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

Espen Eilertsen

High-level Action Planning for Marine Vessels Using Reinforcement Learning

Master's thesis in Cybernetics and Robotics
Supervisor: Anastasios Lekkas
June 2019

Espen Eilertsen

High-level Action Planning for Marine Vessels Using Reinforcement Learning

Master's thesis in Cybernetics and Robotics
Supervisor: Anastasios Lekkas
June 2019

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

Abstract

Autonomous marine vessels have been the subject of much research in the past few years. The motivation behind this is the potential financial gain from the shipping industry, the environmental gain and an increase in safety. This thesis aims to investigate the added value of reinforcement learning (RL) when applied to marine vessels control to increase autonomy.

A Deep Q-Network (DQN) agent is tasked with selecting optimal actions, from a predefined action set, to guide a vessel from outside a port to a designated docking space. The agent should also be able to make intelligent decisions and avoid dynamic obstacles like traffic.

The DQN agent could act as a high-level decision support system for the vessel, where the actions are either sent to a captain for approval, or directly control the lower level controllers. The action set takes the vessel dynamics into account in the final implementation, and the agent is tested on several scenarios in a simulated environment provided by DNV GL. The agent will only be afforded information that could potentially be gathered locally from a physical vessel. Examples of such information would be position and heading for all vessels in the area and also map information. This information could be collected via GPS, AIS data, satellite images or onboard sensors.

The results show that the DQN agent is fully capable of learning the environment, guiding the vessel and avoiding traffic. It therefore has the potential to increase autonomy as it goes beyond what can be accomplished using classical control theory and path-finding algorithms. The final result of this thesis is a practical implementation strategy for a DQN agent as a high-level decision support system.

Preface

This thesis was completed during the spring semester of 2019 at the Norwegian University of Science and Technology (NTNU). The thesis is a summary of all my findings on the added value of applying a DQN agent to increase the autonomy of a marine vessel. The subject of this thesis was suggested by my supervisor Anastasios Lekkas, who recognized the under-usage of RL methods in marine vessel autonomy and its potential.

The thesis is in small part inspired by a project completed the previous semester. In this project, I applied Q-learning and SARSA to a modified version of the frozen lake problem. The similarities between the frozen lake scenario and the port navigation problem presented in this thesis sparked interest in the possibilities of RL implemented on a marine vessel.

The libraries and equipment used to complete this thesis is as follows:

- Python 3.7
- Tensorflow
- Port environment simulator, provided by DNV GL
- Dell computer, provided by NTNU
- Lenovo Yoga laptop
- LaTeX
- GitHub
- Forums and learning material posted online by the AI community

It is expected that the reader has some prior knowledge of reinforcement learning and neural networks. Prior knowledge regarding marine control theory, like LOS guidance systems, is also beneficial. These subjects will be covered to some degree in this thesis, however, only so that an inexperienced reader might understand the results.

Acknowledgement

I would like to thank Anastasios Lekkas for his guidance and support throughout this semester. I would also like to thank Andreas Bell Martinsen for all his help and advice. The simulated environment and vessel model provided by DNV GL has been instrumental to this thesis. I would like to thank DNV GL and Jon Glomsrud specifically for their assistance. My Family, friends and fellow students also deserve my thanks for keeping me grounded and dragging me out of the office from time to time.

Trondheim
June 3, 2019
Espen Eilertsen

Contents

List of Figures

List of Tables

CHAPTER 1 INTRODUCTION

1.1 Background and motivation

Autonomous vehicles have been the subject of much debate and research in recent years. Googles research on autonomous automobiles [1], Teslas$^{©}$ "Autopilot" function installed in their latest model [2] and Kongsberg's research on autonomous marine vessels [3] are all examples of the increase in research and application of autonomy. These all have varying levels of autonomy and different implementation strategies but the goal remains the same: to create a fully autonomous vehicle within an operational environment.

As of today, marine navigation relies heavily on humans when it comes to high-level reasoning. The decision to go into dynamic positioning mode due to a busy dock, is one example of an action that could complement the lower level controllers and help increase autonomy. This thesis will try and add to this field by applying reinforcement learning to assist in the decision making process, and thus potentially increasing autonomy.

In order to assess the current state of marine vessel autonomy a literary review was conducted in preparation of this thesis. This review discovered several papers suggesting reinforcement learning (RL) and neural networks applications to increase autonomy in marine vessels. The subject of these papers varied heavily in regards to which aspect of the marine vessel they were applied in. Multiple papers have been released on the subject of trajectory tracking and path following [4; 5; 6; 7], as well as path optimization [8]. Using neural networks to design behaviour based high-level control of underwater vehicles has also been researched to some degree [9; 10]. The optimal control problem has also been researched, one paper suggests using the RL method policy iteration together with two neural networks to reach optimal control [11].

There are also examples of implementations of adaptive neural network controllers for underwater vehicles [12], and implementations of deep RL for lower level control of autonomous underwater vehicles [13; 14]. Papers implementing RL as obstacle avoidance were also uncovered [15; 16; 17], as well as a study implementing RL to solve stowage problems on cargo ships [18]. Research has also been done on approximation of unknown model parameters using neural networks [19]. This short summary demonstrates the scope in which modern RL and neural networks are being researched. The motivation for the increase in research comes from multiple places, and is shared by this thesis.

From a financial perspective, there are multiple motivations for increasing autonomy in marine vessels. If fully autonomous vessels were realised, this would most likely lead to a complete redesign of both the business model and vessel design. The vessels themselves would probably have less need for crew quarters and systems related to humans, like ventilation, as crew size would decrease, which would free up more space for cargo. An autonomous fleet would likely also be easier to optimize in regards to in regards to fuel and energy consumption, further cutting operational costs.

Aside from the financial motivation, safety is also a large motivating factor. If the shipping industry implement more autonomy this would likely reduce the number of accidents due to human error. As long as the system is robust enough, an autonomous shipping industry would be far safer than a human operated one. In addition to the financial and safety motivations, there is also an environmental motivation to consider. If implemented efficiently, the transportation of goods by sea could remove the need for transportation by air and land.

This would greatly reduce the emission of greenhouse gasses as aeroplanes are far worse then ships with regard to emission. Compared to a land-based transportation vehicle, cargo ships are far better suited for transportation of goods, as it can carry more cargo. This would also free up more space on roads and reduce the need to expand vehicle-based infrastructure. The motivation also comes from challenging the implementation of reinforcement learning. As no similar implementation was uncovered, the thesis also holds added value to the field of marine autonomy itself and could uncover new ways to implement reinforcement learning methods.

It is expected that the reader is somewhat familiar with the core concepts of reinforcement learning, as well as general control and optimisation theory. There will be short sections on what is deemed relevant material, though this will in no way be exhaustive, detailed explanations, only short sections covering the basics. If the reader's goal is to learn about the details of reinforcement learning it would be recommended to look elsewhere first, as the main focus of this thesis is the results gathered from the implementation of such methods. The reference list might be a good place to start for a more in-depth look.

1.2 Goal

The goal of this thesis is to investigate how reinforcement learning (RL) methods can be applied as a high-level decision support system to increase autonomy in a marine vessel. The RL agent will be tasked with choosing the optimal actions, from a predefined set of actions, in order to guide a vessel from outside a port to a designated docking space, while avoiding traffic. Several implementations will be tested at various levels of control in order to find a viable strategy. The focus will be to keep the agent as a decision support system, above the lower level controllers.

The algorithms implemented in this thesis are the Dyna Q-learning algorithm and Deep Q-network, which are both based in Q-learning. The main difference between the two is that Deep Q-network utilizes a neural network, while Dyna Q-learning does not. This gives both their own sett of pros and cons. For instance, Dyna Q-learning is easier to implement, while Deep Q-learning is more versatile. The algorithms will be implemented, tested and evaluated with different reward functions, action sets and implementation strategies.

The suggested final approach is to use a Deep Q-network agent to explore the environment while taking the vessel dynamics into account. The agent will be tested in multiple relevant scenarios, the most challenging scenario also including traffic. The agent will receive feedback from the environment regarding its position, heading, distance from goal and traffic position, as well as a reward designating good and bad actions.

1.3 Contribution

Though many industries stand to gain for the possible solutions that may be uncovered, this thesis is first and foremost focused on research. Therefore its contributions will mostly be expanding the application of reinforcement learning (RL) in the maritime domain. Identifying pitfalls and shortcomings of RL methods can also be considered as contributions.

By using multiple variants of the Q-learning algorithm, individual pros and cons with each variation will be discovered. Issues with larger state spaces, required training time and implementation difficulty, for instance, are examples of negative elements associated with some of the Q-learning variations. However, the most significant contribution of this thesis is a feasible strategy for guiding a ship through a port using reinforcement learning as a high-level decision support system. The high-level decision support system will also be able to make intelligent decisions in a dynamic environment. Not only learning the environment and guiding the vessel to the goal area, but also avoiding traffic while doing so.

The suggested strategy presented in this thesis holds additional value in several regards. Firstly, the final implementation takes into account the ship's dynamics while choosing actions. This means that the agent will not suggest actions that the vessel can not physically complete. It also proves that additional intelligent decisions can be made by the RL agent, like dodging traffic, making it capable of doing more than simply finding an optimal path.

1.4 Outline

The thesis is divided into seven main chapters, the first chapter being this introduction. Chapter two contains a problem description where the problem this thesis attempts to solve will be fleshed out. After this, Chapter three will introduce relevant background information and theory necessary to understand the results. Here reinforcement learning will be presented in a bit more detail, as well as a brief introduction to classical optimization theory, Line-of-sight guidance and explainable AI.

Chapter four will contain a system description which will cover the simulation space and the vessel model used in this thesis. This chapter will also contain a short introduction to Tensorflow. Next, Chapter five will present the implementations strategies. This chapter will present the strategies tested while attempting to solve the problem, the thought behind them and what was learned from them. Following this, Chapter six will present the results, before the thesis ends with Chapter seven, where the results are discussed and a conclusion is drawn.

CHAPTER 2 PROBLEM DESCRIPTION

A cargo vessel needs to be guided from outside a port area to a designated docking space. The vessel needs to be able to navigate both the open area before before reaching the port, and the narrow areas of the port itself. While navigating, the vessel also needs to be able to avoid moving obstacles in the form of other vessels operating in the port area. Solving this environment requires the ability to learn highly non-convex areas and adapt to a dynamic environment.

This thesis will implement reinforcement learning on a a marine vessel in an attempt to solve the presented docking problem. The thesis aims to prove that reinforcement learning has added value by showcasing a level of intelligence that is not possible to achieve with traditional control theory and path-finding algorithms. Using the simulated model for DNV GL's autonomous concept vessel, Revolt [20], reinforcement learning will be implemented on a vessel as a high-level decision support system. The agent will then be tasked with find its way from outside the port to a designated docking space, in an optimal fashion, and avoiding traffic.

As it is not beneficial for the algorithm to keep exploring when an optimal path is found, the algorithm will be designed to gradually place less importance on exploring. Eventually almost stopping to explore entirely. This way, the algorithm could potentially be trained on board the vessel before reaching the port. And the user can be confident that the agent will suggest the appropriate actions once the port is reached.

As the simulated Revolt model can be considered a digital twin, any result will be seen as directly applicable to a physical model. An additional motivation is to help DNV

GL get closer to a functional autonomous vessel. Such a vessel could be beneficial for the environment and represents a large financial potential for the company, as well as increase safety. Proving the usefulness and benefits of reinforcement learning on such a system is also a motivation.

CHAPTER 3 BACKGROUND AND THEORY

3.1 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning that focuses on goal-oriented learning. In the book "reinforcement learning: An introduction" by Sutton and Barto [21], it is compared to how humans learn by interacting with the environment. This is a very accurate comparison, as a reinforcement learning agent also learns by interacting with the environment. In short, reinforcement learning is a way to train a virtual agent by allowing it to explore an environment while rewarding good actions and penalizeing bad actions. This reward/penalty usually comes in the form of a scalar returned to the agent after completing an action. The agents objective is to learn how it can take actions to maximize this reward.

Say that there is a reinforcement learning agent that can execute actions to interact with an environment. Every time the agent executes an action it is given a reward from the environment. This reward is either a positive scalar or a negative scalar, depending on the outcome of said action. Every time the agent executes an action it moves from its current state to a new one. The state, in this case, would be some information given by the environment. If the actions taken by the agent moves it around in a grid, for instance, the state could be the agent's position in the grid. This sets up the flowchart seen in Figure 3.1. This process continues until the agent ends up in a terminal state. A terminal state could be the goal state or perhaps that the agent ended up in a state deemed extremely negative. This terminal state ends the episode and the environment resets so the agent can try again. After several episodes, the agent will have learned the most efficient way of maximizing the reward for a given episode.

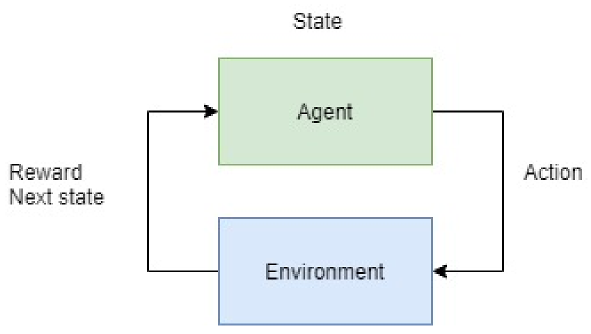


Figure 3.1: Reinforcement learning flowchart

As an example, the goal of the agent might be to find its way through a maze of some sort. In this example the agent might be given a positive reward for finding its way through the maze, ending up in some goal area. Along the way, there might be traps set up for the agent to fall in, areas of the maze which give a huge negative reward and terminate the episode. Using reinforcement learning it is possible for the agent to find its way through the maze to the goal area while avoiding the negative areas, using only information gathered from the environment. This is the basic concept behind reinforcement learning. The following sections will go over a few key concepts that explain how an RL agent does this.

Terminology

In reinforcement learning an agent is often referred to be in a state $s \in S$, and can from this state take action $a \in A$. The reward returned for completing this action is most often labelled R and is usually a scalar value as mentioned earlier. The returned policy, labelled π, can be described as a mapping from states to actions. The policy can also be either deterministic, meaning that a certain state will always result in the same action, or probabilistic, meaning the policy draws a sample from a distribution over actions, $a \sim \pi(s, a) = P(a \mid s)$ [22]. Here P is the transition probability, meaning the probability of action a resulting in state s.

Markov Decision Process

Markov decision process, often labelled MDP for short, is a mathematical formulation of an RL problem. It can be explained as a set of states, actions, rewards and decision probabilities that captures the dynamics of the system [22]. An MDP is what is used to describe the environment to a reinforcement learning agent. In order to form a MDP it is necessary for all states to fulfil the Markov property. The Markov property means that all necessary information exists in each given state. What this means is that when a state is reached all information encountered thus far can be disregarded. in short, a Markov decision process is a sequence of states that all have the Markov property.

The Bellman equation

The Bellman equation is an important and powerful tool in reinforcement learning. As mentioned, the reinforcement learning agent tries to maximise the reward gained over time, and this is usually handled by the Bellman equation. Probably the greatest benefit of the Bellman equation is that it makes it possible to express the value of each state as a function of other states. This makes it possible for a lot of iterative solutions for MDPs to be implemented.

$$V(s) = \underset{a}{max}(R(s,a) + \gamma \sum_{s'} P(s,a,s')V(s')) \tag{3.1}$$

$$V(s) = \underset{a_1}{max}(R(s_1,a_1) + \gamma \sum_{s'_2} P(s_1,a_1,s'_2)\underset{a_2}{max}(R(s_2,a) + \gamma \sum_{s'_3} P(s_2,a_2,s'_3)...] \tag{3.2}$$

$$Q(s,a) = R(s,a) + \gamma \sum_{s'} P(s,a,s')\underset{a'}{max}(Q(s',a')) \tag{3.3}$$

The equation (3.1) is referred to as the state value equation in the literature, and gives the value of a given state as a function of the reward, state, action, transition probability and the next state. By expanding $V(s')$ the resulting equation is (3.2). This equation has an infinite sequence, as the next state value can also be expanded using the next state and action in the sequence. By grouping infinite sequence together after the first max operator we can rewrite this as (3.3). This equation gives the quality of a given action in a given state. It turns out that (3.3) is far more useful in the context of reinforcement learning as it will be possible to find the expected value without having to know the transition function and reward function in advance. Meaning it can be solved with information gathered from interacting with the environment.

Exploration Vs Exploitation

An important topic in machine learning is exploration versus exploitation. Using the maze from earlier as an example, this topic deals with how the agent should act when exploring the environment. For the agent to find the path to the goal it is necessary to allow it to explore the environment. If, for instance, the agent selects random actions in states where no action has been explored before, and the best possible action otherwise, it can be understood that the agent will eventually find its goal. After the goal is found, however, the path will be set. The agent will now keep using the same path every episode when moving through the maze. This result would be good enough if the only objective was moving from A to B, but it is often desired to do so optimally. In this case, if the agent did not find the optimal path through the maze the first time, it has no way of finding it later, this is where exploration comes in.

Exploration allows the agent to make sub-optimal choices, allowing it to explore different paths that could potentially be better than the first path found. There are multiple ways of incorporating this into an algorithm [23; 24], perhaps the easiest way would be the "epsilon-greedy" strategy. The epsilon-greedy strategy lets the agent chose a random action, instead of the optimal one, at random times. This strategy has proven quite effective and is well known in the field. It is often implemented at a decreasing rate, with the goal being that the agent will explore a lot at first and eventually converge to the optimal path. Regardless of strategy, it is important that there exists a balance between exploration and exploitation. This balance is necessary as the objective is to find the optimal path, but not move away from it once it is found. The element of exploration also means that the agent might execute actions that lead to a negative outcome, just for the sake of exploration. In the maze example, this means that the agent might jump into a negative "trap state", even if it is aware that this is a terminal negative state.

3.2 Q-learning

Q-learning likely one of the most well known and explored reinforcement learning algorithms. The main concepts of Q-learning are relatively easy to understand. Say we have a Q-learning agent that exists in a simulated world, and we want to complete some goal. It is possible for the agent to interact with this environment by executing actions. When an action is finished the environment sends a response. This response comes in the form of a positive reward or a negative penalty. What the Q-learning agent does is to simply interact with the world, observe response and evaluate how good that action was based on that response. The quality of executing a given action, in a given state, is often referred to as the Q-value and is recorded in what is known as a Q-table. Table 3.1 is an example of what a Q-table might look like.

$\mathrm{Q}(s_1, a_1)$	$\mathrm{Q}(s_1, a_2)$
$\mathrm{Q}(s_2, a_1)$	$\mathrm{Q}(s_2, a_2)$
$\mathrm{Q}(s_3, a_1)$	$\mathrm{Q}(s_3, a_2)$

Table 3.1: Q-table example

The Q-table is most often a matrix where the rows represent states and the columns are the actions that can be taken from the given states. When calculating the Q-value for this table, a modified version of the Bellman equation, (3.4), is used.

$$Q(s,a) = Q(s,a) + \alpha[R(s,a) + \gamma max(Q'(s',a')) - Q(s,a)] \tag{3.4}$$

In (3.4), $Q(s,a)$ is thought to be the quality of an action taken from a given state. While $Q'(s',a')$ is the quality of the next move given that a previous action was taken from a given state. The variable α is known as the learning rate, and γ is the discount factor. By varying the learning rate we can decide what information the agent should consider. If the learning rate equals one the agent only considers the most recent information gathered. However, if the learning rate equals zero the agent will not learn anything new at all. The discount factor, γ, decides the importance placed on future reward versus immediate reward. If the variable is equal to one, the agent will focus on the long term gain. On the other hand, if the discount factor is equal to zero the agent will focus on short term gain.

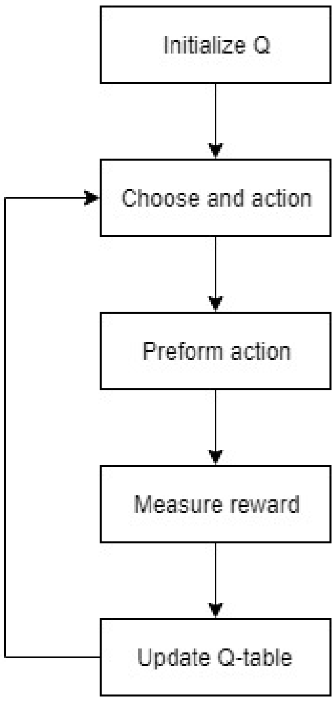
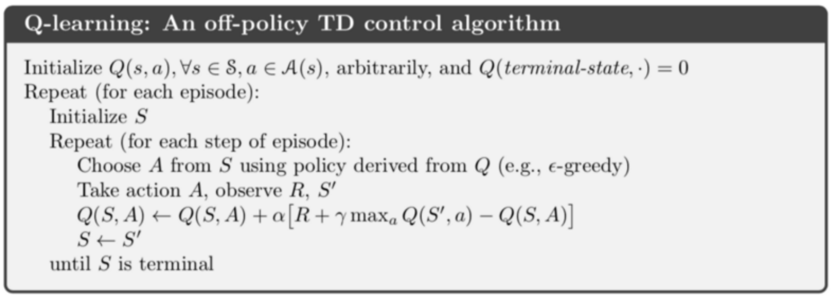


Figure 3.2: Flowchart of Q-learning

Figure 3.2 is a flowchart explaining the workflow of the Q-learning algorithm. As the algorithm learns via interacting with the environment it is common practice to train the agent on a simulated environment first, before implementing it on a physical one. This is due to Q-learnings interactive way of learning and the nature of exploration. As the agent tries multiple actions before figuring out what to do, it will likely fail multiple times. If this was implemented on a physical system initially, say an autonomous car, the car would likely crash multiple times before the agent can figure out what to do. This would create many dangerous scenarios both for the system and the environment.



```
Q-learning: An off-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(terminal\text{-}state, \cdot) = 0$
Repeat (for each episode):
   Initialize $S$
   Repeat (for each step of episode):
      Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy)
      Take action $A$, observe $R$, $S'$
      $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
      $S \leftarrow S'$
   until $S$ is terminal
```

Figure 3.3: Q-learning pseudo code [21]

3.3 Dyna Q

A different variety of Q-learning, called Dyna Q, is introduced in the book "Reinforcement learning: An introduction" by Sutton and Barto [21]. This algorithm is an expansion on the traditional Q-learning algorithm, and adds a planning phase between each interaction the agent does with the environment. The algorithm can be explained as follows. First, the normal Q-learning steps are executed, however, after updating the Q-table the algorithm does three additional steps. It first estimates a transition model and reward function, then it "hallucinates experience" using only previously discovered state-action pairs. Finally, the Q-table is updated again before starting back at the top. This means, that in Dyna Q, planning, acting, model-learning and direct reinforcement learning all occur continuously. Figure 3.4 explains the steps involved in Dyna Q in a more visual manner.

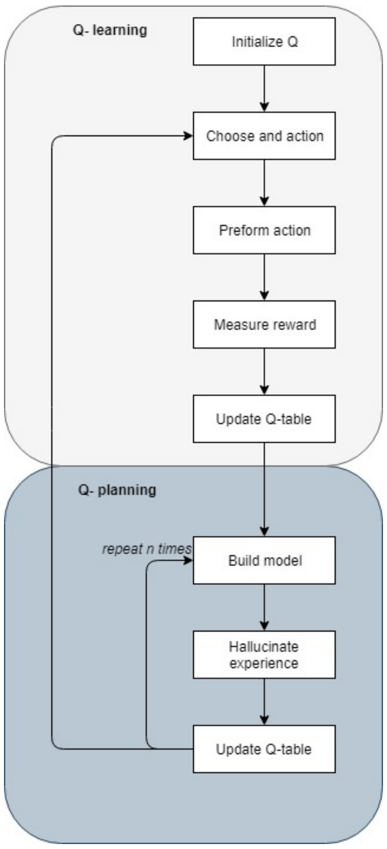
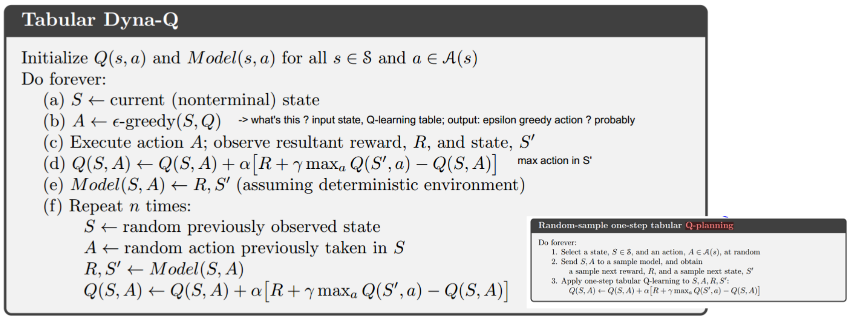


Figure 3.4: Flowchart of Dyna Q

The algorithm assumes a deterministic environment in the new "planning phase" and creates a $Model(s, a) = r, s'$. Then, for a given amount of steps, it chooses a random state-action pair that has been observed before and updates r, s' with respect to $Model(s, a)$ before updating the Q-table. $Model(s, a)$ is, in other words, the state-action reward pairs that have been discovered so far, a model of the world as seen from the agent perspective. For every step the agent takes it then goes back and updates the model with this new piece of information. This results in a faster convergence towards an optimal policy. The pseudo code for this algorithm is shown in Figure 3.5.

The reason Dyna Q is faster then traditional Q-learning is mainly due to the fact that the "planning" is computationally inexpensive. While interacting with the environment is an expensive method of learning, simply going back and "hallucinating" new experiences is a pretty cheap operation. By doing this multiple times for each interaction with the environment, new information spread faster to previously discovered state-action pairs. This means that when the agent finally discovers the goal that information moves faster backwards through the other states with Dyna Q then with Q-learning. Resulting in the agent having to discover the goal area fewer times before converging to an optimal policy. This also means that Dyna Q is not an entirely model-free algorithm, as it does create a model of the environment as it learns.



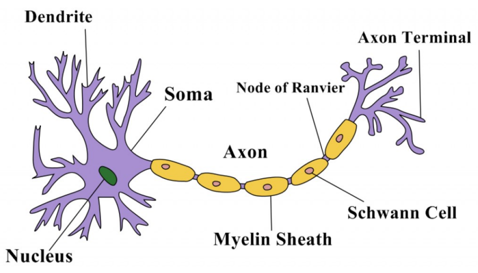
```
Tabular Dyna-Q
Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
    (a) $S \leftarrow$ current (nonterminal) state
    (b) $A \leftarrow \epsilon$-greedy$(S,Q)$    -> what's this ? input state, Q-learning table; output: epsilon greedy action ? probably
    (c) Execute action $A$; observe resultant reward, $R$, and state, $S'$
    (d) $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$    max action in S'
    (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
    (f) Repeat $n$ times:
            $S \leftarrow$ random previously observed state
            $A \leftarrow$ random action previously taken in $S$
            $R, S' \leftarrow Model(S,A)$
            $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
```

```
Random-sample one-step tabular Q-planning
Do forever:
    1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(s)$, at random
    2. Send $S, A$ to a sample model, and obtain
           a sample next reward, $R$, and a sample next state, $S'$
    3. Apply one-step tabular Q-learning to $S, A, R, S'$:
           $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
```

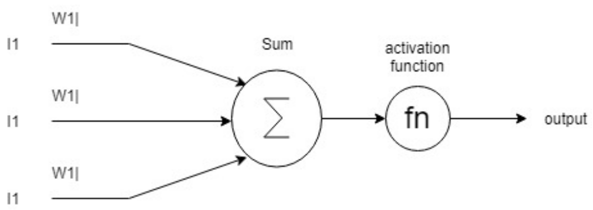
Figure 3.5: Dyna Q pseudo code [21]

3.4 Artificial Neural Networks

The Idea of an artificial neuron was first formed by McCulloch and Pitts in their paper ”A logical calculus of the ideas immanent in nervous activity” [25]. Here McCulloch and Pitts outlined the idea of an artificial neuron based on the neurons existing in the human brain. Biological neurons communicate with each other by sending electric pulses through their ”neural wiring”. A basic model of a neuron can be seen in Figure 3.6. If we consider the inputs to a artificial neuron as the Dendrites, the Soma as the node and the Axon as the output, it is easy to see the similarities between an artificial neuron and a biological neuron. This kind of single artificial neuron is referred to as a Perceptron in the literature, and when connected as shown in Figure 3.7 they form an artificial neural network (ANN).



(a) Model of neuron



(b) Model of artificial neuron

Figure 3.6: Comparison between artificial and natural neurons

An artificial neural network can be described as multiple layers of interconnected nodes, or neurons, which are used for machine learning purposes. The nodes are connected to each other via edges between individual layers. First, information enters through the input layer. Then it is passed to the hidden layer before ending up in the output layer. If a neural network has multiple hidden layers, as in Figure 3.7, it is referred to as a deep neural network (DNN). The network in Figure 3.7 is also what is referred to as a fully connected neural network. Meaning that each node in a given layer is connected to every node in the next layer, which is not always the case in ANN.

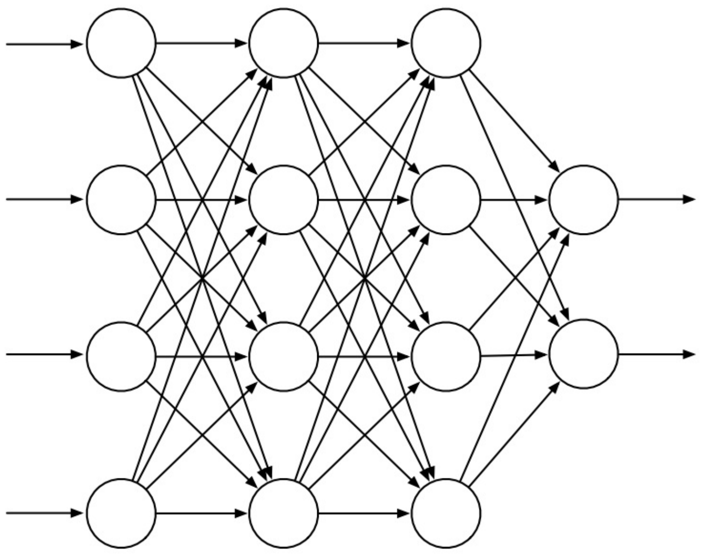


Figure 3.7: An artificial neural network with two hidden layers

An ANN can also be modelled as a function where, upon receiving some input, it will produce an output. In fact, the entire network can be thought of in this way, (3.5) gives us a mathematical representation if a single layer. Each layer can be represented as a weighted input matrix, a bias in the form of a vector and an activation function. If the network is then given an input, x, it will produce an output, y.

$$y = f(Wx + b) \tag{3.5}$$

The activation function decides when the node should "fire", meaning that it sends a signal to the next node. An example of a typical activation function would be the Sigmoid function, for instance. This is an S shape function that exists between 0 and 1. This particular function is very useful if the goal is to predict probability as an output. Another function that has become more used recently is the "rectified linear unit" function [26], or "Relu" for short, shown in Figure 3.8. This function returns zero if it receives any negative input, but returns a value back for any positive input. Softmax is an activation function that is often used in the output layer. This is because it normalizes all the values such that the sum of all the output nodes is equal to one, which is favourable when a probability distribution as output is desired.

When choosing the activation function it is important to have an understanding of what you would like to accomplish and what the input to the system is. For example, if the input to the system is a vector of numbers in the range minus one to one. It might not be favourable to use the Relu activation function, otherwise, all input information between minus one and zero would be lost. In this case, a "tanh" function might work better, as it returns a value from minus one to one. If the output is to be a probability distribution, Softmax might be the way to go. But if not, a linear function might be better suited.

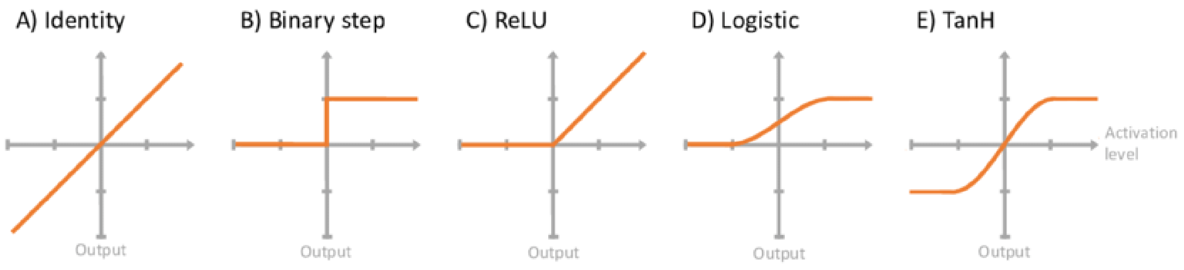


Figure 3.8: A few common activation functions[27]

When training an ANN it is common to use labelled data sets. This data consists of a set of inputs and outputs, where the output data corresponds with the given input data and is the correct response to it. The ANN uses this data and adjusts the weights to each node until it can reach a result which is as close to the correct result as desired. A commonly used algorithm for this purpose is called backpropagation. This algorithm trains the network by executing forward and backward passes through the network [28].

A forward pass is exactly what it sounds like. The input set is passed through the network and the output is recorded. This output is then evaluated against the correct output set and the error between them is calculated. The backwards pass then calculates the gradient of the error function with respect to the weights in the network. As the gradient is defined to move towards the greatest ascent, the weights are adjusted in the opposite direction, towards the minimum.

3.5 Deep Q-Networks

Deep reinforcement learning can be explained as the combination of deep neural network and reinforcement learning. One such method is the deep Q-network (DQN), which will be implemented later in this thesis. DQN is the marriage between Q-learning and deep neural network. This algorithm implements deep neural networks to approximate the Q-values and allows for the use of a larger state space. The algorithm was originally created by "DeepMind", who presented the algorithm in their paper "Playing Atari with Deep Reinforcement Learning" [29], which was later followed by the paper "Human-level control through deep reinforcement learning" [30]. The paper presented the algorithm along with the result gathered by implementing it to solve classic Atari 2600 games.

This was done by using the pixel values of the screen itself as state descriptions, and giving the agent the same actions as would be available to a human player. By stacking a number of frames and passing them through a deep neural network as input, the network manages to correlate actions with the change in input-frames. The network then estimates and outputs the Q-values for the individual actions in the given state. Following the same logic as a traditional Q-learning algorithm, the action associated with the highest Q-value will be the optimal action once the agent is sufficiently trained. The neural network structure is a combination of first convolutional, then dense layers using the "ReLU" activation function [29].

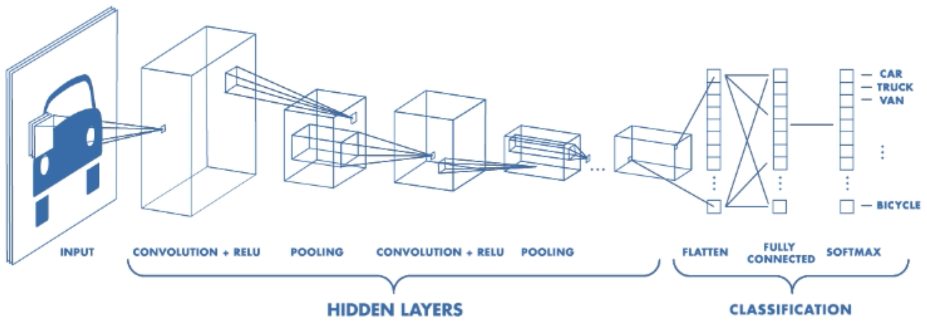
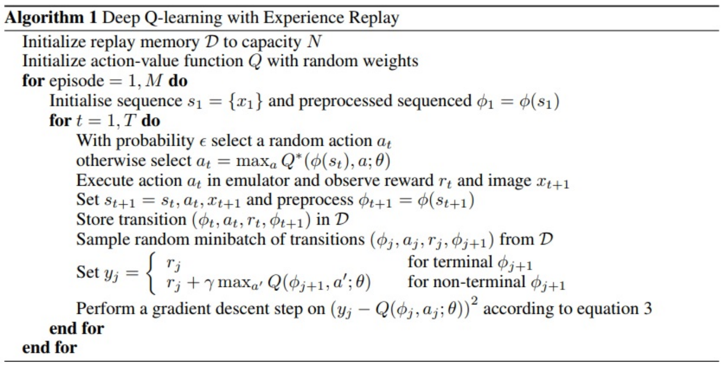


Figure 3.9: A simple convolutional network structure [31]

A convolutional layer is a type of neural network architecture that is often used when the goal is to solve image classification tasks. These layers are different from dense layers, as all nodes in one layer do not point to all nodes in the next layer. Instead, a region of nodes is connected to every node in the next layer. Figure 3.9 is a visual representation of a simple convolutional network. The region that covers a number of nodes is referred to as a filter.

Although this might seem like a trivial achievement, it was a huge step towards general AI. This was due to the fact that the algorithm was able to reach a near super-human level of skill at any Atari game it was applied to. More importantly, it was able to do this without being adjusted for the specific game.

It has been proven that a single hidden layer neural network can approximate any continuous function, provided there are no constraints on any weights or number of nodes [32]. As this is the case one might wonder why bother with deep neural networks. It turns out there are two advantages to deep neural networks. Firstly, they generalise well, which helps avoid overfitting. And secondly, a shallow network will probably require more nodes per layer than a deeper network. This gives deep neural networks a computational advantage [32]. The pseudo code for a deep Q-learning network can be seen in Figure 3.10.



```
Algorithm 1 Deep Q-learning with Experience Replay
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
    end for
end for
```

Figure 3.10: Deep Q-learning pseudo code [21]

3.6 Deep Q-Extensions

Though an effective algorithm, DQN can be regarded as rather old in reinforcement learning years. However, that does not mean that the algorithm has been abandoned. Over the years multiple strategies for improving the DQN algorithm has been developed. This section will cover four such methods, fixed Q-targets, double DQN, duelling DQN and prioritised experience replay.

Fixed Q-target

In traditional DQN we calculate the temporal difference error (TD error) as the difference between the Q-target(Q') and the current Q-value(Q), however, since the Q-target is not known it needs to be estimated. In traditional Q-learning, these values are found by interacting with the environment and using the Bellman equation. When using a DQN, these values are estimated by the network. This becomes a problem as the same parameters (weights) are used to calculate both the Q-value and Q-target. In other words, there is a huge correlation between the Q-target and the changing parameters. The end result is that for every step of training, both the Q-value and Q-target shifts.

We can visualise the effects of this by thinking of a famous scene from the movie Rocky, where Sylvester Stallone (as Rocky) tries to chase down and catch the elusive chicken. Suppose Rocky is the Q-value and the chicken is the Q-target. Rocky tries to catch the chicken, however, every time Rocky moves closer, the chicken moves further away. This is similar to the effect of using the same network to estimate both values. This causes large oscillations during training, which is not desired. Luckily, the issue can be solved by implementing fixed Q-targets.

Fixed Q-targets are relatively simple to implement in the DQN algorithm and was introduced by DeepMind in the paper "Human-level control through deep reinforcement learning" [30]. First, two networks are created from the same template. These networks are often referred to as the local and the target-network. Then a function that copies the weights from the local network and ads them to the target-network is created. The last step is to use the target-network to calculate the Q-target during training. The target-network is then updated with the local-network weights at the desired interval. This interval becomes another tuning parameter that must be tuned for the desired implementation.

Double Deep Q-Networks

Another problem with DQN is the tendency to overestimate the Q-values. This can be solved by implementing double DQN, which was first introduced by Hado van Hasselt [33]. The problem itself can be boiled down to a simple question, how can we know that the best action in the next state is the action associated with the highest Q-value? At the beginning of the training process, this value is very noisy and can often lead to a false positive. This complicates learning and increases the time required for an agent to be sufficiently trained.

The solution is again to utilise multiple networks to solve the problem. If we assume that the local and target-network from the previous section is already implemented, this network can be used to decouple the action selection from the Q-target generation. This is done by using the local-network to select what the best action is in the next state and use the target-network to calculate the target Q-value of taking that action in that state. This way, the double DQN method reduces the overestimation of the Q-value and cuts down training time in the process.

Duelling DQN

As mentioned before, the Q-values tells us how good it is to be in a given state and take an action at that state. With this in mind, it is possible to decompose the Q-value into two parts, resulting in (3.6).

$$Q(s,a) = A(s,a) + V(s) \tag{3.6}$$

Here $A(s,a)$ is the value for taking an action in a state and $V(s)$ is the value of being in the state. What a Duelling DQN tries to accomplish is the separation of these two aspects of the Q-value. The network separates into two streams, with one aspect on either side, before combining them again at the end to get the full Q-value. The Idea is that this will give the network the ability to learn which states are valuable, without having to explore the effects of all the actions in each state. This implementation was presented in the paper "Duelling Network Architectures for Deep Reinforcement Learning" [34].

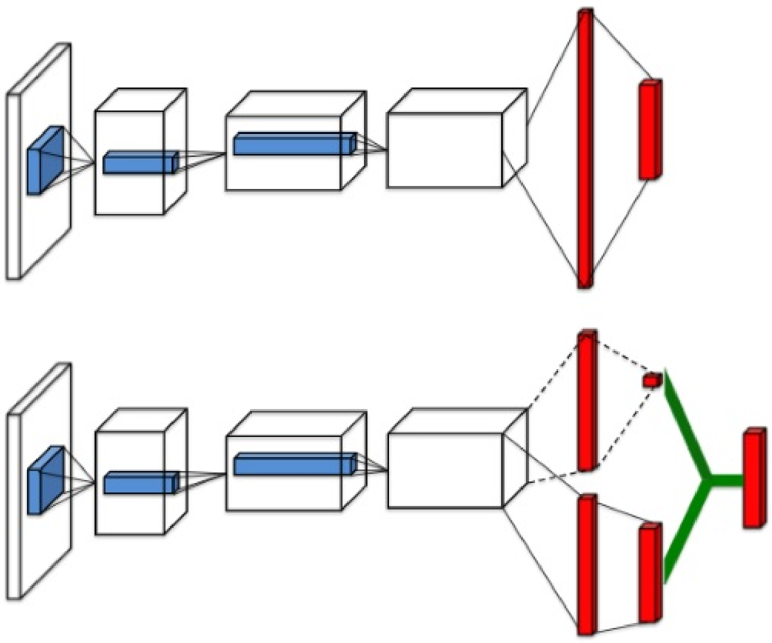


Figure 3.11: Figure visualizing the two different streams and their reintegration [34]

Prioritized Experience Replay

Prioritized experience replay (PER) introduces the concept that some interactions with the environment will contain more information than others. Assuming that this is true it would obviously be better to replay these experiences more often when training the network. In traditional DQN this is not the case, the algorithm simply replays a random sample from the memory. PER introduces a way to prioritise the most important experiences and replay them more often. In the paper "Prioritized Experience Replay" [35] this method was implemented on a DQN and proved to outperform the standard DQN in 41 of 49 games.

What constitutes a valuable experience can be found in the calculation of the TD error, which is the difference between the Q-value and the Q-target. If this difference is large it means that something significant happened during this interaction. This difference can therefore be used as a priority variable to implement PER. It is important to realise, however, that implementing a simple greedy solution will lead to the network only being trained on a single experience. Therefore PER introduces stochastic prioritization, which generates a probability of being chosen for replay, see (3.7). Here a is the priority scale, where $a = 1$ will give full priority sampling, and $a = 0$ will give normal random experience replay.

$$P(i) = \frac{p(i)^a}{\sum_k p_k^a} \tag{3.7}$$

It is also necessary to adjust how the weights in the network are updated, since the memory selection now has a bias in the form of a probability transition. The adjustments suggested in the paper "Prioritized Experience Replay" [35], is to simply adjust the weights only a small portion when replaying an important memory, see (3.8).

$$w = (\frac{1}{N} \cdot \frac{1}{P^i})^b \tag{3.8}$$

3.7 Classical Optimization Theory vs Reinforcement Learning

Problems that can be solved with reinforcement learning can often also be solved using classical optimization theory. An optimization problem is usually formulated mathematically either as a minimization or maximization problem that is subject to some constraints. The problem is usually a function referred to as a ”cost function”, which is based on the model of the system. The fact is that RL is, in part, designed to do exactly the same job as classical optimization. The main differences are how the problems are formulated and solved, and both methods come with their own set of advantages and disadvantages. Two classical optimization methods will be presented in this section, MPC and LQR, as to get a better feel of the similarities. These methods are outlined in the paper ”Merging optimization and control” [36]

MPC stands for ”Model Predictive Control” and is a well know and widely used optimization method for optimal control. MPC runs different forecasts for what the optimal control input should be for a system to reach a particular goal based on a model. It then applies the first calculated input, observes results, and recalculates the forecasts. In other words, the MPC first calculates the set of optimal inputs needed to reach the goal, applies the first input, then recalculates all the optimal inputs again. This makes MPC a very flexible, although somewhat computationally expensive, optimization strategy.

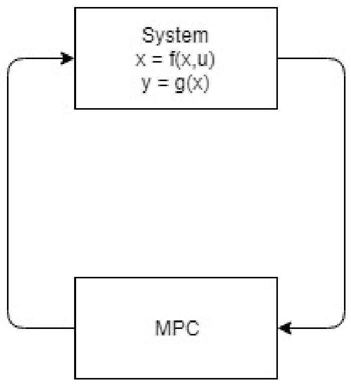


Figure 3.12: Feedback MPC flowchart

The LQR, Linear quadratic regulator, is another optimization method that is often used. This method comes with a given cost function that enables the designer to put restraints, or cost, on states and inputs, (3.9). This is done via two matrices designed in the cost function. By designing the values in these matrices it is possible to penalize the system for deviation of a state and restrict how much input is used. It has been proven that there exists an optimal function that minimises this cost function. This is (3.10), which is the linear quadratic regulator. Its a linear matrix, K, times a state, that minimises a quadratic cost function, (3.9).

$$J = \int_0^\infty (x^T Q x + u^T R u) \tag{3.9}$$

$$u = -Kx \tag{3.10}$$

One of the most fundamental differences between RL and classical optimization theory is the dependency on a model. Optimization theory is heavily dependant on a model, as we can see from the MPC and LQR. Though simply having a model is not enough, they are also dependant on this model being very accurate to get successful results. This means that if the model deviates slightly from the actual physical system it might be impossible to get the desired results using optimization. In addition to this, creating accurate models is a very time consuming and expensive task. Model-free RL algorithms do away with this problem by not considering the model at all and instead learn the system via interaction.

RL has the advantage of being far more adaptable than optimization. As the classical methods are based heavily on a model, this causes serious problems if the model changes or is inaccurate. It is not hard to imagine that an exact model of the ocean, for instance, would be impossible to produce as it is highly dynamic. Again, several RL methods can forgo the problem by not considering a model and can therefore adapt to a dynamic environment. On the other hand, It is easier to prove optimality with classical optimization, as long as the model is adequate. This is not a trivial task in RL, especially in complex problems, due to the nonlinear mapping between states and actions.

RL decisions can be hard for humans to interpret. This is not a problem with classical optimization, due to the fact that a model exists. If there are some questions as to why a system is behaving as it is, the solution is simply to open the model and check. RL, often being model-free, does not afford the user this luxury. Because of this, RL methods are harder to trust as one can not always fully understand why it is taking any given action. This leads into a topic that has been generating traction in the field lately, namely how to make AI and machine learning more explainable.

3.8 Explainable Artificial Intelligence

In recent years machine learning and AI technologies have seen an increase in application and use. The success of these methods has prompted more research that keep pushing the limits and application of these technologies in modern society. Though this has led to a problem with machine learning and AI becoming more apparent, they are non-intuitive and hard to understand. The fact is that the more complex the system, the harder it is to find out why the agent is acting the way it is. If the user does not know why an agent is taking a given action, whether it is succeeding or failing, it is hard to trust that the agent is behaving as intended. This trust is essential, as it stands in the way of using machine learning and AI technologies on important and vital applications. In other words, the fact that machines can not explain their decisions to users limits their effectiveness and usability [37; 38].

This has prompted research focusing on how to make these technologies more explainable. DARPA, for instance, has launched "The explainable AI (XAI) program" [37] with this goal in mind. The program aims to create machine learning techniques that:

- Produce explainable models without sacrificing high-level learning performance
- Enable users to trust, understand and manage the emerging generation of AI

Creating such models would not only help user trust, understand and manage the system, it could also be a powerful tool in regards to debugging. The idea thus far is to develop or modify ML techniques that produce explainable models, using human-computer interface techniques to translate the model into understandable explanation dialog [37].

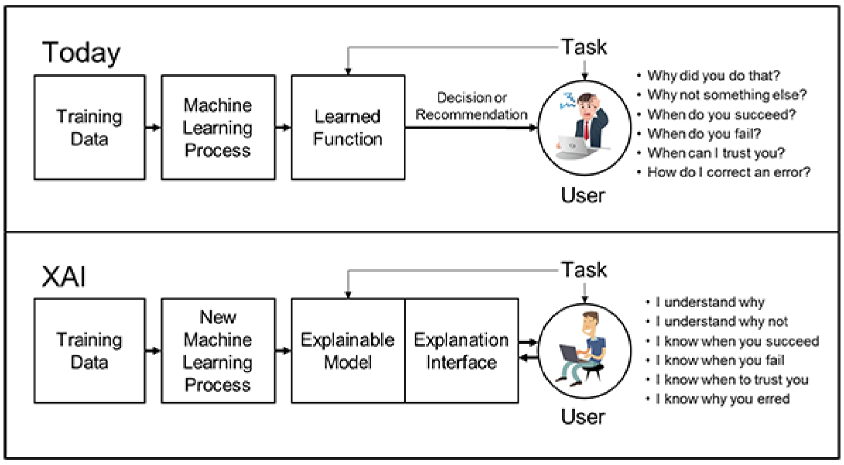


Figure 3.13: XAI concept by DARPA [37]

3.9 Line-Of-Sight Guidance

LOS, which is short for Line-Of-sight, is a form of guidance system very commonly used in marine vessels. The guidance system has been proven very effective at guiding a marine vessel along a path generated by two or more points, LOS does this by controlling the heading of the vessel in such a way that it will converge to a given path. This is done by working in tandem with a motion control system that controls the rudder to track the desired heading angle. Figure 3.14 shows a block diagram of a generic LOS guidance system.

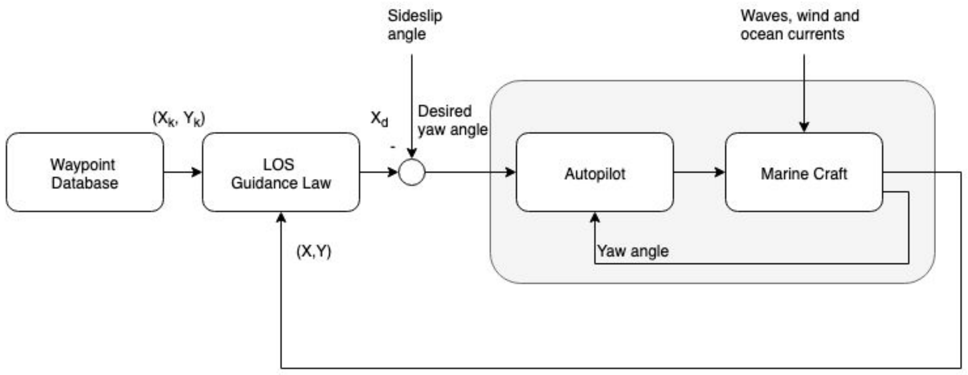


Figure 3.14: LOS guidance system architecture

Implementing a LOS guidance system on trajectory tracking, for instance, usually involves the LOS guidance calculating the desired heading and feeding it as an input into a PID autopilot. This PID controller would be the motion controller in this example and is implemented in series with the LOS system [39].

According to the book "Handbook of Marine Craft Hydrodynamics and Motion Control" by Thor I.Fossen [40], LOS guidance is classified as typically a three-point guidance scheme. The three points referring to a stationary reference point, the interceptor and the target. The guidance system is illustrated in Figure 3.15. As an example, consider a straight line path created by two points, $P_k^n = [x_k, y_k]^T$ and $P_{k+1}^n = [x_{k+1}, y_{k+1}]^T$. And a fixed path reference frame with P_k as its origin, whose x-axis has been rotated by a positive angle relative to the x-axis, as shown in (3.11).

$$\alpha_k := atan2(y_{k+1} - y_k, x_{k+1} - x_k) \tag{3.11}$$

The position of the vessel in the path fixed reference frame can then be calculated using

(3.12), where $R_p(\alpha_k)$ is a rotation matrix about angle α_k. In (3.13), $s(t)$ is the along-track distance and the $e(t)$ is the cross-track error.

$$\epsilon(t) = R_p(\alpha_k)^T (p^n(t) - p_k^n) \tag{3.12}$$

$$\epsilon(t) = [s(t), e(t)]^T \tag{3.13}$$

In this scenario, the control objective is to make the cross-track error tend towards zero. The book by Fossen [40] outlines two ways of using LOS systems to do this, enclosure based and look ahead based. Continuing the example, the enclosure based method will be explained to get a better intuition on how a LOS guidance system works.

Consider a circle with a radius (R) enclosing $p^n = [x, y]^T$. Given that the circle is chosen large enough it will intersect with the line between P_k^n and P_{k+1}^n at some point $p_{los}^n = [x_{los}, y_{los}]^T$. Driving the velocity vector of the vessel toward this point would then drive $e(t)$ to zero, achieving the desired effect. The desired course is then found using (3.15).

$$tan(X_d(t)) = \frac{\Delta y(t)}{\Delta(x(t)} = \frac{y_{los} - y(t)}{x_{los} - x(t)} \tag{3.14}$$

$$X_d = atan2(y_{los} - y(t), x_{los} - x(t)) \tag{3.15}$$

Finally, the two unknowns in the p_{los}^n vector can be found by solving the following two equations.

$$[x_{los} - x(t)]^2 + [y_{los} - y(t)]^2 = R^2 \tag{3.16}$$

$$tan(\alpha_k) = \frac{y_{k+1} - y_k}{x_{k+1-x_k}} = \frac{y_{los} - y_k}{x_{los} - x_k} = constant \tag{3.17}$$

As mentioned there is also a method called "lookahead based steering" which has the LOS vector pointing at some point located a given distance along the path. This method will not be explained here, as the enclosure based method explanation should be enough to give the necessary intuition needed to understand the use of LOS guidance in this thesis.

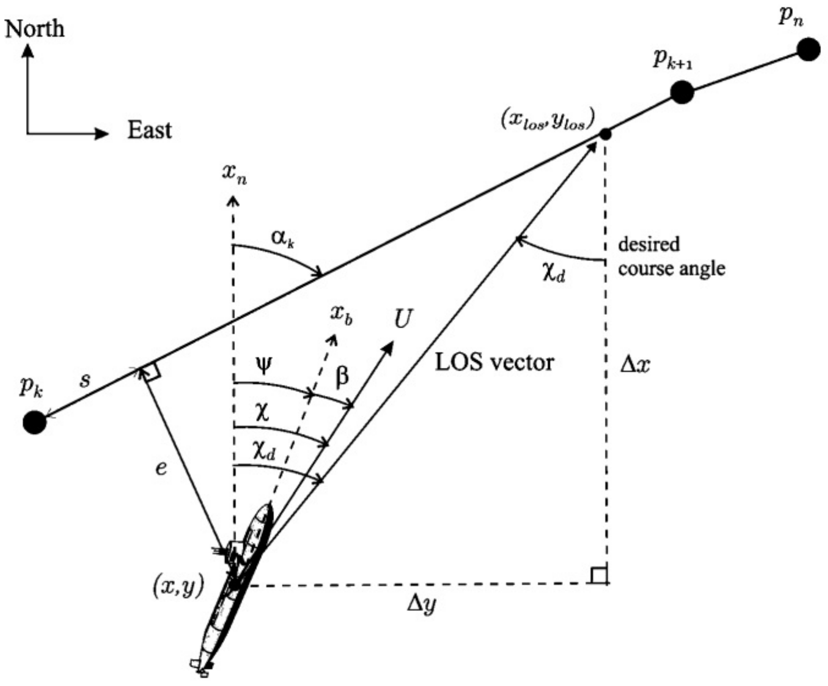


Figure 3.15: LOS guidance system [40]

CHAPTER 4 SYSTEM DESCRIPTION

4.1 Revolt

Revolt is an autonomous concept vessel designed by DNV GL. The vessel exists as both a small scale physical model and a simulated model. Though no real scale model exists today, the ship is designed to be approximately 60 meters in length. The vessel is also designed to be electric, driven by batteries stored aboard the vessel. As well as supplying the vessels thrusters, the batteries also supply a multitude of sensors that makes autonomy possible.

On the subject of thrusters, Revolt has two driving propellers in the aft as well as an Azimuth thruster further towards the bow. These thrusters can be used for dynamic positioning [41] as well as helping the vessel turn in tight spaces [20]. According to DNV GL, the main motivation behind Revolt is the potential financial gain and reduced emissions from the transport sector.

4.2 Simulation

The simulated model of the Revolt is developed by DNV GL and can be considered a "Digital twin" of the physical model, according to DNV GL. This is because the simulated model has been tested and compared to the small scale physical model to ensure that the two are as similar as possible. DNV GL has done a multitude of tests and experiments to make sure that this is the case. Because of this, all results gathered in the simulated model can be expected to be equal to results from an experiment on a physical model. This makes the simulated model a very powerful tool for testing varying solutions for autonomy. It also opens for easier testing on the small scale physical model if desired.

The simulated environment comes equipped with several features that allow users to control the vessel. For instance, a LOS guidance system has already been implemented. This means that it is possible to simply design a set of waypoints, and the ship will generate a path and move along these by itself. There are also a set of manual controls available.

```
sims[0].val('manualControl', 'UManual',Desired_speed)
sims[0].val('manualControl', 'PsiManual',Desired_heading)
```

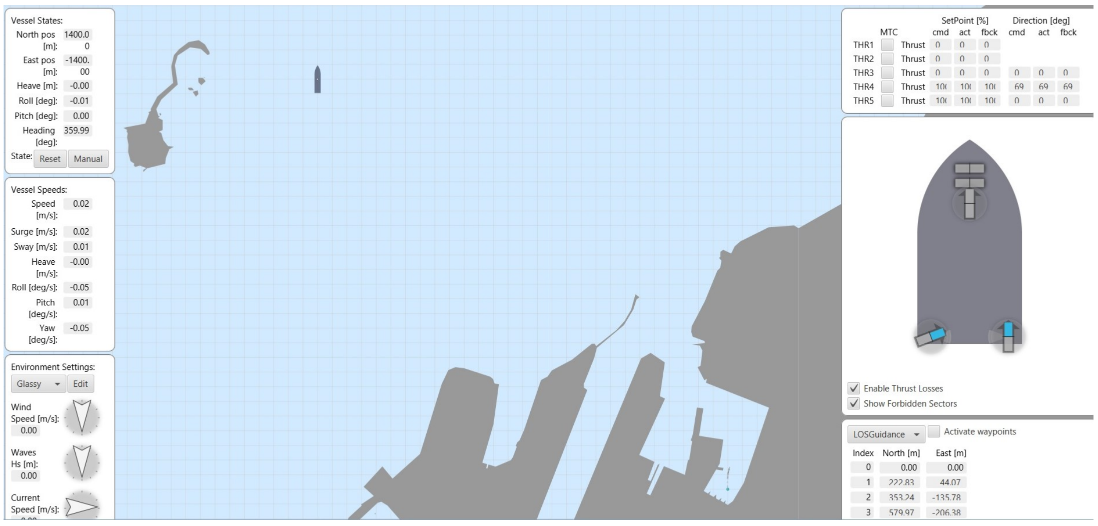
These are used to control the speed and heading of the vessel. The *UManual* command simply sets the desired speed, while the *PsiManual* command takes a desired heading reference. This reference is fed into a PID controller that minimises the error between the desired and actual heading.

There are also features to simulate different types of environmental disturbances, as well as traffic in the form of other vessels. In regards to environmental disturbances, there are several settings which vary the intensity of the disturbance. These are as follows:

Degree of disturbance	Low	Medium	High	Extreme
Low	Glassy	Slight	Very Rough	
Medium	Rippled	Moderate	High	
High	Smooth	Rough	Very high	
Extreme				Phenomenal

Table 4.1: Degrees of disturbance available organized by column

In regards to traffic, it is possible to insert other vessels that follow a given path. This path can be chosen as desired or generated to purposefully intersect with the controlled ship's path. This feature can be utilised when designing collision avoidance to test how a system reacts if another ship enters its path. In regards to the problem posed in this thesis, it is interesting to see if the agent can learn to avoid other vessels by itself, given it knows their position.


Vessel States:

North pos [m]:	1400.00
East pos [m]:	-1400.00
Heave [m]:	-0.00
Roll [deg]:	-0.01
Pitch [deg]:	0.00
Heading [deg]:	359.99

State: Reset Manual

Vessel Speeds:

Speed [m/s]:	0.02
Surge [m/s]:	0.02
Sway [m/s]:	0.01
Heave [m/s]:	-0.00
Roll [deg/s]:	-0.05
Pitch [deg/s]:	0.01
Yaw [deg/s]:	-0.05

Environment Settings: Glassy Edit

Wind Speed [m/s]: 0.00

Waves Hs [m]: 0.00

Current Speed [m/s]:

			SetPoint [%]			Direction [deg]		
	MTC		cmd	act	fbck	cmd	act	fbck
THR1	☐	Thrust	0	0	0			
THR2	☐	Thrust	0	0	0			
THR3	☐	Thrust	0	0	0	0	0	0
THR4	☐	Thrust	10(	10(	10(	69	69	69
THR5	☐	Thrust	10(	10(	10(	0	0	0

☑ Enable Thrust Losses
☑ Show Forbidden Sectors

LOSGuidance ☐ Activate waypoints

Index	North [m]	East [m]
0	0.00	0.00
1	222.83	44.07
2	353.24	-135.78
3	579.97	-206.38

Figure 4.1: Simulation environment

4.3 Tensorflow

The neural networks utilised in this thesis was designed using Tensorflow, which is a software library for building and deploying neural networks. It was originally created by a division within Google called Google Brain Team [42] for internal use, but was later released as a free open-source library. Using the library cuts down development time significantly and makes tuning the architecture and variables much easier. Tensorflow has also made neural networks far more accessible to the general public and has helped companies and amateur developers create AI solutions since its launch. It is designed to run on multiple CPUs, GPUs and mobile operating systems, and can be coded in several coding languages.

Tensorflows main application is machine learning and Deep Neural Networks. It has a C/C++ backend, which results in it running faster than if it utilised a python backend, for instance. Tensorflow is also based on data flow graphs. A data flow graph has two basic units, nodes and edges. The nodes represent a mathematical operation and the edges represent the tensors, which are multidimensional arrays [42; 43].

When using the Tensorflow library to create a neural network one usually starts by creating a placeholder. A placeholder is a variable that will be assigned data at a later time. These are created for the variables that we wish to feed into the nodes of the graph. The user can then design the neural network architecture, create a session and run it to train the network, with relevant input.

CHAPTER 5

DESIGN AND IMPLEMENTATION

Different strategies were implemented and tested in an attempt to solve the docking problem presented in this thesis. This section will present the most significant strategies implemented. Some of these strategies did not yield any results but were steppingstones in order to get to a functional implementation. These strategies will also be presented in this section along with the initial planned functionality and an explanation on why they did not function.

The first strategy was based on Dyna Q-learning and involved the agent choosing waypoints for the LOS guidance to follow. The environment was first discretized into a grid of $100x100$ cells. Potential waypoints were then placed on the corners of each cell. The agent was then given the option of choosing between the waypoints surrounding the vessel, see Figure 5.1. After the choice was made the LOS guidance system would take over and guide the vessel to the waypoint. When the vessel arrived at the chosen waypoint the agent would choose a new waypoint, and so on.

The agent would be rewarded for reaching the goal area and penalized for hitting land. In order to encourage the agent to find the most optimal path to the goal area, the agent would also receive a small penalty for moving. Since the agent also had the option of keeping the vessel still this also helped the agent to avoid getting stuck.

Due to the design of the LOS guidance system, it was necessary to generate a fake waypoint to make the vessel move. This fake waypoint was simply a transposed point further away from the chosen point. This was necessary due to the LOS system being designed to move to the final waypoint extremely slowly. Creating a fake waypoint was an easy

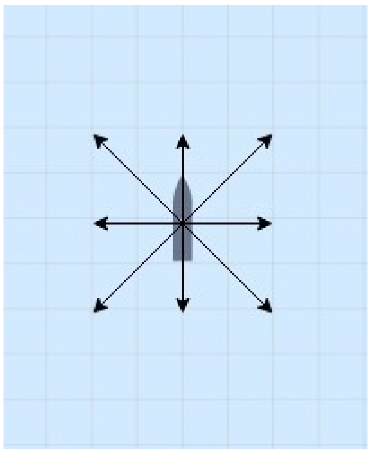


Figure 5.1: First vessel movement based on agent interaction

way to circumvent the problem. The intent was for the vessel never to reach the fake waypointpoint, as it would choose a new waypoint when the first, not the fake, waypoint was reached.

Initially, the strategy seemed to function as intended. The vessel would move as desired, and the transition between waypoints was done in a smooth fashion, without the vessel alternating between slowing down and speeding up. An issue revealed itself after a significant amount of training however. Apparently, there was a chance that the vessel would somehow miss the circle of acceptance for the desired point and simply proceed to the fake waypoint before halting. The reason for this is believed to be the LOS systems circle of acceptance being far larger than desired. Due to a conflict between the agent and the LOS system, the LOS would regard the desired point as reached, while the agent would not. At this stage of the project, the LOS guidance system was unavailable for users so the circle of acceptance radius could not be changed so easily.

There was another issue with the training strategy implemented. The strategy at this point was to simply allow the agent to move around until it found the goal area and see if it converged to an optimal path. However, the goal area was placed in the docking space itself which is extremely difficult to find, as can be seen in Figure 4.1. Though the agent at some point would be able to find the goal area, this would take a very long time.

Since the LOS guidance system circle of acceptance could not be changed at this point the strategy was abandoned. It was decided that a different approach was needed to avoid the issue with the vessel passing the desired point. The training process also needed to be reworked, as the current method was deemed to take to long, regardless of strategy. The next implementation tried to avoid the first problem by simply not utilising the LOS system at all

In the second implementation the agent would control the lower level controllers more directly. The agent's action set consisted of either moving forward, turning or standing still. The state description was the position, heading, velocity vector and velocity vector angle. The environment was again discretized into a grid world of $100x100$. One move forward would take the vessel from one square to the other. This was a simple way of having to have the forward movement of the ship be relatively consistent. The turning action would turn the vessel ten degrees either left or right, and the wait action would have the ship stay in place.

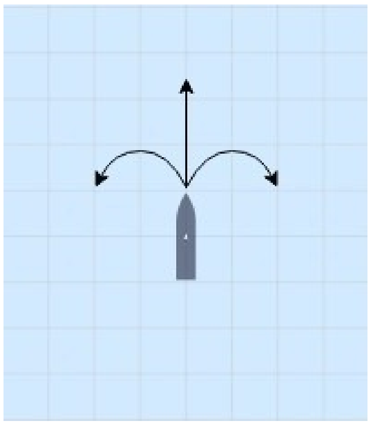
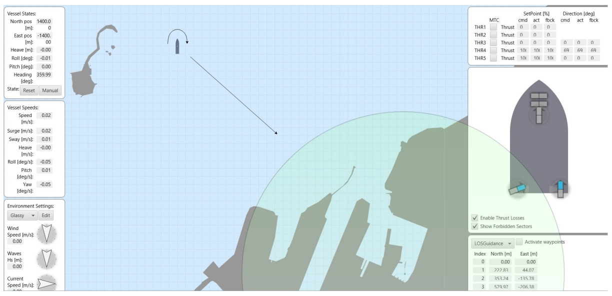


Figure 5.2: Second vessel movement based on agent interaction

The agent was also changed from being a simple Dyna Q-learning agent to a Deep Q-learning agent. The change was done in order to manage the significant increase of states. It has been proven from other implementations that DQN has significant advantages when working with large state spaces [30]. The network was implemented as a multi-layered, fully connected dense network with *tanh* as an activation function. The state values were normalized between minus one and one so no information would be lost when being passed through the network. And so that no state variable would dominate the network. Multiple network structures were tested on this implementation. The structures ranged from two to three hidden layers, with various amount of nodes per layer. The longest running training process that yielded the most promising results was with three hidden layer with 50, 150 and 50 nodes respectively.

The training strategy also changed with this implementation. The idea was to divide the testing into several scenarios and train the agent on them separately, This would reduce training time significantly, as the goal areas could be made easier to find, and several scenarios could be trained simultaneously. The underlying concept was based on that proving that the agent could solve the scenarios separately, also proved that it could solve them collectively.

Three core training scenarios were devised. The first scenario had the agent attempt to take the vessel from outside the port area to close proximity to the port entrance, as seen in Figure 5.3a. The second scenario continued from where the first scenario ended. Placing the vessel in the entrance of the port and having the agent find its way to the docking station, as seen in Figure 5.3b. The final scenario would be a repeat of the second scenario, only now there would also be an enemy vessel trying to exit the port.


Vessel States:
North pos [m]: 1400.00
East pos [m]: -1400.00
Heave [m]: -0.00
Roll [deg]: -0.01
Pitch [deg]: 0.00
Heading [deg]: 359.99
State: Reset Manual

Vessel Speeds:
Speed [m/s]: 0.02
Surge [m/s]: 0.02
Sway [m/s]: 0.01
Heave [m/s]: -0.00
Roll [deg/s]: -0.05
Pitch [deg/s]: 0.01
Yaw [deg/s]: -0.05

Environment Settings: Glassy Edit
Wind Speed [m/s]: 0.00
Waves Hs [m]: 0.00
Current Speed [m/s]:

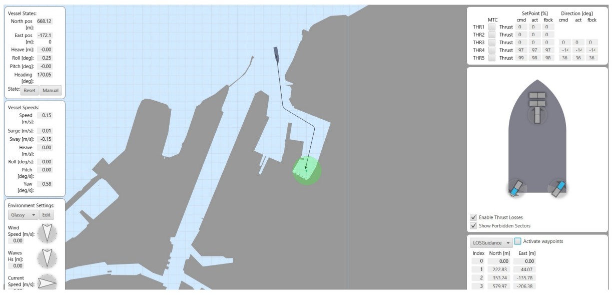
	MTC		SetPoint [%] cmd	act	fbck	Direction [deg] cmd	act	fbck
THR1		Thrust	0	0	0			
THR2		Thrust	0	0	0			
THR3		Thrust	0	0	0	0	0	0
THR4		Thrust	10(	10(	10(	69	69	69
THR5		Thrust	10(	10(	10(	0	0	0

✓ Enable Thrust Losses
✓ Show Forbidden Sectors

LOSGuidance ☐ Activate waypoints

Index	North [m]	East [m]
0	0.00	0.00
1	222.83	44.07
2	353.24	-135.78
3	579.97	-206.38

(a) Simple "get to port" scenario


Vessel States:
North pos [m]: 668.12
East pos [m]: -172.10
Heave [m]: -0.00
Roll [deg]: 0.25
Pitch [deg]: -0.00
Heading [deg]: 170.05
State: Reset Manual

Vessel Speeds:
Speed [m/s]: 0.15
Surge [m/s]: 0.01
Sway [m/s]: -0.15
Heave [m/s]: 0.00
Roll [deg/s]: 0.00
Pitch [deg/s]: 0.00
Yaw [deg/s]: 0.58

Environment Settings: Glassy Edit
Wind Speed [m/s]: 0.00
Waves Hs [m]: 0.00
Current Speed [m/s]:

	MTC		SetPoint [%] cmd	act	fbck	Direction [deg] cmd	act	fbck
THR1		Thrust	0	0	0			
THR2		Thrust	0	0	0			
THR3		Thrust	0	0	0	0	0	0
THR4		Thrust	97	97	97	-14	-14	-14
THR5		Thrust	99	98	98	36	36	36

✓ Enable Thrust Losses
✓ Show Forbidden Sectors

LOSGuidance ☐ Activate waypoints

Index	North [m]	East [m]
0	0.00	0.00
1	222.83	44.07
2	353.24	-135.78
3	579.97	-206.38

(b) Getting to docking station scenario

Figure 5.3: Training scenarios

The second implementation was only tested on the first scenario in a significant degree before being abandoned. While the implementation functioned as intended it was abandoned due to an issue with the simulator. After some 5000 episodes the agent had found the correct general direction to move and had a goal vs failure hit-rate of 25%. Though the results seemed somewhat promising, these 5000 episodes took close to one week to complete. This was due to the fact that the simulator was not optimised for the purpose of training deep neural networks.

Being a digital twin of the Revolt concept ship, the model was extremely mathematically dense. The simulator also lacked any functionalities for speeding up the process. As such, training the agent would take a significant amount of time. In an attempt to combat this problem efforts were made to make the DQN agent more advanced, hoping this would significantly speed up the training process. Fixed Q-targets, prioritized experience replay (PER) and a double DQN strategy was implemented in an attempt to make the training process go faster. However, the efforts did not have the desired effect.

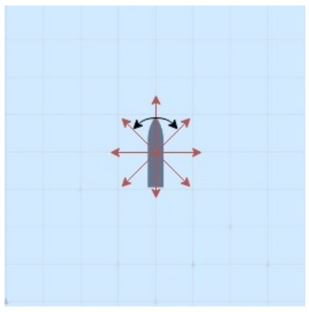
Despite the issues with time constraint an effort was made to try and train the agent on the second scenario as well, hoping this might go faster as the state space was much smaller. At this point, the problem with non-convexity had not been thought of so the agent was placed randomly around the port area at episode start and expected to find the goal area. Due to the difficulty in finding the goal area the agent would for the most part stand, or turn, in place in order to survive the longest amount of time. The agent never got to the point of finding the goal area at a consistent rate. As such, the implementation strategy was abandoned in order to figure out a way around the slow simulator problem

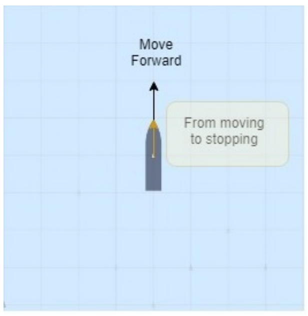
As running the vessel was such a slow process, the focus shifted to finding a strategy that would prove DQN could solve the problem without simulating the actual vessel movement. With this in mind, a new strategy was developed based somewhat on the first implementation. If the agent was regarded as a high-level agent only, there was no need to actually move the ship to make any decision. The agent could in fact simply design actions the vessel should take, trusting that the LOS guidance system would be able to execute the chosen action.

Based on the action set and potential waypoints from the first strategy, see Figure 5.1, the agent would now choose a point, check if it was on land or not, then choose a new point without the vessel moving. This way the agent would be able to learn the environment and find the optimal actions, without simulating the vessel movement. Once this was done, a path could be generated for the LOS guidance system to follow. The initial state description for this strategy was the vessel position. However, it was realised that the agent might benefit from having some information on where it was going and that this might speed up training. The vector between the vessel position and the goal area was therefore also added to the state description. The DQN agent from the second implementation, which included PER, fixed Q-targets and double DQN, was kept for the remainder of the thesis.

The strategy was tested on the three scenarios outlined earlier, and gave the most significant and promising results of all the tested strategies so far. The results themselves will be expanded upon in the Chapter six. As the results were far better using this strategy then any of the others, it was decided to focus on this strategy and try tuning the network parameters and optimize the performance. The number of hidden layers, learning rate, target network update rate and varying the information in the state description were all tuned and tested.

The action set was also changed to try and incorporate the dynamics of the vessel. This was done by first running a set of tests in the simulator to find the vessels turning radius, speed up and stopping distance. These variables were then used to make it so that the agent's actions mimicked the behaviour of the vessel. When using this action set, the vessel heading, and a variable expressing whether the vessel was moving or standing still (*moving)*, was included in the state description. This action set will be referred to as the *dynamic actions* from here on.


(a) Turning motion while standing still



(b) Transitioning: standing-moving

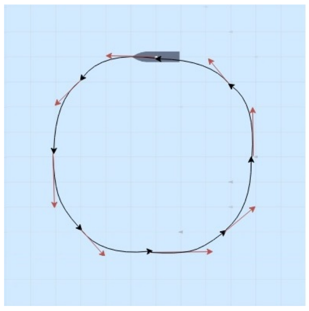

(c) Turning while moving

Figure 5.4: Dynamic actions explanation

The Figure 5.4 shows how the dynamic actions interact with the vessel. Note that in Figure 5.4c and 5.4a, the red arrows indicate the heading of the vessel after executed action. In Figure 5.4b, transitioning from stop to moving gives the same result as "Move Forward".

In a sense, the vessel now had two different modes, moving and standing still. Depending on the mode, the vessel would behave differently when a given action was executed. If the vessel was standing still, $moving = 0$, the vessel would be able to turn in place and could stand still if desired, see Figure 5.4a. If the vessel was moving, $moving = 1$, the turning radius would be around $200meters$ for a full circle, see Figure 5.4c. Transitioning from moving to standing still would take $50meters$, and when moving forward the vessel would move $100meters$ at a time, see Figure 5.4b.

A problem arose when trying to solve the second scenario. The agent seemed incapable of finding the goal area consistently. This was most likely due to the complex nature of the environment. It turned out that the agent could find its way to the designated goal area as long as it did not start in the docking space next to the goal. When the agent ended up here it would try to enter the goal area directly and crash into the land area separating the two docking spaces.

Several different strategies were implemented to try and solve this issue. One such strategy was to try and give the agent information on land areas as it discovered them. The agent state description would be made far larger by adding multiple state descriptions initialized at zero. The agent would then add the points it had crashed to the state description as it discovered them, replacing the zeros. Adding the vector to all the points the agent had crashed while exploring was also implemented and tested. However, none of these methods yielded any significant improvements.

It was recognized that the non-convexity of the area was a problem. Though the agent would in time learn to avoid the land area separating the two docking spaces, this would take far longer than desired. The solution to the problem was to split the area into two areas, which were as convex as possible, and train the agent on one first, before adding the other.

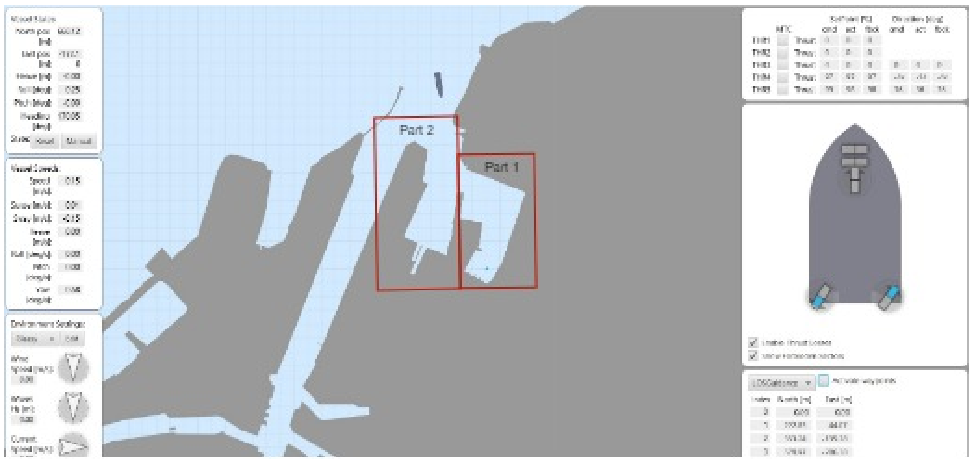


Figure 5.5: More-convex training area

This was done by restricting the training area for the agent as shown in Figure 5.5. The red line in Figure 5.5 separating the two parts can be thought of as a gate. At first, the agent starts training in the first area, labelled *Part one* in the figure. When some condition is fulfilled the gate opens. When this happens the agent starts each episode in the second area, *Part two*, and has to find its way to the goal area. The condition that needs to be fulfilled is designed to prove that the agent is sufficiently trained in the first area. In this implementation, this was simply hitting the goal area 20 times in a row. The neural network is able to remember what was learned in the first area when starting in the second. This means that once the agent finds its way back into the first area it already knows what to do. Hence the ”goal” in the second area can be regarded as getting to the first area, rather than getting to the goal area.

During testing the DQN suggested in the paper by DeepMind [30] was also implemented. This was done mainly to be used as a comparison for results, as this method is proven to work well on similar problems. Using the *skimage* package for python, a screenshot capturing the agent's working environment was taken. This image was then cropped, grayscaled and normalized before being passed to the network. The network itself was changed to a convolutional network as outlined in the paper by Deepmind. The results from this implementation can also be found in the section on Results.

CHAPTER 6

SIMULATION RESULTS

6.1 Introduction

The final strategy was tested on three different scenarios using two different action sets, one based on the first strategy, see Figure 5.1, and one also taking the vessel dynamics into account. These results were then recorded and will be presented and compared in this section. These two types of action sets will be referred to as *simple actions* and *dynamic actions*.

The agent is rewarded for hitting the goal area and penalized for hitting land or moving out of bound. A small energy penalty was also added to keep the agent moving towards the goal and incentivise it to find the optimal path. The *wait* action had a slightly lower energy penalty associated with it. This way, the agent would be encouraged to wait in place for traffic to pass instead of moving back and forth or turning in place. When using the dynamic actions the agent would also receive a penalty for trying to execute actions that were deemed unavailable. An example of this is trying to wait while $moving = 1$, the agent would have to first stop, $moving = 0$, then wait, to avoid this penalty.

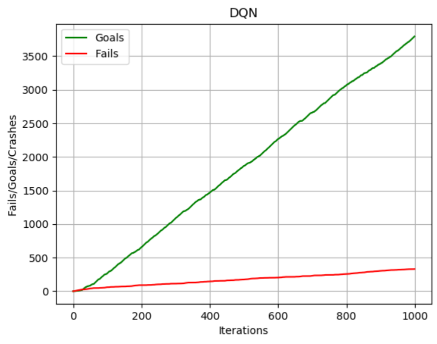
Hitting the goal area would not end the episode, however. To allow the agent to accumulate positive reward the agent had to hit the goal a set number of times before the episode would end. Unless specified otherwise, the state description in the following section consisted of the vessel position and the distance to the goal area when using the simple actions. While the dynamic actions also included heading and the moving variable, telling the agent whether it was moving or standing still. In Scenario three the enemy vessel position was added. Note that these variables were also present in the first and second scenario, however, they equalled zero as there were no traffic included.

In addition to presenting the accumulated goals versus failures and episode score graphs, scenario three will include a series of images explaining the movement of the vessel. Only Scenario three will present these images as Scenario two is the same problem, only simplified, and the solution to Scenario one is sufficiently explained in Figure 5.3a. At the tail-end of the thesis there was also time to run a few additional scenarios, these will be presented briefly in the end of this chapter.

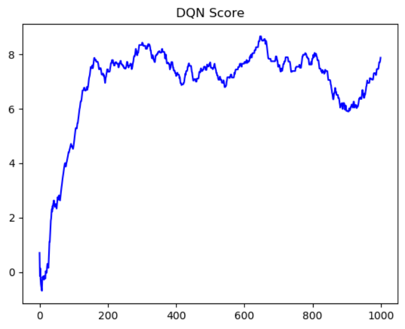
6.2 Scenario One

Scenario One: Simple Actions

The first scenario was solved with ease by the agent. The network in this scenario was not very sophisticated, consisting of only a single hidden layer with 100 nodes using a *tanh* activation function. however, the agent managed to find the goal area within a few minutes and kept consistently hitting the goal from there, see Figure 6.1.



(a) Goals versus fails



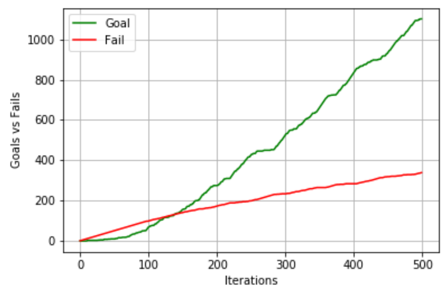
(b) Score

Figure 6.1: Results for first scenario with simple actions

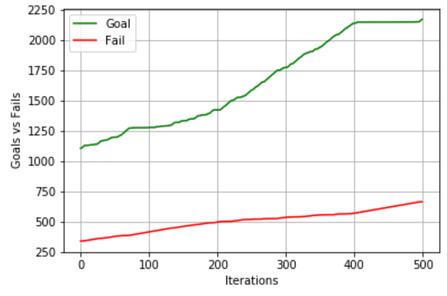
Figure 6.1 also illustrates the learning process of the agent in the first 1000 episodes. The score plot makes it clear that the agent quickly learns which actions improve the accumulated reward, and reaches an average of around eight points per episode. It can also be seen that there are some elements of exploration in the figure, especially around episode 900. Here we see a dip in the accumulated score, most likely due to the agent finding itself in a less explored area of the map. We can also observe that it bounces back after a few episodes, as the score increase.

Scenario One: Dynamic Actions

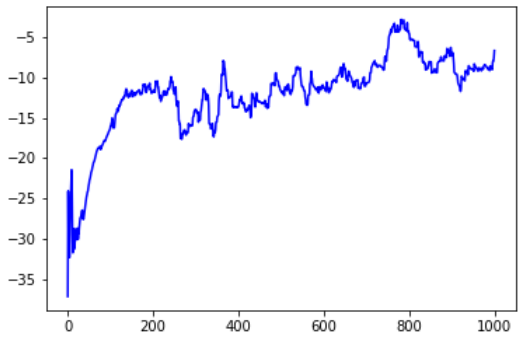
When including the dynamics of the vessel the results were equally promising. In Figure 6.2 we can observe the results from the agent training in the first scenario. As shown in the graph, the agent finds the goal state relatively fast and consistently steers the vessel this way. There are some failures along the way and we see an uptick in fails at 400 episodes in the second run. This is not completely unexpected, however. With dynamic actions, the agent is far more complex and the state space is also far larger. It is possible that the dip we see is due to underfitting, as the network still only has a single hidden layer with 100 nodes.



(a) First 500 runs



(b) Second 500 runs

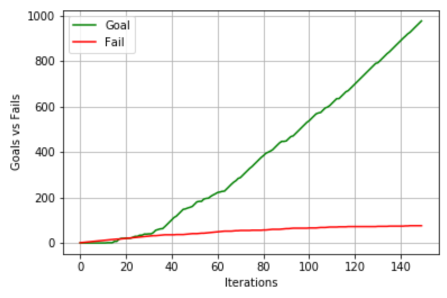


(c) Score for first and second run

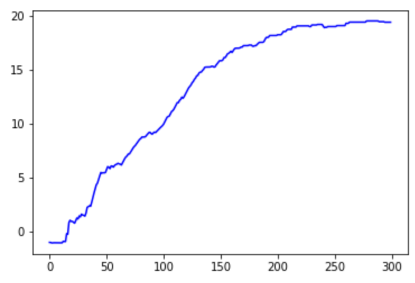
Figure 6.2: First and second run the first scenario with dynamic actions

Scenario One: Revisit

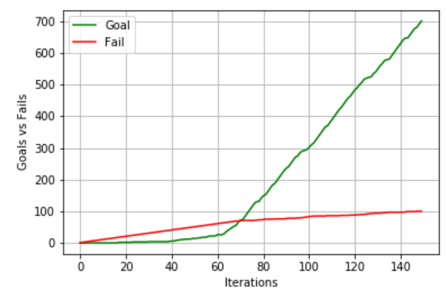
As we will see when we get to the second scenario, the neural network structure had to change to complete this part. It was therefore necessary to revisit the first scenario and verify that the results were still valid. In this test the episode would end if the agent went out of bound, hit land, or hit goal five times in secession.



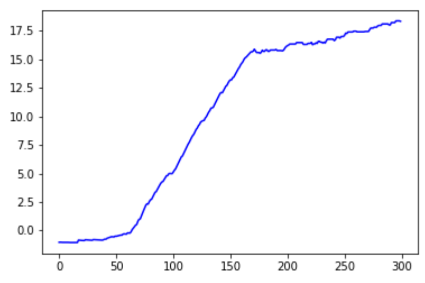
(a) Goals versus fails simple actions



(b) Score simple actions



(c) Goals versus fails dynamic actions



(d) Score with dynamic actions

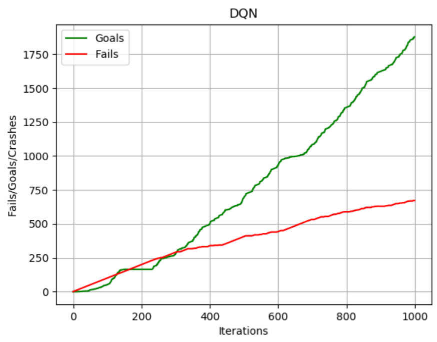
Figure 6.3: Results form first scenario using final network structure

As we can see in the results in Figure 6.3, the results are more or less the same this time around. From the plots we can observe that the agent using the dynamic actions takes longer to learn the environment, which is expected. It is also expected that the score for the agent using the dynamic actions will surpass that of the agent using the simple actions at some point. This is due to the fact that the dynamic action agent needs fewer actions to find the goal. Therefore the agent accumulates less energy penalty. However, the simulation was not run long enough to reflect this, as the results show in Figure 6.3 was deemed more than adequate.

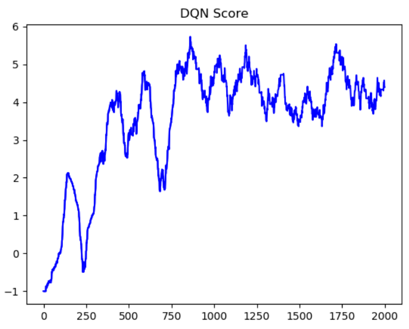
6.3 Scenario Two

Scenario Two: Simple Actions

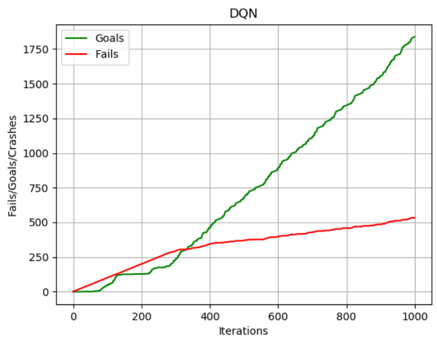
When applying the agent to the second scenario, things turned out to be a bit more difficult. This is where the non-convexity problem came into play, and the agent struggled to find the goal area when starting in the docking area neighbouring the goal area. However, this was solved by implementing the more-convex area separation outlined earlier and shown in Figure 5.5. The following results are from using this training method.



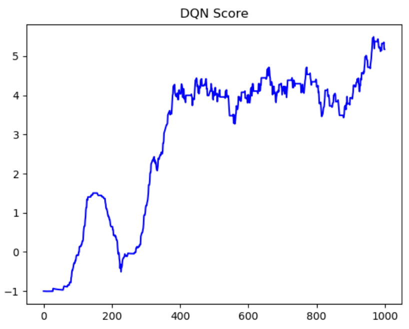
(a) Goals versus fails without PER



(b) Score without PER



(c) Goals versus fails with PER



(d) Score with PER

Figure 6.4: Results for second scenario, with and without prioritized experience replay (PER)

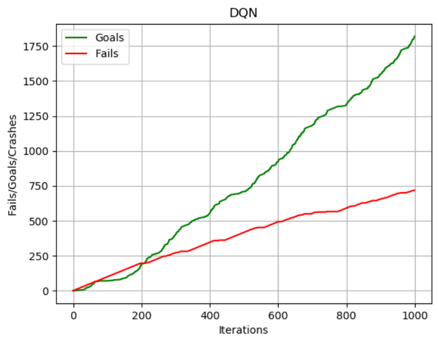
While testing this scenario multiple network structures were tested to see if an optimal one could be found. The results shown in Figure 6.4 was generated using a network consisting of three hidden layers of 64, 128 and 64 nodes respectively. The two first layers using *relu* activation function and the final hidden layer using the *tanh*. This was the network that yielded the best results in this scenario and is considered the final network structure for a viable strategy.

As a test, the results are gathered both with and without prioritized experience replay (PER). This was done in order to show the effect of PER and to solidify that the implementation was working as it should. Looking at the graphs in Figure 6.4 depicting goals versus fails, we can see that in both cases the agent managed to hit the requirement and unlock the second area at around 150 iterations. They then both need some time before managing to find the way back to the goal area. However, when they do the agent with PER enabled is far more reliable. This can be seen from how the amount of fails flat out sooner and that the line representing the accumulated goals seems to be almost linear. The PER agent also has less accumulated failures at the end of 1000 iterations.

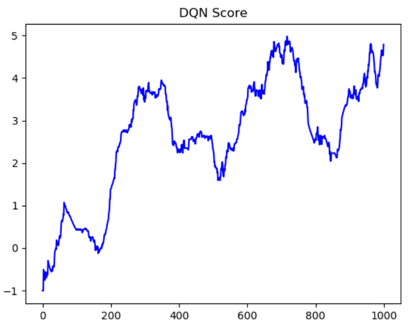
The Score graphs depicted in Figure 6.4 tell a similar story. Here we can see that the start is strikingly similar. Both agents creating a peak in accumulated points early on before decreasing. Both agents also pick themselves up after this, however, the agent with PER is far more stable. This is also despite the graph for the agent without PER being longer, depicting 1000 more iterations than the graph for the PER agent. These results prove not only that the agent is capable of solving the problem, but also that the PER has added value for the DQN network and makes it a more efficient and reliable learner.

Scenario Two: Dynamic Actions

The results from adding the vessel dynamics can be seen in Figure 6.5. As we can observe, the results show that the agent is learning as the score increases over time. When compared to Figure 6.4 we can see that both the score graph and the goal graph are somewhat more erratic. This is to be expected as the two differ a great deal when regarding the size of the state space complexity.



(a) Goals versus fails



(b) Score

Figure 6.5: Results for second scenario with vessel dynamics

The dynamic action set is far more complicated than the simple action set. Though there are fewer actions that can be taken, they behave differently depending on whether the vessel is moving or standing still. The actions also take the vessel further each time they are executed. This means that the land areas are ”closer” to the agent, in a sense. Because of this, the agent is more likely to hit land multiple times.

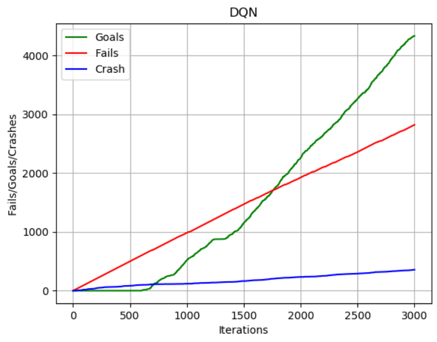
The action set has not only made the vessel movement more complicated, but also made the state space far larger. The inclusion of the moving variable alone doubles the number of states. Since we also take the vessel heading into account the state space increases eight-fold. Though the DQN is fully capable of handling this amount of states, it is to be expected that it would take more time to train the agent sufficiently

Even so, the plots suggest that the agent is learning and that both the goal graph and score graph will stabilise. Note that the element of exploration never goes completely away, as epsilon never goes lower than 0.1. This means that there will always be some variation in regards to score, and that the agent still might fail at times due to random actions.

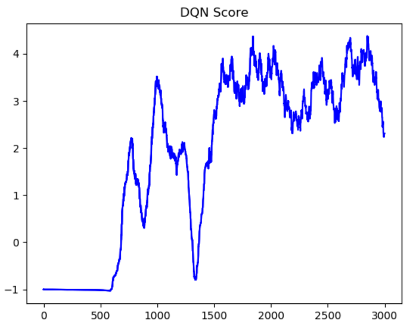
6.4 Scenario Three

Scenario Three: Simple Actions

As mentioned, the final scenario was similar to the second scenario only including an enemy vessel exiting the port area. This enemy vessel was designed to move at the same speed as the agent vessel and would move in the reverse path as depicted in Figure 5.3b. As this test was thought to be an extension of the second scenario, the network would already be trained in the environment without the enemy present. In other words, the network was first trained on the second scenario and once finished the enemy vessel was added. The results can be seen in Figure 6.6



(a) Goals versus fails



(b) Score

Figure 6.6: Results for third scenario with traffic

As we can observe from the plots, the agent did not seem to have too much trouble with finding the solution. Note that the red line in this graph represents failures in general, which also includes crashing. The blue line indicates which of these failures were collisions with the enemy vessel. Initially, the agent both crashes into land areas and the enemy vessel before figuring out what to do. The agent would most often crash with the enemy around the narrow area separating Part one of the environment from Part two. This leads to the agent thinking that this area has turned inaccessible. It then tries to find an alternative way into the goal area, which accounts for most of the failures as there only ever was one entrance. Eventually, though, it does learn the behaviour of the enemy ship and waits until it has passed before moving to the goal area. This is reflected in both the score plot and the accumulated goals.

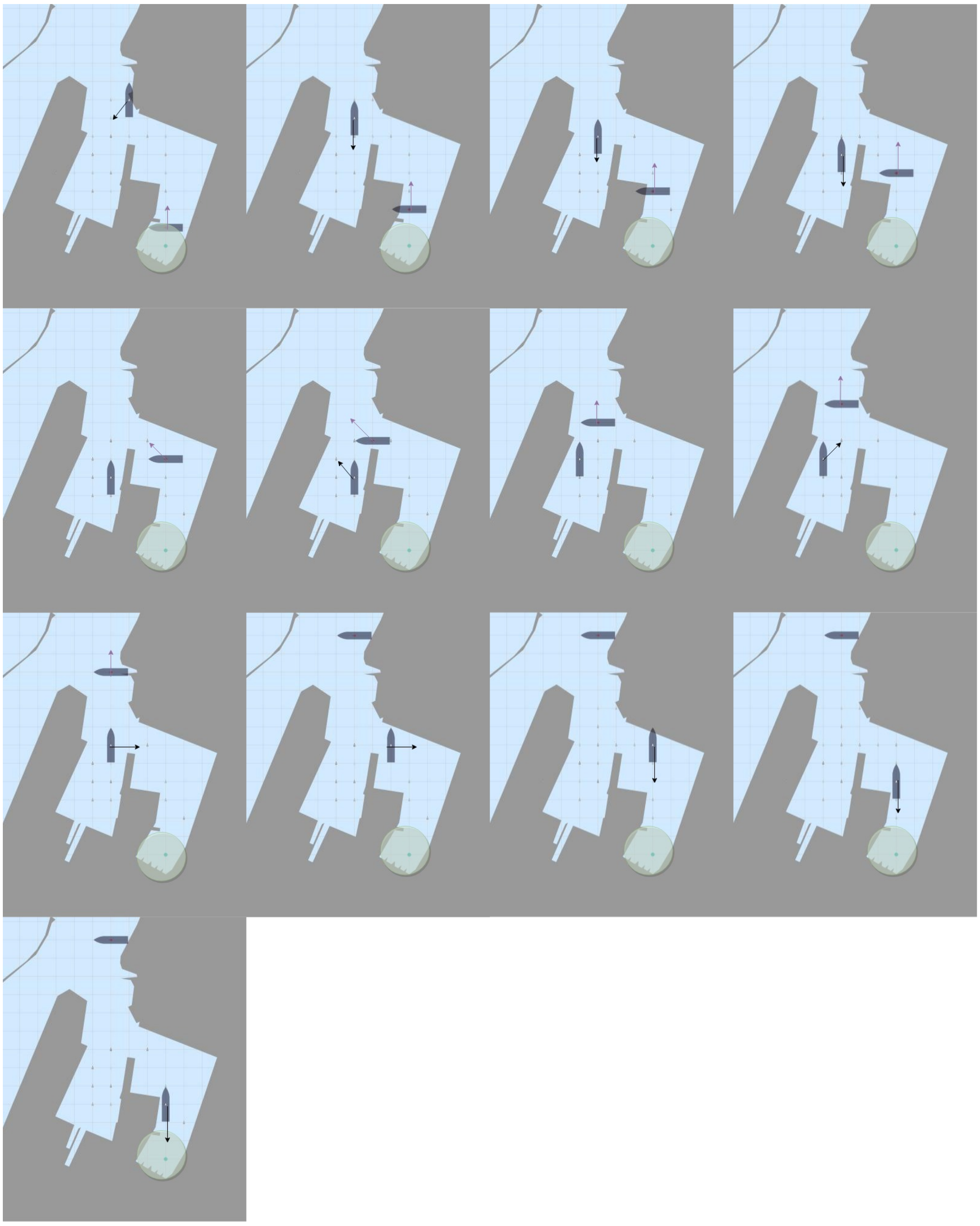
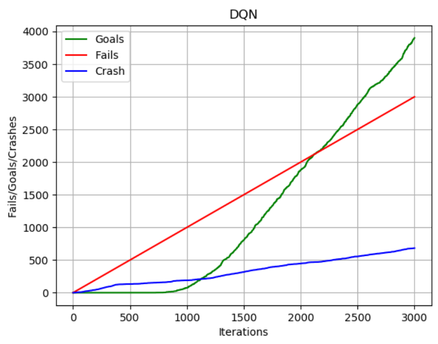


Figure 6.7: Successful episode from third scenario with simple actions

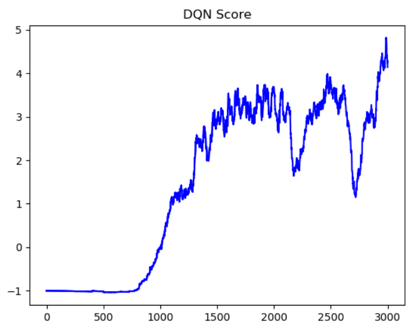
Figure 6.7 shows the agent completing the scenario using the simple actions. The enemy vessel movement is represented by the purple arrow, while the agent vessel is movement is represented by the black arrow. Reading from the top, left to right, the enemy vessels starts in the goal area and moves toward the port exit. The agent vessel starts directly in the path of the enemy vessel in this instance, and must therefor move out of the way to avoid collision. As we can see from the figure, the agent does just that. The agent can also be observed choosing to wait, instead of moving, while the enemy passes. It then moves directly to the goal area. These images were taken during the training process and is only one example of a successful episode.

Scenario Three: Dynamic Actions

Finally, scenario three was repeated again with the dynamic actions. The network remained unchanged, as did the strategy of first training the agent without the enemy and then adding it. The results recorded can be seen in Figure 6.8. These results are somewhat surprising when compared to the results from the simple actions test, seen in Figure 6.6.



(a) Goals versus fails



(b) Score

Figure 6.8: Results for third scenario with traffic and dynamic actions

It was expected that the agent would take longer to find the solution for the same reason as in scenario two. The state space is larger and the actions are more complicated, making the environment itself more complicated. Considering this, it was expected that the agent would take longer to find the correct actions to the goal area and fail more often. It is also expected that the agent would gain a higher total score with dynamic actions. This is due to the agent moving further per action, so fewer actions need to be taken to get to the goal area. However, both plots seem to suggest a far smoother transition from untrained to trained. Both the accumulated goal line and the score graph are far smoother than when the simple actions were applied.

Figure 6.9 shows an example where the agent finds its way to the goal area with traffic. Again, reading from the top, left to right, the agent is the vessel standing still in the first image. The enemy vessel can be seen moving away from the goal area, its movement indicated by the purple arrow. We can observe that the agent waits until the enemy vessel has passed before making its move. The agent vessel heading is indicated by the red arrow, and the vessel movement is indicated by the black arrow. Note that the simulated environment has no visualisation functionality unless actually simulating the vessel movement. As such it is difficult to keep track of the vessel heading. At this point, the agent is not fully trained, as the movement is not completely optimal.

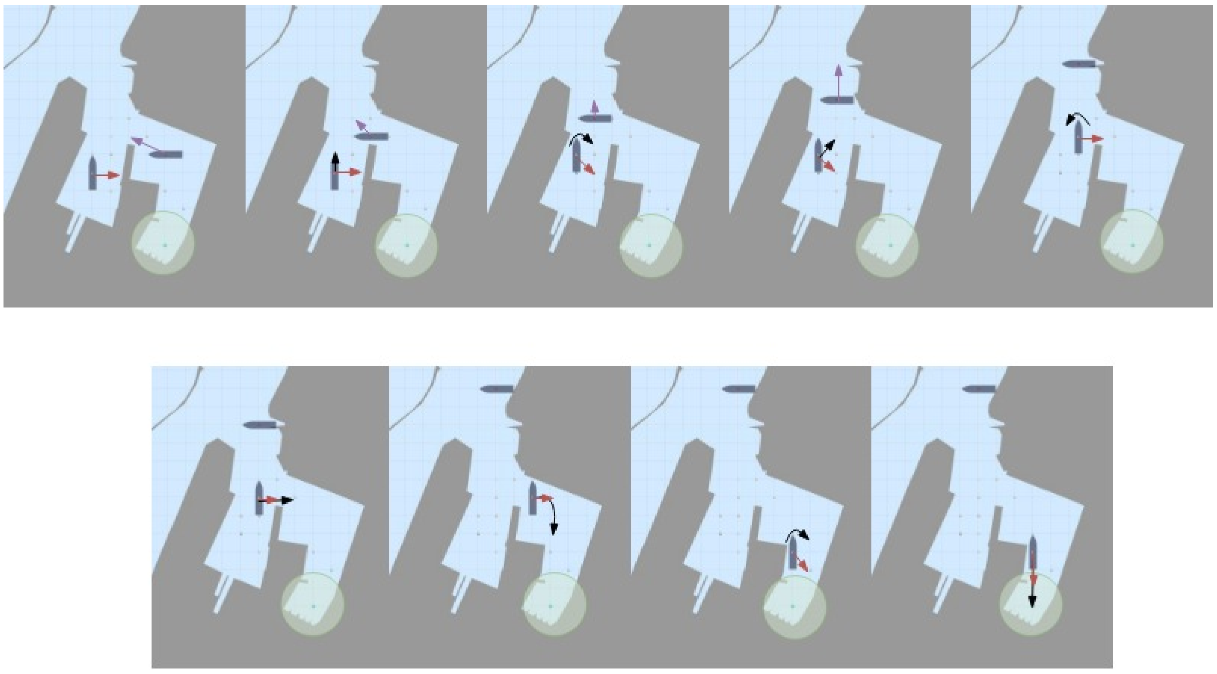


Figure 6.9: Successful episode from third scenario with dynamic actions

Not always successful, Figure 6.10 shows the agent waiting for the enemy vessel to pass in the wrong position, early in the training process. This results in a crash with the enemy vessel exiting the port. The red circle shows the area around the enemy vessel which constitutes a crash. If the agent exists inside this circle, it counts as hitting the enemy vessel. Another common failure was the agent waiting for the enemy to pass in a safe position, but close to the land area. When this was the case, exploration would kick in from time to time and send the vessel into the land area.

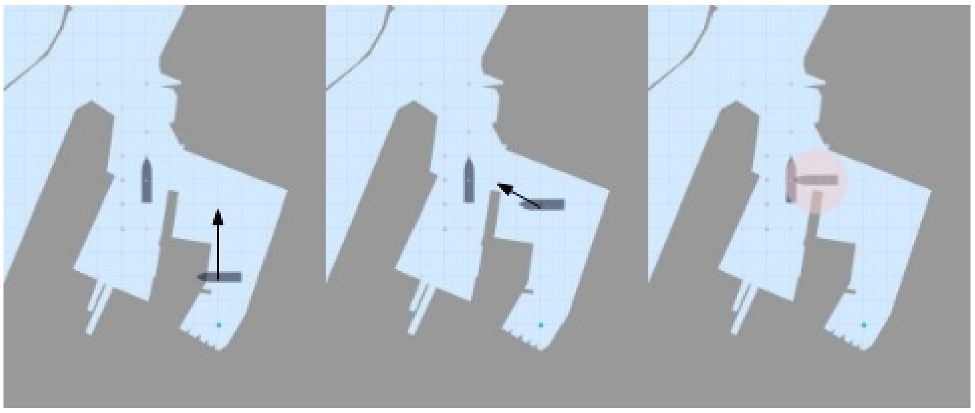


Figure 6.10: Crash from early training

Resulting Network and Variables

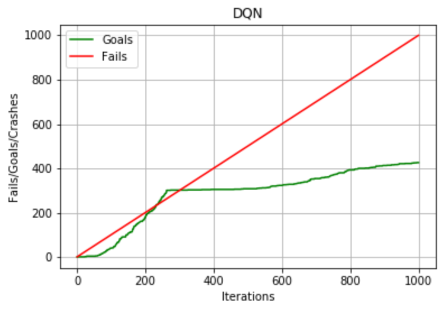
Variables		
a	0.7	Priority scale
tau	0.01	Target network update rate
gamma	0.97	Discount rate
alpha	0.001	Learning rate
epsilon	max = 1.0, min = 0.1	Exploration rate
epsilon decay	0.99	Exploration decay rate
Reward: goal	2	Reward for finding goal
Reward: fail	-1	Reward for failing
Reward: invalid	-2	Reward for trying invalid action
Reward: energy	-0.001	Energy reward
Reward: waiting	-0.0009	Energy reward for waiting
Network		
hidden1	64	First hidden layer: Activation=tanh
hidden2	128	Second hidden layer: Activation=tanh
hidden3	64	Third hidden layer: Activation=ReLU
output	action size	output layer: Activation=None
buffer	100000	Memory buffer length
batch size	120	Memory sampling size

Table 6.1: Neural network variables used in the final implementation

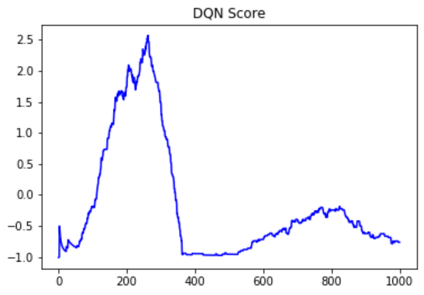
6.5 Supplementary Implementations

DeepMind DQN Implementation

As mentioned, the DQN DeepMind outlined in the paper by DeepMind [30] was also tested. There was interest in exploring how well this method would do, and use the results as a comparison for the results gathered in this thesis. The main difference between the strategy suggested in this thesis and the one implemented by DeepMind is the use of state description. The goal of this thesis was to find a solution using only information that could be gathered locally from a vessel. This differs from the DeepMind DQN which uses the pixel values from a screen as state descriptions. The results from the test can be seen in Figure 6.11. These results are from running the strategy implemented on the second scenario.



(a) Goals versus fails



(b) Score

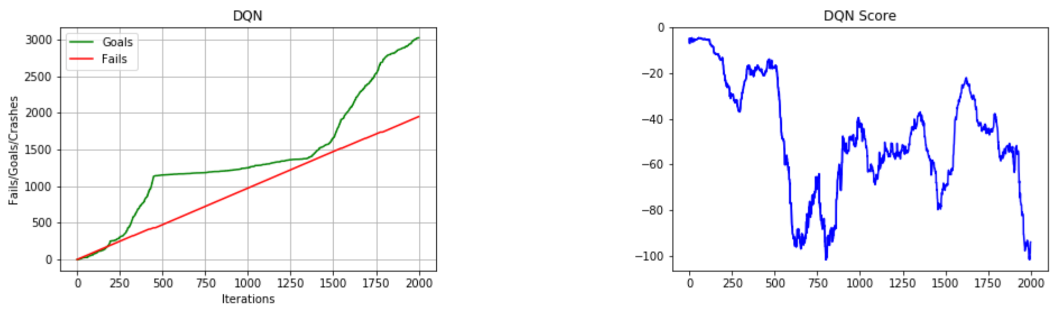
Figure 6.11: Results for DeepMind implementation

There are some differences in the method outlined in DeepMinds paper and what was implemented here. The DQN strategy from DeepMind did not include double DQN or PER, while this implementation does. Apart from this the tuning parameters, reward function and state description are practically the same. By examining the results we can see that the agent unlocked the "gate" after around 200 episodes. This is comparable to the method suggested in this thesis, which also unlocked the second area in around the same amount of episodes. However, after this, the method seems to struggle.

It is not until after another 600 episodes that the agent seems to find its way back to the goal area. And even when doing so, the increase in accumulated goals is slow. Comparing the result here to the results in Figures 6.4 and 6.5 we can see a drastic difference. The results seen in these figures had almost a linear growth in accumulated goals after 400 iterations. Though this is not completely unexpected, as the state description is far larger in the DeepMind network. Due to this, training it from scratch would take some time. It was however expected that this network would work better considering previous results presented by the DeepMind network. Though, the DeepMind networks main strength would be its versatility, as it can be applied to any similar problem as long as there is a depiction of the environment, and not its effectiveness in this specific problem.

Full Environment Guidance

Though this thesis had some time restrictions, there was time for some supplementary testing of the agent. This section contains the results from testing the DQN agent on two extra scenarios. The first supplementary test was testing the agent on the entire environment using the dynamic actions. This test was an extension of the second scenario, only that the agent would add the exterior port area when sufficiently trained inside the port. This effectively proves that the agent is capable of finding its way through the entire environment and solve the guidance part of the problem in its entirety.



(a) Goals versus fails (b) Score

Figure 6.12: Results from supplementary scenario with full environment and dynamic actions

As we see from the plots shown in Figure 6.12a, the agent has some issues when the final part of the environment was added. This area is rather large and the agent struggled for some time before finding its way back to the port. The score graph shown in Figure 6.12a also reflects this. We can observe that the agent has collected a large amount of negative reward during the exploration. This is most likely due to trying to use invalid actions, like waiting while moving, which gives a larger negative reward. However, this will only improve over time, as the graphs show that the agent is solving the environment. The Figure 6.13 shows the agent starting outside the port, close to the island. It then proceeds to find its way in to the goal area.

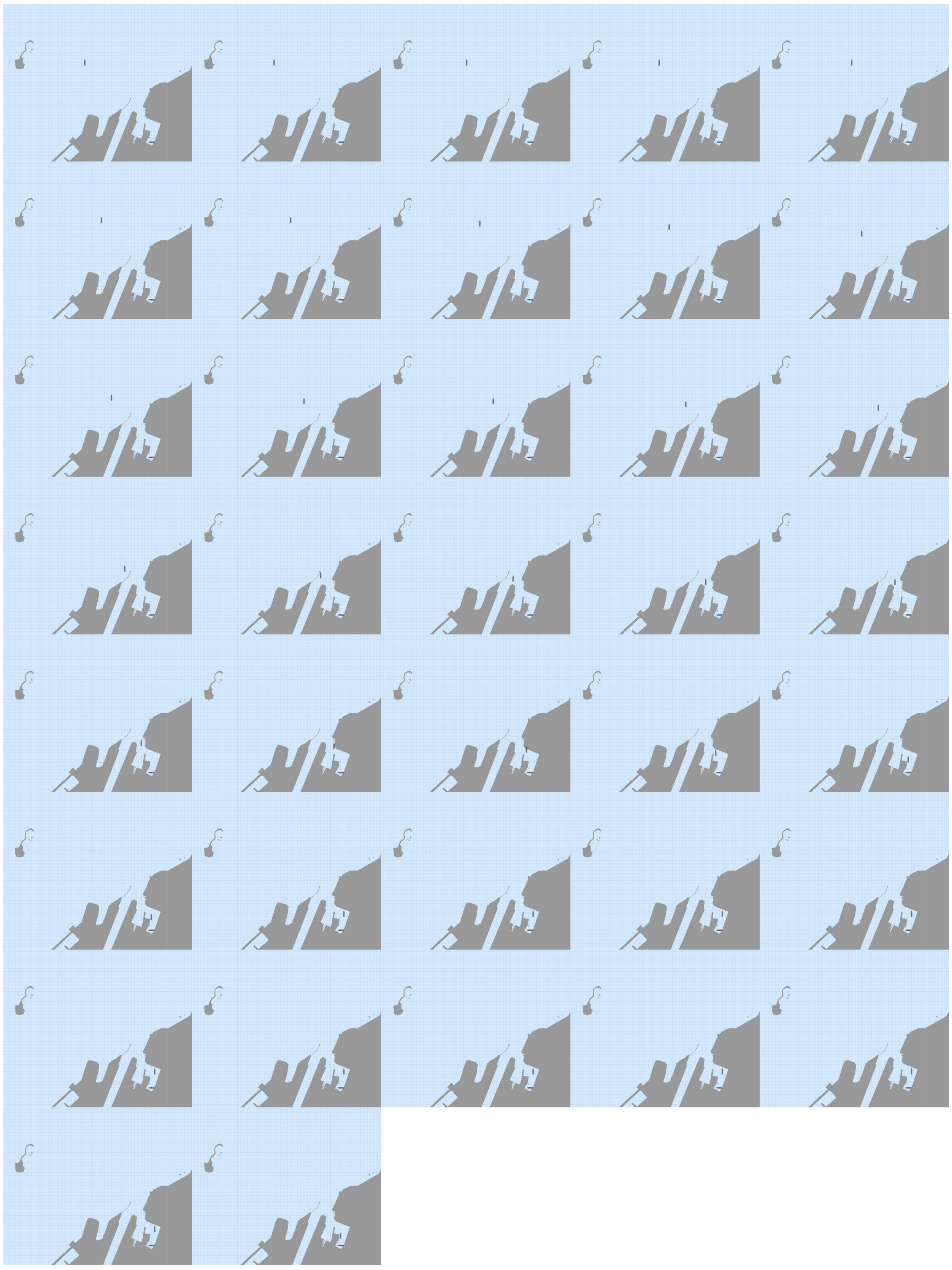
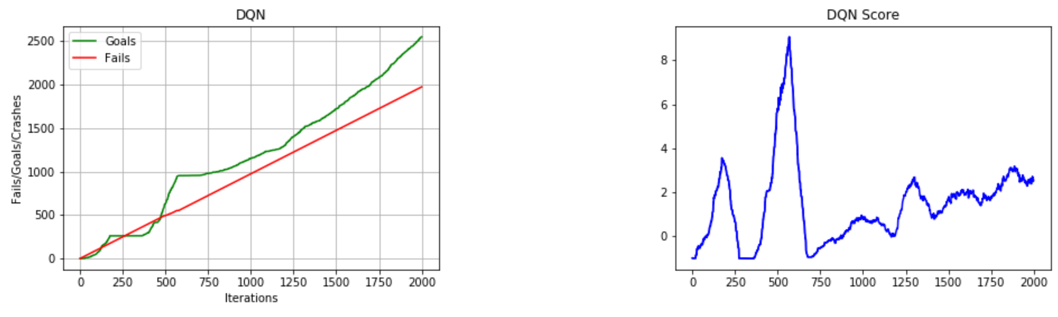


Figure 6.13: Supplementary scenario with full environment and dynamic actions

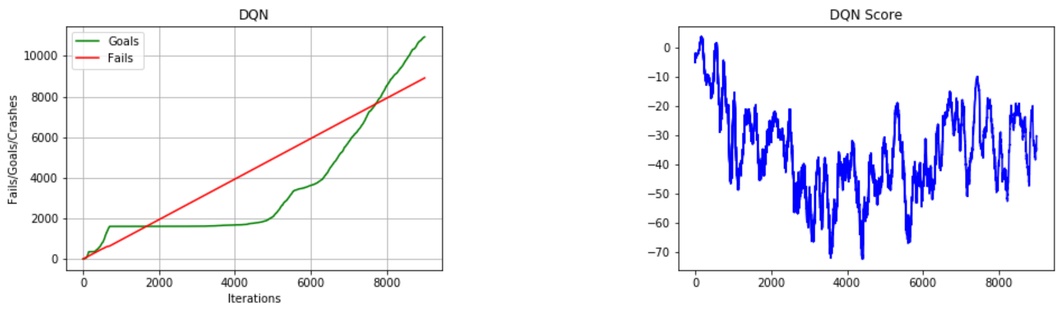
River Guidance

In the second supplementary scenario, scenario two was again expanded. Only this time the agent had to find its way to the goal by navigating the narrow ”river” area next to the port. This was done to see if the agent was capable of navigating difficult, narrow spaces. In this test the simple actions and the dynamic actions where used. Figure 6.15 shows the agent finding the path from the river to the goal area using the simple actions.



(a) Goals versus fails, simple actions

(b) Score, simple actions



(c) Goals versus fails, dynamic actions

(d) Score, dynamic actions

Figure 6.14: Results from supplementary river test

From the plots depicting in Figure 6.14 the results using the dynamic actions we can observe that the agent struggled quite a bit. The plots show that the first two areas were solved easily by the agent, using no more episodes than with the simple actions. When solving the final river area, however, the agent struggles for a long time before finding the goal area. As the river part is rather narrow the agent has problems moving through using the dynamic actions, as these actions move the agent further then the simple actions do. The agent does solve the environment eventually, and we can observe that once it does it finds the goal consistently from that point on.

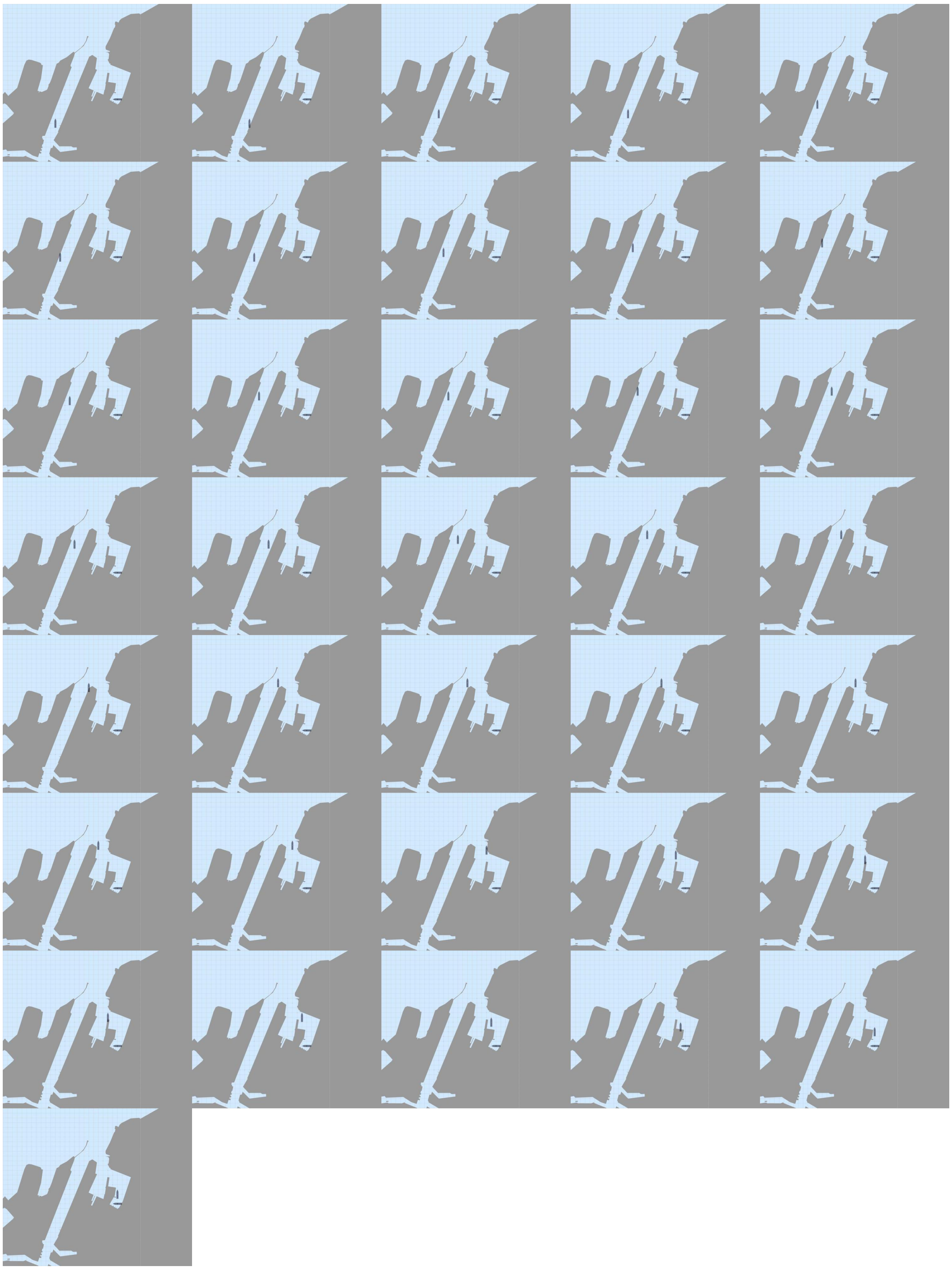


Figure 6.15: Supplementary river test using simple actions

CHAPTER 7 DISCUSSION AND CONCLUSION

7.1 Challenges

When designing and implementing any reinforcement learning solution there will always be challenges along the way, and this thesis was no exception. Several of the attempted solutions to solve the presented problem did not yield any results. Designing and implementing solutions only to see them fail can often be disheartening. It is important to remember, however, that there is also much to learn from failure. This section will discuss some of the challenges faced in this thesis.

One major challenge which is always present in RL problems is the state description, or, how does one describe the states to the agent in an effective manner? In the paper by DeepMind [30] the pixel values of the screen as used as the state description. This way, all information is afforded to the agent. If we look at the problem presented in this thesis, then using the pixel values from the simulator gives the agent information on land, obstacles, traffic, the controlled vessel and the goal area all at the same time. But how can we know that all this information is useful? By inspecting the results from the suggested network in this thesis and comparing it with the DeepMind network, it is clear that all this information is not necessary to solve the problem. This is something that will always be difficult to figure out, and something all RL applications must deal with.

When designing a reinforcement learning solution there is also the very challenging task of tuning all the variables. In addition to this, when using DQN, the network structure also needs some tuning. In this application, the tuning variables were the learning rate, discount rate, update frequency for the target network, priority scale, step-size and reward function to name a few. In regards to the network structure, the number of hidden layers and the number of nodes per hidden layer also needs to be tuned.

It is often said that designing a neural network is more art than science. This is due to the fact that there does not seem to be any general rules for how many hidden layers or nodes one should have. These variables are all tuned based on experience, which makes this an extra long process when sufficient experience is lacking. Another design challenge was designing how the agent should interact with the environment. As shown in the sections on design and implementation, multiple different interaction strategies were tested. This was done both in regards to the controller level of the agent and what the actions afforded to the agent would do. Some of these did not pan out due to design flaws and others suffered from lacking functionalities in the simulator

The simulator provided by DNV GL is indeed an extremely powerful tool. The accuracy of the model allows the agent to be tested on near identical conditions as in a test with a physical system. This advantage can not be understated, as an agent trained in the simulated environment could likely be directly transferred to a physical test, and do just as well at any given task. However, it turns out that the simulator lacks the ability to run fast enough to be used to train RL agents. This is likely due to the fact that it was not created with this in mind.

This became a huge problem for any implementation that relied on actually simulating the vessel movement. The second implementation was one such strategy that was not feasible as it would take to long to train the agent. Since the agent was controlling the vessel at a lower level, it would naturally have to simulate the vessel movement. This implementation was one that had great initial expectations tied too it. Certain that the agent would be able to learn to control the vessel. The plan was to test the agent with multiple enemy vessels and environmental disturbances to prove that DQN could be a very powerful tool for increasing autonomy in marine vessels. Sadly this never happened as the agent was never able to complete its training. It is, however, a firm belief that this method will work if given an environment that can sufficiently train the agent within a realistic time frame.

As the second implementation did not turn out as intended due to the slow simulation speed. There was probably far too much time spent on trying to get this implementation to work. Though this lead to the implementation of fixed Q-targets, double DQN, PER and a lot was learned concerning GPU compatibility, time might have been better spent focusing elsewhere. For instance, if the focus had been on trying to implement a simpler model that could be simulated at a faster rate the strategy might have worked.

7.2 Concerning Results

The results presented at the end of this thesis do indeed show that DQN has merit as a high-level decision support system. It is able to solve the outlined scenarios with relative ease and displays intelligence decision making capabilities while doing so. It shows this by understanding that it has to wait for the the enemy vessel to pass in scenario three. However, there are some criticisms that can be directed toward these results. Firstly, the agent is never showed to be able to handle the problem in its entirety. The fact that it can solve different scenarios does not prove that is can solve all when put together.

This is a valid critique of the suggested strategy. The agent has not been shown solving the entire environment with traffic. This is, however, heavily implied by the results. The fact is that DQN has solved far more complex problems in the past, so the one presented here should not pose any issue. In addition, the amount of training time needed to solve the three scenarios is minimal. Compared to other DQN implementations on different systems, solving these scenarios in under a thousand episodes is quite impressive. Though solving the entire problem might take longer in regards to training, the results and history of DQN point to it being possible. The way this would be done would be to expand the use of the more-convex area segregation introduced.

After training on the first two parts of the map the outside area would be added. The vessel would then start in this area and find its way back to the goal area. This is precisely how the supplementary scenario containing the full environment was done, and the agent proved that it could solve this with ease. The next step would be to add traffic to the environment and apply the trained agent. Another valid critique is that this strategy places a lot of trust in the lower level controllers.

There is no denying that this is the case. As the agent is designed to be a high-level decision support system, it has complete trust in the lower level controllers. This also means that the agent has no way of compensating for any environmental disturbances. As such, the agent does not take into consideration whether the actions found is at all feasible for the vessel to execute, as it can not account for environmental disturbances. However, a high-level decision support system is just that, high-level. It would have to trust that the lower level controllers would be able to correct for disturbances and execute a given action. Taking the ship dynamics into account while choosing the actions also prevents the agent from asking the vessel to take actions that would be impossible.

It might, however, be the case that some actions are initially feasible, but becomes in-feasible due to disturbances alone. The current strategy does not take this into account. It might be a good idea for the agent to test the optimal policy once found and see if the vessel is able to execute all the actions given by the agent. The agent could then be given a reward dependant on the performance of the vessel. If the vessel hits land due to environmental disturbances, this might prompt the agent to guide the vessel further from land to compensate, for instance. This functionality does not exist as of now, but it is a great idea for a future extension.

Another critique could be that the agent action set is rather simplistic. And this is true, even when using dynamic actions. It is important to remember, however, that these actions are in a sense only placeholders. In an expanded implementation the action set would indeed be more complicated. The wait function would, for example, be replaced with a dynamic positioning controller that would keep the ship stationary for a set amount of time. Since the agent is able to solve these scenarios with two different actions sets which behave very differently, it is implied that how the actions move the vessel is not a deciding factor. As long as the agent is given actions that make it possible for it to solve the problem, it will be able to do so. It is, however, not recommended to make these actions needlessly complicated. This would only serve to increase training time and possibly necessitate expanding the network structure.

The results also suffer from the same problem as most DNN implementations in regards to explainability. These scenarios are very clear in what the proper action is, as the scenarios were designed to have clearly distinct optimal actions. This was done in order to more easily prove that the agent was, in fact, acting in a correct fashion. However, this is not always the case, proving what factors have the greatest influence over the agent's choices is not an easy task. Adding some form of explainability to the algorithm would certainly be an improvement. This especially since the strategy is meant to be utilized on a physical system.

A design choice made in this thesis was to try and solve the problem using only information that could be gathered locally on the vessel itself. The main argument for this choice was that it would avoid the need for a graphical interface. If the DQN algorithm by DeepMind was to be implemented directly it would be necessary to have a graphical interface for each port area. This interface would also have to include enemy vessels and obstacles. Creating such an interface seemed like a tedious process, that could be avoided by using local information. It is assumed that the agent would have access to its own position, heading and speed. It was also assumed that the agent would be able to receive the positional information of any enemy vessels. As well as knowing the position of the designated docking area, or goal area.

It is not unrealistic that the vessel would have access to this information when close to the port area. This is proven by the existence of cellphone applications, like ”Gule sider på sjøen”, that are able to position the phone in a port area with ease. If sending and receiving GPS locations is possible on such a small device. It is fair to assume that a large vessel could be equipped to do the same. One might argue that GPS positions are not always correct and can be subject to drift. However, this can be countered by creating an area around obstacles. Punishing the agent for getting to close to land or an enemy vessel instead of hitting them would give a safe distance for the vessel to operate within.

7.3 Future Work

Despite the positive nature of the results presented in this thesis, there is always room for improvement. Some possible expansions have already been mentioned in the previous discussion. This section will present some ideas on what could be improved in a potential continuation of the research presented in this thesis.

It was mentioned earlier that the agent has no way to correct for environmental disturbances. Even though the agent does take the vessel dynamics into account, this does not guarantee that a suggested policy is feasible. The policy might lead the vessel close to land, where environmental disturbances could make the vessel ground, for instance. It would probably be a good idea to give the agent some way to take this into account. This could be done by adding a function for testing the policy once found. The agent could then perhaps be rewarded based on the vessel performance. This would make the entire system more robust, and also allow the agent to possibly correct for poor LOS guidance design to some degree.

There is also an issue with the current strategy that it does not have any way of knowing when the problem is solved. Say the agent is done exploring the environment and has found the goal area. When do we decide that the agent is sufficiently trained? The current implementation does not have any functionality do decide this. It would be useful if there was a way for the agent to know that it was done training. It could then turn off exploration entirely and be ready to complete the task in a real scenario. This could be that the agent successfully reaches the goal a number of times in succession. Or perhaps that the agent reaches the goal in a given number of actions, or achieves a given amount of accumulated reward. Whatever the case, this would be a great expansion on the suggested strategy.

Though the results show that the agent is able to make intelligent decisions in regards to avoiding traffic. There was not enough time to test this aspect exhaustively. It might be a good idea to test the entire problem, from outside port to the docking area, with more traffic. This would reveal the limit of what the agent can keep track of. It might be that the agent would have trouble navigating a port if there are too many enemy vessels around. In this case, the agent might need to prioritise which vessels to keep track of. This might be based on only vessels within a certain range, for instance, or enemy vessels heading. Taking the enemy vessel heading into account, one could decide to only consider vessels that might intersect with the given path from the agent.

Explainability is also something that has been touched on in this thesis. As of now, there are no elements of explainability present in the final implementation. This is not a particularly important issue for the purpose of this thesis. However, if this strategy is to be implemented and utilized successfully we need to be able to trust the system. Implementing an element of explainability would greatly improve the trust in the system, and is a worthy pursuit for future research. LIME [44] is one such algorithm that was researched at the tail end of this thesis, however, there was not enough time to implement this solution.

7.4 Conclusion

This thesis explored the possibility of using RL methods as a high-level decision support system for a marine vessel. The vessel was set in a "close to port" scenario, and the agents goal was choosing the correct actions in order to guide a vessel to a designated docking area. This was completed using a DQN network to choose actions, make intelligent decisions and design an optimal policy to guide the vessel. The agent also takes the dynamics of the vessel into account when choosing actions, increasing the probability that the policy is feasible for the lower level controllers to execute.

The neural network structure suggested in this thesis consists of an input layer, three hidden layers and an output layer, which are all dense. The hidden layers have 64, 128 and 64 nodes respectively. As for the activation function, the two first layers use $ReLU$ and the final layer uses *tanh*. Input information given to the network was the vessel position, distance to goal and enemy vessel position for the simple actions, with heading and moving variables added for the dynamic actions. These values were normalized between zero and five before being passed to the network. The output of the network corresponded to the Q-value of the different actions the agent could take.

The agent was be rewarded for finding the goal area and penalized for hitting land or colliding with an enemy vessel. While using the dynamic actions the agent would also be penalized for trying to use actions not available when moving. In addition to this, there would be a small energy penalty given to the agent for every action. This would encourage the agent to find the fastest way to the goal.

In order for the agent to learn complex, non-convex areas at a faster rate a training strategy was developed. This training strategy was based on dividing the environment into as convex areas as possible. The agent starts training in the area containing the goal first. When sufficiently trained, a new area is added and the agent starts training from here. The goal would then be to find its way back to the goal area, from the new area.

The results gathered shows that the agent is able to find its way through highly complex, non-convex areas. In addition to this, the agent is also shown to be capable of making intelligent decisions. Despite the weaknesses discussed, the suggested strategy shows the potential of RL method being implemented on such a system. It also lays a powerful foundation that can be improved upon in future research.

BIBLIOGRAPHY

[1] M. Birdsall, “Google and ITE : the road ahead for self-driving cars,” *ITE Journal (Institute of Transportation Engineers)*, vol. 84, no. 5, pp. 36–39, 2014.

[2] M. Dikmen and C. M. Burns, “Autonomous Driving in the Real World: Experience with Tesla Autopilot and Summon,” in *8th International Conference on Automotive User Interfaces and Interactive Vehicular Applications*, (Ann Arbor, MI, USA), pp. 225–228, Association for Computing Machinery (ACM), 1 2017.

[3] K. B. Ånonsen and O. K. Hagen, “The HUGIN AUV Terrain Navigation Module,” tech. rep., Kongsberg Maritime Subsea, FFI, Horten, Kjeller, 2013.

[4] A. B. Martinsen, *End-to-end training for path following and control of marine vehicles.* PhD thesis, NTNU, 2018.

[5] A. B. Martinsen and A. M. Lekkas, “Straight-Path Following for Underactuated Marine Vessels using Deep Reinforcement Learning,” *IFAC-PapersOnLine*, vol. 51, no. 29, pp. 329–334, 2018.

[6] A. B. Martinsen and A. M. Lekkas, “Curved Path Following with Deep Reinforcement Learning: Results from Three Vessel Models,” in *OCEANS 2018 MTS/IEEE Charleston, OCEAN 2018*, (n.l), Institute of Electrical and Electronics Engineers Inc., 1 2019.

[7] S. Haiqing and G. Chen, “path following control of underactuated ships using actor-critic reinforcement learning with mlp neural network,” tech. rep., Dalian Maritime University, Dalian, China, 2016.

[8] B. Yoo and J. Kim, “Path optimization for marine vehicles in ocean currents using

reinforcement learning," *Journal of Marine Science and Technology*, vol. 21, pp. 335–343, 2015.

[9] M. Carreras, J. Batlle, and P. Ridao, "Hybrid coordination of reinforcement learning-based behaviors for AUV control," in *Proceedings 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems. Expanding the Societal Role of Robotics in the the Next Millennium (Cat. No.01CH37180)*, (Maui, Hawaii, USA), pp. 1410–1415, 2001.

[10] M. Carreras, J. Yuh, J. Batlle, and P. Ridao, "A Behavior-Based Scheme Using Reinforcement Learning for Autonomous Underwater Vehicles," *IEEE Journal of Oceanic Engineering*, vol. 30, pp. 416–427, 4 2005.

[11] Z. Yin, W. He, C. Sun, G. Li, and C. Yang, "Adaptive Control of a Marine Vessel Based on Reinforcement Learning," in *The 37th Chinese Control Conference*, (Wuhan, China), pp. 2735–2740, 2018.

[12] K. Ishii and T. Ura, "An adaptive neural-net controller system for an underwater vehicle," in *Control Engineering Practice*, pp. 177–184, Pergamon, 2000.

[13] I. Carlucho, M. De Paula, S. Wang, Y. Petillot, and G. G. Acosta, "Adaptive low-level control of autonomous underwater vehicles using deep reinforcement learning," *Robotics and Autonomous Systems*, pp. 72–86, 2018.

[14] R. Cui, C. Yang, Y. Li, and S. Sharma, "Adaptive Neural Network Control of AUVs with Control Input Nonlinearities Using Reinforcement Learning," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 47, pp. 1019–1029, 6 2017.

[15] Y. Cheng and W. Zhang, "Concise deep reinforcement learning obstacle avoidance for underactuated unmanned marine vessels," *Neurocomputing*, pp. 63–73, 2018.

[16] Q. Xu, Y. Yang, C. Zhang, and L. Zhang, "Deep Convolutional Neural Network-Based Autonomous Marine Vehicle Maneuver," *International Journal of Fuzzy Systems*, vol. 20, pp. 688–699, 2017.

[17] R. Zhang, P. Tang, Y. Su, X. Li, G. Yang, and C. Shi, "An adaptive obstacle avoidance algorithm for unmanned surface vehicle in complicated marine environments," *IEEE/CAA Journal of Automatica Sinica*, vol. 1, pp. 385–396, 10 2014.

[18] Y. Shen, N. Zhao, M. Xia, and X. Du, "A Deep Q-Learning Network for Ship Stowage Planning Problem," *Polish Maritime Research*, vol. 24, pp. 102–109, 11 2017.

[19] W. He, Z. Yin, and C. Sun, "Adaptive Neural Network Control of a Marine Vessel

with Constraints Using the Asymmetric Barrier Lyapunov Function," *IEEE Transactions on Cybernetics*, vol. 47, pp. 1641–1651, 7 2017.

[20] DNV GL, "The ReVolt - DNV GL," *https://www.dnvgl.com/technology-innovation/revolt/index.html.*

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* The MIT press, seconed ed., 2018.

[22] Y. Shi, C. Shen, H. Fang, and H. Li, "Advanced Control in Marine Mechatronic Systems: A Survey," *IEEE/ASME Transactions on Mechatronics*, vol. 22, pp. 1121–1131, 6 2017.

[23] A. D. Tijsma, M. M. Drugan, and M. A. Wiering, "Comparing exploration strategies for Q-learning in random stochastic mazes," in *2016 IEEE Symposium Series on Computational Intelligence, SSCI 2016*, (n.l), University of Groningen, Technical University Eindhoven, Institute of Electrical and Electronics Engineers Inc., 2 2016.

[24] S. B. Thrun, "Efficient Exploration In Reinforcement Learning," tech. rep., Carnenie-Mellon University, Pittsburgh, Pennsylvania, 1992.

[25] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[26] P. Ramachandran, B. Zoph, and Q. V. Le, "Searching for Activation Functions," tech. rep., Google Brain, 10 2017.

[27] C. Oberndorfer, *Research on new Artificial Intelligence based Path Planning Algorithms with Focus on Autonomous Driving.* PhD thesis, University of Applied Science Munich, 2017.

[28] P. J. Werbos, "Generalization of Backpropagation with Application to a Recurrent Gas Market Model," tech. rep., U.S Department of Energy, 1988.

[29] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing Atari with Deep Reinforcement Learning," tech. rep., DeepMind Technologies, 2013.

[30] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning.," *Nature*, vol. 518, no. 7540, pp. 529–33, 2015.

[31] S. Patel and J. Pingel, “Introduction to Deep Learning: What Are Convolutional Neural Networks? Video - MATLAB,” 2017.

[32] Ø. H. Gulbrandsen, *AI Planning and Low-Level Control for a Robotic Manipulator*. PhD thesis, NTNU, 2018.

[33] H. V. Hasselt, “Double Q-learning,” in *Advances in Neural Information Processing Systems 23* (J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, eds.), pp. 2613–2621, Curran Associates, Inc., 2010.

[34] Z. Wang, T. Schaul, M. Hessel, and M. Lanctot, “Dueling Network Architectures for Deep Reinforcement Learning,” tech. rep., Google Deepmind, London, UK, 2016.

[35] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized Experience Replay,” tech. rep., Google DeepMind, 2015.

[36] B. Foss and T. A. N. Heirung, “Merging Optimization and Control,” tech. rep., NTNU, Trondheim, 2016.

[37] D. Gunning, “Explainable Artificial Intelligence,” 2016.

[38] M. T. Ribeiro, S. Singh, and C. Guestrin, “”Why Should I Trust You?”: Explaining the Predictions of Any Classifier,” tech. rep., University of Washington, Seattle, USA, 2016.

[39] T. I. Fossen, M. Breivik, and R. Skjetne, “Line-of-sight path following of underactuated marine craft,” tech. rep., NTNU, Trondheim, 2003.

[40] T. I. Fossen, *Handbook of Marine Craft Hydrodynamics and Motion Control*. Trondheim: John Wiley & sons,Ltd, first ed., 2014.

[41] H. L. Alfheim, K. Muggerud, M. Breivik, E. F. Brekke, E. Eide, and Ø. Engelhardtsen, “Development of a Dynamic Positioning System for the ReVolt Model Ship,” *IFAC-PapersOnLine*, vol. 51, no. 29, pp. 116–121, 2018.

[42] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng, and G. Research, “TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems,” tech. rep., Google Research, n.l, 2015.

[43] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, and G. Brain, "TensorFlow: A system for large-scale machine learning," tech. rep., Google Brain, Savannah, GA, USA, 2016.

[44] M. Stiffler, A. Hudler, E. Lee, D. Braines, D. Mott, and D. Harborne, "An Analysis of Reliability Using LIME with Deep Learning Models," tech. rep., USA military academy, IBM research, Cardiff University, Cardiff, UK, 2018.