning for autonomous marine vessels using reinforcement learning. In this chapter a few of these studies will be looked at more in depth. Their problem formulations will be compared to the problem at hand in this thesis. Their methods for solutions will also be discussed to see how they may be used or adapted for the given problem.

3.1 Path planning

In recent years there have been several studies on path planning for autonomous robots in uncertain environments. Most of these studies are done on simplified forms of the problem using discrete, finite state and action spaces. This simplification makes it easier to compute the value function and policy, however it does not translate well into real world scenarios, such as autonomous docking, where both the state and action spaces are continuous. Despite this fact, there may still be several aspects of the design and implementation of such solutions which could be used to solve the continuous case as well. We will now look at some chosen solutions containing different elements which we may be able to take advantage of later when designing our own system. For each solution the most relatable aspects as well as some of their results are considered in this section without going into all the details of each system. For a full understanding of each of them the papers themselves should be read.

3.1.1 Simulated Annealing Q-Learning

An approach to the path-planning problem is presented in (5). Here the problem is presented as a robotic fish whose goal is to reach a goal state in a unknown deterministic discrete state space. This approach uses Simulated Annealing (SA) to ensure sufficient exploration while using Q-Learning to find the optimal policy.

Reward function

For designing their reward function, the goal-oriented principle was used. This means that the reward is mostly based on the distance from the agent, in this case the robotic fish, to the goal state. This was done by defining the variable d_t as the linear distance from the current state s_t to the goal. After performing an action the new distance d_{t+1} is calculated. A reward of 5 would then be given if $d_{t+1} < d_t$ and a reward of -5 if $d_{t+1} > d_t$. In addition to this goal-oriented reward a reward of 10 is given for reaching the goal state and a reward of -10 is given for encountering an obstacle. The overall reward function can be described as

$$R = \begin{cases} 10 & \text{if arrive at goal} \\ 5 & \text{if } d_{t+1} < d_t \\ -5 & \text{if } d_{t+1} > d_t \\ -10 & \text{if encounter obstacle} \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

Pseudo code of the full solution is shown in algorithm 2. After implementing the algorithm on the robotic fish it was then run 6 separate times from scratch. Each time the fish was able to find its way to the goal in a minimal number of steps without crashing. Based on the objective indicated by the reward function all paths, despite being different, are in fact optimal solutions.

3.1.2 Improved Q-learning

In (6) the problem of path planning for a mobile robot in an unknown environment is considered. A new variation of Q learning, called Improved Q-learning (IQL), is proposed and tested. The new algorithm differs from classical Q learning in that the action value $Q(s, a)$ is at a certain point declared to no longer be in need of updating and therefore locked. This results in a much faster computation of the entire Q table since we no longer need to iterate through all states and actions for each update sweep. Certain assumptions are made about the system for IQL to be possible. First one must have a discrete finite state and action space. It is also assumed that the distances from the current state to any next state and the goal are known. Based on these assumptions the paper suggests four properties such that if at any time at least one of these is upheld and either the current or next state has a locked Q value the unlocked state may update its Q value only once then lock it.

The findings of the paper are that the IQL algorithm outperforms the classical QL algorithm in multiple ways. First of all, and most obvious, it significantly reduces the time complexity. In addition the new algorithm saves significant storage by only storing the Q value for the best possible action at each state, rather than storing the Q value for all actions at all states. Also the path planning preformed using the stored Q table from the new algorithm has a better traversal time and number of states traversed, as well as minimizing the number of 90° turns. The later was a condition added to reduce the energy consumption of the robot.

Algorithm 2 Simulated Annealing Q Learning

1: Initialize temperatures T_0 and T, policy $\pi(s)$, max number N, learning rate α and discount factor γ
2: **for** Each episode **do**
3: Initialize current state s_t, save into linked list now_route_state
4: Initialize $T = T_0$
5: **for** each step **do**
6: Select action a_r from $A(s_t)$
7: Select action a_p according to $\pi(s_t)$ and let current action $a_t = a_p$
8: Generate random number δ uniformly distributed on interval $[0, 1]$
9: Compute $p = \exp\left(\frac{Q(s,a_r)-Q(s,a_p)}{T}\right)$
10: **if** $p > \delta$ **then**
11: $a_t = a_r$
12: **else**
13: $a_t = a_p$
14: **end if**
15: Execute a_t, save new state s_{t+1} to now_route_state, observe reward r_t
16: Update $Q(s, a)$ by formula 2.18
17: **if** $s_t =$ goal state **then**
18: Next
19: **else**
20: $t = t + 1$
21: Go to step 5
22: **end if**
23: **end for**
24: Cool down temperature value T
25: **if** $now\_route\_state = last\_route\_state$ **then**
26: $route = route + 1$
27: **else**
28: $last\_route\_state = now\_route\_state$
29: **end if**
30: **if** $route < N$ **then**
31: $episode = episode + 1$
32: **else**
33: Algorithm has ended
34: **end if**
35: **end for**

3.2 Path following

Seeing as the path planning problems discussed above work with discrete action-spaces their methods may prove difficult to transfer to our continuous action space. There have however been done studies on path following for continuous action spaces. Although the problems of path following and path planning have their differences, the RL methods used for path following might prove better suited to solve our path planning problem due to their continuous action spaces. We will therefore look at one of these approaches to see what methods can be taken advantage of.

3.2.1 Deep Deterministic Policy Gradient Method for path following

In (7) a Deep Deterministic Policy Gradient method is implemented for path following in a continuous action space. The full algorithm for the implementation is shown in algorithm 3. As discussed earlier, DDPG is an off-policy method, meaning it learns an optimal policy without having to follow it. This again leads to it being able to learn from someone else performing the exploration. This is brought forward as one of the reasons for the choice of this algorithm, since it is beneficial for a marine vessel to be able to learn from a simulation rather than having to explore several policies in real life.

The DDPG method uses neural networks to approximate the action-value function $Q(s, a)$ and the policy $\pi(s)$ separately. Both networks consisted of two hidden layers, containing 400 and 300 hidden nodes respectively. The relu activation function

$$relu(x) = \max x, 0$$

was used between layers. For the policy approximator the output layer had the number of outputs needed to perform the control task. Each had a hyperbolic tangent activation function with output between -1 and 1 which was scaled by a linear transformation to fit the saturation of the corresponding actuator, giving the control output vector $\boldsymbol{u}$. The network can be represented as the following function:

$$\begin{aligned}
\boldsymbol{h}_1(\boldsymbol{s}) &= \text{relu}(\boldsymbol{W}_1\boldsymbol{s} + \boldsymbol{b}_1) \\
\boldsymbol{h}_2(\boldsymbol{s}) &= \text{relu}(\boldsymbol{W}_2\boldsymbol{h}_1(\boldsymbol{s}) + \boldsymbol{b}_2) \\
\pi(\boldsymbol{s}) &= \tanh\left(\boldsymbol{W}_3\boldsymbol{h}_2(\boldsymbol{s}) + \boldsymbol{b}_3\right)\boldsymbol{u}_{\text{scale}} + \boldsymbol{u}_{\text{mean}}
\end{aligned}$$

where $\boldsymbol{s}$ is the input state vector, $h_i(s)$ is the function of hidden layer i, $\boldsymbol{W}_i$ is the weight matrix for layer i and $\boldsymbol{b}_i$ is the bias vector for layer i, $\boldsymbol{u}_{\text{scale}}$ and $\boldsymbol{u}_{\text{mean}}$ are the adjustment vectors for the control output vector.

The network for the action value approximator had a similar architecture with two hidden layers of 400 and 300 nodes and relu activation functions. Here the state vector $\boldsymbol{s}$ was input for the first layer and the second layer had both the output from the first layer and the action vector $\boldsymbol{a}$ as inputs. The output was the scalar value from the output layer.

The network is represented as functions as follows:

$$\begin{aligned}\boldsymbol{h}_1(\boldsymbol{s}) &= \text{relu}(\boldsymbol{W}_1\boldsymbol{s} + \boldsymbol{b}_1)\\ \boldsymbol{h}_2(\boldsymbol{s},\boldsymbol{a}) &= \text{relu}(\boldsymbol{W}_{2,\boldsymbol{s}}\boldsymbol{h}_1(\boldsymbol{s}) + \boldsymbol{W}_{2,\boldsymbol{a}}\boldsymbol{a} + \boldsymbol{b}_2)\\ Q(\boldsymbol{s},\boldsymbol{a}) &= \boldsymbol{W}_3\boldsymbol{h}_2(\boldsymbol{s},\boldsymbol{a}) + \boldsymbol{b}_3\end{aligned}$$

The next crucial element is the reward function. This is as mentioned what defines the task of the agent and is thus very particulate to the task. For this path following case it was based on a performance measure of how close the vessel was to the desired path. In addition a term was added to the reward function which added penalty for aggressive control actions. This was done to reduce both mechanical wear on the control surfaces and passenger discomfort.

When training the algorithm two different approaches were tested. First the algorithm was trained from scratch for both straigh-path following and curved-path following. For the second approach the algorithm was trained from scratch for straight-path following, which is the simplest of the two tasks. The policy and value function learned for the straight-path task was then used as the starting point for the training of the more advanced curved-path task. This is called transfer learning and has been explained in chapter 2.1.5.

The results showed especially two things of interest. Fist of all it showed that the algorithm was able to learn both the straight-path and curved-path following better than a traditional path-following algorithm. Secondly it was proved that the performance of the curved path follower was improved when using transfer learning from the straight line follower compared to training from scratch. This could be useful to keep in mind for the implementation of our autonomous docking solution.

Algorithm 3 Deep Deterministic Policy Gradient

Initialize critic $Q_{\boldsymbol{\theta}_Q}(s, a)$ and actor $\pi_{\boldsymbol{\theta}_\pi(s)}$ randomly with weights $\boldsymbol{\theta}_Q$ and $\boldsymbol{\theta_\pi}$.
Inilialize target networks $\boldsymbol{\theta}_{Q'} \leftarrow \boldsymbol{\theta}_Q$ and $\boldsymbol{\theta}_{\pi'} \leftarrow \boldsymbol{\theta}_\pi$
Initialize replay buffer
for each episode from 1 to M **do**
    Initialize random process $\mathcal{N}$ for action exploration
    Receive initlial observation state s_1
    for each t from 1 to T **do**
        Select action $a_t = \pi_{\boldsymbol{\theta}_\pi}(s_t) + \mathcal{N}_t$
        Perform action a_t
        Observe s_{t+1} and r_t
        Save transition (s_t, a_t, r_t, s_{t+1}) in replay buffer
        Sample N transitions (s_t, a_t, r_t, s_{t+1}) from replay buffer
        Set $y_i = r_i + \gamma Q_{\boldsymbol{\theta}_{Q'}}(s_{i+1}, \pi_{\boldsymbol{\theta}_{\pi'}}(s_{i+1}))$ for $i \in 1 \cdots N$
        Update critic by minimizing loss: $\frac{1}{N}\sum_i \left(y_i - Q_{\boldsymbol{\theta}_Q}(s_i, a_i)\right)^2$
        Update policy with: $\frac{1}{N}\sum_i \nabla_{u_i} Q_{\boldsymbol{\theta}_Q}(s_i, a_i) \nabla_{\boldsymbol{\theta}_\pi} \pi_{\boldsymbol{\theta}_\pi}(s_i)$
        Update critic target network: $\boldsymbol{\theta}_{\pi'} = (1-\tau)\boldsymbol{\theta}_{\pi'} + \tau\boldsymbol{\theta}_\pi$
        Update actor target network: $\boldsymbol{\theta}_{Q'} = (1-\tau)\boldsymbol{\theta}_{Q'} + \tau\boldsymbol{\theta}_Q$
    end for
end for

Chapter 4

Analysis and Design

In more conventional autonomous control of vehicles one will usually have separate subsystems for path-planning and path-following. The path-planning subsystem will be fed input containing goal coordinates along with waypoints avoiding any obstacles. Based on these inputs and any limitations, such as turning rate and radius of acceptance, an optimal path will be calculated. This path will then be fed into the path-follower subsystem which will generate control output based on the position of the vehicle relative to the desired position along the path.

Recent studies have been done on the use of RL in the path-following subsystem where RL algorithms have been used to generate control output to follow pre-generated paths of various complexity (7; 8). These studies show promise in this field and have shown that various RL algorithms are able to optimally follow different paths.

In this thesis, however, we wish to use RL not only to follow a pre-generated path but also to generate the path itself. Given the fact that RL requires actual interaction with the environment to be able to learn anything it could even prove most useful to combine the path-generation and -following into a single RL algorithm which will allow the vehicle to find its way to the goal in an optimal manner.

4.1 Scope and assumptions

Before any decisions are made in regard to the algorithm and other deciding factors it is important to specify the scope we want for our solution, as well as any assumptions necessary for a solution to be feasible. In the case of autonomous docking there are several ways in which the scope may be altered based on the desired degree of simplification. One might wish to simplify the problem to a discrete state and action space such as in (6; 5). This allows for use of a wider range of algorithms which are not able to handle continuous state and action spaces. In this thesis the goal is to come up with an algorithm which could in fact, at least in theory, be implemented on a real life marine vessel, thus it is not beneficial to simplify the state and action spaces in this way. We therefore need to implement any simplifications in other ways. For example we would have to limit the

state vector based on what measurements actually are available to us. To begin, however, it will be simplest to assume that we are provided with all necessary measurements for the proposed solution. Once we have a working solution it would then be possible to make adjustments so that the system will work even with sub-optimal measurements.

4.2 Algorithm

Based on the scope chosen for the problem we must choose an algorithm capable of working with a continuous state and action space. This is, as mentioned, to make the solution as realistic as possible such that it may be implemented in real life. From the theory and previous works mentioned in this thesis we know that this reduces the number of possible methods. Although there are still plenty of methods to choose from the choice in this case falls on using the Deep Deterministic Policy Gradient method. This choice is based in part on personal preference among the methods researched and also the fact that this method was shown to work well in the continuous state and action space problem of path following in (7). Given the similarities in the problem considered in that thesis and the one at hand it is reasonable to believe that the method will lead to a feasible solution.

4.3 Reward function

Now that the algorithm/method has been chosen another vital element to consider is the reward function. As mentioned earlier the objective of any RL problem is defined by the reward function, since any RL system seeks to maximize the return. It has also been mentioned that the reward function can include elements corresponding to various sub-objectives to better define the behaviour of the agent, meaning we must take several considerations besides the main objective when designing our reward function.

The objective for our problem is to be able to safely maneuver a marine vessel to a desired position in a dock. This can be split into the two separate goals of maneuvering to the desired position and doing so safely. We start by considering the first of these goals.

In the simplified discrete cases considered in chapter 3 we have two different approaches to achieving this goal. In (5) there is only given a reward at the actual goal, while in (6) there is also given a reward based on the distance to the goal. Although this addition is not necessary for a RL agent to be able learn to find the goal, it represents a sub-objective of always moving in the right direction. Also, since this leads to rewards being given in more states than just the goal, it helps increase the convergence speed of the value function. Therefore we propose including this in some way in our own reward function. So far we can formulate it as

$$R(s, a) = r_g + r_d \tag{4.1}$$

where the term r_g represents the reward given at the goal state and r_d represents the sub-objective of always moving in the direction of the goal.

In (7) we also see that the reward given for being sufficiently close to the desired path is given as a Gaussian distribution. This was shown to lead to better performance in the path-following. This is intuitive seeing as the reward would grow as the vessel moved

closer within the boundary rather than having the same reward for being barely inside the boundary as for being exactly on the optimal path. This could be utilized for our reward for reaching the goal state so that

$$r_g(s) = r_{g_{max}} e^{-\frac{(d_g-\mu)^2}{2\sigma}} \tag{4.2}$$

where d_g is the distance from the vessel to the goal position, $r_{g_{max}}$ is the maximum reward which will be given only exactly at the goal, μ is the mean and σ is the standard deviation. By tuning these values we can adjust the distribution and size of the reward.

The reward for moving in the right direction r_d can be implemented in several different ways. One possibility is to evaluate the distance to the goal from the current state and compare it to the distance from the goal to the previous state. The reward could then look something like

$$r_d(s) = \begin{cases} r_{closer} & \text{if } d_g(s) < d_g(s') \\ r_{further} & \text{if } d_g(s) > d_g(s') \end{cases} \tag{4.3}$$

The values for r_{closer} and $r_{further}$ would then be chosen such that r_{closer} is more favorable than $r_{further}$. They could also be designed to be proportional to the change in distance such that moving a lot closer is more beneficial than moving a little closer.

Now that we have a way of giving reward for moving to the right place we must also assign some reward, or rather punishment, for encountering obstacles. Here it would in theory suffice to assign a negative reward for actually encountering the obstacle, however since we wish to implement on an actual vessel it might be seen as a better idea to start assigning some negative reward when getting close to an obstacle. This will make it increasingly beneficial to keep obstacles at a certain distance to avoid crashing. This can also be done by implementing the obstacle punishment as a Gaussian function around the obstacle in a similar way as the reward for being close to the goal. We suggest the term

$$r_o(s) = -r_{o_{max}} e^{-\frac{(d_o-\mu)^2}{2\sigma}} \tag{4.4}$$

where d_o is the distance to the obstacle. The term $r_{o_{max}}$ represents the amplitude of the function and can be adjusted along with μ and σ to give the desired distribution of the reward. So far our proposed reward function is given by

$$R(s, a, s') = r_g(s) + r_d(s, s') + r_o(s) \tag{4.5}$$

We have now defined sufficient rewards for the system to be able to learn to find a given goal state while avoiding obstacles, however we still have several sub-objectives which may still be implemented through the reward function. As described earlier a term was added in (7) for smoothing the control output. This has benefits in our implementation as well and could therefore be added. Given a desire to minimize fuel consumption and a way of measuring said consumption a term could also be added for this. Any other desired restrictions could also be formulated as sub-objectives in the reward function to alter the behaviour of the trained agent. The benefits or drawbacks of increased complexity of the reward function should be tested by training the agent for various versions of the proposed reward functions and comparing the results.

4.4 State augmentation

Apart from the reward function the only other information available to the agent is the state vector. Although a DRL agent in theory should be able to learn a reliable and accurate accurate value function and policy from only a limited number of input states it is important to supply the agent with sufficient and valuable information through the state vector. Thus there are several considerations to take when deciding on what to include in the state vector.

Since we overall are looking to control the 2-dimensional surface movement of a vessel, a good starting point is to use the pose and velocity vectors for a 3-DOF vessel from equation 2.35. This could give the state vector

$$\boldsymbol{s} = \begin{bmatrix} x \\ y \\ \psi \\ \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} \tag{4.6}$$

This state vector should in theory be sufficient for the DRL agent to learn an optimal policy, however there are several other measurements or variations of the same measurements which will speed up the learning process. For example the state vector proposed above will require the DRL agent to learn function approximators that are able to perform quite complex transformations from these measurements to desired control actions. There is no information about the desired position for the vessel or an other information to indicate a desired behaviour. Providing such information would allow for a simpler architecture of the DRL network since the complexity of the transformation needed would be reduced. This information could be provided in multiple ways. Provided knowledge of where the goal state is one could calculate the offsets or error parameters to give

$$\boldsymbol{s} = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{\psi} \\ \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} = \begin{bmatrix} x_g - x \\ y_g - y \\ \psi_g - \psi \\ \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} \tag{4.7}$$

Alternatively one could provide a distance d_g and direction ψ_g to the goal giving

$$\boldsymbol{s} = \begin{bmatrix} d_g \\ \psi_g \\ x \\ y \\ \psi \\ \dot{x} \\ \dot{y} \\ \dot{\psi} \end{bmatrix} \tag{4.8}$$

These are simply two possible augmentations of the state vector which may improve training and performance of the DRL agent. Seeing as the distance to obstacles plays a role in the return the agent receives it might also be a good idea to provide the agent with information related to this. Similarly any other conditions playing a role in the rewards should be considered as useful information when augmenting the state vector. A more thorough analysis of possible state augmentations must be done for further work on the problem.

4.5 Actions

The system design so far is intended to combine the path-planning and path-following tasks into a single agent. The output from this agent, the actions, are sent to the vessel to control its movements. We therefore need to define these actions in a way such that they can control the vessel. This can be done in a number of different ways. One possibility is to output desired movement, much like a traditional path-follower, which the vessels control system turns into control actions. Another possibility in which the agent has most direct influence in the behaviour of the vessel is to output control actions directly to the actuators of the vessel. This should yield better overall performance since the vessels control system may not always be optimal. This is therefore chosen as the preferred solution for further work. Similar to all previously discussed system design choices there is also here the opportunity to test different implementations to see what is in fact optimal.

4.6 Exploration vs Exploitation

Now that we have established a method of creating a reward function it is important to consider the aspect of exploration versus exploitation. As discussed earlier we want to find a perfect balance between taking advantage of action patterns we have found to be rewarding and exploring new actions which could potentially prove even more rewarding. This can be implemented in several different ways. Adding noise to the policy such as in the DDPG algorithm used in (7) is a quite straight forward solution to this problem.

The slightly more systematic approach of using simulated annealing such as in (5) might, however, prove even more efficient as it controls the exploration to go from being broad in the beginning to more narrow as the policy approaches the optimal policy. However, this is also more advanced to implement and might therefore prove to be somewhat more challenging. For further work it could be beneficial to attempt to implement both solutions to examine the difference in performance from the two.

4.7 Future work

We have now considered various aspects of designing a solution to our problem and proposed methods to handle these aspects. As mentioned earlier this work is intended as preparation for future work in the form of a Master's thesis. The proposed methods will therefore be analyzed further, if needed, and implemented at that time. Once an acceptable solution is created there are several additional aspect which may be considered and implemented if possible. One of these is how the RL agent can handle moving obstacles. This

is a necessary consideration if a solution is ever to be implemented on an actual vessel. In most docks there will be other vessels present. While some may be viewed as stationary as they are fastened to the dock there is a great chance of several of these vessels moving around in the dock area. Ways to handle such situations should be included in future work.

In this work we have assumed that we have sufficient measurements of all the information we need, such as vessel and goal positions and obstacle detection. These measurements may be difficult to obtain in real life. A thorough analysis of possible sensors and other measurement units is therefore also necessary for a real life implementation to be possible.

When it comes to marine vessel control we have only introduced the most common ways of representing the kinematics of a vessel. To be able to make good choices for both the state augmentation and our desired output (actions) we need to do a more in depth study in vessel control theory. It is also necessary for creating a simulation model to train and test the RL implementation. This study will be done as part of the continuation of this work.

Chapter 5

Conclusion

This project has been written in preparation for a Master's thesis. The possibilities of using reinforcement learning to automate the docking process of marine vessels have been researched. After presenting theory on several methods and looking at previous works in related areas, a proposal is made for a possible base of a solution. Based on both the theory and previous works it is suggested that Deep Deterministic Policy Gradient method shows promise. A proposal of a possible reward function based on the overall goal with certain additions representing sub-objectives is made. Further, several other aspects of the problem are discussed for future work. In the end we can conclude that, although there are still several areas which need more work, the proposed methods should lead to a feasible solution.

Bibliography

[1] 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. [Online]. Available: http://arxiv.org/abs/1312.5602

[2] 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. [Online]. Available: http://arxiv.org/abs/1712.01815

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

[4] 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 *Experimental Robotics IX, The 9th International Symposium on Experimental Robotics [ISER 2004, Singapore, 18.-21. June 2004]*, 2004, pp. 363–372. [Online]. Available: https://doi.org/10.1007/11552246_35

[5] 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: Springer International Publishing, 2018, pp. 248–257.

[6] 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.

[7] A. B. Martinsen, "End-to-end training for path following and control of marine vehicles," 2018. [Online]. Available: http://hdl.handle.net/11250/2559484

[8] E. A. Lund, "Path following in simulated environments using the a3c reinforcement learning method," 2018. [Online]. Available: http://hdl.handle.net/11250/2563040

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

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

[11] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *CoRR*, vol. cs.AI/9605103, 1996. [Online]. Available: http://arxiv.org/abs/cs.AI/9605103

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

[13] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Improving neural networks by preventing co-adaptation of feature detectors," *CoRR*, vol. abs/1207.0580, 2012. [Online]. Available: http://arxiv.org/abs/1207.0580

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

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

[16] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3, pp. 229–256, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992696

[17] 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. [Online]. Available: http://arxiv.org/abs/1509.02971

[18] 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.

[19] P. J. M. van Laarhoven and E. H. L. Aarts, *Simulated annealing*. Dordrecht: Springer Netherlands, 1987, pp. 7–15. [Online]. Available: https://doi.org/10.1007/978-94-015-7744-1_2

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