Masteroppgave

NTNU
Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

Anne Marie Mathisen

Motion planning for Autonomous Underwater Vehicles (AUVs) in challenging sea-cage environments using Reinforcement Learning

Masteroppgave i Kybernetikk og Robotikk
Veileder: Associate Professor Martin Føre
Medveileder: Dr. Marios Xanthidis, Dr. Eleni Kelasidi
Desember 2022

NTNU
Kunnskap for en bedre verden

Anne Marie Mathisen

Motion planning for Autonomous Underwater Vehicles (AUVs) in challenging sea-cage environments using Reinforcement Learning

Masteroppgave i Kybernetikk og Robotikk
Veileder: Associate Professor Martin Føre
Medveileder: Dr. Marios Xanthidis, Dr. Eleni Kelasidi
Desember 2022

Norges teknisk-naturvitenskapelige universitet
Fakultet for informasjonsteknologi og elektroteknikk
Institutt for teknisk kybernetikk

NTNU

Kunnskap for en bedre verden

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Anne Marie Mathisen

Motion planning for Autonomous Underwater Vehicles (AUVs) in challenging sea-cage environments using Reinforcement Learning

TTK4900

December 2022

Master's thesis in Cybernetics and Robotics

Department of Engineering Cybernetics

Supervisor: Associate Professor Martin Føre

Co-supervisor: Dr. Marios Xanthidis, Dr. Eleni Kelasidi

Abstract

Autonomous Underwater Vehicles (AUVs) are already used in underwater operations where the environment or the operation is challenging due to for instance weather conditions or the duration of time spent under water in order to assist humans. At the same time the increasing demand for sustainable production of food results in a growing potential in fish farms. However, the farms mainly being located at coastal shores is a limiting factor, and and more people are looking to relocate the facilities off shore. With off shore locations maintenance procedures like cleaning, observation of the behavior of the fish and maintenance of the net become more difficult to conduct, but are important for the welfare of the fish. By incorporating autonomous technology in fish farms human personnel can be assisted in tasks that are important for the welfare of the fish while at the same time creating a safe work environment for the personnel.

AUVs rely in motion control systems to be able to operate autonomously. The motion control system is used to minimize the error between the AUVs current state and a desired state, where obtaining the desired state is done by a motion planner. The motion planner is responsible for providing a path that can handle the dynamic environment of a fish cage.

This thesis proposes to use the deep reinforcement learning algorithm Deep Deterministic Policy Gradient (DDPG) as a reactive motion planner in order to allow for obstacle avoidance during the AUVs operation. DDPG handles continuous state and action spaces, which is necessary for a motion planner.

The reinforcement learning- based motion planner operates online, and outputs the desired heading at each step that allows the AUV to avoid obstacles detected by sensors, while approaching a defined goal configuration. The thes is is based on the work done in the specialization project the spring of 2022, where Rapid Exploring Random Tree (RRT) is proposed as a global planner producing waypoints that form a path from a start configuration to a goal configuration. The reactive planner presented in this paper has as its objective to plan a collision free path between the waypoints.

The results from testing the planner are presented in this thesis, and are conducted as a series of stepwise simulation experiments. The experiments are starting with tests to verify basic elements of the proposed method before the method is tested in a extended environment.

The main results of the work is presented in the following order:

- The proposed method is first tested in a 1D environment to verify the functionality of the method.
- Randomly located obstacles are placed in a 2D environment together with a randomly placed goal. The results show the motion planners obstacle avoidance abilities while it tries to reach the goal.

The experiments show that the deep reinforcement learning-based motion planner is able to avoid collision with obstacles while reaching the goal, however the performance can be seen to be considerably affected by hyperparameters and architecture of the neural networks.

Sammendrag

Autonome Undervannsfartøy (AUVer) brukes allerede i undervannsoperasjoner hvor miljøet eller operasjonen er utfordrende for mennesker å gjennomføre på grunn av for eksempel værforhold eller tiden operasjonen tar. Samtidig er det et økende behov for bærekraftig matproduksjon i verden, noe som fører til et stort potensiale for oppdrettsnæringen. I dag er de fleste oppdrettsanlegg lokalisert langs kystområder, noe som er en begrensningsfaktor med tanke på plass. Flere av aktørene i bransjen ønsker derfor å flytte oppdrettsanlegg offshore. Offshore oppdrettsanlett medfører at vedlikeholdsprosesser som rengjøring, observasjon av fisken og annet vedlikehold blir mer utfordrende å gjennomføre, men de er viktige for fiskens vellferd. Ved å inkorporere autonom teknologi i oppdrettsanlegg kan menneskelig personell bli assistert i oppgaver som er viktige for fiskens vellferd og som sikrer et tryggere arbeidsmiljø for de ansatte.

AUVer bruker bevegelseskontrollsystemer for å kunne operere autonomt. Slike systemer minimerer avviket mellom AUVens nåværende tilstand og den ønskede tilstanden, hvor det å finne den ønskede tilstanden blir gjort ved å bruke en bevegelsesplanlegger. En slik planlegger vil være ansvarlige for å finne en sti som kan håndtere det dynamiske miljøet i et oppdrettsanlegg.

Denne masteroppgaven foreslår å benytte dyp læring (deep learning) sammen med forsterkningslæring (reinforcement learning) i en algoritme som heter Deep Deterministic Policy Gradient (DDPG) som en reaktiv bevegelsesplanlegger for kollisjonsunngåelse for AUVen. DDPG kan håndtere kontinuerlige tilstands- og handlingsrom, noe som er nødvendig for en bevegelsesplanlegger.

Bevegelsesplanleggeren opererer online og resultatet er en ønsket retning for hvert steg som lar AUVen unngå kollisjoner oppdaget ved bruk av sensorer mens den samtidig beveger seg mot mål. Masteroppgaven er basert på prosjektoppgaven som ble gjennomført våren 2022 hvor Rapid-exploring Random Tree (RRT) blir foreslått som en global planlegger som produserer veipunkter som former en sti fra startkonfigurasjonen til mål. Den reaktive planleggeren som er presentert i denne oppgaven ønsker å planlegge en kollisjonsfri sti mellom veipunktene.

Resultatet fra testingen av den foreslåtte metoden er presentert i denne oppgaven og er gjennomført som en rekke simuleringseksperimenter:

- Den foreslåtte metoden er blitt testet i et 1D miljø for å verifisere funksjonaliteten av metoden.
- Tilfeldig lokaliserte hinder er plassert i et 2D miljø sammen med tilfeldig lokaliserte mål. Resultatet viser hvordan bevegelsesplanleggeren prøver å unngå kollisjon mens den prøver å nå mål.

Eksperimentene viser at den foreslåtte metoden får til å unngå kollisjoner med hinderne mens den når mål, men hvor godt den presterer blir påvirket i stor grad av hyperparametre i de neurale nettverkene.

Preface

This report presents the work done in a Master thesis from Cybernetics and Robotics at the Norwegian University of Science and Technology (NTNU), under the Department of Engineering Cybernetics, and the work done in this thesis has been carried out in collaboration with SINTEF Ocean. The thesis has been developed the autumn of 2022 and is based on my specialization project from the spring of 2022.

First I would like to thank Martin Føre for the guidance and support throughout the writing of this thesis and for valuable and motivating meetings. Many thanks to Marios Xanthidis for his highly appreciated contributions in the field of motion planning, and to Eleni Kelasidi for the shared knowledge and for guiding the work in the thesis in the right direction.

Trondheim, 19.12.2022
Anne Marie Mathisen

Contents

List of Figures

List of Tables

Chapter 1

Introduction

During recent years academic research in the field of autonomous vehicles has reached high popularity[1]. At the same time the growing demand for food worldwide, while meeting environmental conditions, requires new solutions for food productions in sustainable ways. Aquaculture is destined to play an important role in solving the food crisis as the ocean covers 70 percent of the earths surface[2]. However with the increasing production in aquaculture it is important that the industry is both environmentally sustainable, ethical, productive and safe both for the fish and for personnel working in the industry.

SINTEF Ocean conducts research and innovation related to the ocean space for national and international industry[3]. They are working on creating autonomous, intelligent systems that can be a part of shifting the aquaculture industry from being based of manual labor and experience to being based on technology[4]. RACE Fish-Machine Interaction is a SINTEF project where the main objective is to gain fundamental knowledge about methods to identify changes in behaviour of the fish, develop new methods for modelling and control of underwater vehicles enabling autonomous operations in fish farms while considering the interactions with the living fish[5]. Another SINTEF project is CHANGE has the primary objective to develop new control systems for UUVs to enable autonomous operations in highly complex and dynamic environments that contain live fish and flexible structures[6]. These two projects are examples of work being done today to allow for sustainability and safety within the aquaculture industry.

This thesis investigates the potential for using Reinforcement Learning in motion control systems by using it to perform online motion planning. The results represent an important step in the development of a new control method for underwater vehicles that enable autonomous operations while taking living fish into consideration. The presented solution is a result of a literature study and simulations, and is based on the work done in the specialisation project conducted in the spring of 2022. Some of the work done in the specialisation project will be reused or rewritten in this thesis.

1.1 Background

Aquaculture involves the farming of fish and other marine animals[7], and has the last decades had a rapid growth in production volume and economic yield, being a key provider of seafood[8]. In 2017 the salmon sales reached 1.2 million tonnes, and they are expected to reach 5 million tonnes annually by 2050 provided that environmental and important production challenges are met[9].

An important tasks at the farms consist of cleaning and monitoring the net to remove biofouling and detect tears in the net. Biofouling is unwanted growth of organisms on the net in a fish farm, and it is one of the main challenges in salmon farming[10]. Typical fish cages used at norwegian fish farms today are created for handling rough conditions by being robust but flexible. At the top of the construction there is a floating element, keeping the cage floating at the surface. Connected is the net, which has a sinking element at the bottom. The shape of the cage can vary, but they are often circular. Today there are methods in place to remove the biofouling using high pressure washing, but the process is labor-intensive and has a negative impact on fish health and welfare[10].

Aquaculture has for the most been based on experience and the interaction between man and animal, similar to livestock farming. However due to the location of the farms as well as the volume of fish, building a relationship with the fish and evaluating the status of the population through direct observation is near impossible[8]. Therefore having technology enabling monitoring of the fish through data collection, surveillance of the feeding process and other welfare critical processes is another important task for the future development of the aquaculture industry[4].

Todays fish farms are mainly established in areas close to the shore, making them accessible to human personnel. However, the coastal areas are limited, and in to meet the demand potential solutions are investigated. SALMAR has developed and deployed the Ocean Farm 1 facility for offshore fish farming[11] to engage in offshore fish farming to accommodate the need for food in the world, while making a minimal environmental footprint. Moving farms offshore is a potential solution to the space problem, but it does however require solutions that are able to assist human personnel in the challenging environments.

With the growth of production volume the likelihood that the industry will face emerging challenges that can influence the ability to maintain productive, ethical and environmental friendly production of fish[8]. The director of the Norwegian Labour and Inspection Agency states that the employees in the aquaculture industry have one of the most risk involved occupations, and the goal is to increase the focus on the safety of the workers to exploit the potential of the industry[12]. A study of Fatalities in the Norwegian fishing fleet[13] shows that the Norwegian aquaculture industry has the second highest incident rates per 10 000 employee for fatalities after the fishing fleet. With production shifting towards more exposed locations the workers have to manage increasingly challenging environments, which could result in an even higher risk for incidents.

The aquaculture industry today does in other words face several challenges as the demand form farmed fish increases. Moving the farms to more remote sites, or even offshore, makes it possible for the industry to continue to grow. However to maintain sustainability and safety for the fish, while still having a safe work space for the personnel new solutions are required. It is therefore expedient to investigate technological solutions that do not depend on being controlled on site, but rather operate autonomous or can be remotely controlled.

1.2 Motivation

By moving the fish farms to exposed areas offshore, the goal is to contribute to a more sustainable way to meet the growing food requirements in the world, but doing so also presents some challenges. Depending on the location the necessary operations on the farm can be too dangerous or even near impossible to be performed by human personnel.

Today SINTEF has multiple different projects aiming to develop technology directed towards making the aquaculture industry more sustainable through using technology. The vision is to develop autonomous and intelligent systems for operations in complex and dynamic environments, due to the often exposed areas the fish farms are located in. By assisting the workers using autonomous technology like autonomous underwater vehicles, putting workers in dangerous situations can be avoided. This is especially relevant when it comes to cleaning and maintenance of the net, but autonomous underwater vehicles are also relevant for data acquisition by having them carry sensor systems.

An Autonomous Underwater Vehicle (AUV) is an undersea system containing its own power and controlling itself while accomplishing a pre-defined task[14]. This makes them well suited to perform various tasks in fish farms where there is a demand for as little as possible impact on the fish. However, fish farms are dynamic environments influenced by hydrodynamic forces due to wind and current which can result in deformation of the cage. In addition moving obstacles on the cage structure and the fish within the cage are exposed to collisions, and it is therefore important that the AUV can avoid collisions with obstacles detected by its sensors during the operation.

To avoid collisions with obstacles detected during the operation a motion planner that is able to plan online is required. That means that the AUV should plan towards its destination or some goal, and if an obstacle is encountered the plan should change locally to avoid the obstacle before continuing towards its goal.

1.3 Scope

The focus of this project is to investigate a novel motion planner using Deep Reinforcement Learning (DRL) as a reactive planner in the multilayer motion planner proposed in the specialization project which can be found in Appendix A. RRT is proposed as a global planner, producing waypoints as an initial path, while DRL is the reactive planner that handles obstacle avoidance during the operation.

Motion planners based in reinforcement learning methods are able to observe the environment it operates in using sensors, and based on the observation it makes a non-linear mapping between where it is now and what it should do to get closer to its goal. Where the mapping from the observation to the action is based on previous experiences. The goal can for instance be to reach a physical goal configuration, maintain some distance to a moving point or follow some constraints like avoiding collisions. Due to motion planners main task being to predict the effects of the execution of an action in to find the one that suits the object of the planning, reinforcement learning is a suitable candidate for motion planning.

1.4 Structure

The report is structured into six chapters: after the introduction in Chapter 1, Chapter 2 presents theory about motion planning and reinforcement learning, focusing on the DRL in continuous action spaces. Chapter 3 concerns the implementation as well as the structure of the experiments conducted in the project. Chapter 4 contains the results of the experiments, showing both the learning rate in the different cases as well as the actual performance after training. Following these results are discussed in Chapter 5, and finally Chapter 6 concludes the thesis.

Parameters for running the code are attached in separate files to run the code without training to watch the performance.

Chapter 2

Theory

This chapter presents the theory behind motion planning and how it is used to allow a under water vehicle to operate autonomously. One of the main challenges when operating in a fish farm is the dynamic environment, and a motion planner must therefore be able to plan around obstacles discovered during the operation. Further the chapter will therefore present the theory behind reinforcement learning, a promising solution for solving motion planning in environments containing obstacles that are unknown prior to the execution of the operation.

The theory chapter is based on the work done in the specialization project conducted the spring 2022, and therefore not all relevant theory will be explained in detail. Some om the most central topics about motion planning and reinforcement learning are re-explained in this chapter, as the understanding of them is central for the work done in the thesis. The reader is however encouraged to turn to the specialization project, as some of the topics of the theory section are explained in greater detail there.

2.1 Motion control systems

The concept of motion control was introduced during the literature study conducted in the specialization project, however this section will expand on that theory to connect it to the experiments presented in this thesis. In this chapter the main focus will first be a short introduction to the concept of motion control, and what role motion planning has in controlling an autonomous vehicle.

The concept of guidance, navigation and control (GNC) deals with the design of systems that automatically or remotely control vehicles or devices moving in space, on the surface of the earth or under water. This thesis will mainly discuss guidance, however the GNC system in total will briefly be introduced to provide an over all understanding of the concept.

Guidance

In Figure 2.1 the guidance system can be seen as the part of a motion control system that is concerned with the transient motion behavior associated with the achievement of motion control objectives[15]. The guidance system can use data from an human operator, like a joystick, inputs about environmental conditions, knowledge about topology like known obstacles and structures of the environment, sensor inputs and information about the state of the vehicle given by the navigation and sensor system[15]. Based on this data a motion planner can calculate the desired path of the vehicle. Optimization techniques can be used to compute an optimal path that the vehicle should follow, where features like fuel optimization, minimum time navigation or collision avoidance can be included[15].

For ships and under water vehicles the guidance and control system often consists of an attitude control system and a path-following control system[15]. This thesis will focus on motion planning in two dimensions, and therefore the desired attitude is given by the desired heading ψ_d of the vehicle. However for a system in three dimensions the desired attitude would also include roll ϕ_d and pitch θ_d. To follow the desired path the control system should therefore be given the desired heading from the guidance system.

When the desired path is found using the motion planner, the desired heading can be calculated. Line Of Sight guidance is widely used to do so, where the focus is to calculate the desired heading ψ_d by minimizing the cross-track error y_e between the desired path and the current position. Path following control is often a speed guidance system, however for underwater vehicles a depth controller is also required in tree dimensions[15].

Navigation

The control systems of conventional ships and under water vehicles are implemented with navigation systems. To use the the control system to reduce the error to the desired state, given by the guidance system, the current state has to be known. The navigation system is a model-based state estimator that is used to process sensor and navigation data. This data is passed to the control system using a feedback connection, allowing the control system to compare the current state of the vehicle to the desired state given from the guidance system. In Figure 2.1 the navigation system is included in the vehicle block. In Figure 2.1 a feedback connection from the vehicle to the guidance system can be seen as a dashed line. This allows for optimization of the motion planner and gives a systematic method for inclusion of static and dynamic constraints, however it is challenged by solving the optimization problem online[15].

Control

Figure 2.1 shows that the control system takes two inputs. One is from the guidance system and one is from the navigation system, where the first produces the desired state and the latter produces the actual state. The control system is used to minimize the error between the two, making the vehicle converge to and follow the desired motion from the guidance system.[15] distinguishes between trajectory tracking and path following. The objective of the first is to force the system output $y(t)$ to track a desired output in the presence of both spatial and temporal constraints. The latter wishes to follow a predefined path independent of time considering spatial constraints.

The control system is used to control the motion of the vehicle by using actuators[15]. This is done using a controller that calculate the necessary control forces and moments that should be provided by the vehicle by using a PID controller for instance. The generalized control forces calculated using the control law should then be distributed to the actuators of the vehicle using control allocation.

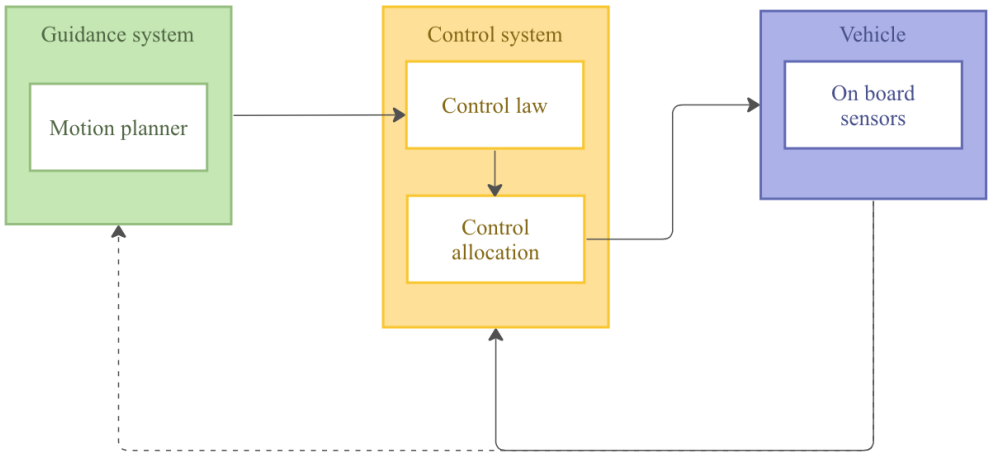


Figure 2.1: GNC system illustrated showing the interaction between the sub-systems. The control system gets the desired state from the guidance system which is based on the desired motion produced by the motion planner. The navigation system of the vehicle is based on sensors and measurements that estimates the current state. The current state is then compared to the desired state in the control system, and the control law is used to produce the generalized forced needed to minimize the error, before the forces are distributed to the vehicles actuators using control allocation. The feedback connection in the dashed line allows for optimization of the motion planner based on information on the current state of the vehicle.

Figure 2.1 shows how the sub systems explained above are structured and connected to allow vehicles

operate with little or no interactions with humans. It can be seen that the motion planner and the guidance system is used to produce a desired path which is given to the control system as for instance a desired heading or desired speed. The controller uses this as a reference, and compares it to information on the vehicles current state which is given from sensors on board the vehicle and state estimators. The vehicle is then given the control forces to apply to its actuators to reach the control objective, which is given by the guidance system. In this thesis the focus will be on the on the motion planner, and the following sections will therefore further explain the concept of motion planning and how it can be performed in environments like fish farms.

2.2 Motion planning

For a motion control system to control a vehicle towards some goal the guidance system must provide information on what the desired behaviour is. The motion planning system is responsible for producing a feasible path which represents the desired behaviour of the vehicle. Following the control system uses this desired path as a reference, trying to minimize the error between the current state and the desired state given by the motion planner. The desired path can for instance be defined as the path from a start configuration to a defined goal configuration, or a path avoiding obstacles along its way. To create an autonomous vehicle the motion planner should be given a description of the task to be solved which then should be executed without further human intervention. This can be done by giving input descriptions on what should be done rather than how, and then having the motion planner decide what motions to perform.

In a dynamic environment, such as a fish farm, there is also a need for the motion planner to be able to avoid collisions with obstacles that are not known prior to the operation, but rather discovered during the execution of the operation. In a fish farm this could be parts of the net structure or the fish in the farm, and to avoid collisions, both for the safety of the vehicle and the fish, it is important that the planner can operate online. If an obstacle is detected by the sensors mounted on the vehicle, the planner should be able to plan a path that results in avoiding a collision. However it should at the same time try to carry out the other objectives like reaching a goal or minimizing fuel consumption.

2.2.1 Multilayer motion planners

The specialization project presented the three types of motion planning: global, local and reactive. More details on each of them can be found there, however the main concepts will be summarized in the following.

Global motion planning

The global algorithms take all information about the environment into account and plan from a start to a goal configuration[16]. Global methods rely on availability of a topological map defining the vehicles workspace and the location of obstacles[17]. The path can consist of waypoints from the start configuration to the goal configuration, giving an end-to-end path from the start position to the goal position. The global planner is however not able to take obstacles unknown before the start of the operation into consideration. Running the global planner each time a previously unknown obstacle is encountered would result in a high computational cost as the whole path would have to be re-planned. Some of the most well known and used global planner algorithms are A^*, Rapid-Exploring Random Tree (RRT) and Probabilistic Road Map (PRM).

Local motion planning

Local planners are used when the start and goal configurations are close together[16] and is therefore suitable for planning between waypoints produced by a global planner. Due to operating over shorter distances the local planner is more suited to handle obstacle avoidance for obstacle detected during the operation, as only as short section needs to be re-planned. Local planners that use pure obstacle avoidance methods do however suffer from the inability to generate an optimal solution, and the vehicle can get ensnared into a local minimum[17]. By combining them with global planners the distance is split up by the waypoints, creating sub-goals for the local planner.

Reactive motion planning

The term reactive planning refers to a broad class of algorithms that uses only the local knowledge about the obstacle field[18]. Reactive planners are used in cases where there is uncertainty about the obstacle field, and can avoid last minute collisions even with the obstacle position only known within a small radius. They are however seldom used on its own, but rather combined with a global planner to handle obstacle avoidance by receiving sensor inputs.

[19] presents a reactive motion planner that is based on Elastic Band method (EBM) to perform motion planning for a net cleaning AUV in fish farms. The method was introduced by Quinlan and Khatib in 1993[20], and uses an initial global plan which is optimized by the EBM to minimize the length of the path while still taking obstacles into consideration. The EBM based reactive planner optimises the path incrementally, such that the longer the vehicle moves, the better the output.

Another reactive planner is presented in[21] which is based on fuzzy logic, which due to its short response time is suitable for online motion planning. The fuzzy rule-base in the proposed motion planner combines information about the distance and angle between the vehicle and nearby obstacles with information about the distance to and angle between the goal and the current position. The method does however only take the nearest obstacle into consideration, increasing the possibility of getting suck in a local minima.

Reinforcement learning is a machine learning technique where an agent learns sequential decision making from trial and error. With the growth of machine learning during recent years, the application of reinforcement learning algorithms to motion planning problems[22], vehicle decision making and control problems[23] is a new trend that has emerged. Compared with traditional motion planning, methods based on machine learning have a strong generalization ability and robustness, and the trend continues to grow with new findings in areas like deep learning. By creating scenarios where collision with obstacles, avoidance of obstacles and the goal being reached occurs, this experience can be used to create a motion planner that is able to plan towards a goal while avoiding obstacles detected during the operation.

In[24] autonomous driving for a vehicle is implemented using reinforcement learning in combination with using artificial potential field for collision avoidance. The algorithm was tested in a TORCS environment, and the results show that the agent successfully controls the steering of the vehicle.[22] presents a motion planner for a robotic arm which uses a reinforcement learning based method which the paper refers to as Residual Reinforcement Learning, that is successfully applied to a robotic arm performing tasks in a virtual environment.[25] presents a study that develops a reactive two-dimensional path planning method based on reinforcement learning. Results from methods compares different neural network architectures, where the best performance being a successful reaching of the destination in 89%.

Approach used in this work

In the specialization project Rapid-exploring Random Tree (RRT) was suggested as a global planner used to produce waypoints from a start position to a goal position. The experiments conducted for the RRT as a global planner showed that the algorithm is able to plan a path consisting of waypoints from the start configuration to the main goal configuration while avoiding known obstacles given by a topological map. The suggested RRT global planer will create the path consisting of waypoints before the operation, and should therefore be combined with another planner that enables avoidance of obstacles detected during the operation.

In this thesis the focus will be on the reactive planner that will operate between a set of given waypoints from the RRT global planner. The literature study conducted during the specialization project will be the base of this thesis, where a reactive planner that is able to avoid obstacles between the waypoints based on reinforcement learning was proposed.

In the following sections the concept of reinforcement learning will be explained, as well as the concept of deep learning and how the two can be combined into deep reinforcement learning to solve complex problems. Following the chapter will focus on Deep Deterministic Policy Gradient (DDPG), a deep reinforcement learning algorithm for learning in continuous action spaces.

2.3 Reinforcement Learning

Reinforcement Learning is a general class of algorithms in machine learning that deals with the problem of having an agent with learning abilities achieve a goal while dealing with constraints. The research of reinforcement learning has a long history, however the dynamic programming algorithm[26] proposed by Bellman in the 1950s has been the foundation[27]. Figure 2.2 illustrates the learning process in reinforcement learning. It summarizes the interaction between a learning agent whose objective is to reach some goal. The agent uses a mapping function called a policy to choose the action it believes is the best to reach the goal base on the state it is currently in. Based on the outcome of the action the agent is rewarded with a positive reward if the action resulted in approaching or reaching the goal. The reward can then be used to improve the policy.

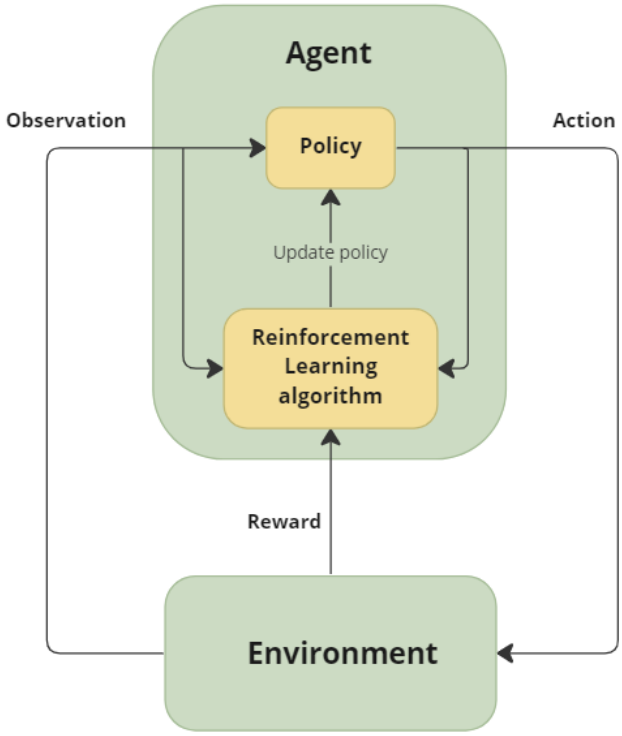


Figure 2.2: The figure shows how the agent learns from interacting with the environment. An observation is made of the vehicles state environment and is mapped to an action using a function called a policy. The action is executed in the environment, and an observation can be made of the vehicles state in the environment after the action. Depending on how good the action was in terms of making the vehicle reach its goal, a scalar reward is given. The reward is feedback to the policy on how good the mapping from observation to action was, and therefore the policy can improve based on the reward.

Reinforcement learning differs from unsupervised learning and supervised learning. Unsupervised learning attempts to find similarities and differences between data without labels, while supervised learning learns a structure by finding some pattern based on a given data set. Contrary, reinforcement learning learns how to act by trial and error[1] with the objective of finding the best behaviour, an action or a label for each particular situation. In reinforcement learning an agent has to observe the current state of the environment that it lives in, and based on the information it has to choose an action which will result in a new state. Depending on how good the new state is in terms of the agents goal and the constraints, the agent receives a reward. An action resulting in the agent approaching or reaching its goal would result in a positive reward, while a violation of constraints, like a collision, would result in a negative reward. To reach its goal the agent should therefore make actions that maximize the total reward.

The following sections will introduce some of the main concepts of reinforcement learning to give an overall view of the general concept. Later deep learning is presented and how it can be combined with reinforcement learning to allow for learning in complex environments, creating deep reinforcement learning (DRL).

Finally the deep reinforcement learning method Deep Deterministic Policy Gradient (DDPG) is presented and explained in detail. This algorithm is suitable for continuous action spaces, making it relevant for motion planning applications such as the multilayer motion planner that is proposed in the specialization project.

2.3.1 Reinforcement Learning for motion planning

[1] states in a survey about using reinforcement learning for self-driving cars that "*even with full knowledge of the current state of the traffic, the future intentions of the surrounding drivers are unknown, making them partially observable.*" Like most real world problems, the future states of the environment the motion planner plans in will not be known before it occurs, making the decision process for motion planners partially observable. To plan the motion of the vehicle the inputs to the planner are travel destination, sensor information and other information on the network the vehicle will travel in. In addition the dynamics of the vehicle also has to be known to the planner to create a feasible solution. The output is a feasible planned path, however it can also be represented as for instance the desired heading ψ_d of the vehicle at the current step.

Reinforcement learning provides motion planners that are based on artificial intelligence and are able to make decisions for the planning by directly establishing patterns from sensor data[27] which reduces the need for processing of the sensor measurements. Another advantage is the generalization abilities of reinforcement learning based motion planners[28] which determines the vehicles abilities to operate safe and reasonable in previously unseen scenarios. The abilities for generalization do however rely on the training and how well the experience data used during training allows for exploration of the environment where the planning happens. A large amount of research based in deep learning has been conducted based on implementing automated driver applications[23], however methods relying on supervised learning are in need of large amounts of labeled data to generalize driving tasks. Gathering these amounts of data, and labeling it requires massive amounts of human labor[23], and will still not necessarily represent the complexity of the task. Reinforcement learning avoids this problem by its trial-and-error learning approach which does not require humans to label data, in addition to the data being based on the experience of the agent in the environment it will operate in rather than human produced data trying to resemble experience.

When an autonomous agent interacts with a physical world it is often necessary to handle large continuous action spaces[1]. In the case of motion planning steering and acceleration can be adjusted by the agent to create the desired path, however both are continuous parameters and it is therefore important that the algorithm can handle the infinite number of actions. One option is to discretize the action space, however this will result in a rapid growth of the number of possible actions, while making the problem less similar to reality. The other option is to use an algorithm with the possibility of handling continuous action spaces. Deep Deterministic Learning is a reinforcement based algorithm implemented with the intention of handling large action spaces. In[24] the autonomous cars steering abilities is implemented using DDPG, and in[29] DDPG was also implemented to solve the lane-keeping problem for self-driving autonomous cars with highly effective results.

2.3.2 Formulating a RL problem using Markov Decision Process

Markov Decision Process

An important aspect of formulating a reinforcement learning problem is modelling the transitions of the states of the environment based on the actions of the agent[1]. Such transitions are often described using Markov Decision processes (MDP) which can be defined by the set

$$(S, A, P, R) \tag{2.1}$$

The MDP consists of a set of states *S*, including the initial state s_0 and a set of actions, *A*. It also contains a transition model $P(s_{next}|s_t, a_t)$, where s_t and a_t are the state and action at step t, that describes the next state of the environment based on the current state and the action taken to obtain the next state, as well as a reward function, $R(s_t)$, that indicates how good it is to be in a state[30]. The process can be summarized as:

The cycle of a MDP agent:

1. Given the current state s_t, execute the action a_t. Where $s_t \in S$ and $a_t \in A$.
2. Get the new state s_{t+1} from the transition model P and reward $r_t = R(s_t, a_t)$.
3. Set the new state to be the current state.
4. Use r_t to improve decision making of action.

[30]

Partially Observable Markov Decision Process

However, MDPs assume that the environment is fully observable which in dynamic environments is not an realistic assumption if there is a lack of knowledge about the future states of the environment[1]. Examples are not knowing the future intentions of other drivers in traffic or where the fish will swim next within the cage of a fish farm. This leads to the formulation of a Partially Observable Markov Decision Process (POMDP) which is defined by

$$(S, A, T, R, \Omega, O) \tag{2.2}$$

where S is the set of environment states, A is the set of actions, T is the transition function between states based on actions, R is the reward function which depends on state-action pairs, while O is the set of observations and Ω is the sensor model[1]. The difference is that the POMDP instead of working with the actual state of the environment uses the observed state of the environment, which is related to or given by some observation of the environment made through the sensor model Ω. Using POMDP the process can be summarized as:

The cycle of a POMDP agent:

1. Given the observed state o_t at step t, execute the action a_t. Where $o_t \in O$ and $a_t \in A$.
2. Receive sensor measurements based on the sensor model, Ω.
3. Set the new observed state o_{t+1} based on o_t, a_t and the sensor measurements and calculate the reward $r_t = R(o_t, a_t)$. Where $o_{t+1} \in \Omega$
4. Set the new observed state to be the current observed state.
5. Use r_t to improve decision making of action.

[30]

The T in the tuple in Equation 2.2 contains information on the modeling environment of the learning process, which in the case of motion planning includes modeling the dynamics of the vehicle, the surrounding environment of both static and dynamic objects and the topology of the environment.

States and observations

The observed state is a perception of the vehicles state in the environment, and will in the rest of this thesis be referred to as an observation, $o_t \in O$. This is because that states in the real world are only partially observable by perception, which provides an agent with information about the world the agent exist in by interpreting responses from sensors[30]. States and observations are different in that the state includes all information on the environment, while an observation only contains information on what is sensed by the sensors. In addition it is not possible to have an observation of the next state of the environment, as it would require the sensors to collect measurements in the future.

Getting information about the surroundings of the vehicle and representing it to the learning agent can be done using different types of sensors. Examples are computer vision, lidar and radar. The structure of the sensor model is important to take into consideration when implementing motion planners as it affects the neural network structure of DRL agents[1]. This is because the dimension of the observation determines determines the structure of neural networks. If computer vision is used the observation dimension will be large due to consisting of all of the networks pixels. If lidar or radar measurements are used the dimension will be given by the number of measurements made. If the vehicle has one lidar sensor attached to it the dimension of the observation will be much smaller than when using computer vision. The type of neural network as well as the architecture will also be affected, something that will be explained later in the thesis.

Actions and action spaces

Actions are what the agent can use to change its current state to control the systems state towards a goal. The action can be of multiple dimensions, and when using reinforcement learning for motion planning the action vector can contain information like change of orientation and acceleration. The limitations of the actions are given by the dynamics of the vehicle, and the set of possible actions define the action space of the agent.

An important aspect of the action space of the problem is whether it is discrete or continuous. Continuous action spaces are limited by a upper and lower limit, and are infinite in size as they contain all possible values between the limits. Discrete action spaces are defined by a set number of actions like up, down, left or right. However in most motion planning problems the controllable variables like steering are continuous[1], and many reinforcement learning algorithms are based on having a finite set of actions. A solution is to discretize continuous variables, however this is challenged by pushing the solution far from reality if there is a small number of discrete choices and a large number slows down the learning process. It is therefore desirable to use a reinforcement learning algorithm able to handle continuous action spaces when dealing with motion planning problems.

Rewarding

An episode is a series of steps, where one step consists of observing the state of the vehicle, choosing an action based on the observation, executing that action in the environment and observing the reward and new state of the vehicle in the environment. During training the episode is over if:

- The agent reaches its goal
- A terminal condition occurs, like a collision
- The predefined maximum number of steps is reached

Each step in an episode results in the agent being in a new state, and in each state the agent receives a reward, $R(s, a)$[30], determined by a reward function that depends on how good the agents choice of action was. The reward can therefore be used to criticize the agents way of making decisions for it to improve. It is therefore important that the reward function reflects the wanted behavior. In motion planning some relevant performance factors that can be a part of the rewarding is: reaching the goal configuration, time spent finishing the task, fuel consumption and other.

The reward can be positive or negative, but it must be bounded[30]. The designer of the reward function has to determine what details are necessary. If there is only given a reward for reaching the goal, the agent will only receive feedback at the end of an episode, if the goal is reached. This can result in a slower training process, however the feedback given to the agent is then less shaped by the designer of the reward function, and more by the algorithm. Adding rewards based on each step will speed up the learning process, and can be based on minimizing the distance to the goal or encouraging the vehicle direction towards the goal.

The agent wants to choose actions that increase the sum of rewards in the long run[31]. In particular, it needs to be specified how the agent should take the future into account when making decisions about how to behave now[31]. The parameter γ is used to describe how much future rewards depend on the actions being made at current time.

$$R_{inf} = \lim_{t \to \infty} \sum_{t=0}^{\infty} \gamma^t r_t \tag{2.3}$$

In Equation 2.3 the infinite-horizon discounted return takes the long-run reward of the agent into account, but future rewards are discounted by the discount factor γ^t[31]. Where r_t is the reward at step t, and t goes from 0 to infinity, and $\gamma \in (0, 1)$, which makes it determine how much better a reward is at current time than in the future. γ also makes the sum, which would be an infinite sum, finite[32], and it is therefore possible to search for a maximized reward.

2.3.3 What is a policy?

When the agent tries to choose the best action based on the current observation of the environment it does a mapping from state to action through a policy function π. In reinforcement learning, the goal is to improve this mapping for the agent to choose the best possible action given the current state by maximizing the future long-term reward.

In this thesis the focus will be on deterministic policies, which is a direct mapping such that π: S $\rightarrow$ A. However stochastic policies returning a distribution also exist, where π: S$\times$A $\rightarrow$ $[0, 1]$ such that for each state there is not one clear action that is the best, but there is a probability distribution for actions based on the state.

With the policy being a function it depends on parameters. For example, the mapping from state to action is linear it can be written on the form $\pi(s_t) = \theta s_t = a_t$ where θ is the parameter of the mapping. To obtain the optimal policy, the parameters are adjusted to improve the mapping based on the rewards it has received. In most real world cases the mapping is not linear, and the policy-function $\pi(s_t)$ is in some cases approximated using neural networks. This topic will be discussed in greater detail later.

Exploration vs. exploitation

One of the main differences of reinforcement learning and supervised learning is that the agent in reinforcement learning has to explore the environment in order to learn[31]. An important challenge of training a reinforcement learning agent is choosing the trade-off between exploration and exploitation[33]. For the agent to improve its policy, it has to try different actions and observe the results, it is learning through trial and error. The agent is however motivated by choosing an action that results in the highest possible reward, and by exploiting the current policy the agent chooses the action that based on the current policy is believed to result in the highest possible reward. The trade-off is therefore between taking actions to gain information on the environment to improve the policy and choosing an action based on what at this point in the training is the best to make a greedy choice.

One way to handle the exploration vs. exploitation trade-off is by using an exploration rate which determines how often an action should be based on exploration or exploiting the current policy. The rate can either be a static number or it can change. By having a exploration rate that decreases over time during training the exploitation of the policy increases as the policy improves. Another solution to the trade-off is using off-policy learning. This approach is called the ϵ-Greedy selection in[33], as ϵ is the scalar that determines the exploration rate.

On-Policy learning vs. Off-Policy learning

After establishing that the agent learns the optimal policy from experience, on-policy and off-policy learning can be explained. In on-policy learning updates of the policy happen on the basis of the experience that is gained from executing that policy[34]. When learning on-policy the latest policy is used to collect experience and then that experience is used to improve the policy. The process can be explained in the following steps:

On-policy learning process

1. Use the current policy π to choose an action a_t based on the current observation o_t.
2. Execute a_t in the environment, observe the new state of the environment as o_{t+1} and the resulting reward r_t.
3. Use the reward to update π, and then repeat the process.

In on-policy learning the policy π is improved by using it to choose an action given the current state, and then improve the policy based on the result of the chosen action. One of the main challenges with on-policy learning is the trade-off between exploration and exploitation as in on-policy the exploration process cannot be separated from the learning, and it must therefore be handled directly[34]. One of the main ways to solve this is using the exploration rate previously discussed.

Off-policies on the other hand use two different policies, and learns from actions other than those actually executed[34] by using one policy to create experiences while the other is the actual policy that should be optimized. One policy, π^b generates experiences and stores them in a replay buffer. A replay buffer is a memory where episodes of transitions on the form (o_t, a_t, o_{t+1}, r_t) where $o_t, o_{t+1} \in O$, $a_t \in A$ and r_t is

the reward from the reward function *R*. These stored transitions represent experiences from being in one observed state o_t, taking an action a_t and ending up in the observed state o_{t+1}.

The other policy π^a is the policy used to perform actions and when improving π^a, experience data from the replay buffer is used. π^a is a greedy policy, always choosing the best possible action, and it is the policy that is to be optimized. After the training is completed π^a will be the policy used to choose actions during the motion planning.

Initially the policy π^b can be any policy that generates samples, the point is to have examples, both good and bad that can be used to update the policy π^a. Therefore choosing π^b to randomly choose actions initially can be a good choice as it allows for exploration over the range of the action space. When the replay buffer is filled with samples generated from the initial π^b, some of the samples are used for optimizing π^a, how this is done will be explained later. To criticize its own performance samples are then generated using π^a, and stored in the replay buffer. This results in a combination of samples created using π^b and π^a in the buffer. π^b will however still represent the samples in the replay buffer, but it will approach π^a due to an increasing number of the samples being from π^a. This is due to the size of the buffer being fixed, and old samples are replaced with the new. At the end of the training process the replay buffer will be filled with samples from π^a, making $\pi^b = \pi^a$.

The process can be explained in the following steps:

Off-policy learning process

1. Use some policy π^b to generate samples for the replay buffer B. An example is to choose π^b as randomly choosing a_j.
   (a) Choose an action a_j based on some state o_j using some policy π^b.
   (b) Execute a_j in the environment, observe the new state of the environment as o_{j+1} and the resulting reward r_j.
   (c) Store the tuple (o_j, a_j, o_{j+1}, r_j) in B.
2. Based on samples in B, update the policy π^a.
3. Use π^a to choose an action a_t based on the observed state o_t.
4. Execute a_t in the environment, observe the new state of the environment as o_{t+1} and the resulting reward r_t.
5. Store the tuple (o_t, a_t, o_{t+1}, r_t) in the replay buffer B, replacing some of the samples made using π^b. B will now consist of a mix of samples from the old π^b and π^a, which updates π^b.
6. Update π^a based on samples from B, and repeat the process after 2.

The main reason for distinguishing between the two is to handle the trade-off between exploration and exploitation. When using off-policy training the agent is able to explore by choosing π^b such that the exploration is large. At the same time π^a is a greedy policy, thus also enabling exploitation.

This far the update of the policy based on the reward has not been explained other than that the goal is to maximize the future reward and explaining how the discount factor γ. In the following sections the two main approaches for optimizing the policy, Policy gradient and Value-based learning, will be explained.

2.3.4 Learning the optimal policy

Processes in the real world are POMDPs, therefore processes like motion planning should be modeled using this formulation[1]. The challenge of working with POMDPs over MDPs is the fact that current actions affect future states and therefore also affect the future rewards. Maximizing the future expected return will therefore require knowledge about future states, which is difficult when dealing with POMDPs. To solve this there are two main approaches, Policy Gradient and value-based methods. In the following sections details about policy-based and value-based methods will be explained before it is finally explained how DDPG utilizes both techniques to obtain the optimal policy.

Policy gradient

Policy gradient methods are based on gradient descent[22], optimizing the policy directly. The parameters of the policy are optimized with respect to the long-term cumulative reward shown in equation 2.3. The

policy is updated by applying the chain rule to the expected return from the start distribution J with respect to the parameters θ^π of the policy[35]

$$J(\theta^\pi) = E_\pi[R] \tag{2.4}$$

where the start distribution J depends on the policy since the better the policy, the higher the expected long-term cumulative reward, R. $E_\pi[R]$ is the expected total discounted reward when following policy π.

The optimal policy is then found by

$$\pi^* = \arg\max_\pi J(\theta^\pi) \tag{2.5}$$

where π^* is the optimal policy. Solving the optimization in Equation 2.5 problem is done calculating the gradient of Equation 2.4. π is defined by a set of parameters θ^π, and by using gradient ascent the parameters defining π can be moved in the direction suggested by the gradient $\nabla_{\theta^\pi} J(\theta^\pi)$. This results in a update of the policy that is based in maximizing the expected reward in Equation 2.3.

$$\nabla_{\theta^\pi} J(\theta^\pi) = E[\nabla_{\theta^\pi} Q(s_t, a_t|\theta^Q)] = E[\nabla_a Q(s_t, a_t|\theta^Q)\nabla_{\theta^\pi}\pi(s_t|\theta^\pi)] \tag{2.6}$$

[35]

In Equation 2.6 the gradient of the performance of the policy $\nabla_{\theta^\pi} J(\theta^\pi)$ with respect to the parameters defining the policy θ^π is given. Where $Q^\pi(s, a)$ is the value for a state-action pair when following a policy π, θ^Q are the parameters defining $Q^\pi(s, a)$, and $\pi(s_t|\theta^\pi)$ is the policy.

Value-based learning

Value function methods are based on estimating the value (expected return) of being in a given state[36]. They learning rely on learning an action-value function that predicts the expected discounted reward from a given state if an action is taken and a policy π is followed forever after**??**. In value based learning the optimal action will therefore be the action that maximizes the returned value from the value function.

Value-based learning is often referred to as Q-learning, and the returned value is a Q-value. A Q-value will show how good an action a is given a state s when all following decisions are made using a policy π[35]. If the policy is the optimal policy, π^*, the optimal action can be found by

$$\pi^* = \arg\max_a Q^{\pi^*}(s, a) = \arg\max_a Q^*(s, a) \tag{2.7}$$

where the optimal action is found by choosing the action that maximizes the Q-value when acting according to the optimal policy π^* for all future.

Q-values are given by Q-functions. There are two main Q-functions, the On-policy Q-function and the Optimal Q-function. Both functions are based of the Bellman equation. The main goal of reinforcement learning is to maximize the reward long term like in Equation 2.3. When an action is taken the the long term possible reward will be affected, and the Bellman equation describes the expected long term value when in a state s_t an action a_t is taken, and for all future steps after a policy π is followed. Therefore the Bellman equation describes how good it is to act according to the policy π.

$$Q^\pi(s_t, a_t) = E[R(s_t, a_t) + \gamma E[Q^\pi(s_{t+1}, a_{t+1})]] \tag{2.8}$$

In Equation 2.8 the value of being in state s_t and taking action a_t is given by the reward of that step, $R(s_t, a_t)$ and the expected value of the following states discounted by γ, where the policy π is being followed.

- The **On-policy Q-function**, $Q^\pi(s, a)$, returns the expected return when starting in state *s*, take some action *a*, and then acting according to the policy π forever like described in the Bellman equation.

$$Q^\pi(s_t, a_t) = E[R(s_t, a_t) + \gamma E[Q^\pi(s_{t+1}, a_{t+1})]] \tag{2.9}$$

- The **Optimal Q-function**, $Q^*(s, a)$, is the expected return when starting in state *s*, taking an action *a*, and then acting according to the optimal policy forever.

$$Q^*(s_t, a_t) = E[R(s_t, a_t) + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1})] \tag{2.10}$$

In Equation 2.9 the Q-value of taking action a_t when in state s_t is given according to the Bellman function. It represents how good it is to take an action when following the current policy for all future steps. In a similar manner Equation 2.10 is the result of taking action a_t when in state s_t and then acting according to the *optimal* policy π^* for all future steps. From Equation 2.7 the optimal policy π^* can be obtained when the optimal Q-function is known.

$$y_t = r(s_t, a_t) + \gamma \max_{a_{t+1}} Q^*(s_{t+1}, a_{t+1}|\theta^Q) \tag{2.11}$$

The expression in Equation 2.11 is from Equation 2.10 and is often referred to as the target. Where θ^Q are parameters defining the Q-function and a_{t+1} is given by the optimal policy $\pi(s|\theta^\pi)$ which is defined by the parameters θ^π. The goal is for the On-policy Q-function Q^π to be close to or the same as this target, as Q^π then can be used to find the optimal action. The target is however not known, and has to be approximated which can be done using a neural network. The topic of approximating the target will be discussed later in this chapter.

When the target is approximated, the loss between the target and the current on-policy Q-function, Q^π, is calculated as Equation 2.12. By minimizing the loss Q^π can be optimized to approach the goal.

$$L(\theta) = E[(Q^\pi(s_t, a_t|\theta) - y_t)^2] \tag{2.12}$$

The loss between the target and the current is presented in Equation 2.12. If the problem being solved is of some high dimension, both Q^π and Q^* can be represented using neural networks, where the weights describing the networks are θ. The following section will introduce how neural networks are used in reinforcement learning. Later the thesis will return to how the value-based learning presented in this section is used together with policy gradient.

2.4 Deep Reinforcement Learning

So far the topic of Reinforcement Learning has been introduced as the process where an agent learns to choose an action based on its current observation of the environment to reach a goal. Although reinforcement learning has had some success in the past, previous approaches have been limited to fairly low-dimensional problems[36] and finding a pattern that represents how the observation should be mapped to an action is due to the low dimensionality of the observation often a linear mapping. However if the dimension of the observation is large, the mapping from observation to action, which is the policy, is not linear. To approximate the optimal policy based on the experience data neural networks can be used as approximators. To use reinforcement learning in complex real-world situations the agent should be able to handle high-dimensional situations with multiple observations, and applying reinforcement learning to problems like motion planning will therefore often require the use of neural networks.

Deep learning has in the last years been prevailing in reinforcement learning in areas like games, natural language processing and robotics[37]. This is due to its abilities for end-to-end learning using gradient descent as well as feature engineering, making it possible for video games to be learnt from pixels or creating control policies in robotics based on sensors like camera inputs[36]. Applying deep learning to reinforcement learning provides powerful function approximators in neural networks, and is an important step in overcoming the challenge of high dimensional observation and action spaces in reinforcement learning.

2.4.1 Neural networks and deep learning

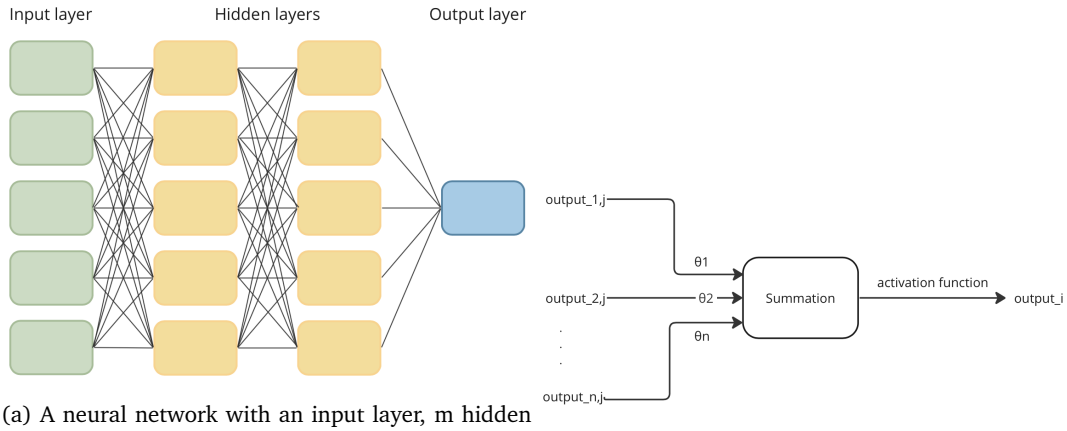
Neural networks represent a method for computation where the solution to a problem is learned from a set of examples, a method inspired from studies of information processing in biological nervous systems like the human brain[38]. Feed forward neural networks are mathematical non-linear functions that transforms its input variables to output variables on the basis of weights, where the value of the weights plays an important role in the mapping from input to output. To determine the values of the weights the network is trained, a concept that will be described in more detail later. The process of training the network can be time consuming as it can be a computationally intensive undertaking[38]. However, once the training process is over, and the weights are fixed, processing new data over the network can happen rapidly[38].

The networks consist of a number of layers, where each layer consists of a number of nodes. In deep learning there is an input layer, an output layer, and between there are a number of hidden layers. In Figure 2.3a a neural network with an input layer consisting of five nodes, two hidden layers consisting of five nodes and one output layer consisting of one node is illustrated. At each of the layers, except the input layer, the input to each node is computed as the weighted sum of of units from the previous layer. Then some transformation or activation, often non-linear, is applied to the sum[37]. Between the layers the inputs/outputs are weighted as illustrated in Figure 2.3b. In the figure the computation from input to output is illustrated, showing n inputs to a node, who each are the outputs of the nodes from the previous layer. A weight is added to each of the inputs before they are summarized. The activation function added can be of different types, but are often logistic, tanh or rectified linear unit (ReLU)[37]. The result will be the output of the current layer, and an input to the nodes of the following layer.

$$\text{output}_i = a(\sum_{k=0}^{n} \theta_{j,i}\text{output}_j) \tag{2.13}$$

Equation 2.13 shows that from the first layer j to the next layer i, the output of the nodes in layer i is given by the sum of the outputs from all n nodes in the previous layer j weighted with the respective weights θ_i, j, which is the weight from layer j to layer i. This sum is given by z_i Finally, to obtain the outputs in layer i, the activation function a is applied to z_i. The equation is illustrated in Figure 2.3b.

When training a neural network to reach a target, the weights θ are what is being adjusted during the training process in to minimize the loss between the function represented by the neural network and some target. Therefore, when trying to minimize the loss in Equation 2.12 between the on-policy Q-function and the target describing the optimal Q-function, the Q-function can be approximated using a neural network, and the parameters θ^Q will be the weights of the network approximating the Q-function. This will be further elaborated on in the section on backpropagation.



(a) A neural network with an input layer, m hidden layers of dimension n and an output layer.

(b) Calculation done inside network.

Figure 2.3: The figure illustrates the feed-forward process in neural networks, showing both the structure of the networks and how layers are connected as well as the computation process in each neuron.

Figure 2.3 illustrates how neural networks are structured and how the calculations are done inside each neuron. In Figure 2.3a a neural network with an input layer, two hidden layers and an output layer is

illustrated. The dimension of the input layer and the hidden layers is five due to the five neurons in each of the layers, and the dimension of the output layer is three. This illustration show how each signal enters into each neuron in all layers. This is illustrated in a greater detail in Figure 2.3b where all n inputs are multiplied with one weight $\theta_{j,i}$ each, before they are summarized. Next they are sent from the neuron as the output, an activation function, a, is applied to the neuron. This process is shown in Equation 2.13.

Backpropagation

After the output is calculated the network can learn the best values of the weights by looking at the difference between the output of the network and some target, often called the loss or loss function. It is possible to backpropagate the error from the output layer to the hidden layers[30]. In Equation 2.12 the loss between the approximated On-policy Q-function and the target is given by $L(\theta)$, due to θ being the only parameters that can be adjusted to make the loss smaller. The goal is to minimize the loss by adjusting the weights of the network which can be done by evaluating how each weight contributes to the loss. This concept is called backpropagation, as it starts at the output of the network, and works its way back adjusting the weights layer by layer to minimize the loss.

Using the loss function $L(\theta)$ in Equation 2.12 as an example, the concept of backpropagation can be explained. The backpropagation process emerges directly from a derivation of the overall loss gradient[30] One weights contribution to the loss is given by finding the gradient of the loss-function with respect to that weight. For one single weight $\theta_{j,i}$ from i to j the gradient is

$$\frac{\partial L(\theta)}{\partial \theta} = \frac{\partial L(\theta)}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial \theta} \tag{2.14}$$

where $z_i = \sum_{k=1}^{n} \theta_{j,i} x_i^k$ is the weighted sum of the inputs x^k to layer i, and n is the number of inputs. a is the activation function, which will be introduced in more details in the following section.

Equation 2.14 shows how the gradient of the cost function can be calculated based on one weight using the chain rule. If there is more than one weight, which will be the case if the dimension of the input or output is more than one or if there is one or more hidden layers, the chain rule can also be used to find the partial derivative of the loss function with respect to all weights to find out how much they each contribute to the error, and how they should be adjusted. In Figure 2.4 the process of backpropagation is illustrated with a simple network consisting of a one dimensional input, one hidden layer of one dimension and an output layer of one dimension. The objective is to update the connections between the input units and the hidden units, by defining a quantity analogous to the loss term for the output for each weight[30].

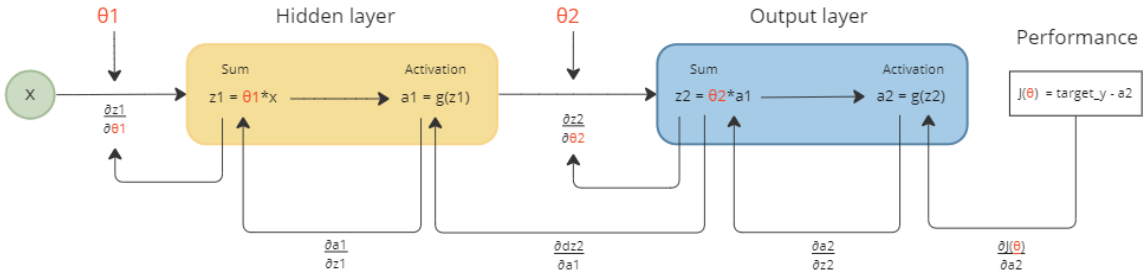


Figure 2.4: The process of back propagating the error between the calculated output and the target output. The figure illustrates how the gradient of the loss/error depends on the weights of the network θ in a case with one input layer, one hidden layer and one output layer where each layer consists of one neuron.

The process in Figure 2.4 starts at the loss function, given by $J(\theta)$ as the loss depends on the weights θ of the network. The gradient of θ_2 can be calculated using Equation 2.14, however the gradient of θ_1 is not as simple. Due to the chain rule of partial derivatives the gradient of a loss function can be represented as a product of gradients of all the activation functions of the nodes with respect to the weights. Therefore the update of the weights will depend on the activation functions of each of the nodes, which is presented in the following equation

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{\partial J(\theta)}{\partial a^{(2)}} \frac{\partial a^{(2)}}{\partial z^{(2)}} \frac{\partial z^{(2)}}{\partial a^{(1)}} \frac{\partial a^{(1)}}{\partial z^{(1)}} \frac{\partial z^{(1)}}{\partial \theta} \tag{2.15}$$

From Equation 2.15 it can be seen that the backpropagation of error in weights close to the input layer will depend on gradients of the activation functions of each node. This will be discussed further in the section about activation functions, as it is an important factor in choosing the correct activation function.

Based on the gradient and a hyperparameter called the learning rate, α, of the network the weights are updated to minimize the loss.

$$\theta_{new} = \theta_{old} - \alpha \frac{\partial L}{\partial \theta_{old}} \tag{2.16}$$

Equation 2.16 shows how the new weight is calculated based on the gradient and α which is called the learning rate when we are trying to minimize loss in a learning problem[30]. The gradient $\frac{\partial L}{\partial \theta_{old}}$ describes the weights contribution to the loss, and the learning rate $\alpha \in [0, 1]$ controls how large of a step to take in the direction of the negative gradient[39]. A if $\alpha = 1$ the whole error contribution from the weight should be corrected in one step, while if $\alpha = 0$ the weight will not be corrected at all. A learning rate that is too low will therefore result in a slow training, however a too large value can result in learning sub-optimal weights as the model converges too quickly, causing the system to diverge in terms of the objective function[39].

Activation functions

Activation functions are important to the performance of neural networks due to the amount of non-linearities in the real world. The activation functions help in learning and making sense of non-linear mappings between the inputs and corresponding outputs[40]. Linear activation functions creates an output directly proportional to the input. It would therefore not be many benefits of using a linear activation function as the network would not be able to identify complex patterns in the data[40].

The tanh activation function takes the input value and outputs a value in the range $(-1, 1)$ thus bounding the input which helps to make sure the gradient does not get too large as this would lead to large updates of the weights during training[40]. The tanh activation function is given by

$$tanh(x) = 2\sigma(2x) - 1 \tag{2.17}$$

where x is the input and $\sigma(x) = \frac{e^x}{1+e^x}$.

The challenge of the tanh activation function is that it will result in the inputs being bounded between -1 and 1. This can result in the gradient $\frac{\partial J(\theta)}{\partial \theta}$ approaching zero if the network has several layers due to the chain rule in Equation 2.15. With a gradient approaching zero Equation 2.16 illustrates how the learning will slow down and stop as the gradient approaches zero. If the gradient approaches zero a stall in the update of the parameters occur, since the algorithm uses the gradient to calculate the next step[41].

The problem arises when the activation function results in a small gradient, and therefore one solution is to use a different activation function. Rectified linear unit (ReLU) is another non-linear activation function that is widely used in neural networks[41]. If ReLU is used as an activation function the output will be equal to the input if the input is positive, and zero if not. The derivative of the output of the activation function when using ReLU will therefore be either 0 or 1, which prevents the gradient from vanishing.

$$ReLU(x) = \max(0, x) \tag{2.18}$$

Types of neural networks

There are different ways of connecting the neurons in the network together. In the previous sections on neural networks the concept of feed-forward networks is explained, where the connections between the networks are only in one direction. In this type of network there are no loops, and the network represents a function of its current inputs which makes the weights inside the network the only internal state of the network[30]. Feed-forward networks are on the form presented in Figure 2.3. They will be the main focus

in the rest of the thesis, however a short introduction to some other popular neural network types are given in the following paragraph.

Other than feed-forward networks, convolutional neural networks and recurrent neural networks are common types of neural networks. Convolutional Neural Network (CNN) is similar to feed-forward networks, but is primarily developed when handling images as it allows for encoding image-specific features into the neural network architecture[42] where the input to the network consists of the pixels of the image. CNN handle the large dimension of the input by having a weight sharing structure and pooling methods.

Recurrent networks are neural network with feedback connections designed to learn sequential or time-varying patterns[43]. The feedback connections creates an internal memory that enables them to have an understanding of sequences, suitable for handling text, time-series, video etc.

The structure of the input is important for determining which type of network should be used, both in terms of size and structure. If the input data to the neural network is unstructured a CNN is needed[1]. Structured data refers to data that resides in a fixed place in a file[1] and contrary to unstructured data the more simple feed-forward networks can be used.

The architecture of neural networks

An important aspect of a neural network is its architecture. That is the dimension of the layers and the number of hidden layers in the network. Many researchers do agree that the quality of the solution of a network depends on the size of the network as it affects the networks capabilities to find accurate patterns outside its training data[44]. Fitting a neural network is like a regression problem for polynomials which is concerned mainly about two challenges: finding the order of the polynomial and finding the coefficients of the polynomial. If the order of the polynomial is too high it can lead to overfitting, which makes the approximated function bad at generalizing, however if the order of the polynomial is too low the approximated function will struggle to understand complex patterns. Similarly, a network with dimensions that are too large will perform nicely for patterns in the training data, but worse for unknown test data. If the dimensions are not large enough the network can have problems being accurate enough[44].

2.5 DDPG: Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient is a deep reinforcement learning algorithm that combines policy gradient and value-based learning to obtain the optimal policy. It uses off-policy learning to learn a Q-function and then the Q-function is used to learn the policy.

The algorithm is based on two neural networks. The actor network, which is the approximator to the policy function, proposes an action given a state. The second network is the critic network, which estimates the Q-function that predicts whether the proposed action is good. Then the actor network is adjusted based on the feedback given by the critic through gradient calculations to improve the policy.

2.5.1 Handling continuous action spaces

Since many physical control tasks have continuous and high dimensional action spaces the algorithm used has to be able to handle this. Other, similar approaches to DDPG is Deep Q-Networks (DQN), however DQN cannot be straight forwardly applied to continuous action spaces as it relies on finding the action that maximizes the Q-function which requires an iterative optimization process at each step[35]. A suggested approach for such algorithms suggested by both[1] and[35] is to discretize the action space, however this would result in limitations. The number of actions will increase exponentially with the number of degrees of freedom[35]. The DQN method is therefore combined with an actor-critic approach, where neural networks are used as function approximators.

The challenge of applying Q-learning to continuous action spaces is due to locating the best action a_t using the greedy policy will require solving the optimization problem in Equation 2.7 at every time step over all actions in the continuous action space[35]. Instead a policy $\pi(s)$ is learned using a gradient based method which is based on $Q^*(s, a)$ being differentiable with respect to the action. Based on this the optimization problem in Equation can be approximated by $\max_a Q(s, a) \simeq Q(s, \pi(s))$. This solution allows for the

continuity to be handled when calculating the policy instead of finding the value that maximizes the Q-value.

2.5.2 DDPG algorithm

A short introduction to the main concepts of the networks used in DDPG has been given, however this section will give a more detailed explanation. In addition the concept of target networks will be presented.

Replay Buffer and off-policy learning

DDPG is an algorithm that learns off-policy by utilizing a replay buffer. This allows the algorithm to benefit from learning across a set of uncorrelated transitions[35], allowing for exploration. The replay buffer consists of a finite number of samples in the format (s_t, a_t, r_t, s_{t+1}). The the buffer is filled up by sampling using a different policy than what the actor uses to choose actions, and when the replay buffer is full the oldest samples are discarded[35].

Calculating the target using target networks

The value-based side of DDPG is based on minimizing the loss, stated in Equation 2.11, between the target in Equation 2.11 and the current on-policy Q-function in Equation 2.9. This loss is used to update the on-policy Q-function by adjusting the weights of the critic network. A problem with this is that it can lead to unstable or diverging learning, which is due to the critic network that represents the on-policy Q-function also is used to calculate the target[35]. This can be seen in Equations 2.11 and 2.9, where both depend on the critic network parameters θ^Q, resulting in that every time the on-policy Q-function is updated to reach the target, the target is also updated. This has the on-policy Q-function trying to reach a moving target.

The solution suggested for solving this problem is in the original DDPG paper[35] presented as Target networks. A copy is created of both the actor and critic network, $\pi(s|\hat{\theta}^{\hat{\pi}})$ and $\hat{Q}(s, a|\theta^{\hat{Q}})$ respectively. These networks are used for calculating target values by updating the weights such that they slowly track the original networks. The result is that the target values are changing to obtain the final target, but they are changing slowly. Doing so should make the the result more stable, however due to the target changing slowly it will take longer to reach the final target, making the training last longer.

$$\theta^{\hat{\pi}} \longleftarrow \tau\theta^{\pi} + (1-\tau)\theta^{\hat{\pi}} \tag{2.19a}$$

$$\theta^{\hat{Q}} \longleftarrow \tau\theta^{Q} + (1-\tau)\theta^{\hat{Q}} \tag{2.19b}$$

Equation 2.19 show how the weights for the target networks are copied from the original networks. The parameter $\tau \in [0, 1]$ determines how strong the copy is. The higher the value of τ, the stronger the copy will be.

Symbol	Description
θ^Q	Weights of critic network that approximates on-policy Q-function
θ^{π}	Weights of actor network that approximates policy
$\theta^{\hat{Q}}$	Weights of target critic network that approximates the optimal Q-function
$\theta^{\hat{\pi}}$	Weights of target actor network that approximates policy used in calculating target
α_{actor}	Learning rate of the actor network
α_{critic}	Learning rate of the critic network
γ	Discount factor for the long-term reward
τ	Update factor for target networks

Table 2.1: Symbols for the parameters of the networks and hyperparameters introduced in this chapter.

In Table 2.1 an overview of the parameters belonging to the different networks are given together with a description of the networks role. The target actor and target critic parameters are soft copies of the original actor and critic parameters like given in Equation 2.19. The table also gives an overview over hyperparameters used in this chapter.

When calculating the target using the target networks, Equation 2.11 can be rewritten as

$$y_i = r_t(s_i, a_i) + \gamma \hat{Q}(s_{i+1}, \hat{\pi}(s_{i+1}|\theta^{\hat{\pi}})|\theta^{\hat{Q}}) \tag{2.20}$$

in Equation 2.20 the target at time step t is given by y_t. It is based on the Bellman equation in 2.10. In this updated equation however, the target actor and target critic networks are used. This can be seen where the target policy $\hat{\pi}(s_{t+1}|\theta^{\hat{\pi}})$, which is approximated by the target actor network, is used as the optimal policy seen in Equation 2.11. The optimal Q-function is determined by the weights from the target critic network $\theta^{\hat{Q}}$.

Updating the critic

The weights of the critic network, θ^Q are updated by minimizing the loss between the target y_t and the on-policy Q-function following the current policy. This is done by first sampling a batch of N transitions (s_i, a_i, r_i, s_{i+1}) from the replay buffer, and then use the sampled transitions to calculate the target in Equation 2.20 and the on-policy Q-function $Q(s_i, a_i|\theta^Q)$.

Over the N samples from the batch the loss is given by:

$$L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{2.21}$$

TO DO: explain symbols and make sure it is necessary to have this equation and the other one

Updating the actor

The actor network is an approximator to the policy, and the weights of the network, θ^π, are therefore updated over iterations for the approximated policy to approach the optimal policy. The optimal policy will be the policy that maximizes the Q-function. The update of the actor is based on Equation 2.22.

$$\nabla_{\theta^\pi} J \simeq \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\pi(s_i)} \nabla_{\theta^\pi} \pi(s|\theta^\pi)|_{s=s_i} \tag{2.22}$$

The equation states that to change the Q-function, the weights of the actor network, θ^π, has to be changed. This can be seen when looking at the equation on the form $\frac{dJ}{d\theta^\pi} = \frac{dQ}{da}\frac{da}{d\theta^\pi}$. Due to the continuous action space, the Q-function is assumed to be differentiable with respect to the action such that gradient ascent can be performed with respect to the actor network parameters.

The way the update is calculated is by first using the actor network together with the states of the batch of transitions that is sampled from the replay buffer, that is, the same batch used when calculating the target and update for the critic. The actor network/policy is then used to calculate an action for each of the states, before the states are sent to the critic network who calculates the resulting Q-values. Then gradients of the Q-values are calculated with respect to the actions chosen by the actor network. Next the states are sent to the actor again, mapping them to actions before the gradient of the actions is calculated with respect to the actor network weights. By multiplying the two gradients and dividing by the number of transitions sampled from the replay buffer, the result will be the gradient of the objective function J, which is what should be maximized. The maximization can be done using gradient descent on the negative of $\nabla_{\theta^\pi} J$ ziped with the network parameters. It can be noted that the parameters of the critic network are treated as constants in this process.

2.5.3 Summary of algorithm

To summarize the previous sections that describe the elements of the DDPG algorithm the algorithm from the original DDPG paper[35] is presented and explained here.

Algorithm 1 summarizes the previous chapters on the DDPG algorithm. First the actor network, critic network and the target networks as well as the replay buffer are initialized. Following, for a set of M

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor network $\pi(s|\theta^\pi)$ with weights θ^Q and θ^π.
Initialize target networks $\hat{Q}$ and $\hat{\pi}$ with weights $\theta^{\hat{Q}}$ and $\theta\hat{\pi}$.
Initialize replay buffer *R*.
for episode = 1,M **do**:
    Get initial state s_t
    for t = 1,T **do**:
        $a_t = \pi(s_t|\theta^\pi)$
        Execute a_t in environment and observe r_t and s_{t+1}
        Store (s_t, a_t, r_t, s_{t+1}) in *R*
        Sample batch from R of N transitions (s_i, a_i, r_i, s_{i+1}) from *R*
        Calculate target:

$$y_i = r_t(s_i, a_i) + \gamma\hat{Q}(s_{i+1}, \hat{\pi}(s_{i+1}|\theta^{\hat{\pi}})|\theta^{\hat{Q}}) \tag{2.23}$$

        Update critic by minimizing loss:

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2 \tag{2.24}$$

        Update actor policy using the sampled policy gradient:

$$\nabla_{\theta^\pi} J \simeq \frac{1}{N}\sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\pi(s_i)} \nabla_{\theta^\pi}\pi(s|\theta^\pi)|_{s=s_i} \tag{2.25}$$

        Update the target networks:

$$\theta^{\hat{\pi}} \longleftarrow \tau\theta^\pi + (1-\tau)\theta^{\hat{\pi}} \tag{2.26a}$$

$$\theta^{\hat{Q}} \longleftarrow \tau\theta^Q + (1-\tau)\theta^{\hat{Q}} \tag{2.26b}$$

    end for
end for

episodes the initial state s_1 is observed, before a series of T steps is performed. The policy $\pi(s_t|\theta^\pi)$ chooses an action based on the latest observed state, and executes the action in the environment. The resulting reward r_t and new state s_{t+1} of the environment is observed after the step is taken, and the transition (s_t, a_t, r_t, s_{t+1}) is stored in the replay buffer.

For the agent to learn and update the weights of the network a batch of N transitions is sampled from the replay buffer. The replay buffer is an important part of the algorithms off-policy learning, as it at first is filled with samples from an other policy than the policy of the agent. One way to solve this is to initialize the replay buffer with transitions that are based on random sampled actions. As seen in line 9 transitions based on actions chosen by the agents policy will however be stored in the buffer, replacing the transitions originally stored there. When sampling the batch of transitions in line 10, the samples will be used for calculation of the current target in line 11, where the target networks are also used. The target is then used in calculation of the loss L together with the Q-function and the sampled transitions.

The policy is then updated to choose the action that maximizes $Q(s, a)$. The policy update is based on the equation in line 13, where the goal is to maximize the gradient of the objective function. Finally the weights of the target networks are slowly updated to follow the weights of the original networks.

Chapter 3

Methodology

This chapter describes the development of the algorithm, and the tools used for the implementation. The simulation setup will also briefly described, but it is based on the environment presented in the specialization project. The chapter ends with an overview of the different cases studied in the experiments, as well as a detailed description of each case.

3.1 Implementation

The implementation of the project is an extension of the code presented in the specialization project. In this thesis the results are based on extending the code with an agent that is based on the algorithm presented in 1. The implementation contains functionality for training the agent and updating the parameters of the target networks which is based on the theory presented in Chapter 2.

First the algorithm was implemented in a 1D environment to observe the agents abilities to learn, verifying the functionality of the implemented DDPG agent. A simple case where the agent had to choose the size of a step to reach the goal position. By choosing an action the same size of the distance to the goal the agent would be rewarded with the highest possible reward, while having to take multiple steps to reach the goal would increasingly reduce the reward. Not reaching the goal at all resulted in no reward.

With the verification of the functionality of the case implemented in 1D, the implementation of the algorithm is done in a 2D environment. This is firstly due to the main objective of this thesis being to investigate the possible performance of the decision making of the agent without being limited by computational resources. The agent lives in a 2D world, and is given the position of the goal it is supposed to reach while ideally taking as few steps as possible. By first filling the replay-buffer to a set size using random samples, and then training the agent for a number of episodes, the agents learning can be observed. In this thesis the learning process is illustrated using learning curves, which will be presented later in the thesis. Following the learning process the agent was tested in to see how well it performs in an unknown situation. During testing the agent will at each step output the desired course, which based on its own experience will result in reaching the goal without colliding with any obstacles detected during the mission.

The following sections will go into further detail on the implementation of the project, including important specifications and choices made. First the tools used to implement the code are presented, which includes choice of programming language and other software tools. Following is details on how the environment is represented presented, describing the POMDP of the reinforcement learning problem. This includes a representation of the observation space, action space, the implementation of the reward function and the movement of the agent in the environment modeled using a step-function. The implementation of the agent is based on the DDPG algorithm presented in Algorithm 1, and this chapter goes into further detail on specific choices made to apply DDPG to the motion planning problem presented. Following the process of training the algorithm is described. Finally a section introducing the experiments that will be conducted for training and testing of the agent, including a description of the plots and performance metrics.

3.1.1 Tools of implementation

The implementation is done done in Python due to a good selection of tools and libraries for neural networks, deep learning and reinforcement learning. Some of the frameworks used are Keras and TensorFlow for handling the learning side of the algorithm. In recent years a variety of software libraries has been released that significantly has eased and accelerated the application of neural networks, with the most popular being TensorFlow[45]. TensorFlow builds a computational graph consisting of nodes and edges, providing basic building blocks for neural networks like fully connected dense layers, convolutional layers, recurrent networks and nonlinear activation functions[45]. In addition functions for computing mean squared error (MSE), functions for gradient computation and optimizers[45]. Keras is a high-level Python library for deep learning that can run on top of TensorFlow[46]. In Keras a model can be constructed in a high-level manner by stacking predefined layers, and calling methods to compile and train the model[45]. To represent the environment, which is formalized as a POMDP[47], for interaction with the agent OpenAI Gym is used, a toolkit for predefined environments that also enables creating costume environments. To visualize the results Matplotlib is used, which is a graphics package for data visualization in Python[48].

Both actor and critic networks are implemented using Dense layers from Keras to represent feed-forward networks, and a replay-buffer is also implemented to perform off-policy learning. In addition there is also functionality enabling storing the parameters of the networks in files to a specified directory. This is important to load the network parameters into the test function after the training is finished, or to pause the training and resume it at a later point by saving the parameters and then loading them back.

3.1.2 Representation of environment: OpenAI

For the agent to learn it receives observations of the environment before and after a step is made. This allows it to learn how the action made affects the current state. Having a representation of the environment that is being observed is therefore necessary. The environment contains information on the observation space, action space, obstacles and the motion of the vehicle. OpenAI Gym is a open source Python library for developing Reinforcement Learning algorithms. It provides a standard API for communication between the learning algorithm and environments[49].

The environment is modeled as a Python env class, where the environment is initialized with an observation space, action space and other details of the environment. The standardized class also contains a step function used model a step made by the vehicle, returning the new observation of the environment, and a reset function that resets all environmental information for starting a new episode.

In this thesis a custom environment is implemented, and will be described in the following sections. The environment is based on the work done in the specialization project, however some changes are made and the main concepts are therefore explained again.

Observation space

The observation space defines the structure of the observations given to the agent of the environment. In the case presented in this thesis the observations are given by continuous values, assumed to mainly be represented by lidar measurements giving the distance to elements in the environment. The continuous observation space structure is therefore defined using max and min values for each of the observations, while the shape of the observation space depends on the problem, and will be presented in the following sections and in Table 3.1.

The observation made by the robots sensors describes the environment to the agent. It is therefore important that the observations contain sufficient information about the environment for the agent to choose an appropriate action. In this thesis some relevant observation elements include the state of the environment, topological information like the net of the fish farm cage as well as obstacles.

When getting information about the surroundings of the vehicle there are several options presented in current literature. In a real world scenario the AUV would retrieve environmental information through sensor measurements like camera images, radar or lidar information, in this thesis the sensor measurements are assumed to be by lidar, as lidar measurements can provide good depth measurements of the environment. In section 2.4.1 different types of neural networks are presented and it is explained how the structure of the input data to a neural network affects the structure of the neural network, where unstructured data

Environment	Observation dim	Observation	Min and max values
1D	1	obs = [goal_dist]	[0,env_size]
2D with one obstacle	3	obs = [goal_dist,yaw_diff, obstacle_distance]	[(0,-180,0),(env_size,180,sensorrange)]

Table 3.1: The table gives an overview over the observation spaces in the different cases of environments.

requires using CNN, while when using structured data feed forward networks are sufficient. In this thesis the input of the neural networks will include the observation of the environment, therefore the structure of the observation is important to determine to determine the necessary structure of the neural networks.

An AUV operating in a cage is faced with multiple potential obstacles, and to avoid colliding with those the agent will need information about the distance to those obstacles. There are different ways of solving this problem, but to maintain a fixed input of the neural networks the size of the observation has to be fixed as well. One way to solve thus is to set the observation space to contain n predefined observations for obstacles. These observations will contain the distance to the n closest obstacles, and if there are less than n obstacles within the range of the sensors, the additional distances are set to be a high, fixed value. One of the main challenges with solving the problem like this is that there will be a trade off between keeping the dimensionality of the observation from being unnecessarily high while still making room for enough observations.

Since the decision process focuses on the ego-vehicle, the reference frame chosen is centered at the origin of the AUV. By choosing a reference frame that is pinned to the vehicles reference frame the region of state-space in which the policy must perform is reduced[1]. By only having knowledge about the nearest parts of the environment the observation space the policy has to perform in will be reduced. It is also closer to a real deployment where the agent uses sensors mounted at the AUV.

The observation spaces in table 3.1 show how the state of the environment is observed by the agent in the different cases. In the one dimensional environment the observation is the distance the AUV has to the known goal configuration. In the case of a 2D environment the agent is given the information about the distance to the goal, as well as the difference between its current orientation and the orientation of the goal. The difference in orientation is calculated by having the goal position. The 2D environments observation also includes the distance to the obstacles.

Action space

The choice of action space depends highly on the task to be solved. In this thesis the goal is for the agent to control the position of the AUV towards the configured goal trough choosing the AUVs orientation while the step size is fixed. The action space therefore has the shape $(1, 1)$ where

$$a_t = [\Delta\psi] \tag{3.1}$$

for all the cases presented in section 3.2. In Equation 3.1 a_t is the action at time step t, and $\Delta\psi$ is the change of orientation from ψ_{t-1} to ψ_t.

Due to the nature of the problem the action space is set to be continuous, and the space is therefore defined by the min and max values of the action which should be decided by the dynamics of the AUV, however in this thesis it is set to $a_t \in [-20, 20]$. The vehicle is in other words able to turn 20 degrees in both direction at each step.

Rewarding

Rewarding plays an important role in the agents learning process by evaluating how good the agents choices are, which helps to improve the policy. Implementing a reward function that describes the wanted behaviour

is therefore crucial for the training process. The reward function will return the reward that is the result of a step, the step reward.

The main goal of the agent is to have the AUV reach a configured goal without colliding with the surrounding obstacles. The step reward will therefore consist of a positive reward for reaching the goal and a negative reward for colliding with obstacles. The challenge with only rewarding when the goal is reached or a collision occurs is that the learning process will be slow due to only getting feedback at those events. A solution to this is to add lighter rewards at each step resulting in a evaluation of the current state, resulting in a faster learning process. In this thesis this includes a positive reward when the yaw diff decreases as well as a reward for a decreasing distance to the goal. However a small negative reward for each step is also included to make the agent use as few steps as possible.

The rewards will be explained in greater detail in the section presenting the simulation experiments, where the rewards relevant to the different cases is presented.

Step function and vehicle model

The process of motion planning for a dynamic vehicle depends on the model describing the dynamics of the vehicle. The reason is that the same input will result in different reactions when passed to different models. The model used for implementation of the motion planner should therefore be as accurate as possible to avoid large corrections having to be made by the controller[50].

However, when modeling the movement of the vehicle for a reinforcement learning based motion planner there is a trade off problem between model accuracy and computational resources[1]. This is due to the number of episodes necessary for determining the optimal policy. The step-time of each step made during each episode is based on the evaluation time of the vehicle dynamics model, and complex dynamics are therefore affecting the training time.

In this thesis a simple kinematic model is used, where change in position and orientation is given by:

$$\Delta\theta = \theta_t - \theta_{t-1} = a_t \tag{3.2a}$$

$$\Delta x = x_t - x_{t-1} = L\cos\theta_t \tag{3.2b}$$

$$\Delta y = y_t - y_{t-1} = L\sin\theta_t \tag{3.2c}$$

In equation 3.2a the change of the orientation of the AUV is given by the action chosen by the agent. Following the position of the agent will be given by the x- and y-coordinates that results from taking a step of length L in the direction of the new orientation θ_t. As presented in Section 3.1.2 the length of each step is in this thesis set to be $L = 1$.

The equations in 3.2 describe a simple step, and do not include details on the dynamic motion of the agent. For the proposed motion planner to be used in simulations in dynamic environments as well as a physical experiment a model describing the dynamics of the vehicle therefore has to be included. In this thesis the choice of not including the dynamics has been made to investigate the performance of the algorithms abilities to reach a goal and avoiding obstacles without being limited by computational resources.

3.1.3 Implementation of DDPG agent

The agents abilities to make decisions is based on algorithm 1, consisting of four neural networks actor network, critic network, target-actor network and target-critic network. The networks are implemented using Keras Dense layers which are regular, densely-connected neural network layers that implement the operation.

Network architecture

The architectures of the networks are mainly based on the original paper presenting the DDPG algorithm[35]. All four networks are created as feed-forward networks that are fully connected between all layers like the one illustrated in Figure 2.3a. The paper uses two hidden layers with 400 and 300 neurons respectively for networks in a similar to, or higher, dimension-ranges as those presented in Table 3.1 for this thesis.

	dim(input)	n hidden layers	dim(hidden layers)	dim(output)
Initial parameters	Table 3.1	2	400 and 300	1

Table 3.2: Initial architecture of neural networks based on the original DDPG paper.

Table 3.2 gives an overview of the network architecture parameters given in[35]. These values will be the starting point of the experiments presented in the next chapter.

Activation functions

The layers of the actor and target actor networks use ReLU activation function for the hidden layers in to avoid slow learning due to a low gradient. The output layer will however use the tanh activation function to bound the action between -1 and 1. This value is later multiplied with 20 to represent the real values of the actions. For the critic and target-critic the hidden layers the ReLU function is also used to enable finding complex relationships in the data, however the output layers of the critic networks does not use an activation function to not bound the value in the Q-value calculations.

Hyperparameters

Hyperparameters are important for determining the architecture of the neural networks. In Section 3.1.3 the choice of the networks architecture is presented, however other hyperparameters are the learning rate of the networks, the soft update of the target networks and the discount factor. The role of each of the parameters is presented previously in the thesis and later the effect they have on the performance of the algorithm will be discussed.

	α **actor**	α **critic**	γ	τ
Initial parameters	10^-4	10^-3	0.99	0.001

Table 3.3: Initial neural network hyperparameters based on the original DDPG paper.

where α is the learning rate, γ is the reward discount factor and τ is for the soft target network updates. In the same way as the parameters describing the architecture of the networks, the parameters in Table 3.3 will also be from the original DDPG paper as a starting point.

3.1.4 Training process

When training the agent the number of training episodes is set at the beginning. Each episode consists of a series of iterations where at each iteration a step is taken from one state to the next state depending on the action selected. The episode is done when it 1) reaches the goal, 2) collides with an obstacle or 3) reaches the max number of iterations.

The first episodes are used to fill the replay buffer, using random selection of action from the action space as the initial behavior strategy. This is done to enable off-policy learning that allows for exploration and enables a more stable learning process. The transitions are on the format [state, action, new_state, reward, done]. When the replay buffer is filled, the agent is used to choose an action based on the DDPG algorithm, also resulting in transitions on the format [state, action, new_state, reward, done]. These are also stored in the replay buffer, replacing some of the transitions from the random sampling. The agent will therefore at first learn from sampling batches consisting of transitions from the random sampling, but as those transitions are replaced by the ones based on the agents choices, the agent will begin to evaluate its own performance. As mentioned, this is the way DDPG solves the exploration/exploitation trade off, as the random sampling which will result in a large portion exploration at first, but as the amount of transitions based on the current policy increases as the agent starts to make its own actions the exploitation will increase at the expense of the exploration.

3.2 Simulation experiments

The experiments include testing the performance of the agent in different environments with a varying difficulty level. The experiments start with a simple case in one dimension to verify the functionality of the implementation. Training agents in high dimensions is a time consuming process, therefore testing to see if the implementation works is first done in a low dimension.

Following the validation of the method in 1D with successful results testing in a 2D environment is done. Then the agents abilities to perform collision avoidance is tested by adding obstacles in a 2D environment while trying to reach a given goal.

The experiments are conducted in the following order:

1. Testing goal-reaching abilities of algorithm in 1D environment.
2. Motion planning towards goal with obstacle avoidance in 2D environment.

1D environment

After implementing the core DDPG-algorithm the implementation is tested in a 1D environment where the agent has an initial, randomly chosen position at the beginning of an episode which is a number between 0 and 10. A random goal is also set to be a number between 0 and 10, and the agent has to choose an action that minimizes the distance to the goal by using as few steps as possible. The agent receives the highest score, which is $r = 6$ when the distance to the goal is zero by using only one step. Based on the success of the testing in 1D the environment is expanded.

First the agent is trained using the initial parameters presented in Table 3.2 and Table 3.3 from the DDPG paper[35]. To see if the performance of the agent will improve when adjusting the parameters the learning rate $alpha$ will first be decreased to slow down the update of the weights. This will require a longer training process, but it can improve the update of the parameters making the training of the agent improve overall. Then the number of hidden layers is decreased from 2 to 1 for both the actor and the critic. This is done due to the observation space being of a low dimension, and a too large architecture of the networks could result in overfitting the training data, resulting in the trained agent not performing great when tested.

Obstacle avoidance in 2D environment

Then the agents obstacle avoidance abilities will be tested. This will be done in a 2D environment where obstacles are added and collision is detected if the vehicle is within some close radius of the centre of the obstacle.

For reaching the obstacle the agent is rewarded in the following way:

Using the distance and orientation to the goal as the observation of the agent the agent has to choose an action, a change of heading between -20 and 20 degrees, to reach the goal. The agent is rewarded with 30 points if a goal is reached, it is rewarded for approaching the goal based on the function

$$r_{dist} = 1 - \frac{|distance|}{distance_{max}} \tag{3.3}$$

where r_{dist} is the reward, $distance$ is the distance to the goal from the current position, and $distance_{max}$ is the highest possible distance between the current position and the goal. The agent is also rewarded for minimizing the orientation to the goal based on the function

$$r_{\psi} = 1 - \frac{|\psi_{error}|}{\psi_{errormax}} \tag{3.4}$$

where r_{ψ} is the reward, ψ_{error} is the angular difference between the current orientation of the robot and the orientation to the goal and $\psi_{errormax}$ is the max orientation difference. Equation 3.3 and Equation 3.4 show that if the agent approaches the distance and the orientation of the goal it is rewarded with 1 point.

To make the agent not take more steps than necessary it also receives a negative reward of $-0.1 * n_steps$ if the number of steps is higher than 1.5 times the distance between the start position and the goal. For the agent to choose an action that makes it avoid obstacles if there is one near by it is given a negative reward if a collision occurs of -4 points. The goal is for the agent to choose actions at each step that results the goal being approached, while there is no collision with the obstacle. The agent is also encouraged to choose a path that is as short as possible.

Some of the hyperparameters presented in the thesis will also be changed to observe how it affects the performance of the agent. The first test case will be using the initial parameters presented in Table 3.3 and Table 3.2. In the next case the learning rate, α, is decreased in order to observe how the agent reacts to a slower training process while the rest of the parameters are kept at their initial values. It is also interesting to observe how the agents performance changes with an increase in the number of hidden layers. A third hidden layer consisting of 200 neurons is therefore added before the output layer, while the other parameters are kept at the initial values.

3.3 Performance analysis and presentation

To illustrate the performance of the motion planner the following chapter will consist of two main types of plots. A percentage of successful runs will also be calculated to see how well the performance is over different cases.

Learning curves

The learning curves illustrate the learning of the agent based on the rewards received. If the agent chooses actions that suit the state well it receives a high reward and if it chooses an action that is not well suited the reward will be low. By observing how the total reward for all steps in one episode increases during training it can be observed how the agents abilities to choose actions improves over the training episodes.

To show how well the score is the total average score over all training episodes is also in the plot. This is to show how the over all performance is, but also to compare how fast, or after how many episodes, it takes before the performance is over average.

Learning curves will be a part of the results for both the case in the 1D environment and the 2D environment with obstacles. The learning curves will also be shown for all variations of parameters to be able to compare the learning process when the parameters are changed.

Plot of motion plan

To illustrate the performance of the trained agent test cases are run. Depending on the case being tested these cases will include a goal and possibly obstacles. These plots show how the agent chooses to solve a task when the position of the goal is known and obstacles are discovered during the process of reaching the goal. By having an illustration of the path chosen by the agent it can be seen if the path is longer than necessary as well as where it collides if a collision occur.

When creating a plot of the planed path the agent operates in the reference frame of the vehicle, however to create the plots easily using Matplotlib the position of the vehicle is plotted moving relative to the environment. This solution will however still illustrate the movement of the vehicle, the only difference is the plot being made in the environments reference frame.

The plots showing the motion chosen by the agent are not presented for the 1D environment case, but is presented the 2D environment with obstacles. The plots are presented for all variations of the parameters to see if there is any correspondence between the learning curve for the case and the motion plots.

Percentage of successful plannings

To determine how well the trained agent performs in different settings a series of 100 episodes in different environments will be performed. In each episode a flag will be set if the episode terminates due to a collision and one flag if the episode terminates due to reaching the goal. A percentage of episodes resulting in collisions and successful plannings can be calculated and compared to the learning curves.

Chapter 4

Results

This chapter presents the results from the simulation experiments conducted for evaluation of the performance of the proposed motion planner. First the results from verifying the method by training and testing the agent in a 1D environment is presented. Then the an agent is trained to investigate the potential for obstacle avoidance and later tested to observe how the trained agent performs.

The results will be presented using the tools for performance analysis presented in the previous chapter, however a plot of the motion plan will not be presented for the 1D verification case.

4.1 Testing in 1D

The following section presents the results when training an agent in a 1D environment. First the agent is trained with the initial parameters presented in Table 3.3 and Table 3.2. In the next case the learning rate is decreased to see if slowing down the learning process can improve the agents performance. Finally the dimensions of the architecture is changed.

In all of the cases where the agent is trained in a 1D environment a replay buffer consisting of 20 samples is used, and the agent is trained for 2000 episodes. In the cases where the parameters are changed from the initial parameters, for instance where the learning rate is changed, all the other parameters are kept at the initial value.

Initial parameters

The results of the training are presented in 4.1. The figures show how the agents received reward increases as it improves the understanding of what to do when trained to solve the 1D problem described in section 3.2 using the initial parameters for hyperparameters and architecture presented in respectively Table 3.3 and Table 3.2. The average of the total rewards presented in the green line can in Figure 4.1 be seen to be about 5.6, and the average episode reward shown in the blue line crosses the total average after around 400 episodes before it stays above above the total average line for the remaining episodes of the training process. By observing the learning curve in Figure 4.1 it can be seen that the curve mainly stays above the total average reward after crossing. It can also be seen that the process continues to improve and become smoother towards the end of the training process.

When testing the performance of the agent trained in Figure 4.1 100 times it is able to reach the goal 100% of the times, however it does use more than one step each time, as the average number of steps per episode is 3.84.

Lower learning rate

When lowering the learning rate α for both the actor and critic networks from the initial values $\alpha_{\mathrm{actor}} = 0.0001$ and $\alpha_{\mathrm{critic}} = 0.001$ for the actor and critic respectively to $\alpha_{\mathrm{actor}} = 0.00005$ and $\alpha_{\mathrm{critic}} = 0.0005$ the results can be seen in Figure 4.2 where the average reward is 5.5. The reward curve can be seen to

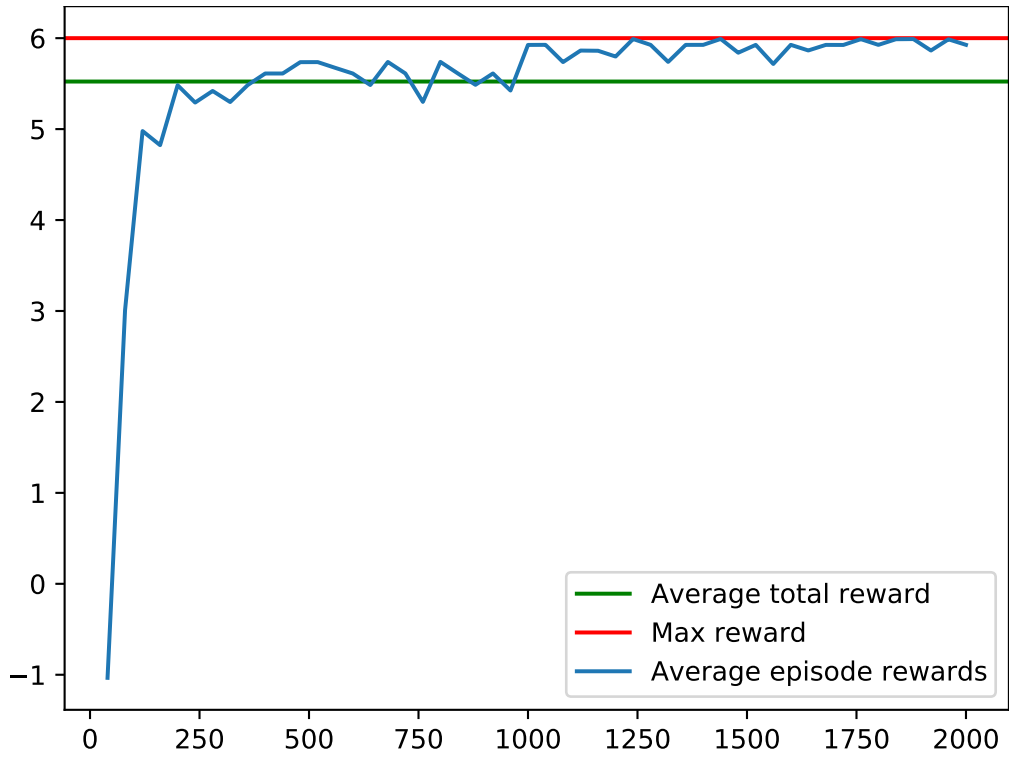


Figure 4.1: The learning curve show an agent trained in a 1D environment using the initial parameters presented in the previous chapter. The agent is trained in the 1D environment first to verify the implemented method.

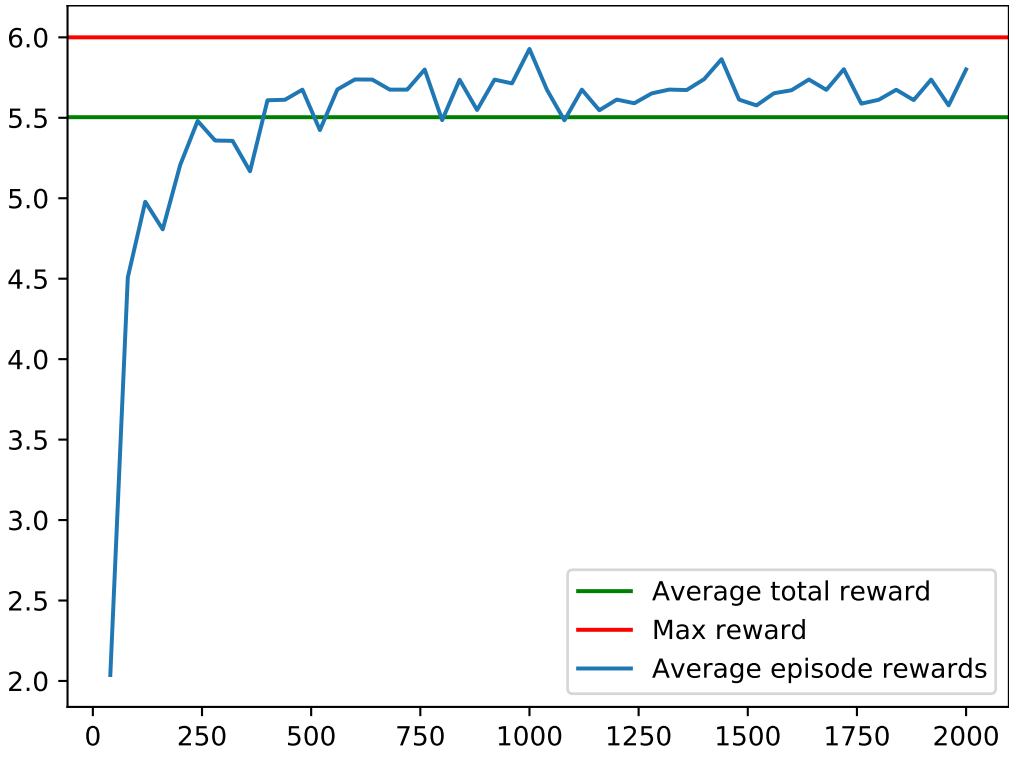


Figure 4.2: The learning curve show an agent trained in a 1D environment using presented in the previous chapter using a lower learning rate than the initial learning rates.

cross the total average line after about 450 episodes, however compared to Figure 4.1 where the initial parameters are used the curve is less steep at the beginning in Figure 4.2. Compared to the case with initial parameters the learning curve for the decreased learning rate can be seen to fluctuate more.

When testing the agent in Figure 4.2 100 times the goal is reached 21 of the times with an average number of steps per episode being 3.26.

One hidden layer

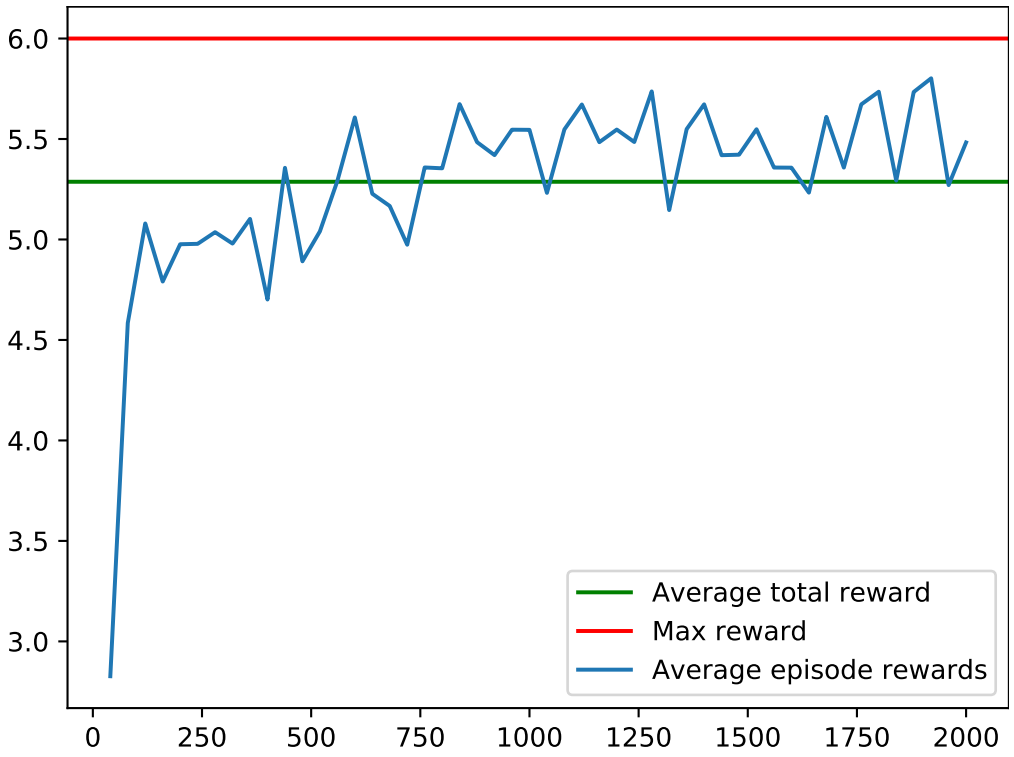


Figure 4.3: The learning curve show an agent trained in a 1D environment presented in the previous chapter using one less hidden layer than the initial learning rates.

When decreasing the number of hidden layers the learning curve can be observed from Figure 4.3 to have the slowest learning process out of the three cases presented in the 1D environment. The total average learning curve is also the lowest out of the three cases at around 5.2. The reward curve does not stay above the line for a longer time before after episode 750 and does fluctuate after this.

When testing the agent using only one hidden layer 100 times it is able to reach the goal 100 times with an average number of steps per episode at 3.73.

4.2 Results in 2D environment

This section will present the experiments in a 2D environment where the agent is trained to avoid obstacles, and tested to see if it can avoid collisions while trying to plan a path towards the goal.

4.2.1 2D environment with obstacle

The following results show the implemented DDPG agents abilities to perform obstacle avoidance. Obstacles are detected by the agent within some set sensor range for the agent to be able to handle obstacles detected during the running time of the algorithm.

Initial parameters

The initial training parameters are chosen to be like in the original DDPG paper[35] where the parameters defining the architecture of the networks are given in Table 3.2 and the initial hyperparameters are given in Table 3.3.

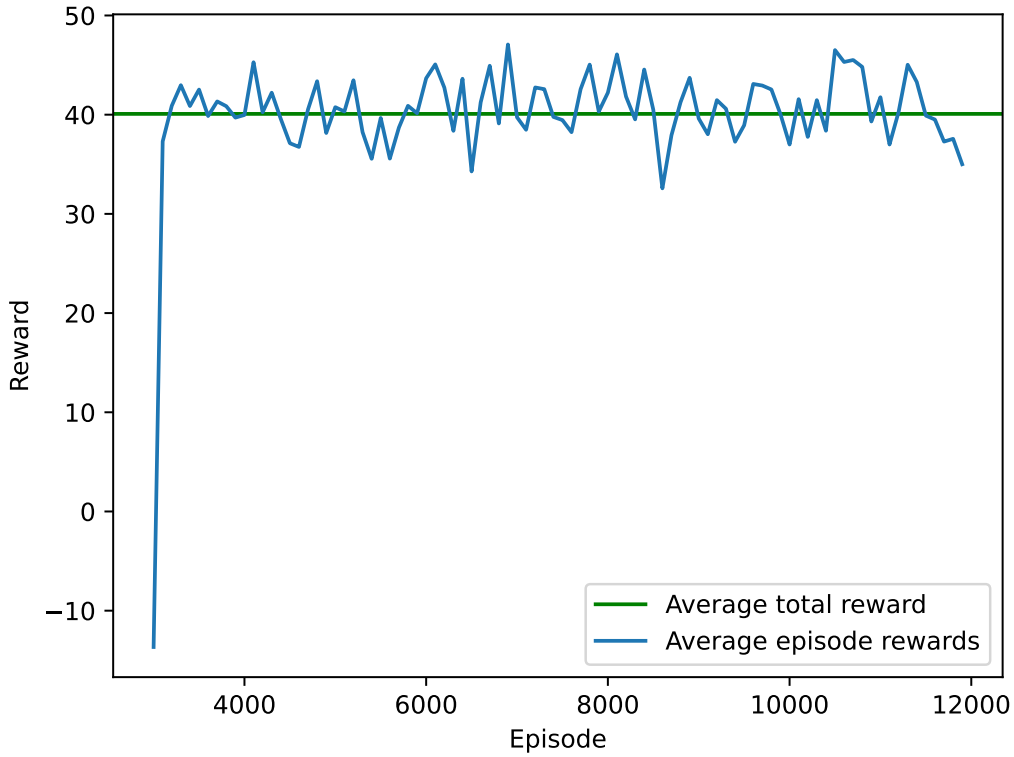


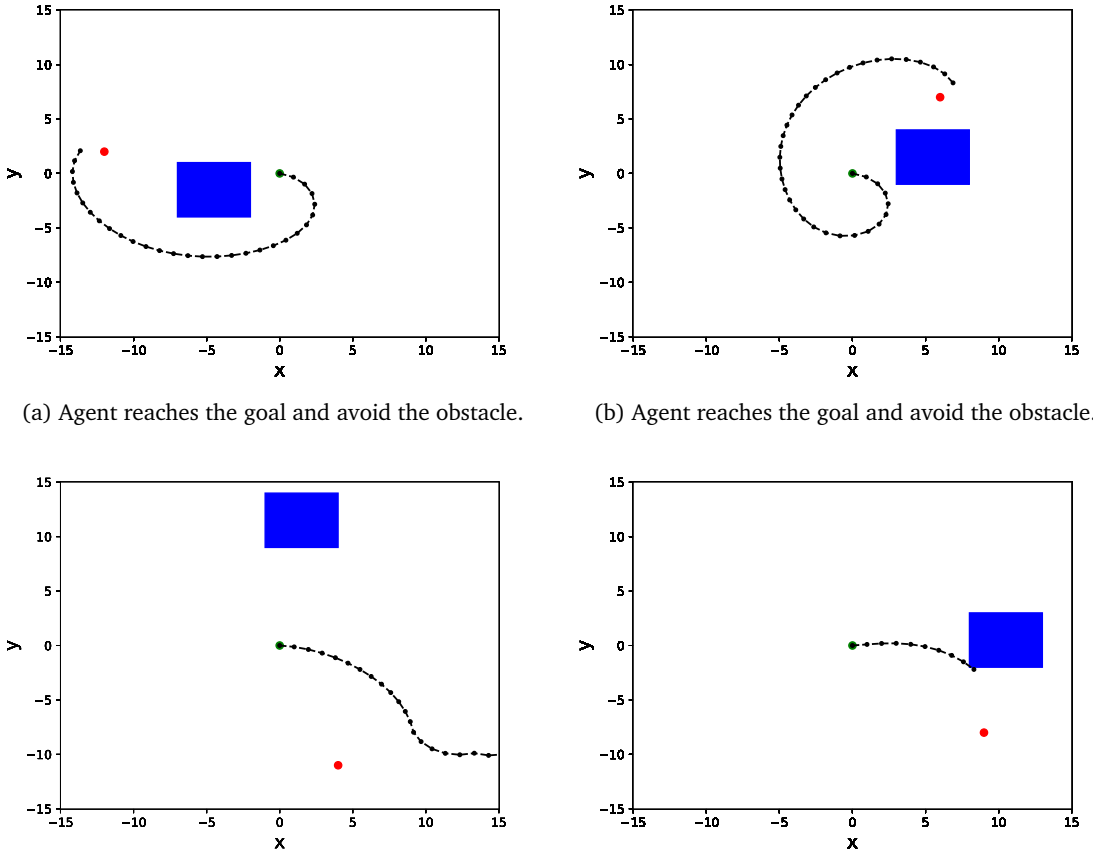
Figure 4.4: The learning curve illustrates the increase in reward for a DDPG agent training to find a collision free path from its starting position towards the goal. This agent is trained using the initial parameters presented in the previous chapter.

In Figure 4.4 the learning curve for an agent trained with initial parameters is shown in the blue curve. The agent is trained for a number of 10000 episodes, which includes filling the replay buffer which consists of 150000 samples. From Figure 4.4 it can be seen that the training of the agent therefore starts around episode 3000, and around 200 episodes later the learning curve crosses the line of the average total reward, which is indicated by the green line. After crossing the total average line the reward signal can be seen to fluxuate between staying over and under the line. During the first approximately 200 episodes the performance of the agent has a steep learning curve, which indicates how fast the agent learns.

Examples of the results from using the agent trained in Figure 4.4 to perform obstacle avoidance are shown in Figure 4.5. The agent is tested 100 times, and the results presented here are both representative but also show the most interesting results. In Figure 4.5 the starting position is indicated by the green point, the goal is indicated by the red point, the blue square is indicating the obstacle and the black dashed line is the path planned by the agent by choosing how the heading of the robot should change at each step.

Three examples of the agent reaching the goal while avoiding collision can be seen in subfigures 4.5a, 4.5b and 4.5c, while in 4.5d a collision with the obstacle occurs. The agent can be seen to choose a similar start to each of the paths despite the changing environment in each of the cases, where the first step in subfigures 4.5a, 4.5b and 4.5c starts by turning -20 degrees in all three cases.

After testing the agent 100 times with a random goal placement and random obstacle placement every time the goal is reached 45% of the test cases, collides 22% of the test cases and wander off as illustrated in Subfigure 4.7c. This corresponds to reaching the goal 40 times, colliding with an obstacle 20 times and wandering off 29 times. Resulting in 11 of the test cases not being valid due to the obstacle randomly being placed on top of the goal or the start position.



(a) Agent reaches the goal and avoid the obstacle.

(b) Agent reaches the goal and avoid the obstacle.

(c) Agent gets lost and wanders off.

(d) Agent collides with obstacles.

Figure 4.5: The four figures show example paths planned by the agent trained in Figure 4.4 with which is trained using the initial parameters presented in the previous chapter for both the actor network and the critic network.

Decreased learning rate

Figure 4.6 illustrates the learning process of an agent learning with a lowered learning rate, α for both the actor and critic networks from the initial values $\alpha_{\text{actor}} = 0.0001$ and $\alpha_{\text{critic}} = 0.001$ for the actor and critic respectively to $\alpha_{\text{actor}} = 0.00005$ and $\alpha_{\text{critic}} = 0.0005$.

Figure 4.6 show the learning curve when the learning rate for both the actor network and the critic network are lowered. Like in the case with initial parameters the first approximately 3000 episodes are used to fill the replay buffer, resulting in 7000 episodes used for training. The total average reward, the green line, is slightly above 40 and the episode reward shown in the blue line crosses the average total reward after 500 episodes. After the reward fluxuates, however mainly stays above the average line.

When using the agent trained in Figure 4.6 to perform motion planning examples of a resulting path can be shown in the Figures 4.7. The examples presented in Figure 4.7 are samples from testing the agent 100 times. Subfigure 4.7a and Subfigure 4.7b show that the agent is able to plan a path avoiding collision with an obstacle while reaching the goal, however in Subfigure 4.7c the agent is not able to train around the obstacle to the goal located behind the obstacle, resulting in a collision. In Subfigure 4.7d the agent can be seen to get lost and wander off.

When the agent trained in Figure 4.6 is tested 100 times the goal is reached 56 times, a collision occurs 14 times and wanders off 12 times. Some of the tests are invalid as the randomly placed obstacle covers either the start position or the goal position. The percentage of times the goal is reached is therefore 68% of the tests, collision happens 17% of the tests and the remaining 15% the agent wanders off and is unable

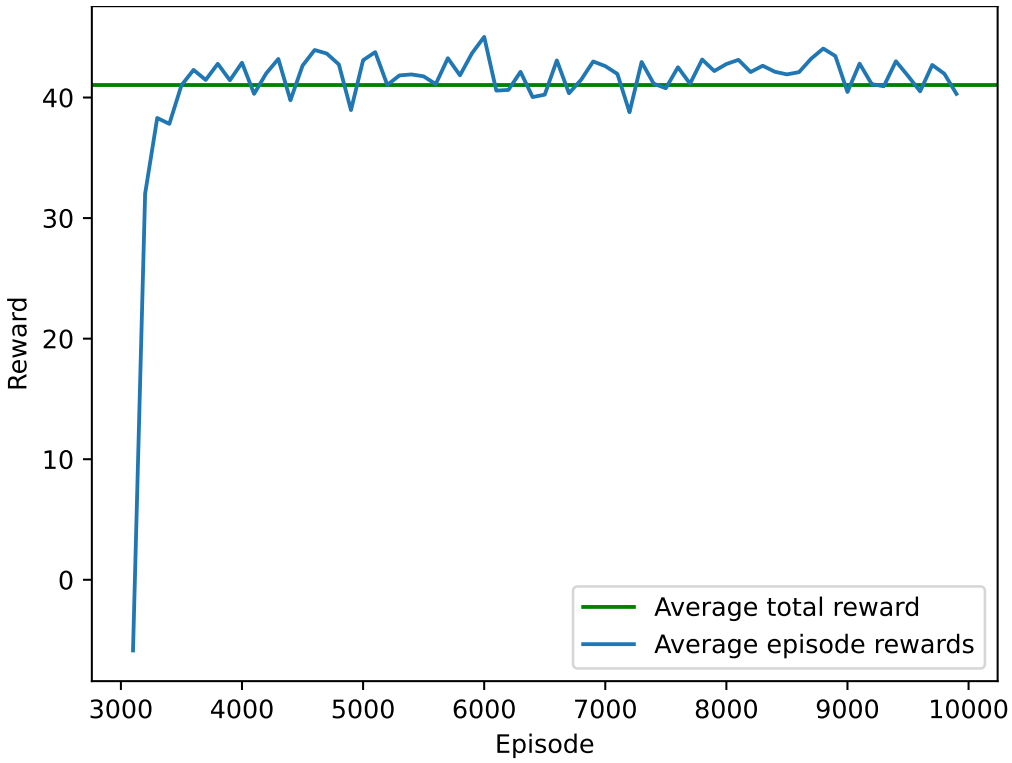


Figure 4.6: The learning curve illustrates the increase in reward for a DDPG agent training to find a collision free path from its starting position towards the goal. This agent is trained with decreased learning rates for both the actor and critic compared to the initial parameters.

to approach the goal again.

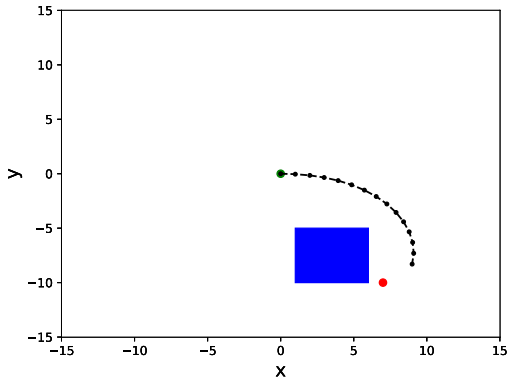
Increased number of hidden layers

The performance of the agent where the networks parameters are increased is shown in this section. From the initial parameters one hidden layer with 200 neurons is added to both the actor and critic network. All other parameters are kept at their initial value, including the learning rates.

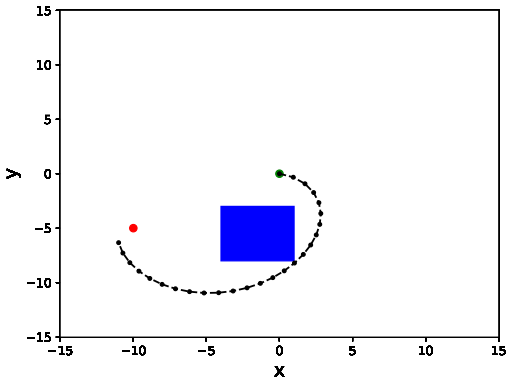
Figure 4.8 show the training process of an agent trained with a higher dimension of both the actor and critic networks, where an additional hidden layer consisting of 200 neurons is added before the output layer. The process of training is done over 10000 episodes, where 3000 of them are used to fill the memory and the remaining 7000 are used to train the agent. From the plot it can be seen that the first 300 episodes of the learning process result in a negative reward between around -15 and -50, however after episode 400 the performance improves and crosses the total average line which can be seen to be slightly lower than 40. After crossing the total average line, the reward stays above the the line.

Examples of testing the agent trained in Figure 4.8 can be seen in Figure 4.9, where Subfigures 4.9a, 4.9b and 4.9c show the agent avoiding the obstacle while also reaching the goal, however in Subfigure 4.9d shows that the agent also does have some collisions.

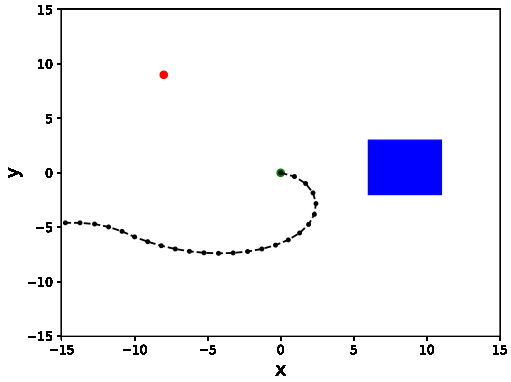
The results of testing the agent trained in Figure 4.8 in 100 cases the goal is reached 78 of the cases, a collision occurs in 7 of the cases and wanders off in 2 of the cases. This corresponds to reaching the goal 90% of the time, collides 8% of the time and wanders off in 2% of the time, when cases where the goal or starting point is covered by an obstacle.



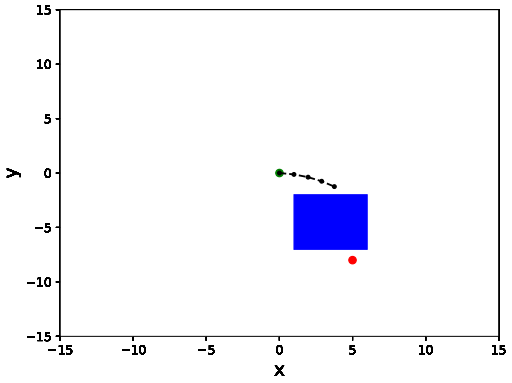
(a) Agent reaches the goal and avoid the obstacle.



(b) Agent reaches the goal and avoid the obstacle.



(c) The agent gets lost moving towards the goal and wanders off.



(d) Collision occurs.

Figure 4.7: The four figures show the agent trained in Figure 4.6 with a decreased learning rate for both the actor network and the critic network.

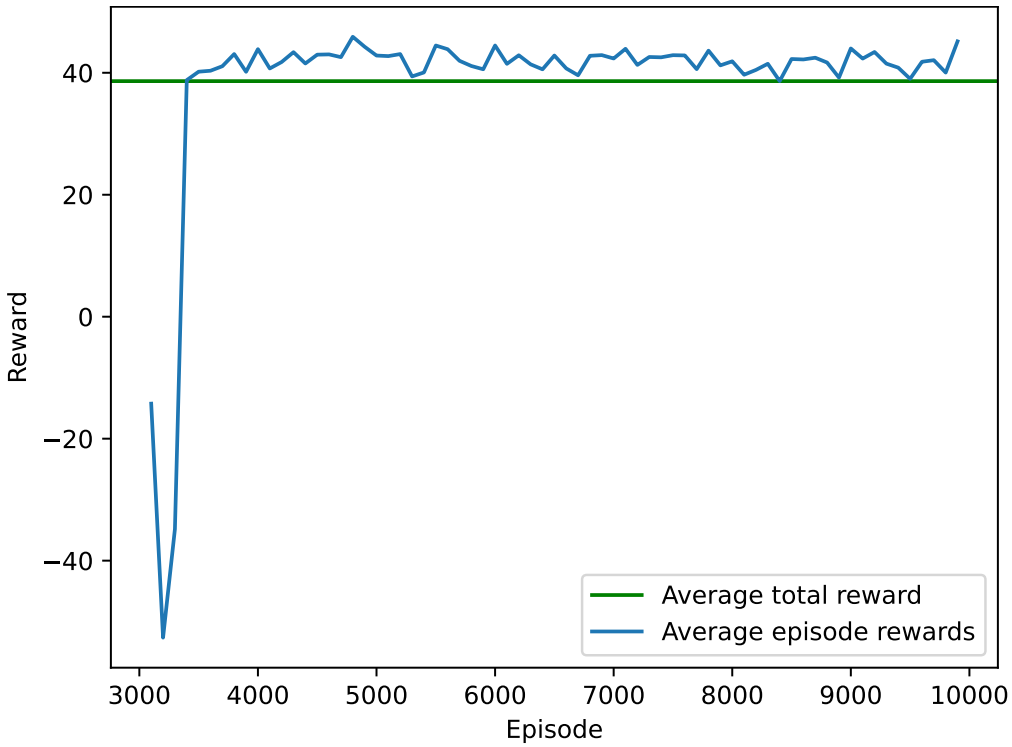
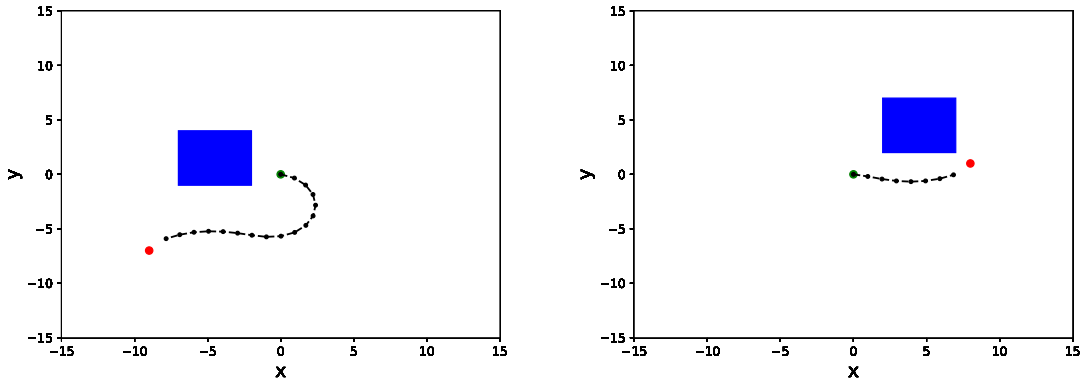
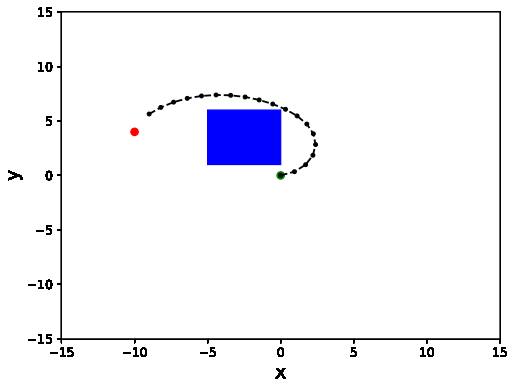


Figure 4.8: The learning curve illustrates the increase in reward for a DDPG agent training to find a collision free path from its starting position towards the goal. The agent is trained with one more hidden layer consisting of 200 nodes compared to the initial parameters.

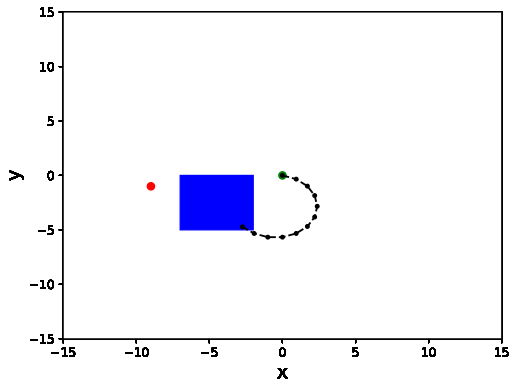


(a) Goal is reached.

(b) Goal is reached.



(c) Goal is reached.



(d) Collision occurs..

Figure 4.9: The four figures show the agent trained in Figure 4.8 with a increased number of hidden layers for both the actor network and the critic network.

Chapter 5

Discussion

The results from the work conducted in this thesis is a reactive motion planner for obstacle avoidance based on the deep reinforcement learning algorithm DDPG. The results from training and testing the implemented method are shown in the previous section, starting with a verification of the method in 1D, which verified the agents abilities to learn. Based on the verification the agent was tested in a 2D environment with obstacles to observe its abilities to reach the goal and perform collision avoidance.

The agents abilities to avoid colliding with obstacles can be observed to depend on hyperparameters and the architecture of the actor and critic neural networks. By adding one more hidden layer consisting of 200 neurons to the network of both the actor and the critic networks the number of times the goal is reached without a collision occurring goes from 45% with the initial parameters to 90% with the increased layer.

The results presented in the simulation experiments show that the proposed method can be a promising tool for in-cage navigation with obstacle avoidance. Combining the desired heading produced by the presented method with for instance desired speed control can be a promising method for utilizing deep reinforcement learning in motion planning and motion control.

5.1 1D environment

Initial parameters

The learning curve from the first test case in the 1D environment show that the agent has strong abilities to learn with the learning curve in Figure 4.1 crossing the total average line after around 400 episodes before showing a promising learning curve. The results from training the agent with the initial parameters 100 times support the learning curve by reaching the goal 100% of the time, however the agent is not able to do it in one step.

Decreased learning rate

When decreasing the learning rates for both the actor and critic networks the learning curve is less steep at the beginning of the training process, which corresponds to Equation 2.16 where the learning rate α determines how much the weights should be updated at each training. The tests show that slowing down the learning process does not improve the results as the agent is not able to reach the goal as many times as when using the initial parameters. This can be due to the slow learning, which results in the weights not being updated enough compared to in the case with initial parameters.

One hidden layer

By using only one hidden layer in the neural networks compared to the two hidden layers used in the initial case the learning process can be seen to be less steep, similar to when the learning rate is decreased. However the performance of the agent trained with one hidden layer from Figure 4.3 is better than when the agent is trained with a decreased learning rate, reaching the goal 100 out of 100 times. This can be

because the training process is not slowed down, as the weights of the neural networks are updated at the same rate as in the initial case using the initial learning rates for both actor and critic. However the learning process displayed in Figure 4.3 does not perform as well with the training data as the learning process in the initial case in Figure 4.1 which can be seen by the learning curves. This could be because the initial case overfit the data, and using one hidden layer is the most appropriate choice of architecture. Both cases do however perform well when tested, and based on those results choosing the architecture consisting of one layer can then reduce the computations having to be made, and thereby potentially also the training time.

5.2 2D environment

5.2.1 With obstacles

In section 4.2.1 the results from training an agent to perform obstacle avoidance are shown. Some important parameters for the agents performance are varied, creating different cases where the performance of the agent can be seen to be affected.

Initial parameters

The learning curve illustrating the agent trained with the initial parameters is shown in Figure 4.1. While the learning curve rapidly converged to cross the average total reward after only 250 episodes, the fluctuation of the curve illustrate that the agent does have some collisions, bringing the average over the 100 episodes down towards 30. Having those fluctuations at the beginning of the training process could benefit the training as it would have examples of collisions that could improve the performance, however in this case the fluctuations can not be seen to decrease over the training period.

The performance when testing the agent is also not ideal, with the agent only being able to reach the goal in 45% of the times. Both the amounts of collisions and times the agent gets lost and wanders off are quite high at respectively 22% and 33%. It is therefore also possible that the fluctuations in the learning curve can be caused by the agent not finding its way towards the goal at all, but rather getting lost.

Decreased learning rate

When the learning rate is decreased from the initial parameters the learning curve in Figure 4.6 can be seen to change slightly from from the learning curve of the initial process in Figure 4.1. By decreasing the learning rates α_{actor} and α_{critic} the learning is slowed down, as the weights of the neural networks are updated less at each time the network is trained. This requires more training episodes to minimize the loss between the output of the network and the target.

The curve in Figure 4.6 is less steep at the beginning of the training process, and uses around 500 episodes to cross the total average line. This corresponds to the decreased learning rates that result in a slower learning process. After crossing the total average line the fluctuation of the reward can be seen to be less prominent during the remaining training episodes, mainly staying above 40 points. Compared to the initial parameters there is a clear improvement in the agents learning process when training with a lower learning rate. Over the last 3000 episodes the performance of the agent also looks like it has some improvement compared to the performance over the previous episodes, indicating that the agent is improving. Based on these results slowing down the learning process can be observed to improve the agents training, indicating that the initial learning rates update the weights too fast, resulting in a sub-optimal solution.

The tests of the agent trained with a decreased learning rate also shows that the performance of the agent has improved as the increase in reaching the goal is 23%, from 45% to 68% compared to the initial parameters. Looking at the learning curve the increase in reaching the goal makes sense as the learning curve show little tendency to collide or wander off as the reward stays close to the highest possible reward during major parts of the training process. The testing of the agent therefore supports that slowing down the learning process can be seen to improve the performance of the agent compared to when the initial parameters are used.

Looking at the motion plots for both the initial parameters in Figure 4.5 and the parameters with decreased learning rate in Figure 4.7 the agent can be seen to choose to turn -20 degrees for the first steps in all of the

cases. However in Subfigure 4.7b the optimal path would be found by choosing to plan over the obstacle rather than under.

Increasing number of hidden layers

The final parameter to be adjusted is the number of hidden layers in the neural network. The training during the first 300 episodes results in a total average reward that is lower than in the cases of initial parameters and lower learning rate. This can be a coincidence as the learning of the agent at this stage is highly influenced by the initial data in the replay buffer which is randomly sampled, and with it being at the beginning of the learning process the agents abilities to choose suitable actions is not good.

Despite the low scores at the beginning of the training process the total average reward is still close to the same as the two other cases, indicating that the agent with one additional layer overall performs slightly better during the rest of the learning process, compensating for the performance during the first 300 episodes.

Looking at the learning curve the training seems to be stable after crossing the total average line, after 400 episodes, compared to the case of the initial parameters. Based on the learning curves adding an additional hidden layer improves the performance of the agent, allowing the neural networks to find more complex patterns in the data compared to when the dimensions of the network are lower. This observation is supported by the results from testing the agent with an increased number of hidden layers as the goal is reached 90% of the times and collides or wanders off only 8% and 2% respectively. This is the best performance presented out of the three cases of parameters presented for obstacle avoidance in this thesis, and for future work within this field it is clear that the dimension of the network should be increased compared to the initial parameters presented.

Chapter 6

Conclusion

6.1 Conclusion

In this thesis the results from a local motion planner based on the deep reinforcement learning algorithm Deep Deterministic Policy Gradient are presented. The planner is based on the multilayer motion planner proposed in the specialization project in Appendix A, where Rapid-exploring Random Tree is suggested as a global planner producing waypoints. Deep reinforcement learning is proposed as a reactive planner with the objective to perform online motion planning between the waypoints and avoiding obstacles based on sensor measurements.

The proposed method, DDPG, is first validated using a 1D case, where the initial parameters proposed in the original DDPG paper show the most promising learning curve. The method does however also perform well when the number of hidden layers is decreased. Both methods are able to reach the goal 100% of the times they are tested, but both use more than one step on average to reach the goals. If the results in 1D are compared to motion planning it is possible to say that the agent in both cases finds a path to the goal, but it is not optimal.

The results of testing the agents abilities as an online planner show that it is able to plan towards a goal and make decisions enabling obstacle avoidance of an obstacle not known before it is detected at some distance. The performance of the agent is observed to largely be affected by the parameters influencing the training process. Both decreasing the learning rates of the actor and critic networks, thus slowing down the training process, and increase the number of hidden layers in the networks, enabling the networks to make more complex mappings, result in improving the performance of the agent. The cases in the 2D environment do however show similarities to the 1D case, as the shortest path to the goal while avoiding obstacles is not always found.

The final case where the additional hidden layer is added to the networks provides the best results, reaching the goal 90% of the 100 times it was tested. This agent can be a promising tool for in-cage navigation with obstacle avoidance.

6.2 Future work

An important aspect of a motion planner is that the planned motion is feasible, being consistent with the dynamics of the vehicle. In this thesis the main objective has been to investigate the potential of deep reinforcement learning as a online motion planner by investigating its abilities to make sequential decisions. In order for the proposed local planner to be applied as a motion planner for a given vehicle a model of the movement of the vehicle is crucial for the planned motion to be feasible. It would therefore be interesting to investigate the performance of the agent when the dynamics of the vehicle are included in the calculations of the movement of the vehicle.

The results show that the agent is able to perform obstacle avoidance of obstacles that first are known when detected by sensors, however the number of obstacles presented in the thesis is static. This is due

to the neural networks introducing a challenge, as a unknown number of obstacles will result in a varying dimension of the observation which results in the input of the neural network not being consisted during training. In this thesis the proposed solution is to have a set number of obstacles possible to detect, and the distance to a discovered obstacle is set whenever the obstacle is within some set distance determined by the sensor range, otherwise the distance is set to be a large number. This would result in a set number of inputs in the neural network, while still enabling a varying number of obstacles to be detected. However this results in a trade off between allowing a large enough number of obstacles to be detected at once while not increasing the complexity of the calculations more than necessary due to a large observation dimension. For future work an interesting approach could therefore be to investigate other solutions to having a varying number of obstacles.

Bibliography

[1] Szilárd Aradi. Survey of deep reinforcement learning for motion planning of autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 2020.

[2] SINTEF. Global ocean governance, . URL https://www.sintef.no/en/sintef-research-areas/global_ocean_governance/. (accessed: 01.11.2022).

[3] SINTEF. Sintef ocean, . URL https://www.sintef.no/ocean/. (accessed: 01.11.2022).

[4] SINTEF. Havbruksrobotikk og autonome løsninger, . URL https://www.sintef.no/ekspertise/ocean/havbruksrobotikk-og-autonome-losninger/. (accessed: 01.11.2022).

[5] SINTEF. Race fish-machine interaction, . URL https://www.sintef.no/en/projects/2020/race-fish-machine-interaction/. (accessed: 02.11.2022).

[6] SINTEF. Change - an underwater robotics concept for dynamically changing environments, . URL https://www.sintef.no/en/projects/2021/change-an-underwater-robotics-concept-for-dynamically-changing-environments/. (accessed: 02.11.2022).

[7] Hans Vanhauwaert Bjelland. Aquaculture technology. URL https://www.sintef.no/en/sintef-research-areas/aquaculture_technology/. (accessed: 04.10.2022).

[8] Martin Føre, Kevin Frank, Tomas Norton, Eirik Svendsen, Jo Arve Alfredsen, Tim Dempster, Harkaitz Eguiraun, Win Watson, Annette Stahl, Leif Magne Sunde, et al. Precision fish farming: A new framework to improve production in aquaculture. *biosystems engineering*, 173:176–193, 2018.

[9] Jingzhe Jin. Offshore fish farm. URL https://www.sintef.no/globalassets/sintef-ocean/factsheets/offshore-fishfarm.pdf. (accessed: 04.10.2022).

[10] SINTEF. Netclean 24/7, . URL https://www.sintef.no/prosjekter/2019/netclean-247/. (accessed: 19.12.2022).

[11] Jingzhe Jin, Biao Su, Rui Dou, Chenyu Luan, Lin Li, Ivar Nygaard, Nuno Fonseca, and Zhen Gao. Numerical modelling of hydrodynamic responses of ocean farm 1 in waves and current and validation against model test measurements. *Marine Structures*, 78:103017, 2021.

[12] Trude Vollheim. Havbruksnæringen må ivareta arbeidstakernes sikkerhet og helse. URL https://arbeidstilsynet.no/nyheter/havbruksnaringen-ma-ivareta-arbeidstakernes-sikkerhet-og-helse/. (accessed: 04.10.2022).

[13] Edgar McGuinness, Halvard L Aasjord, Ingrid B Utne, and Ingunn Marie Holmen. Fatalities in the norwegian fishing fleet 1990–2011. *Safety science*, 57:335–351, 2013.

[14] D Richard Blidberg. The development of autonomous underwater vehicles (auv); a brief summary. In *Ieee Icra*, volume 4, pages 1–12, 2001.

[15] Thor I. Fossen. *HANDBOOK OF MARINE CRAFT HYDRODYNAMICS AND MOTION CONTROL*. John Wiley Sons, 2021.

[16] Yong K Hwang and Narendra Ahuja. Gross motion planning—a survey. *ACM Computing Surveys (CSUR)*, 24(3):219–291, 1992.

[17] Voemir Kunchev, Lakhmi Jain, Vladimir Ivancevic, and Anthony Finn. Path planning and obstacle avoidance for autonomous mobile robots: A review. In *International Conference on Knowledge-Based and Intelligent Information and Engineering Systems*, pages 537–544. Springer, 2006.

[18] Chad Goerzen, Zhaodan Kong, and Bernard Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57(1):65–100, 2010.

[19] Sverre Fjæra. Optimal and adaptive path planning and following for permanent resident cleaning robot operating in fish farms. Master's thesis, NTNU, 2021.

[20] Sean Quinlan and Oussama Khatib. Elastic bands: Connecting path planning and control. In *[1993] Proceedings IEEE International Conference on Robotics and Automation*, pages 802–807. IEEE, 1993.

[21] Panagiotis G Zavlangas, Spyros G Tzafestas, and Kasper Althoefer. Fuzzy obstacle avoidance and navigation for omnidirectional mobile robots. In *European Symposium on Intelligent Techniques, Aachen, Germany*, pages 375–382, 2000.

[22] Dongxu Zhou, Ruiqing Jia, Haifeng Yao, and Mingzuo Xie. Robotic arm motion planning based on residual reinforcement learning. In *2021 13th International Conference on Computer and Automation Engineering (ICCAE)*, pages 89–94. IEEE, 2021.

[23] Fei Ye, Shen Zhang, Pin Wang, and Ching-Yao Chan. A survey of deep reinforcement learning algorithms for motion planning and control of autonomous vehicles. In *2021 IEEE Intelligent Vehicles Symposium (IV)*, pages 1073–1080. IEEE, 2021.

[24] Xi Xiong, Jianqiang Wang, Fang Zhang, and Keqiang Li. Combining deep reinforcement learning and safety based control for autonomous driving. *arXiv preprint arXiv:1612.00147*, 2016.

[25] Milton Vicente Calderón Coy and Esperanza Camargo Casallas. Training neural networks using reinforcement learning to reactive path planning. *Journal of Applied Engineering Science*, 19(1):48–56, 2021.

[26] Richard Bellman. On the theory of dynamic programming. *Proceedings of the national Academy of Sciences*, 38(8):716–719, 1952.

[27] Zichen He, Jiawei Wang, and Chunwei Song. A review of mobile robot motion planning methods: from classical motion planning workflows to reinforcement learning-based architectures. *arXiv preprint arXiv:2108.13619*, 2021.

[28] Chunxi Cheng, Qixin Sha, Bo He, and Guangliang Li. Path planning and obstacle avoidance for auv: A review. *Ocean Engineering*, 235:109355, 2021.

[29] Sahisnu Mazumder, Bing Liu, Shuai Wang, Yingxuan Zhu, Lifeng Liu, and Jian Li. Action permissibility in deep reinforcement learning and application to autonomous driving. *KDD'18 Deep Learning Day*, 2018.

[30] Peter Norvig Stuart Russell. *Artificial Intelligence A Modern Approach*. PearsonEducation, 2010.

[31] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[32] Marco A Wiering and Martijn Van Otterlo. Reinforcement learning. *Adaptation, learning, and optimization*, 12(3):729, 2012.

[33] Melanie Coggan. Exploration and exploitation in reinforcement learning. *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University*, 2004.

[34] Satinder Singh, Tommi Jaakkola, Michael L Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine learning*, 38(3):287–308, 2000.

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

[36] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. Deep reinforcement learning: A brief survey. *IEEE Signal Processing Magazine*, 34(6):26–38, 2017.

[37] Yuxi Li. Deep reinforcement learning: An overview. *arXiv preprint arXiv:1701.07274*, 2017.

[38] Chris M Bishop. Neural networks and their applications. *Review of scientific instruments*, 65(6):1803–1832, 1994.

[39] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[40] Sagar Sharma, Simone Sharma, and Anidhya Athaiya. Activation functions in neural networks. *towards data science*, 6(12):310–316, 2017.

[41] Matías Roodschild, Jorge Gotay Sardiñas, and Adrián Will. A new approach for the vanishing gradient problem on sigmoid activation. *Progress in Artificial Intelligence*, 9(4):351–360, 2020.

[42] Farnaz Nazari and Wei Yan. Convolutional versus dense neural networks: Comparing the two neural networks performance in predicting building operational energy use based on the building shape. *arXiv preprint arXiv:2108.12929*, 2021.

[43] Larry R Medsker and LC Jain. Recurrent neural networks. *Design and Applications*, 5:64–67, 2001.

[44] George Bebis and Michael Georgiopoulos. Feed-forward neural networks. *IEEE Potentials*, 13(4): 27–31, 1994.

[45] Bo Pang, Erik Nijkamp, and Ying Nian Wu. Deep learning with tensorflow: A review. *Journal of Educational and Behavioral Statistics*, 45(2):227–248, 2020.

[46] Antonio Gulli and Sujit Pal. *Deep learning with Keras*. Packt Publishing Ltd, 2017.

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

[48] Ekaba Bisong. Matplotlib and seaborn. In *Building machine learning and deep learning models on google cloud platform*, pages 151–165. Springer, 2019.

[49] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[50] Romain Pepy, Alain Lambert, and Hugues Mounier. Path planning using a dynamic vehicle model. In *2006 2nd International Conference on Information & Communication Technologies*, volume 1, pages 781–786. IEEE, 2006.

Appendix A

Specialization project

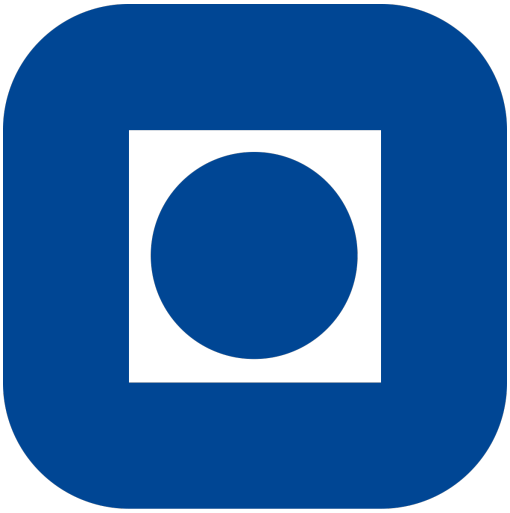
Norges teknisk-naturvitenskapelige universitet

TTK4550

Anne Marie Mathisen

Prosjektoppgave

VÅR 2022



NTNU

Abstract

Autonomous Underwater Vehicles (AUV) are already an important part of underwater operations today by assisting in environments where it is hard, or even impossible for humans to perform operations alone. A potential field to incorporate autonomous technology is fish farms, where operations like cleaning and investigation of the net are important procedures an AUV could be used as assistance. If fish farms are moved further offshore they also have to become more independent from human personnel.

In order to perform autonomous operations an AUV relies on a motion control system with a motion planner that can handle the dynamic environment of a fish cage. The goal of the project has been to investigate methods towards robust motion planning for an AUV in order to enable autonomous operations in fish farms. The project investigates a multilayer motion planning framework that utilizes Rapidly-exploring Random Trees in order to produce a global path consisting of waypoints before the operation, and the deep reinforcement learning technique Deep Deterministic Policy Gradient (DDPG) as a reactive local planner. For the shake of simplicity, a 2D environment was considered, thus it is assumed that the robot maintains depth.

The main contributions to the work consist of:

- RRT is proposed as a global planner in order to produce a global path consisting of waypoints before the operation. The global planner should plan before the operation, and the path should be based on a map of the environment with information on static obstacles known before the operation.
- The interaction between the deep reinforcement learning agent and the environment is modeled in order to enable observation of obstacles, detecting collisions and take a step towards making the agent aware of its surroundings.
- Experimental testing in order to observe the performance of the methods mentioned above.

The project concludes with discussions of the obtained results. The RRT is able to successfully find a path from the start to the goal configuration while avoiding the obstacles in the environment, resulting in a series of waypoints that can be used to bias a local planner. The results from implementing the robots interaction with the environment creates a foundation for the decision making reinforcement learning agent to be aware of its surroundings, and is a step towards fast obstacle avoidance of dynamic obstacles. Finally, the next steps are discussed in a section on future work, where the reward mechanism used for the deep reinforcement learning agent is mentioned as a promising area to investigate further.

Contents

List of Figures

List of Tables

1 Introduction

Nowadays, Autonomous Underwater Vehicles (AUV) are used in many subsea applications such as exploring and mapping the seafloor and ocean. They also have potential to perform operations assisting humans in high risk environments. As salmon farm sites are moved further offshore in locations exposed to more challenging environmental conditions there is an increased need for replacing human personnel with technology.

SINTEF Ocean projects such as "CHANGE" [1], An underwater robotics concept focusing on dynamically changing environments and "RACE Fish, Machine Interaction" [2] address challenges of using AUVs in fish farms. The goal is to minimize the impact the autonomous operations has on the fish. While motion control strategies in static environments exist, they do not handle operations in dynamic environments, an important aspect in order to impact the fish less and ensure safe operation during cage deformations. To avoid collisions, both in order to protect the fish and avoid expensive collisions, it is essential to provide strategies that can robustly perform operations in such challenging dynamic environments.

1.1 Motivation

The motivation behind this project is to contribute to a motion control system for an AUV performing operations in a fish cage. For the AUV to be fully autonomous it requires a motion planner to provide the motion controller with a plan on where it should go. The motion planner should therefore be able to handle the challenging dynamic environment of a fish cage, providing motions for efficiently moving towards the goal and safely avoiding collisions.

1.2 Scope

Previous specialization projects and master theses have focused on using conventional motion planning methods to develop an adaptive motion planner that is able to handle the dynamic environments of a fish cage. However, the interest of exploring deep reinforcement learning (DRL) for motion planning is increasing due to its abilities of solving complex tasks from high dimensional sensory input. It is therefore interesting to investigate the performance of a deep reinforcement learning based motion planner.

This specialization project explores motion planning through first performing a literature study on relevant work. Based on the literature study a multilayer motion planner using Rapidly-exploring Random Trees to produce waypoints along the edge of the net, and a DRL reactive local planner to perform obstacle avoidance between the waypoints is investigated. The goal is for this project to be a step towards implementing a motion planner for an an AUV conducting autonomous operations in a dynamic environment.

1.3 Structure

The project presented in this report is structured into six main chapters, including the introduction. In Chapter 2 relevant theory is presented with a focus on reinforcement learning. Chapter 3 presents how the main aspects of the implementation is done, in addition to structuring the simulation setup. Chapter 4 then presents the results using the structure from the previous chapter. Chapter 5 discusses the results and presents the next steps in a Future Work section. Finally a conclusion of the project is done in Chapter 6.

2 Theory

This chapters main focus is to introduce the concept of motion control and how it uses motion planning in order to allow for a robot to operate autonomously. In the following sections the modules of a motion control system will be explained to illustrate the role of a motion planner, and to provide an overview on how it interacts with the other components of a motion control system. Figure 1 presents a high level system description of the components of the motion control system and their interaction.

First the concept of motion planning is introduced, which is the responsible for the motion control of the robot utilizing the output. Secondly in depth information on motion planning in provided, which is the main focus of the project. The basic concepts of different types of motion planners are presented, along with advantages, challenges they are facing.

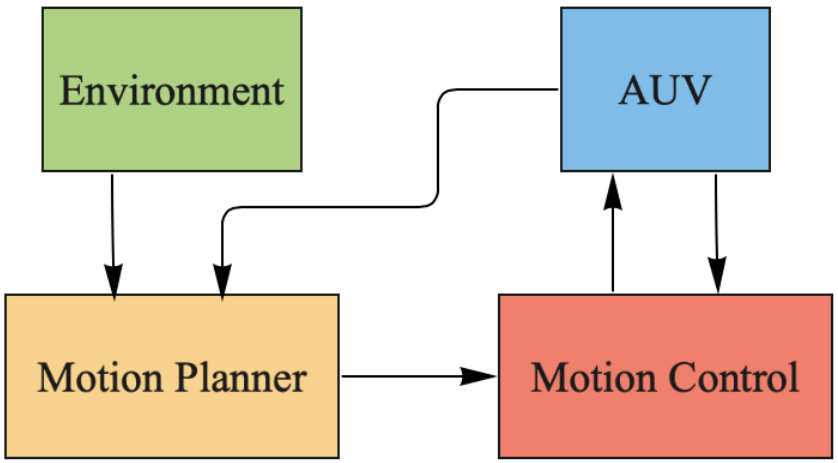


Figure 1: A high level system description of the different parts working together to control the motion of a ROV.

2.1 Motion planning and motion control of AUV

The motion planner and motion controller are two of the main modules in most motion control systems. The output is decided in the motion control block, but it does depend on the other modules in order to know what the robots current state is and which state it wants to be in in order to reach its goal. The latter is decided by the motion planner.

2.1.1 Motion Control

The goal of a motion control system is to compute the necessary control forces and moments in order for the AUV to reach a certain control objective [3]. The control objective is often decided by the motion planner, and could be a path or trajectory representing the motion a robot should follow. In addition it can include more advanced features like collision avoidance, fuel optimization or minimizing time spent.

2.1.2 Motion planning

Motion planners are often categorized by whether they are dependent on robot constraints and time. A trajectory should represent a reference signal as a function of time, such that when following the trajectory the robots position and velocity should track the time varying position and reference signals of the trajectory. [3] Path planning is on the other hand independent of time, where the planed path is described as a series of coordinates or waypoints that the robot should reach. [3]

2.1.3 Multilayer motion planning

There are different types of motion planners that each solve different levels of the motion planning. Often a map representing the environment is used, but the map can be inaccurate or previously unknown obstacles can be encountered during execution. By using sensors the robot can discover obstacles that were not known prior to the operation. AUVs will often operate in challenging and dynamic environments, and do therefore depend on the sensor measurements in order to perform online planning while avoiding all obstacles. Online planning is when the path is replanned during execution, and it must be applied in order to create a collision free path in dynamic environments. [4]

Global motion planning Global motion planners generate a solution from start to goal, often described by waypoints. [5] This is done based on information given by a topological map where static obstacles and the structure of the environment is described. Global planners are often discrete path searchers (DPS), who search the state space to find a discrete path from the start position to the goal [5], representing the result as a discrete number of points.

Discrete path searchers can be divided into two categories. Graph-search based algorithms are mainly used in low dimensional spaces because the algorithm uses a full modelling process of the environment. [5] As a result it is a computationally expensive process in environments of higher dimensions. Sampling-based algorithms search the environment until the goal is reached, or until it reaches a time limit. Sampling based planners are more efficient than graph-search based planners in environments of a higher dimension [5], as they do not need to explore all of the environment. They also tend to guarantee the finding of a solution if they are not limited by time, while the graph based solution is can be unable to find a solution in an infinite amount of time.

Probabilistic Road Maps (PRM) and Rapidly-exploring Random Trees (RRT) are two fundamental sampling based algorithms [5]. By sampling points in the obstacle free subset of the configuration space, which is the space of all possible poses for the robot, a graph is constructed. The points are then connected where a collision free path occurs to get a path from start to goal [6]. The algorithms both assume that a map of the environment is available and that there is a method of finding the distance between the points. The main difference between the two algorithms is that RRT is limited to single query applications, while PRM is for multi query applications. In other words, PRM generates a road map, a graph in the free configuration space, which then can be reused multiple times in order to solve different motion planning queries. Whereas the graph build by RRT has to be rebuilt for each application.

When unknown obstacles are met during execution the planing has to be redone in order to avoid a collision. If a global planner was used alone it would have to recalculate the entire path which would result in a high computational cost, not suitable for online planing. [4] A local motion planner can therefore be used together with the global planner in order to efficiently do online planning.

Local motion planning The local planner is responsible for planning smaller sections of the path created by the global planner. The global planner creates a path consisting of waypoints and the local planner performs point-to-point planning towards the next waypoint. Based on additional information about the environment local planners can make fast specific strategies for avoiding previously unknown obstacles and environmental conditions [5].

Reactive planning A motion planning concept closely related to local planning is reactive planning, where only local knowledge about nearby obstacles is used to perform the planning. This knowledge is often obtained by sensors, resulting in reactive algorithms to plan fast as obstacles only are known to the robot within the radius of the sensor [7]. Reactive planing algorithms are used for planning based in local areas. Much like local planning they cannot be used on their own to find a solution, as it would not lead to an optimal solution, or any solution at all due to getting stuck in a local minima.

Standard motion planning frameworks consists of a multilevel structure where global and local motion planners are combined. [5] The global planner will efficiently create a path to bias the local planner from start to goal, and the local planner will perform the online planing, avoiding the running of the global planner each time a previously unknown obstacle is encountered.

One example of reactive navigation is Braitenberg vehicles, who use sensory input connected with the motor of the vehicle in order to perform an action based on the sensor measurements without an internal representation of the environment [8]. An example of Braitenberg vehicles are robot lawn mowers, who run straight forward until an obstacle is sensed, and then turn by some predefined angle in order to avoid the obstacle, before continuing straight forward in the new direction [9]. The challenge of Braitenberg vehicles is when they are used in larger areas with vehicles that have a limited amount of fuel, like an AUV in a fish cage. The AUV should be able to cover as much as possible of the net if it is going to clean or inspect it, and Braitenberg vehicles are prone to overlapping previously visited areas.

An other approach based in using reactive local planning together with a global initial path is presented in [9]. The reactive algorithm is the Elastic Band Method and it is used to create an adaptive path in a fish cage. The algorithm is based on using a series of bubbles in order to represent an elastic band. A simple environment can be represented by few larger bubbles, but for increasingly more complex environments the number of bubbles increase and their radius decrease. The approach in [9] resulted in successfully planning a path in a dynamic environment. An advantage of the method is its improvement over time, which is done by incrementally optimizing the path. Large environmental changes could however result in problems in finding a path even if one exists. [9] An other challenge of using the elastic band is that for each time the path is optimized the entire path is optimized. Doing this does result in a path that is close to optimal, but it also requires more time to perform the calculations as the path grows.

[10] proposes a motion planner for an Agile Autonomous Underwater Vehicle using a RRT-based approach while also demonstrating Trajopt for online planning. The method proposes approaches for performing obstacle avoidance both using offline and online planning. The results showed that the proposed approach was able to successfully plan a collision free trajectory in a cluttered space, however it is pointed out that the deployment of the proposed framework online high- lighted some computational challenges.

Reinforcement learning, and specifically deep reinforcement learning has been a popular motion planning method lately due to its abilities of solving complex tasks from raw high dimensional sensory input by learning low state characteristics from high dimensional states [11], which makes it suitable for reactive motion planning. By not relying on a prior map, reinforcement learning makes it possible to plan in situations where a detailed map is hard to obtain [5]. It has the ability to learn a mapping from noisy sensor data to actions and after a optimal mapping between state and action is found during training, the time spent on choosing the next action is limited to running the mapping function. This makes reinforcement learning an interesting approach for performing online motion planning, as it might be able to improve the computational time from the previous work mentioned.

A challenge of using reinforcement learning for motion planning is that sparse rewards makes the agent hard to train [12]. This will be explained in greater detail later, but essentially reinforcement learning, like several local planners is vulnerable to getting lost in a local minima if they don't have some global points, or sub-goals to bias their direction. An interesting approach could therefore be to investigate the potential of reinforcement learning as a reactive planner, and later bias the direction towards the goal using RRT as a global path planner.

2.2 Global motion planning: Rapidly-exploring Random trees

Rapidly-exploring Random Trees (RRT) builds a tree structure, where the nodes of the tree represent the state of the robot and the edges represent the transition between the states [13].

The RRT algorithm is presented in Algorithm 1 [14]. It starts at an initial node, and expands by sampling a random state of the robot in the obstacle free configuration space. Then the nearest node, q_{near}, to the random state is chosen, which in the first iteration will be the initial node. Then a new node is created as an extension towards the randomly sampled state. If the random sampled state is less than a given step size away, the node will be placed there. If not it will be placed in the direction of the random sample with the step size as the distance from q_{near}. This is repeated until the goal is reached by placing a node within a given radius of the goal. When the goal is reached, the path can be found by connecting the nearest nodes of the nodes until the initial node is reached [5].

Algorithm 1 Rapidly-exploring Random Tree

Input: Approved goal distance ϵ
$T.add(q_{start})$
while not terminate **do**
    $q_{rand} = randomnode$
    $q_{near} = nearest$ neighbor in T to q_{rand}
    q_{new} = extend q_{near} towards q_{rand}
    if q_{new} can connect to q_{near} **then**
        T.connect q_{near}, q_{new}
    end if
    if Distance to goal $< \epsilon$ **then**
        break
    end if
end while

When RRT is used as a global planner it avoids known obstacles by detecting whether a node is sampled where an obstacle is located. If that is the case the node will be discarded, and a new sample is drawn.

2.3 Local motion planning: Reinforcement Learning

The idea behind learning through interaction with the environment is not new. Humans and animals do it all the time. In order to change the state you are in, you perform an action. Then, by observing the outcome of the action you learn whether the action got you closer to where you want to be or not. In a similar situation the learner will have more knowledge on what to do through previous experiences.

In the next sections the concept of reinforcement learning will be explained in order to introduce key terms and give a basic understanding of the method. Next the combination of deep learning and reinforcement learning is introduced which is called deep reinforcement learning. The final sections will go into depth on a deep reinforcement learning learning technique that can handle continuous action spaces.

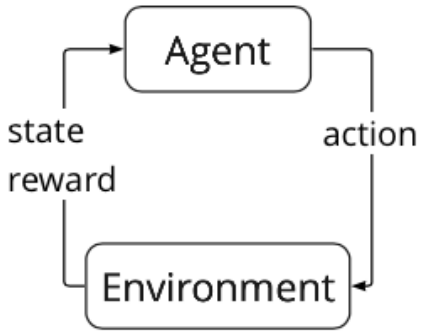


Figure 2: The interaction between the agent and the environment is done by the agent choosing an action based on its state in the environment, then executing the action in the environment and then retrieving the resulting reward and the new state.

2.3.1 The main components of reinforcement learning

Agent and environment Reinforcement Learning is a technique where an agent learns through trial and error. Depending on the outcome of the action chosen by the agent, the agent is rewarded or punished. The agent operates in an environment, and the agents state in the environment changes when an action is made. The interaction between the environment and the agent is described in Figure 2.

Reward The state of the environment after the action will result in a reward, and the goal of the agent is to maximize the total reward. [15] The discounted return is the sum of the rewards obtained by the agent, but by each time step the reward is discounted, such that the rewards furthest into the future are discounted more than the reward obtained at the current time step. [15] In other words, the discount factor γ determines how much the agent cares about the rewards in distant future compared to now. The equation for the total discounted reward is seen in Equation 1.

$$R = \sum_{t=0}^{N} \gamma^t r_t \tag{1}$$

In Equation 1 R is the total discounted reward after N steps, where the reward at each step r_t is discounted by the discount factor γ. The discount factor is a number between 0 and 1, and its dependence of time results in that step rewards further in time will have less impact on the total reward. The discounted step rewards are then summarized to the total discounted reward R.

It is important to reward the agent in a way such that it understands what the goal is. An example is a car driving to reach a target. Is the goal to get there as soon as possible? Should it make some specific stops along the way? The car has to stay within the speed limit, and also keep the passengers safe. The reward must be modeled in a way that reflects how the agent should behave.

A challenge when using Reinforcement Learning for motion planning is sparse rewards [6]. The agents actions depend on getting feedback in the shape of rewards, letting the agent know if its actions makes it approach the goal. For the robot to find its way to the goal the agent therefore has to gain information along the way that illustrate if it is getting closer or not. Without any information on whether it is approaching the goal the robot can get stuck in a local minima [6]. A solution to this is to create sub-goals towards the goal, where the agent is rewarded when the following sub-goal is reached.

States and observations The state, s_t, represents the world of the agent at a given time. It contains all information on the world, whereas the observation is a partial description of the state in the sense that the agent might not be able to observe the whole state [15].

Action space The action space defines what actions the agent is able to make in the given environment, it is the set of all possible actions [15]. In some environments the action space is discrete with a finite number of possible actions. Continuous action spaces on the other hand have continuous set of possible actions. The distinction between discrete and continuous action spaces is important in reinforcement learning, since deep reinforcement learning methods often only can handle one or the other [15].

Policy The agents main challenge is detecting what action it should take, it is not given, but must be learned. The policy is a mapping from state to action [16]. In Deep Reinforcement Learning, Neural Networks are used to determine the optimal policy. Then the policy is parameterized by θ, and the problem is based on finding the parameter values resulting in the best choice of action given the state.

The main goal of Reinforcement Learning is to select the policy that results in a maximized reward when the agent follows it. [17] The policy can be either stochastic or deterministic. A stochastic policy is represented as a distribution over actions, and is denoted by π. For each state there will be a probability distribution for every action. A deterministic policy on the other hand has one given action for a state, $a = \mu(s)$ [15].

MDPs and POMDPs Reinforcement learning is based on Markov Decision Processes (MDP). MDPs are sequential decision problems, and they can be represented as a tuple $[S, A, P, R]$. S is a set of states, A is a set of actions, $P : SxA- > S$ describes the transition dynamics when applying an action when in a state, and $R : SxAxS- > \mathbb{R}$ is the reward. [16] The state in a MDP is fully observable, meaning that the agent has all the information worth knowing to determine the action, the information described by the state.

When the agent is not able to observe the full state of the environment it results in a Partial Observable Markov Decision Process, POMDP.[16] In addition to a MDP a POMDP will contain information on the observation of the current state made by the agent, and the probability for the observation given the state and action.[16] The real world is a POMDP, and when applying Reinforcement Learning to a real world problem this must be taken into account.[18] An example is using noisy sensor measurements for the agent to determine its position instead of assuming that the position is known.

Equation 1 is often referred to as the *utility* of a given sequence of states. To find the best policy, different utilities can be compared by looking at the expected utility when using a policy.[18]

$$U^{\pi}(s) = E[\sum_{t=0}^{N} \gamma^t r_t] \tag{2}$$

Equation 2 show that the expected utility U, when a policy π is used, is the expected sum of the discounted rewards. In Equation 3 the policy with the highest expected utility is said to be the optimal policy, which is the policy the agent should use to map from state to action.

$$\pi_s^* = arg\max_{\pi} U^{\pi}(s) \tag{3}$$

2.3.2 Deep Reinforcement Learning

Deep Reinforcement Learning combines the decision making of Reinforcement Learning with deep learning's ability to approximate functions by using neural networks[11]. The advantage of deep reinforcement learning is that by using neural networks, low-dimensional characteristics can be learned from large state and action spaces through iterative interaction with the environment, which is a limiting factor for reinforcement learning[11].

Deep Learning and Neural Networks Neural Networks are used to approximate functions by simulating the structure of biological neurons in the human brain.[19] The network consists of an input layer, a number of hidden layers and an output layer, illustrated in Figure 3. When the input is processed through the hidden layers it will be affected by an activation function g and weights $w_{i,j}$.[18] Each unit computes a weighted sum of its n inputs.

$$in_j = \sum_{i=0}^{n} w_{i,j} a_i \tag{4}$$

In Equation 4 in_j is the input of the neurons in layer j. It is given by the sum of the weighted outputs, a_i from the previous layer. The output from layer j, j is then found by applying an activation function g to the input. This is shown in Equation 5.

$$a_j = g(in_j) = g(\sum_{i=0}^{n} w_{i,j} a_i) \tag{5}$$

The number of layers and the number of nodes in each layer is called the architecture of the neural network. Figure 2.3.2 illustrates a network with one input layer with dim = 4, two hidden layers with dim = 4 and one output layer with dim = 2. The figure illustrates that the input of each node is the sum of the output of the nodes in the previous layer, multiplied with weights along each edge.

Two important hyperparameters for a neural network are the number of layers in the network and the number of nodes in each layer. Compared to supervised learning, not much research has been done to investigate design of the architecture of neural networks used in reinforcement learning[16], but it is common to use two or tree hidden layer feedforward multilayer perceptrons (MLP)[16]. Configuring these parameters should be done through systematic experimentation by choosing some initial parameters for an initial training of the network. Then training the network again with different parameters to see how the performance is affected, and repeating this for a few parameters in order to decide which parameters result in the best performance.

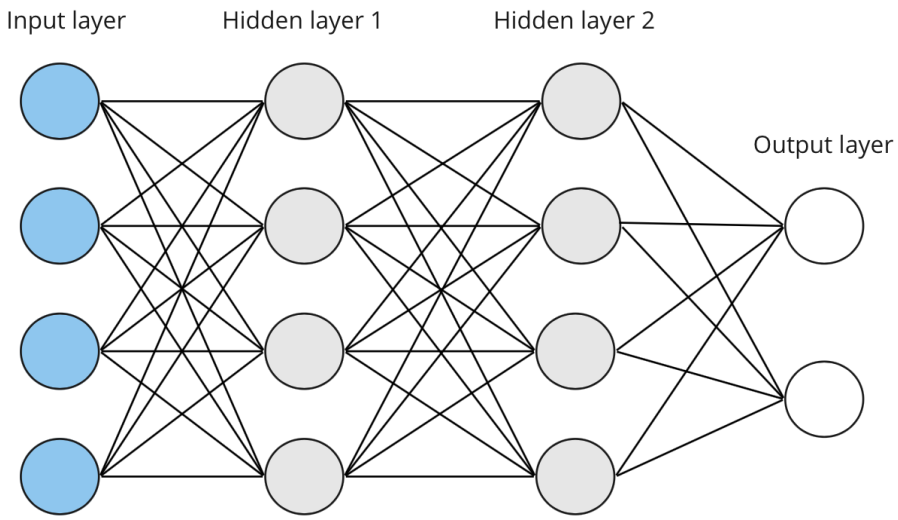


Figure 3: A neural network with an input layer with dim = 4 two hidden layers that each have a dim = 4 and an output layer with dim = 2.

2.4 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is a deep reinforcement learning actor-critic algorithm based on deterministic policy gradient that can operate in continuous action spaces [17]. The policy found using DDPG is deterministic, given a state it will always choose the same action [17]. In Actor Critic algorithms two methods of obtaining the optimal policy are combined [20]. In policy-based methods the policy is found by directly manipulating the policy, while in value-based methods the policy is found indirectly through finding the optimal action-value function.

With DDPG being a deep reinforcement learning algorithm, neural networks are used as function approximators in order to find the policy and the action-value function [17]. The network used to find the policy is often called the Actor being the one choosing the actions and the network used to approximate the action-value function often is called the Critic being responsible for giving feedback, or criticising the policy. The interaction between the two networks is shown in Figure 4. This interaction will be discussed in greater detail in the following sections.

$$a^*(s) = \mu^*(s) = arg\max_a Q^*(s,a) \tag{6}$$

The relationship between the optimal policy $\mu^*(s)$ and the optimal action-value function $Q^*(s,a)$ is shown in Equation 6. The optimal action will be the action that maximizes the action-value function [20] due to the policy obtained from DDPG being deterministic. The following sections will explain the DDPG algorithm: how the optimal action-value function can be found and how it results in finding the optimal policy.

2.4.1 Value-based

In value-based reinforcement learning the policy is found indirectly by finding the optimal action-value function $Q_{\theta^Q}(s,a)$ [20]. The action-value function estimates how good it is to perform an action mapped by the deterministic policy μ from the current state s, and then acting on that policy for all future time steps. An action-value is the return of an action-value function for a given state-action pair, and describes how good it is to choose a specific action while in a given state. This will indirectly give a value indicating how good the mapping from state to action is, in other words, how good the policy is. In DDPG one of the main concepts is to feed the action-value from an action chosen by some policy into the update of the policy. In this way the action-value function gives feedback to the update of the policy on how it is performing.

The optimal action-value function is the function that results in the highest possible action-value for a

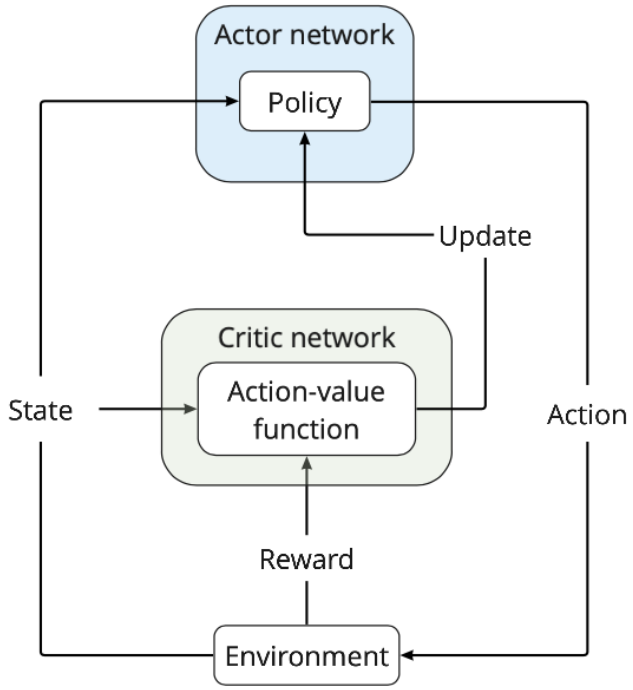


Figure 4: The interaction between the Actor and Critic networks depend on the basic reinforcement learning concepts introduced previously, and the interaction between the two networks in order to obtain the optimal policy.

given state-action pair [20]. From the optimal function the optimal action can be extracted by choosing the action that gives the maximum action-value for for state s. This is shown in Equation 6, where μ^* is the optimal policy, and $Q^*(s,a)$ is the optimal action-value function.

A target is made in order to find the optimal action-value function, and the goal is to minimize the error between the current function and the target, described by the Bellman equation [17].

$$Q^*(s,a) = E[r(s,a) + \gamma \max_{a'} Q^*(s',a')] \tag{7}$$

Equation 7 shows that the target function is the expected value of the reward r at the current time step t summarized with the discounted future reward that is calculated using the optimal value function, where discounted rewards have been discussed in a previous section.

The current action-value function, $Q_{\theta^Q}(s,a)$, is an approximator to the optimal action-value function [17]. The approximator is often a neural network, where the parameters of the network θ^Q describe the action-value function, or the Critic as it is commonly called. The goal is for the parameters to be adjusted in order to minimize the error between the approximator and the optimal action-value function. [17] This is described by the mean-squared Bellman error (MSBE), illustrating how close the parameters of the approximators are to satisfy the target.

$$L(\theta^Q) = E[(Q_{\theta^Q}(s,a)) - (r(s,a) + \gamma(1-d) \max_{a'} Q_{\theta^\mu}(s',a'))^2] \tag{8}$$

The MSBE in Equation 8 is the difference between the approximator $Q_{\theta^Q}(s,a)$, and the target shown in Equation 7. In addition (1-d), where d equals done, is added to the target in order to illustrate that if the next state results in done, the action-value function shows that there are no additional rewards after [20].

Off-policy learning and replay buffers One of the main challenges of using neural networks in reinforcement learning is that the optimization algorithm assumes that the samples used when finding the approximator $Q_{\theta^Q}(s,a)$ and the target are identically and independently distributed [17]. This is

hard to obtain if the samples come from on-policy data, where the samples are collected by taking a sequence of steps in the environment. A solution is to to use replay buffers [17]. Transitions in the form of POMDPs are sampled from the environment according to some policy β and stored in the replay buffer. For every time the network parameters should be updated a batch of transitions are sampled uniformly from the buffer.

An advantage of using off-policy algorithms is that the trade of between exploration and exploitation can be treated trough the off-policy training data by adding noise to it [17]. The trade of between exploration and exploitation in reinforcement learning is a question about how much the agent should explore the environment by taking sub-optimal actions versus using what is already known about the environment in order to get the best results known to the agent [21]. By using a replay buffer it is possible to add noise to the actor policy, solving the trade off problem.

Target networks Equation 7 show that the optimal action-value function depends on itself, and therefore the target also has to be computed through optimization of its parameters. In order to calculate the target, separate target networks are used. This is in order to train the critic network in a stable way by not having the same parameters for the approximator and the target [17].

Network	Parameter	Function
Critic network	θ^Q	Action-value function
Actor network	θ^μ	Current policy
Target Critic network	$\theta^{Q'}$	Create target action-value function
Target Actor network	$\theta^{\mu'}$	Produce actions to create target action-value function

Table 1: Parameters for the four neural networks in DDPG.

The parameters for each of the networks are presented in Table 1. The parameters of the Actor and Critic networks are used to parameterize the current policy and the approximator action-value function, while the target parameters are used in order to find the target action-value function.

When updating the parameters of the critic network to minimize the loss from the target, the parameters of the target are also updated. This update comes from the transitions sampled from the replay buffer, where the target actor is used to find an action from the sampled state or observation in the transition and the target critic uses the resulting actions to compute the target critic. Then state, actions are used in the approximator critic and compared to the target critic in the MSBE presented in Equation 8.

The target network parameters are updated by having them track the network parameters, but with a time delay, resulting in a slow but stable learning [17]. The update is done by copying only a fraction of the main weights by polyak averaging, shown in Equation 9.

$$\theta^{\mu'} \longleftarrow \theta^{\mu'}(1-\tau) + \tau\theta^{\mu} \tag{9a}$$

$$\theta^{Q'} \longleftarrow \theta^{Q'}(1-\tau) + \tau\theta^{Q} \tag{9b}$$

The update from the network parameters to the target parameters is done depending on $\tau << 1$.

2.4.2 Policy based

In policy-based reinforcement learning the optimal policy is found by directly manipulating the policy to maximize the expected reward [16]. The policy is parameterized by θ^μ, where

$$J(\theta) = E[\gamma^t r_t] \tag{10}$$

describes the expected value of the discounted cumulative reward when acting according to policy μ. This is the sum of all rewards r_t retrieved at each time step t, discounted by a factor γ that is a number between 0 and 1. Equation 10 shows that further in time the reward will have a smaller effect

on the expected reward, to what degree depends on the value of γ. J is a cost function, and the best policy can be found by maximizing J. Policy gradient is the gradient of the policys performance [17], and it can therefore be used to find the best performing policy. In this project the focus will be on deterministic policies symbolized by μ as DDPG is a deterministic method.

$$\nabla_{\theta^\mu} J = E[\nabla_{\theta^\mu} Q_{\theta^Q}(s_i, \mu_{\theta^\mu}(s_i))] \tag{11}$$

Equation 11 states that the gradient of the cost function J with respect to the policy parameters θ^μ is equal to the expected value of the gradient of the action-value function. The goal is to find the policy which chooses an action that maximizes the action-value function $Q_{\theta^Q}(s, a)$, with respect to the policy parameters θ^μ. The action-value functions action is chosen by the policy μ, therefore the resulting action-value for the current policy will criticise the performance of the policy.

To summarize the DDPG algorithm the pseudo code from the original DDPG paper [17] is included below.

Algorithm 2 Deep Deterministic Policy Gradient

Initialize actor and critic network parametersθ^ϕ and θ^Q
Initialize target networks $\theta^{\phi'} \leftarrow \theta^\phi$, $\theta^{Q'} \leftarrow \theta^Q$
Initialize replay buffer R
for episode = 1,M **do**
    Initialize a random process $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ for action exploration
    Receive initial observation state s_t
    for time step = 1,T **do**
        Select action from policy with noise $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$
        Perform a_t in environment
        Observe the next state, s_{t+1}, done signal d_t and reward r_t
        Store transition $[s_t, a_t, r_t, s_{t+1}, d_t]$ in R
        Sample a batch of N transitions from R, $B = [s_t, a_t, r_t, s'_t, d_t]$
        Set target using target network

$$y_i = r_i + \gamma Q_{\theta^{Q'}}(s_{i+1}, \mu_\theta^{\mu'}(s_{i+1})) \tag{12}$$

        Update Critic by minimizing the loss

$$L = \frac{1}{N} \sum_i (y_i - Q_{\theta^Q}(s_i, a_i))^2 \tag{13}$$

        Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_{\theta^\mu} Q_{\theta^Q}(s_i, \mu_{\theta^\mu}(s_i)) \tag{14}$$

        Update the target networks:

$$\theta^\mu_{target} \longleftarrow \theta^\mu_{target}(1 - \tau)\theta^\mu \tag{15a}$$

$$\theta^Q_{target} \longleftarrow \theta^Q_{target}(1 - \tau)\theta^Q \tag{15b}$$

3 Implementation and method

Based on the theory in chapter 2 this section presents the development of the investigated algorithms and details about the implementation. The testing of the algorithms was performed in a systematic manner, and is presented in the end of this chapter.

3.1 Tools used for implementation

The investigated framework was developed in Python due to its wide variety in libraries for implementing Reinforcement Learning, and due its popularity and community support. To visualize the simple environment plots were made using the Python library Matplotlib.

Keras is a deep learning API written in Python. It is used on top of the machine learning platform TensorFlow with Keras as the high-level interface of TensorFlow. Using these libraries building machine learning models like neural networks is realized in a straight forward manner with the main building blocks being network layers and models.

The environment of the Reinforcement Learning agent is modeled using the OpenAI Gym framework. It allows for building of custom environments, and can be used together with Keras and TensorFlow.

3.2 2D simulation environment

A 2D environment is developed for testing the investigated technique. This section introduces the main aspects of the environment. The RRT algorithm is mainly concerned about the placement of the obstacles, whereas the reinforcement learning algorithm requires a model of the environment the agent is able to interact with.

3.2.1 Reinforcement Learning in 2D environment

To enable the interaction between the environment and the reinforcement learning agent the environment has to be modeled in a way such that the agent is able to comprehend it, and utilize knowledge from the environment in order to make decisions. To model the environment OpenAI Gym is used, since it enables the construction of a custom environment where the action space and observation space can be defined.

Observation space To decide the robots placement relative to the net the observation space contains information on the distance to the goal and the angular difference of the current orientation and the orientation of the goal as viewed by the robot. It will also contain information on the distance to the edge of the environment from different orientations. The observation space is represented as a vector with the form: $\mathbf{o} = [\mathbf{d}_{collision}, d_{goal}]$, where $\mathbf{d}_{collision}$ is the vector containing the possible collision points detected by a sensor and d_{goal} is the distance to the goal.

Reward function To make the agent reach the goal it is important that it has an understanding of what the goal is, and which constraints has to be respected when trying to reach it. The total reward obtained by the agent during a sequence of transitions shapes the policy, by looking for the policy that results in the highest reward.

In this project the main goal of the agent is to get to the goal by reaching sub-goals provided by the RRT-algorithm. The goal is also to avoid collisions with obstacles and the edge of the environment, which would be the net of a fish cage in a real life scenario. Based on this information the agent should receive positive rewards for reaching the sub goals, $r_{subgoal}$, and negative rewards for colliding with either the net or obstacles, $r_{collision}$. To further guide the agent towards the sub goals it receives positive rewards for making the distance smaller, $r_{goaldist}$, resulting in the agent retrieving at each time step.

Action space The action space defines the minimum and maximum limits of the possible actions the agent can choose. The action space is set to be in one dimension, where the agent has the ability to change the yaw of the robot in each step. The orientation is defined as the angle between the NED

frame, which is the environment frame set by North, East and Down, and the BODY frame, which is the frame of the robot.

The minimum and maximum limits of the action space are set to be $A = [-20, 20]$, meaning that the actions are drawn from the interval in A. In other words, the AUV can turn max 20 degrees either way before taking a step of a set step size in the orientation of the AUV.

Step function and sensor simulation For the agent to compute a mapping between state and action it needs to have a way of obtaining some information regarding the state. The state is in real life scenarios often obtained using sensors that produce observations that often represent only a subset of the actual environment.

In a simulation case it is possible to implement functionality resembling sensor measurements. In order to do so the movement of the AUV has to be modeled with a step function. The step function calculates what the position of the AUV will be after the step is taken. When an action is chosen from the action space and executed in the environment the AUV performs a step that depends on the action. In this project the action is a change of orientation, and a step is then taken in the orientation of the AUV in a given step size.

In the reinforcement learning environment the AUV is in the BODY frame, and the environment is implemented to move in relation to the movement of the robot, while the AUV is kept at the origin of the BODY frame. After the AUV has taken a step the positions of the obstacles and the net will change according to the step taken which will also be the case for the (sub)goal(s) of the system.

The step function will return the new observation of the environment after the step is made along with the reward the step achieved, and a flag indicating if the goal is reached. For the step function to get the information on the observation, a simulated sensor is utilized.

The positions of the obstacles are used to calculate their euclidean distance from the AUV. If the distance is found to be longer than a given max distance, it is set to be unknown, resembling actual sensor measurements. The distances of the possible collision points on the edge of the environment is also found, and is resembling sensor measurements used for collision avoidance with the edge of the environment. The measurements are performed every 5 degree from 0 to 355 degrees by calculating at which point the robot would collide with the edge if continuously moving in that direction. Then the distance to the point is calculated and limited by the max distance of the sensor.

After an action is selected a step is performed by the step function based on that action. A flag detecting collision is set by checking the minimum distances for all observations, and an other flag is set for if the current goal set by a subgoal is reached by being inside the target region of the goal. The reward of the current step is then calculated based on the observed information.

Implementation of the agent The investigated technique is based on the DDPG algorithm which is suitable in this case as it handles continuous action spaces. The agent class provides a function for learning that performs the gradient calculations and update the weights by sampling from the replay buffer based of the DDPG algorithm. The agent class also contains functions to store and load the weights of the networks which is useful in order to easily load the trained parameters into the model when testing the agent. In addition they work as a backup if something were to happen to the agent during training.

The action is chosen by another function in the agent class that represents the mapping between state/observation and action. It that takes an observation and passes it trough the current policy, before adding noise during training. To implement the neural networks used to approximate the policy and the action-value function Dense layers from Keras are used.

3.2.2 Obstacle handling for RRT

The RRT algorithm mainly revolves around knowing the position of the goal and the obstacles. Figure 5 shows the 2D simulated environment where the black squares represent the obstacles.

The obstacles represent the prior map, which the RRT algorithm depends upon in order to plan a path avoiding obstacles. The obstacles are avoided by checking the distance from the robot to the outer

edge of the obstacle. If the distance is less than a given safety distance a collision flag is set, and the node will not be added to the tree. Collision checks on the transition between the nodes in the tree is also performed by using linear interpolating along the line of the edge and checking whether any of the points collide with any of the obstacles.

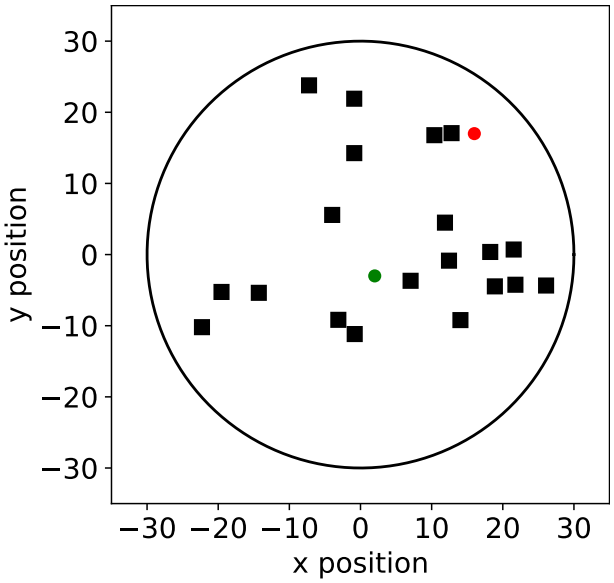


Figure 5: The 2D simulation environment developed is defined by an edge, obstacles, as well as start and goal configuration.

In Figure 5 the 2D simulation environment is shown with 20 obstacles with a dimension of 2mx2m and the environment has a radius of 30m. The black outer line represents the edge of the environment and the black squares represent the obstacles. If the AUV is within a set distance of either the edge or an obstacle, a collision flag is set. The green point represents the initial position of the AUV in the environment, and the goal is represented by a red point.

3.3 Hyperparameters

The DDPG algorithm represents the agent, and to implement a DDPG agent four neural networks are needed: the actor network, critic network, a target actor network and a target policy network. Neural networks necessitate some parameters to be taken into consideration. The Table bellow presents the initial parameters.

Variable	Initial value
Number of hidden layers	2
Dim of layer 1	400
Dim of layer 2	300
Learning rate actor: α_{actor}	0.0001
Learning rate critic: α_{critic}	0.001
Discount factor: γ	0.99

Table 2: The values are initial values chosen for the neural networks.

In this project initial values are chosen for the parameters and presented in Table 2. The architecture of the neural network is given by the number of layers in the networks and the dimension of each network, and due to the lack of guidelines for choosing the network architecture, the initial values are based on similar projects like [22] and [17], where the latter is the original DDPG paper. [16] also states that a common number of hidden layers in feedforward multilayer perceptrons are two or tree.

The other hyperparameters are based on the DDPG paper and a paper on motion planning and control for an AUV [22]. In the original DDPG paper a learning rate of 10^4 is used for the actor and 10^3 for

the critic. Both papers propose the discount factor of the reward, γ, to be 0.99.[17] also proposes a batch size of 64 samples to be used for training.

In order to choose the resulting hyperparameters the agent should be trained with different values, and due to the lack of rules on how to choose those values systematic exploring should be performed. Then the results from each of the rounds of training should be compared, and the parameters resulting in the best performance should be chosen.

3.4 Simulation experiments

The simulations are conducted in the 2D environment described in section 3.2. The idea is first to test the algorithms without any disturbance from environmental conditions, and then at a later stage use the investigated motion planner in a simulation in a fish cage simulation environment with environmental disturbances and unknown dynamic obstacles.

The experiments are conducted in the following order:

1. RRT planning in the 2D environment
   (a) Max step sizes to test: 3m, 7m 12m
2. Reinforcement learning in 2D environment
   (a) Interaction between robot and environment.

3.4.1 RRT planning in the 2D environment

The RRT algorithm is tested using different step sizes to see how the performance is affected by the step size. The average time spent reaching the goal and the average number of samples are calculated to measure the change on the computation time spent during the planning process.

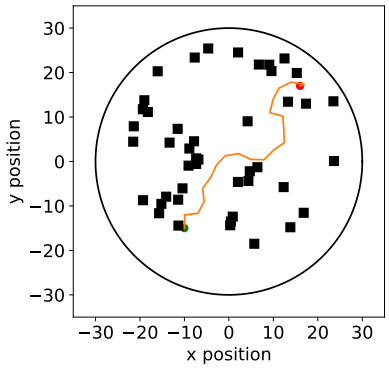
3.4.2 Reinforcement learning in the 2D environment

For the agent to interact properly with the environment, the effect the of each action on the environment has to be modelled correctly. The step function returns the new observation after an action is made. Due to the importance of the step function, the results of how the agent moves in the environment is presented first. Since the AUV is modelled in the BODY frame, the results from the step-function will not be a path, but a change of position for the surroundings while the position of the AUV is at (0,0). The simulated sensor is an important functionality in order to obtain the observation of the environment, therefore its functionality will also be demonstrated in the results section.

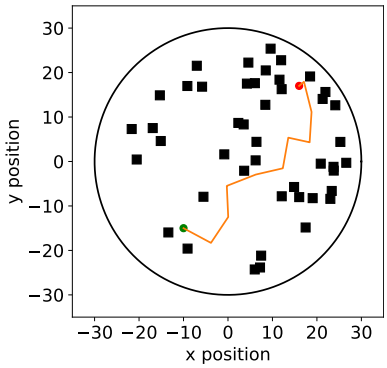
4 Results

This section presents the results from the RRT global motion planner and the results from the reinforcement learning point-to-point motion planner. The experiments have been conducted in a systematic manner, and the results are presented in the same order as introduced in section 3.4.

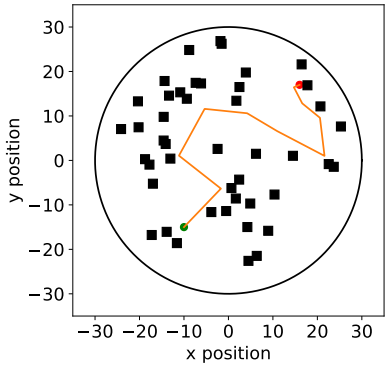
4.1 RRT results



(a) RRT with step size = 3.



(b) RRT with step size = 7.



(c) RRT with step size = 12.

Figure 6: Results from RRT global motion planner.

In Figure 6 the results from running the global RRT motion planner in the simulation environment are shown. The environment consists of the edge and the black, square obstacles. The initial configuration of the AUV is indicated b a green point, while the goal configuration is indicated by a red point. They are connected by a orange line that represents the resulting path from start to goal, where the corners along the path is where the waypoints are located. From the figure it can be seen that the algorithm is able to find a path from start to goal, while avoiding collisions for all tree step sizes.

Case	Step size	Time [s]	Number of samples
1	3	4.60	1323
2	7	1.87	1047
3	12	1.47	935

Table 3: Results from running RRT with different step sizes.

Table 3 shows the average running time of the RRT algorithm using the different step sizes corresponding to Figure 6. The values are obtained by running the algorithm 20 times for each step size, and calculating the average time it takes to reach the goal and the average number of samples needed to reach the goal.

4.2 Reinforcement learning

This section presents the results for the reinforcement learning algorithm. The main components are the modelling of the environment and the agent. First the results from the modelling of the environment will be presented. This includes the step-function, whose role is to model the movement of the environment relative to the robot in the BODY frame, and detect if a collision happens. Following the results from the simulated sensor are presented both for the edge of the environment and the obstacles.

4.2.1 Results from the modelling of the environment

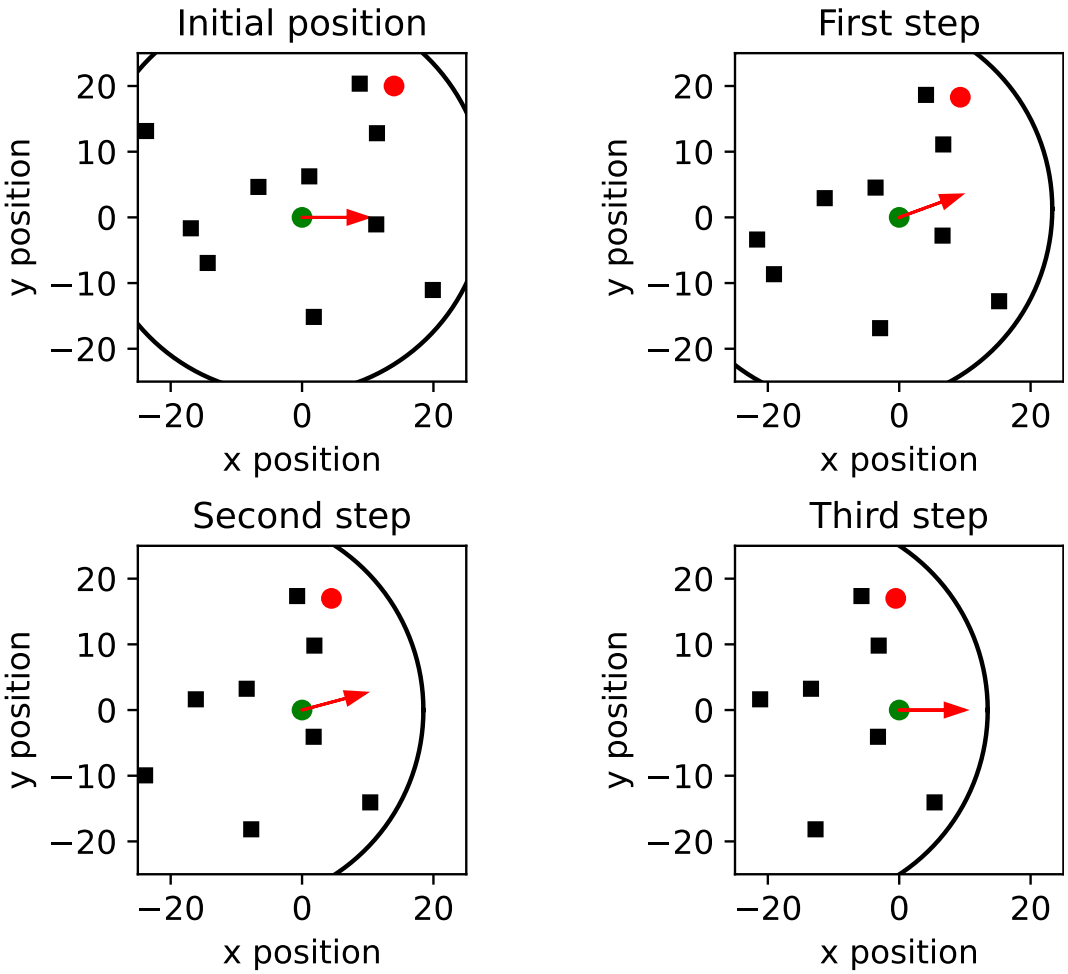


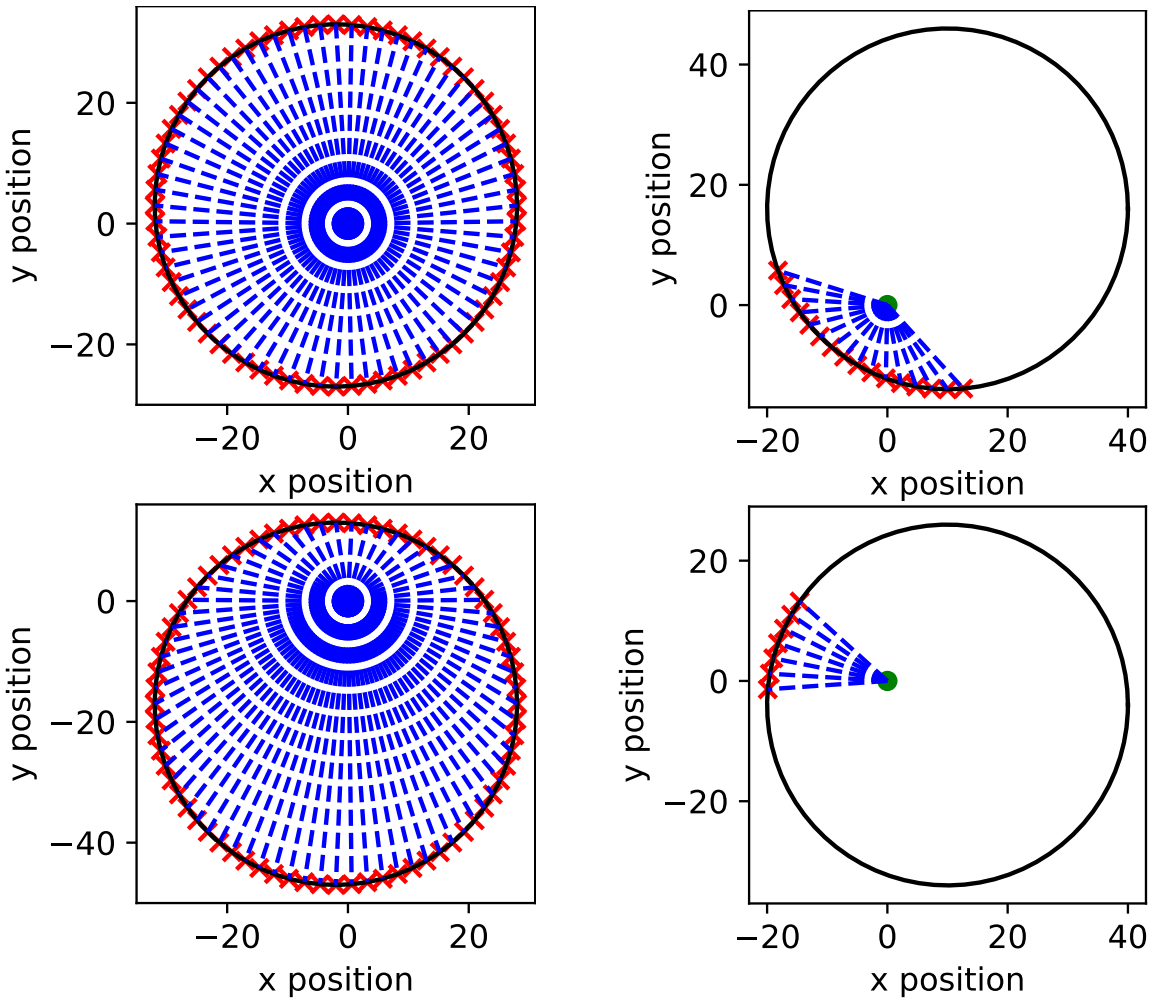
Figure 7: Step function illustrated by a sequence of actions.

Step-function Figure 7 shows a sequence of steps taken by the AUV. The orientation in each step is given by a sequence of actions $a = [20, -5, -15]$, and the step size is set to be 5m. The step size is chosen to be large in order to make the plots clear. The position of the robot is illustrated by the green point, and its orientation is shown using the red arrow. The orientation of the arrow corresponds to the state in each subplot, thus indicating the orientation of the step just taken.

From the plots it can be observed that the position of the robot remains in (0,0) while the environment moves relative to the actual movement of the robot. To show how the obstacles move relative to the robot all the obstacles are included in Figure 7. In the next section the results of how different obstacles are observed depending on their distance to the robot will be presented.

Sensor simulations The reinforcement learning environment contains implementations resembling the functionality of a sensor. The following results illustrate how regions of avoidance are detected.

The results are provided both with an unlimited range to illustrate how the algorithm works, as well as more realistic results illustrating a sensor that only obtains information on obstacles within a given proximity.



(a) Detecting all collision points. Before and after a step.

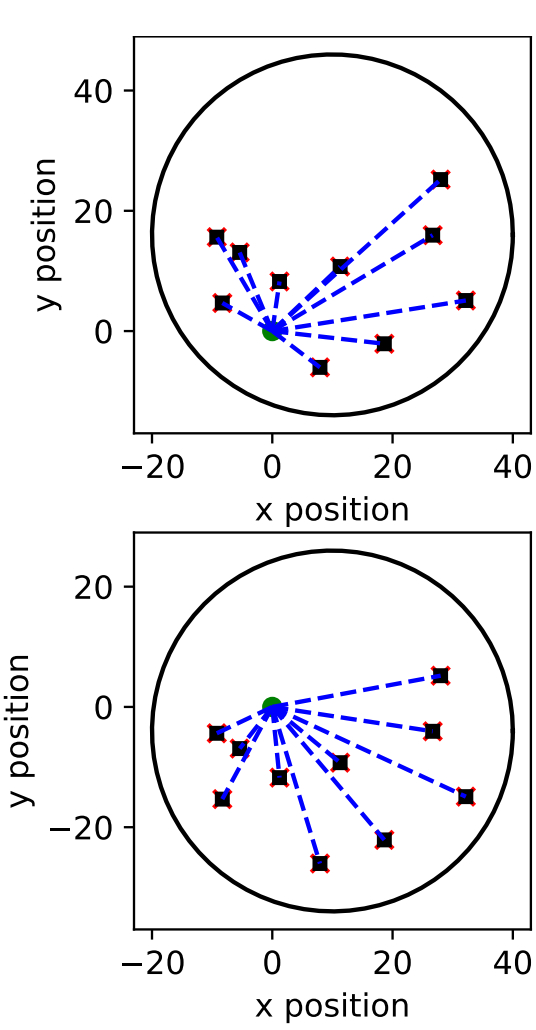
(b) Only detecting collision points within a given distance. Before and after a step.

Figure 8: Sensor function detecting collision points on the net.

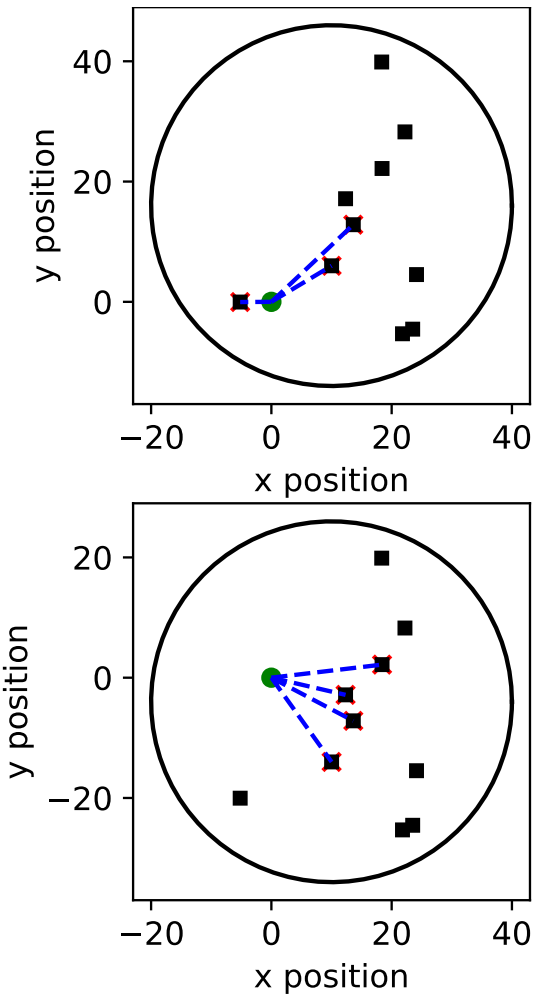
The simulated sensor locates collision points on the net for different orientations if the robot were to travel straight forward in the given direction from the current position. Then it calculates the distance to each of the points. This is illustrated in Figure 8.

In Figure 8a points all along the net are shown with a 5 degree space between. The top plot is in the initial position and the bottom plot is after a step is performed. A scenario more close to actual sensors is represented in Figure 8b, where the range of the sensors is limited. The top plot show the sensor before taking a step, and the bottom plot show the sensor after a step is performed.

The sensor function also keeps track of the centre of the obstacles, and uses this to detect collisions with obstacles shown in Figure 9. As mentioned all obstacles are kept in the plot to clearly illustrate which are detected. Figure 9a show the detection of all obstacles before and after a step is taken. In Figure 9b the range of the sensor implementation is set to be shorter.



(a) Detecting all obstacles, before and after a step.



(b) Only detecting obstacles within a given distance, before and after a step.

Figure 9: Sensor function detecting obstacles.

5 Discussion

In this section the results from the previous section will be discussed. The chapter is structured in the same order as the results in order to provide a systematic presentation of the discussion.

5.1 RRT

Comparing the paths presented in Figure 6 the main difference appears from the plots to be the path smoothness. The plot in Figure 6a has a step size of 3 meters, resulting in the maximum distance between the waypoints to be 3 meters. This results in a path that is smooth, without any sharp changes of direction. As the step size is increased the paths appear to have an increasing amount of sharp turns and less smooth appearance, which can be observed in Figure 6b and Figure 6c. Compared to the two other paths, the path in Figure 6a appears to be shorter than when a longer step size is used. This was also the trend observed when running the algorithm 20 times for each of the three step sizes. The results from Table 3 however, indicate that the smallest step size of 3m out of the tree provided a solution slower than the two longer step sizes of 7m and 12m.

5.2 Reinforcement learning

5.2.1 Performance of step function and sensor function

Step function The correspondence between Figure 7 and the sequence of actions presented in paragraph 4.2.1 can be compared in order to evaluate the performance of the step function. The position of the robot is in $(0, 0)$ at each step. At the first step the orientation of the robot is changed from the initial state to 20 degrees, indicated by the red arrow in the plot of the first step. The environment can be seen to have moved one step in the opposite direction of the state of 20 degrees. In the following two steps the orientation is changed by -5 degrees and -15 degrees, resulting in a orientation state of 15 degrees and 0 degrees, respectively. In the second and third steps in Figure 7 the environment can be seen to have moved a step in the opposite direction of the orientation state at the time.

Sensor simulations The plots in Figure 8 and Figure 9 show how observations are made. The sensor simulations can be combined with the step function, which also is illustrated in the same figures. In both Figure 8b and Figure 9b the range of the sensor functionality is set to be limited by 20m, resulting in several of the obstacles not being detected. To the robot they would therefore not be visible until they get in proximity. This represents a functionality closer to an actual sensor than the case where all obstacles are detected. In a real scenario in a fish cage the sensor would most likely not be able to sense 20m, but the value is used in the plot in order illustrate.

5.3 Future work

With the result of the project being a promising step towards the main goal of creating a motion planner for conducting robustly safe work in fish farms, this section will introduce the next steps by describing challenges and future work to address them.

Firstly, the next step would be to train the agent using the presented observation and action spaces. The agent was previously trained using an observation space in the world reference frame, resulting in a large state space as all states of the agent in the environment must be taken into account. This approach failed to produce a converged policy during the training process even after a run time of 38 hours. A solution to that could be to train the agent using the observation space presented in this project, which is fixated to the body of the robot, as proposed also in [23]. Choosing this approach will reduce the region of the state space that the policy has to perform in. By only having knowledge about the nearby environment from sensor measurements the agent will have less to process when choosing a policy by only considering the obstacles currently relevant for the action. It would also be a better approach in a real deployment situation where the agent would use sensors in order to detect obstacles rather than taking information about the entire environment into account.

In this project as well as in [23] the importance of the design of the reward function is mentioned. Previous attempts to train the agent has used reward functions that only provided a reward at the end

of an episode, that is when the sequence of steps is terminated due to reaching the goal or colliding with an obstacle or the edge. A better solution can be to provide the agent with a small reward at each step like the reward function presented in this report. The agent would be rewarded for making the distance and orientation to the goal smaller, in addition to reaching goals, while still being punished for colliding with obstacles or the edge. The previous training processes failing to produce a converged policy could be a result of the previous rewards being too sparse, which is a known challenge in reinforcement learning.

Another aspect to be explored in future training of the agent is batch normalization. It is suggested in the original DDPG paper [17], and it is used in order to manually scale features to make them in similar range across the different units.

After successfully training the agent in the 2D simulation environment the motion planner should be implemented to handle a 3D environment, where the dynamics of the 6 DOF model of the AUV and the constraints of a fish cage environment are taken into consideration. The agents performance can then be tested in a realistic simulation environment. There it would be interesting to investigate the performance of the agent by tuning of hyperparameters like the learning rate and the network architecture, as the initial values are chosen based on similar work. Currently there are no specific guidelines for how the architecture of the system should be chosen, therefore the values resulting in the best performance must be found trough trial and error.

6 Conclusion

This work presented results from a promising step towards the goal of providing a multi-layer motion planning framework for conducting autonomous operations in fish farms using an AUV. Through a literature study a multilayer motion planning scheme was investigated, using a global sampling-based planner in order to produce waypoints from a start to a goal configuration along the net of a fish cage. Then, a reactive deep reinforcement learning-based local planner for planning and avoiding of detected obstacles was investigated.

RRT was investigated as a global planner for generation of waypoints, and tested with promising results, as it was able to create a path from start to goal while also avoiding obstacles. Further the implementation of local deep reinforcement learning showed promising results towards being used as a local motion planner. Results from implementing the robots interaction with the environment through using a simulated sensor is a step towards fast obstacle avoidance of dynamic obstacles. By creating a foundation for the decision making reinforcement learning agent to be aware of its surroundings, this project is a step towards having an AUV operate in dynamic environments such as fish farms.

The main challenge of the investigated method is to choose a reward function that produces a converged policy, while also representing the wanted behaviour of the DDPG agent. This is something that should be investigated in future work as it is one of the main promising areas for improvement.

References

[1] An underwater robotics concept for dynamically changing environments. https://prosjektbanken.forskningsradet.no/en/project/FORISS/313737?Kilde=FORISS&distribution=Ar&chart=bar&calcType=funding&Sprak=no&sortBy=date&sortOrder=desc&resultCount=30&offset=30&ProgAkt.3=FRINATEK-Fri+prosj.st.+mat.%2Cnaturv.%2Ctek. Accessed: 2022-06-12.

[2] Race fish-machine interaction. https://www.sintef.no/en/projects/2020/race-fish-machine-interaction/. Accessed: 2022-06-12.

[3] Thor I. Fossen. *HANDBOOK OF MARINE CRAFT HYDRODYNAMICS AND MOTION CONTROL*. John Wiley Sons, 2021.

[4] Eduard Vidal, Mark Moll, Narcís Palomeras, Juan David Hernández, Marc Carreras, and Lydia E Kavraki. Online multilayered motion planning with dynamic constraints for autonomous underwater vehicles. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8936–8942. IEEE, 2019.

[5] Zichen He, Jiawei Wang, and Chunwei Song. A review of mobile robot motion planning methods: from classical motion planning workflows to reinforcement learning-based architectures. *arXiv preprint arXiv:2108.13619*, 2021.

[6] Anthony Francis, Aleksandra Faust, Hao-Tien Lewis Chiang, Jasmine Hsu, J Chase Kew, Marek Fiser, and Tsang-Wei Edward Lee. Long-range indoor navigation with prm-rl. *IEEE Transactions on Robotics*, 36(4):1115–1134, 2020.

[7] Chad Goerzen, Zhaodan Kong, and Bernard Mettler. A survey of motion planning algorithms from the perspective of autonomous uav guidance. *Journal of Intelligent and Robotic Systems*, 57 (1):65–100, 2010.

[8] Peter Corke. *Robotics, Vision and Control*. Springer International Publishing, 2017.

[9] Sverre Fjæra. Optimal and adaptive path planning and following for permanent resident cleaning robot operating in fish farms. Master's thesis, NTNU, 2021.

[10] Marios Xanthidis, Nare Karapetyan, Hunter Damron, Sharmin Rahman, James Johnson, Allison O'Connell, Jason M O'Kane, and Ioannis Rekleitis. Navigation in the presence of obstacles for an agile autonomous underwater vehicle. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 892–899. IEEE, 2020.

[11] Chunxi Cheng, Qixin Sha, Bo He, and Guangliang Li. Path planning and obstacle avoidance for auv: A review. *Ocean Engineering*, 235:109355, 2021.

[12] Aleksandra Faust, Kenneth Oslund, Oscar Ramirez, Anthony Francis, Lydia Tapia, Marek Fiser, and James Davidson. Prm-rl: Long-range robotic navigation tasks by combining reinforcement learning and sampling-based planning. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5113–5120. IEEE, 2018.

[13] Wouter J Wolfslag, Mukunda Bharatheesha, Thomas M Moerland, and Martijn Wisse. Rrt-colearn: towards kinodynamic planning without numerical trajectory optimization. *IEEE Robotics and Automation Letters*, 3(3):1655–1662, 2018.

[14] Steven M LaValle et al. Rapidly-exploring random trees: A new tool for path planning. 1998.

[15] OpenAI. Key consepts of rl, . URL https://spinningup.openai.com/en/latest/spinningup/rl_intro.html#the-optimal-q-function-and-the-optimal-action.

[16] Jack Parker-Holder, Raghu Rajan, Xingyou Song, André Biedenkapp, Yingjie Miao, Theresa Eimer, Baohe Zhang, Vu Nguyen, Roberto Calandra, Aleksandra Faust, et al. Automated reinforcement learning (autorl): A survey and open problems. *arXiv preprint arXiv:2201.03916*, 2022.

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

[18] Peter Norvig Stuart Russell. *Artificial Intelligence A Modern Approach*. PearsonEducation, 2010.

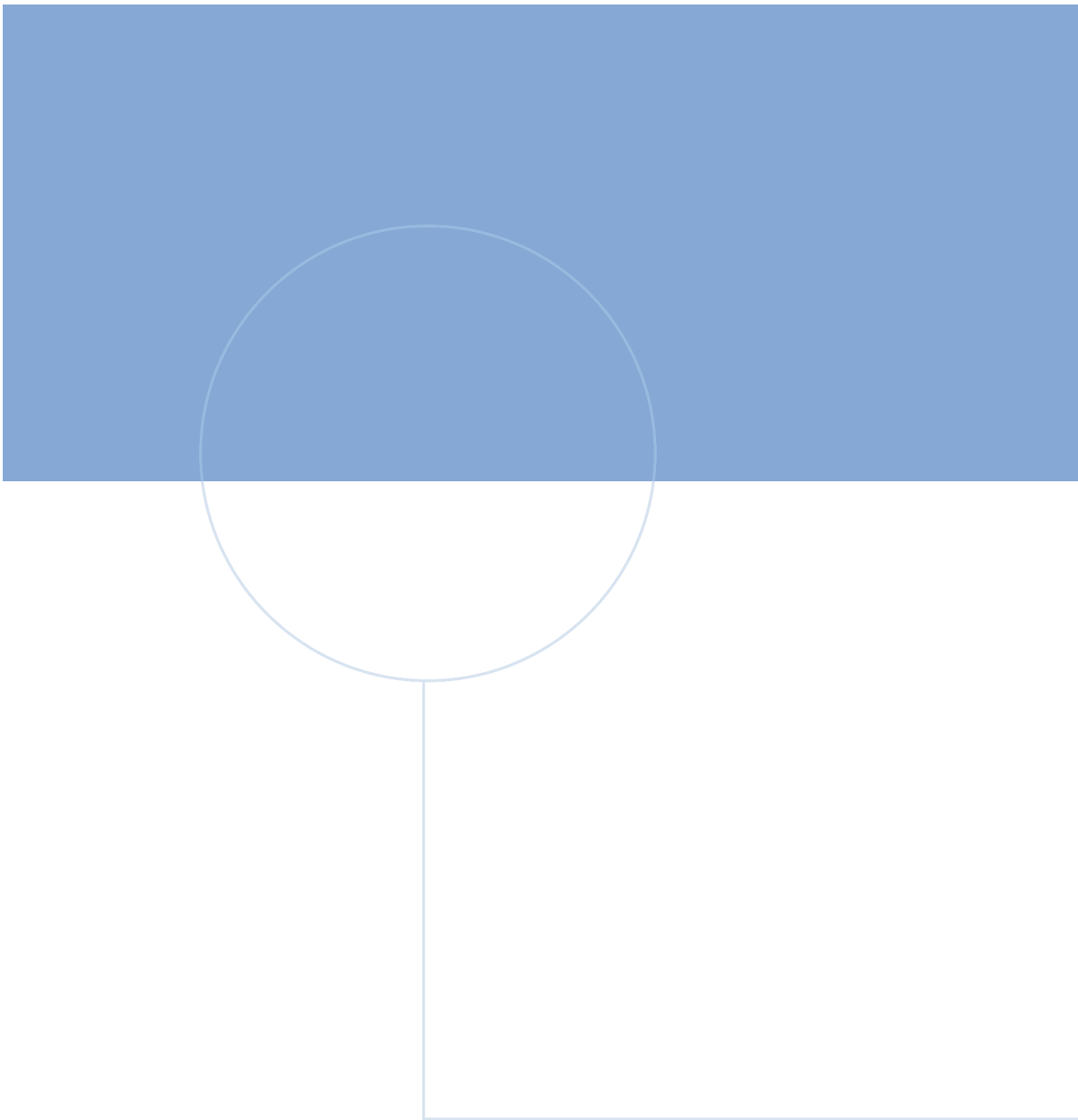
[19] Jinglun Yu, Yuancheng Su, and Yifan Liao. The path planning of mobile robot by neural networks and hierarchical reinforcement learning. *Frontiers in Neurorobotics*, page 63, 2020.

[20] OpenAI. Deep determinstil policy gradient, . URL `https://spinningup.openai.com/en/latest/algorithms/ddpg.html`.

[21] Melanie Coggan. Exploration and exploitation in reinforcement learning. *Research supervised by Prof. Doina Precup, CRA-W DMP Project at McGill University*, 2004.

[22] Simen Theie Havenstrøm, Adil Rasheed, and Omer San. Deep reinforcement learning controller for 3d path following and collision avoidance by autonomous underwater vehicles. *Frontiers in Robotics and AI*, page 211, 2021.

[23] Szilárd Aradi. Survey of deep reinforcement learning for motion planning of autonomous vehicles. *IEEE Transactions on Intelligent Transportation Systems*, 2020.



NTNU

Kunnskap for en bedre verden