Åbo Akademi

Learning autonomous maritime navigation with offline reinforcement learning and marine traffic data

Jimmy Westerlund 39053

Master's thesis in Computer Engineering

Supervisors: Sebastien Lafond & Sepinoud Azimi Rashti

Åbo Akademi University

Faculty of Science and Engineering

Information Technologies

2021

Abstract

Autonomous shipping is a heavily researched topic, and currently, there are large amounts of ship traffic data available but unexploited. Autonomous ships have the potential to reduce costs and increase safety. The challenge is achieving the correct maritime navigation behavior according to the situation reliably, which may be possible by exploiting historical ship traffic data. This thesis explores the possibility of using offline reinforcement learning based on AIS data to learn autonomous maritime navigation.

The hypothesis that AIS data can be used for training a reinforcement learning agent is tested by implementing an offline reinforcement learning agent. For comparison, an online agent that learns without data is also implemented. Both agents are trained and evaluated in a simulator, and the goal of both agents is to learn to navigate to a destination, given a starting point.

The results suggest that offline reinforcement learning can be used for automating maritime navigation, but a more extensive and more diverse dataset is needed to conclude its effectiveness.

Keywords: Reinforcement learning, offline reinforcement learning, autonomous navigation, autonomous ship, AIS

Preface

I want to say a special thanks to my supervisors Sepinoud and Sebastien, for their guidance and motivation throughout the project. Also, thanks to Richard Nyberg for helping me get started with the project and explaining how to use the simulator.

Table of Contents

1. Introduction

Over the past decade, there has been a dramatic development in the field of autonomous technologies. Autonomous systems are being applied in transportation systems, and with the technological breakthrough, the reality of fully autonomous transport systems may not be so distant. Giants like Google and Tesla are two leading developers of fully self-driving systems for road transport, and their partially self-driving systems are already available in many new cars [23]. For air-based transport, uncrewed aerial vehicles (UAVs) are being used for delivery services. In Helsinki, for example, a delivery system using drones is being developed by Wing [24]. Maritime transport is the sector responsible for the majority of all transport. Among the leading developers of autonomous maritime systems are Rolls-Royce and Kongsberg, but no fully autonomous vessels are used commercially yet. Autonomous maritime navigation is an area of broad and current interest, and it is also the topic of this thesis.

A study in the report *Review of Maritime Transport 2020* [25] shows that over 80 percent of global trade by volume is carried out by cargo ships. Moreover, shipping is the only viable option for international trade, as it is by far the most cost-effective option for transporting large volumes of merchandise [26]. However, the shipping industry faces immense economic, environmental, and safety challenges as traditional solutions, such as building larger and more optimized ships, reach their limits.

It is estimated that over 75 percent of maritime accidents involve human error [27]. Furthermore, an analysis of around 15 000 maritime liability insurance claims between 2011 and 2016 was done by Allianz Global Corporate & Specialty (AGCS) and concluded that human error was a primary factor in 75% of all cases, equivalent to 1.6 billion dollars in losses [27].

Crewless autonomous ship systems would enable new ship designs, as the bridge and the living spaces on vessels could be removed to improve aerodynamics and reduce fuel consumption. It would also lower crew costs and ultimately improve safety by removing the human element. Autonomous ships are believed to be a potential solution to the difficulties the shipping industry faces, offering safer and more cost-effective shipping [26].

Reinforcement learning used for autonomous driving and navigation is a reasonably new approach. Even more recent is the idea of using data-driven reinforcement learning in maritime navigation, which is enabled by the collection of automatic identification system (AIS) data.

According to *Regulation 19*, all cargo ships weighing over 500 tons and all passenger ships, irrespective of size, must be fitted with AIS [28]. The AIS data contains information about the vessel, such as the coordinates, speed, and heading. AIS data contains valuable information that can, in theory, be used to learn autonomous maritime navigation. This thesis studies the possibility of using *offline reinforcement learning* with AIS data to learn autonomous maritime navigation.

2. Reinforcement learning

Reinforcement learning (RL) is the approach of computationally automating tasks by specifying a goal and a set of rules to follow. What sets reinforcement learning apart from other machine learning approaches is that the RL agent interacts directly with the environment and learns to make decisions based on direct feedback from the environment [1]. Compared to supervised learning, the strength of reinforcement learning lies in the lack of need of data. In supervised learning, labeled data is used to learn a specific behavior. The data contains information about what the correct action in a particular situation would be [1]. This approach can yield successful results in cases where large amounts of data are available. Still, in unexplored or partially unexplored environments with limited or no data available, reinforcement learning is more applicable. However, there are also hybrid approaches; recently, the interest in data-based reinforcement learning, also known as offline reinforcement learning, has grown as it has shown great potential in solving various problems [5].

Navigating a ship from a starting point to a destination can be defined as a problem with a goal, a set of possible actions, and a set of rules that the ship must follow while maneuvering its way to the destination. With this definition, it can be thought of as a problem that can be solved with reinforcement learning. The problem is, of course, more complex in reality, but the essential idea of RL can be explained through this simple example. Maritime navigation with reinforcement learning is the core of this thesis, and the possible actions, rules, and goal of the thesis will be discussed in greater detail in the next chapter. In contrast, this chapter will review the general history and elements of reinforcement learning.

2.1 History

In the book *Reinforcement Learning: An Introduction*, Sutton and Barton explain that modern reinforcement learning has its roots mainly in two different methods, namely learning by trial-and-error and optimal control [1]. The methods evolved independently and eventually intertwined in the early 1980s to form what is known as reinforcement learning today.

Learning by trial-and-error stems from psychological studies of animal learning and is generally regarded as the earliest studies in artificial intelligence [1]. The idea dates back to the late 1800s when the British psychologist Conway Lloyd Morgan used the term “trial-and-error learning” to describe animal behavior. In the early 1900s, American psychologist Edward Lee Thorndike formulated a principle about trial-and-error learning, known as the “Law of Effect”

principle, which involved positive reinforcement and negative reinforcement. According to Sutton et al., Alan Turing was among the first to approach the idea of implementing trial-and-error learning in a computer [1]. In 1948, Turing wrote a report about a "pleasure-and-pain system" based on Thorndike's "Law of Effect" principle. In the report, Turing describes a system that would take an action randomly and save the results temporarily. When a negative reinforcement occurs, the system will discard all temporarily saved results, and when a positive reinforcement occurs, it will save the results permanently [1].

Optimal control is a term that emerged in the 1950s and it describes the method of designing controllers for dynamical systems [1]. Richard Bellman developed a proposed solution to optimal control. The proposed solution uses a value function and the states of a dynamical system to define a function that became known as the *Bellman equation* [1]. The methods that solve this equation are categorized as *dynamic programming.*

Dynamic programming is the methodology of breaking down a complex problem into smaller subproblems and solving them individually before combining them into a final solution [2]. A dynamic programming algorithm saves the answers to all solved subproblems in tables [2]. The algorithm will only have to calculate the answers to the subproblems once, as the answers can be fetched directly from the table in case the subproblem needs to be solved again. However, one of the most significant drawbacks with dynamic programming is that its computational needs grow exponentially with the complexity of the problem since the number of subproblems grows with the number of state variables [1].

In the early 1960s, the research in artificial intelligence shifted from reinforcement learning to supervised learning, and the distinction between the two types was often confused [1]. Following the confusion, research in reinforcement learning stagnated in the 1960s and early 1970s, but in the mid-1970s, Harry Klopf recognized that the adaptive behavior in artificial intelligence was fading due to most researchers focusing on supervised learning. Through this realization, Klopf linked trial-and-error learning to reinforcement learning again and made a clear distinction between reinforcement learning and supervised learning [1].

Apart from dynamic programming, two other reinforcement learning methods are commonly used, namely Monte Carlo (MC) methods and temporal difference (TD) learning. Unlike dynamic programming, Monte Carlo methods do not require a complete model of the environment, but their downside is that they update the policy only after an episode has terminated, not at every time step [1]. As stated by Sutton et al., MC methods are a way of

solving RL problems by averaging sample returns, and this approach is only suitable for episodic tasks. Lastly, TD learning methods combine the strengths found in DP and MC methods [1]. TD methods learn from experience without a model, and they can also make predictions mid-training, as the policy is updated at each time step.

2.2 Markov Decision Process

Before delving into the elements of reinforcement learning, it is important to first explain the core mathematics behind reinforcement learning, namely the Markov Decision Process (MDP). In case the state and action space of the MDP is finite, it is called a *finite Markov Decision Process* (finite MDP) [1]. According to Sutton et al., finite MDP is a vital part of reinforcement learning. They state that a large amount of modern reinforcement learning can be understood simply by understanding MDPs.

To express a reinforcement learning task as an MDP, it must first fulfill the requirement of the Markovian property [1]. A task is said to have the Markov property if the next state s' at time $t+1$ only depends on the state and action s_t, a_t at time t. A task with the Markov property is called an MDP, and it enables the predictions of the following state and the expected next reward, given the current state and action [1]. With a given state s and action a the probability of every next state s' and reward r is expressed with the following formula [1]:

$$p(s', r|s, a) = \Pr\{S_{t+1} = s',\ R_{t+1} = r \mid S_t = s, A_t = a\}$$

This formula fully expresses the dynamics of a finite MDP, and it enables the computation of other things regarding the interaction between the agent and the environment. An example is the formula for calculating the state-transition probability, which gives the probability of transitioning to state s', given a state s and an action a [1]:

$$p(s'|s, a) = \Pr\{S_{t+1} = s' \mid S_t = s,\ A_t = a\}$$

An MDP is represented as a 5-tuple [1][3]: (S, A, P, R, γ) where:

- S – a finite set of states
- A – a finite set of actions
- P – a set of state-transition probabilities
- $R_a(s, s')$ – the immediate reward distributed by a reward function when transitioning from state s to state s' through action a
- $\gamma \in [0, 1]$ – the discount rate

To clarify, a set of states are values that describe the situation the agent is in, and a set of actions describes what actions the agent performs. The discount factor is a value between 0 and 1 which determines the current value of future rewards, as discussed further in this chapter.

2.3 Elements of reinforcement learning

According to [1], reinforcement learning is defined by the following elements: agent, environment, policy, reward function, value function, and model. These are the building blocks that define what reinforcement learning is, and they are explained in greater detail in the following sections.

2.3.1 Agent and Environment

In reinforcement learning, the two first elements are the agent and the environment. The learner, which is also the decision-maker, is called the agent [1]. What it interacts with, containing everything outside the agent, is called the environment. State and action are two terms used to describe what situation the agent is currently in. A state contains information about where in the environment the agent is at a given moment, and the action describes what action the agent takes. The agent and the environment are connected through their interactions; the agent takes an action, and the environment reacts to the action by presenting a new state to the agent. The interaction between the agent and the environment occurs at each *time step*, where the discrete time step is a sequence [1]: $t = 0, 1, 2, 3, \ldots$ As stated by Sutton et al., time steps can also be continuous, though continuous time steps involve more complex calculations compared to discrete time steps. In this thesis, only discrete time steps are considered as they are simpler to deal with, and it applies well to ship log files, where each row in the data

corresponds to 1 second. The use of discrete time steps will be more explicit in the next chapter, where the ship log files from Aboa Mare are explained in further detail.

Figure 1 is a visualization by Sutton et al., and it is widely used to illustrate the interaction between the agent and the environment. The agent starts in time step t and takes an action A_t based on the initial state S_t and reward R_t. The environment responds to the action by providing the agent with a new state S_{t+1} and a new reward S_{t+1} and the interaction continues. The environment generates rewards that the agent tries to maximize over time. In some cases, it can be difficult to define the boundary between agent and environment, but the general rule is that anything that the agent cannot change is considered outside of it and thus part of its environment. In the case of an autonomous ship, the ship is the agent, and everything else, such as the water, the land, and the sea marks combined, make up the environment.

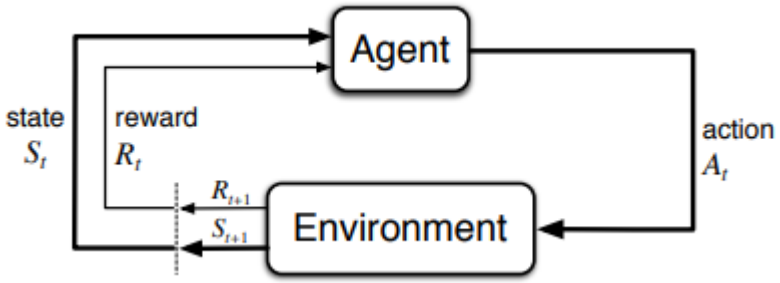


Figure 1: Agent and environment interaction visualized by Sutton et al. [1]

2.3.2 Policy

The policy defines the agent's behavior in any state; in other words, the policy is the agent's strategy. It is a mapping from each state and action in the environment to the probability of taking a specific action when being in a particular state [1]. The policy is an essential part of a reinforcement learning agent, as it alone is enough to define the behavior of the agent. Essentially, solving a reinforcement learning task is to find the policy that brings the largest number of rewards over a long time. As stated by Sutton et al., this policy is called the *optimal policy* [1].

Reinforcement learning methods are usually categorized as being either *on-policy* or *off-policy*, depending on how the agent achieves the optimal policy [1]. In on-policy methods, the agent learns how good actions are based on a specific policy π by observing the rewards generated when following that policy. In off-policy methods, the agent learns how good actions are based on a *behavior policy* π_1 by observing the rewards generated when following a different policy π_2 called the *target policy*.

2.3.3 Reward function

A reward in reinforcement learning is a numeric value that the agent receives at each time step of the training, and it is also what defines the goal of a reinforcement learning problem [1]. The objective of the reinforcement learning agent is to maximize the total reward over a longer time. The reward function defines what behavior should be rewarded and what behavior should be punished. With a well-made reward function, the agent will thus be able to learn the desired behavior. The reward is also a deciding factor for altering the policy since the agent will recognize that an action followed by a low reward may not be optimal. As a result, the policy might change to make the agent take a different action if faced with the same situation in the future [1].

In some reinforcement learning scenarios, the interaction between the agent and the environment naturally breaks up into segments. These segments are called *episodes* and what divides one episode from another is a reset of the environment. In the example earlier described where an agent must learn to navigate a ship from a starting point to a destination, an episode would end if the agent reached the goal, but it would also end if the agent did not reach the destination within a specific time limit, in case there was a time limit defined. Every episode ends with a state called the *terminal state*, which is always followed by resetting the environment to a default starting state [1].

The *expected return* is the total amount of rewards that the agent is expected to gather, and for the most straightforward cases, it can be expressed as the sum of the rewards [1]:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$$

where R_t is the reward in time step t and T is the last time step. To avoid dealing with infinite rewards, the formula can be further developed by introducing a *discount rate* γ, a value within the bounds of $0 \leq \gamma \leq 1$. The discount rate tells the current value of future rewards, which helps the agent choose between immediate and future rewards [1]. A discount rate of 0 will result in the agent prioritizing to maximize the immediate rewards from the next time step, but when the discount rate approaches 1, the agent will also take future rewards into account [1]. Without a discount rate, the agent is at risk of getting stuck doing the same actions repeatedly but never reaching the goal. The *discounted return* is represented by [1]:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The formula says that a reward received k time steps into the future is only worth γ^{k-1} times what it would be worth immediately [1].

Like several other aspects of reinforcement learning, the idea of discounting is also based on human psychology. As presented by Leonard Green and Joel Myerson [4], an example of human discounting is when faced with the option of receiving 100€ now or 120€ in a month, the immediate reward might seem more appealing. However, if both rewards are shifted one year into the future, the choice is to receive 100€ in one year or 120€ in 13 months. The larger reward suddenly seems like the obvious choice, as the sooner reward has become *discounted* because it is so far in the future.

2.3.4 Value functions

Closely related to the reward function are the value functions. While the reward function defines what action yields the best reward in the current state, the value functions specify what actions are best for collecting the most rewards over a longer time. The value of a state is the total amount of rewards an agent can be expected to collect in the future, given a starting state s and a policy π. This value is calculated through the *state-value function* and is expressed as [1]:

$$v_\pi(s) = E_\pi[G_t|S_t = s] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right]$$

where E_π is the expected value, G_t is the discounted return and t is any time step. Another value function is the *action-value function*, which gives the value of taking an action a in state s with a policy π [1]:

$$q_\pi(s,a) = E_\pi[G_t|S_t = s, A_t = a] = E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right]$$

The value functions v_π and q_π are needed because states that yield low immediate rewards might lead to states with high rewards. The reverse can also be true; states which produce high immediate rewards might lead to states with low or even negative rewards.

The state-value function v_π can be used to derive the formula called the Bellman equation [1]:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')]$$

The Bellman equation expresses the relation between the value of a state and the value of future states. Essentially, it is the product of the sums of two probabilities. The first, $\pi(a|s)$, is the probability of taking action a in state s while following policy π and as earlier discussed, $p(s',r|s,a)$ is the probability of ending up in a state s' with reward r while being in state s and taking action a. The value is calculated with the sum over the values of the actions a , future state s' and reward r. For each a, s', r the probability $\pi(a|s)p(s',r|s,a)$ is calculated and weighted by the quantity $[r + \gamma v_\pi(s')]$. Finally, the expected value is given by the sum of all possibilities [1].

2.3.5 Model

The model is the sixth and final element of some, but not all, reinforcement learning systems. As the name indicates, it is a model of the environment, and it can be used in various ways to enhance an RL system. For example, with the input of a state and an action, the model can predict what the next state and next reward will be [1]. A model enables planning, meaning that we can decide what actions to take by considering future situations. Reinforcement learning methods utilizing a model are called *model-based*, while the methods purely built on trial-and-error style learning are called *model-free* [1].

2.4 Offline reinforcement learning

Reinforcement learning methods that utilize previously collected datasets for training without interaction with an environment are categorized as offline reinforcement learning [5]. Offline reinforcement learning algorithms have shown great promise in teaching optimal decision-making based on large datasets. In this section, the main goal is to explain the core idea of offline reinforcement learning and the benefits and drawbacks.

In traditional reinforcement learning, also known as online reinforcement learning, the agent explores the environment and learns by trial and error. This is not an issue with small, simple tasks that can easily be simulated, but a problem arises when dealing with large and complex tasks with no access to a realistic simulated environment in which the online agent can be trained. Building simulated environments for autonomous driving and healthcare tasks, for example, can be both expensive and dangerous. The challenge lies in building accurately simulated environments for the agents to train in, as the tasks can be complex and very different from each other. Furthermore, there is always a risk of error when transferring an agent from training in a simulated environment to solving real-world tasks.

In offline reinforcement learning, the agent does not have to interact with the environment, and instead of *exploring* the environment, it must *exploit* the dataset. With a large enough dataset, the idea is that the agent will be able to learn the optimal policy both reliably and quickly. The algorithm is given a static dataset of transitions $D = \{(s_t, a_t, s_{t+1}, r_t)\}$ and must try to learn the optimal policy based on it [5]. Ultimately, the goal of offline reinforcement learning is to find the optimal policy that performs better than the behavior observed in dataset D.

The great benefit of a successful offline reinforcement learning implementation is the lack of need for a simulated environment, in addition to the possibility to learn based on previously collected data. An added benefit is not having to address the *exploration-exploitation trade-off*. This is the dilemma of choosing between *exploring* new actions or *exploiting* known actions [1]. The agent must exploit known action to collect rewards, but it must also explore new action to make more optimal choices in the future. When learning based on a static dataset, the agent can only exploit the transitions found in the dataset.

However, there are a couple of issues as well that need to be considered. One of the biggest challenges in offline reinforcement learning is the fact that the training relies solely on the transitions in the dataset. This means that the agent will not be able to explore the environment to learn possible high-reward regions that exist outside of the dataset [5]. Another issue is the amount of data required to train an offline reinforcement agent. It is impossible to say an exact amount of data required for successful training as it depends on the complexity of the problem and the complexity of the algorithm, but generally, a larger dataset gives the potential for more successful results. With a small dataset, there is a risk that the agent is never able to find an optimal policy [5].

2.5 Algorithms

Essentially all reinforcement learning algorithms implement the same learning loop: the agent interacts with the environment while following a behavior policy. The agent observes the state, picks an action, observes the next state and the reward. This loop may repeat multiple times, and the policy is updated using the observed transitions [5].

There are many well-built algorithms for solving reinforcement learning problems, and choosing which one to use is not necessarily straightforward. In this thesis, the Deep Q-Network algorithm is used because it is a well-known algorithm. It has successfully been applied to solve tasks in simplified environments with discrete action spaces, such as vehicle navigation among pedestrians [8] and path planning of ships [9]. Additionally, several libraries with the Deep Q-Network algorithm implemented are available, which further speed up the development process.

2.5.1 Q Learning

Many new algorithms have emerged throughout the years, and most are improved versions of older ideas and approaches. Q-learning is regarded as one of the most important breakthroughs in the history of reinforcement learning, and it is the base of several modern algorithms [1].

In 1989, Anthony Watkins presented Q-learning, which is a model-free reinforcement learning algorithm based on dynamic programming methods [6]. It enables the agent to learn by evaluating the consequences of its actions in the environment. The agent tries all possible actions in all possible states and learns which actions are best based on the discounted return [1][6]. As stated by Watkins and Dayan [6], the object in Q-learning is to estimate the *Q-values* for an optimal policy. If the agent can successfully learn the Q-values, it can also decide the optimal actions in states. The Q-values are given by the *Q-function* $Q(S_t, A_t)$, which is equivalent to the action-value function earlier discussed. Each calculated Q-value is updated in a look-up table used in future calculations [6]. The Q-learning algorithm is represented by [1]:

$$Q^{new}(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

As seen in the formula, the Q-value, which is updated each time step, is the sum of three factors:

- $Q(S_t, A_t) + \alpha$: The current Q-value weighted by the learning rate
- αR_{t+1} : The reward in the next time step, weighted by the learning rate
- $\alpha \max_a Q(S_{t+1}, a)$: The action with the highest estimated action value based on all possible actions in the following state

The learning rate $0 \leq \alpha \leq 1$ determines at what rate the agent should overwrite previously learned behavior; in other words, it determines how quickly the agent should learn a behavior. A value of 0 will make the agent learn nothing, while a value of 1 makes the agent overwrite all previously learned information on each time step [1].

2.5.2 Deep Q-learning

Q-learning's main problem is that the look-up table also quickly grows in size in tasks involving many states and actions. Additionally, to accurately estimate the value of an action, the agent must explore every state in the environment, which is unrealistic for larger environments. As an answer to these problems, Minh et al. propose the Deep Q-Network (DQN) in the research paper *Human-level control through deep reinforcement learning*, published in 2015 [7]. The proposed algorithm is a combination of Q-learning and deep neural networks, and its advantage over Q-learning is the ability to approximate the optimal action-value function using a deep convolutional neural network instead of a look-up table [7]. With DQN being able to approximate the action-value function, it is more effective than Q-learning in tasks involving large state- and action spaces.

A deep convolutional neural network (DCNN) is a type of feed-forward neural network (FNN) [9], illustrated in Figure 2. A neural network of this kind works by distributing the input to the hidden layers, which will make decisions from previous layers and weigh whether a stochastic change within itself will be good or bad for the final output [9]. A neural network becomes a deep neural network when the architecture contains multiple hidden layers.

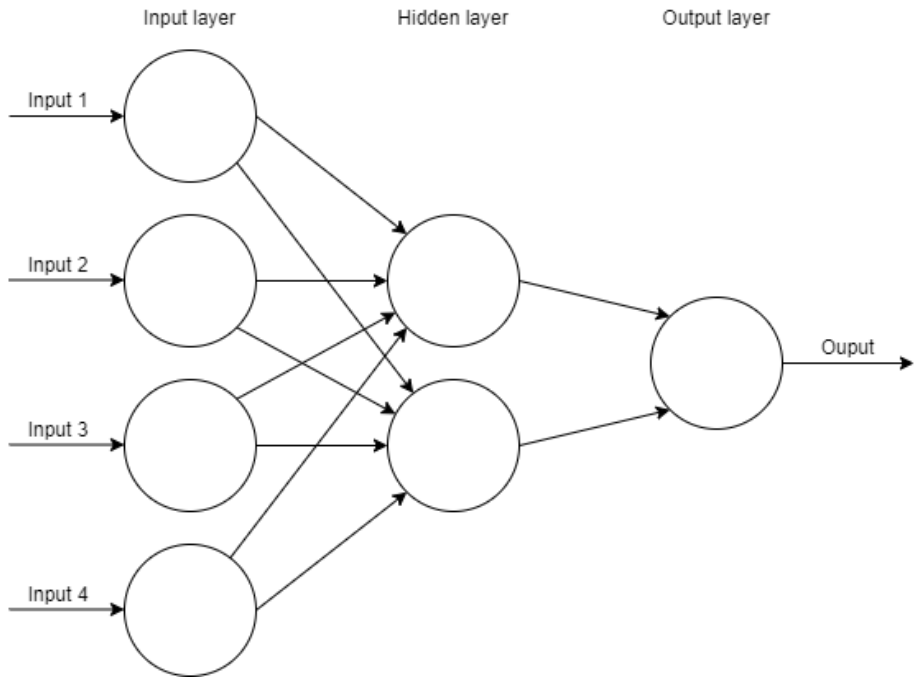


Figure 2: A three-layered feed-forward neural network (FNN), comprised of an input layer, a hidden layer, and an output layer.

Reinforcement learning methods are notoriously unstable when neural networks approximate the action value [7]. According to Minh et al., the reason behind the instability is caused by two things. Firstly, due to minor updates to Q may significantly change the policy, and secondly, due to the correlations between the action-values and the target values. To stabilize the learning, Minh et al. introduce *experience replay* and a *target network* [7].

Experience replay is the method of storing an agent's experience into a replay memory, and at each time step, the replay memory is updated. During the training, random samples from the replay memory are picked out and used for training the agent [7]. Experience replay improves training because it breaks the correlation between consecutive samples. Furthermore, it smoothes out learning and avoids oscillations because the behavior is averaged over past experiences. To further improve the stability of Q-learning, a target network is introduced. At a set interval, the network is cloned to obtain a target network, which is used for computing the estimated Q-values.

3. The project

The idea for this thesis project came from The Institute of Maritime Software Technology (MAST! Institute), a collaboration between Åbo Akademi University and Novia University of Applied Sciences. Research projects in the MAST! Institute focus on maritime digitalization and autonomous ships. With a combination of software expertise from Åbo Akademi University and the tools and knowledge at Novia's Maritime Academy, Aboa Mare, the goal is for the MAST! Institute to contribute to maritime research and develop new solutions for autonomous shipping [10].

The implementation and the testing of reinforcement agents are done in the *Simple Ship Simulator*, a ship simulator developed by Ivan Porres, Kim Hupponen [11], and Sebastian Penttinen [12]. The simulator was built for evaluating machine learning algorithms for autonomous ships.

3.1 Goal

This thesis explores the possibility of training an offline reinforcement learning agent based on previously collected ship data. The ship log data were collected from the simulator at Aboa Mare. The collection process was part of two other master's theses projects done by Richard Nyberg and Tatjana Cucic.

Evaluating an offline agent's performance can be difficult, and as a means of comparison, an online agent was developed alongside the offline agent. The online agent enables comparing performance and training times between the two agents, which can help make a final verdict about the offline RL agent's performance. Both agents are tested in the Simple Ship Simulator, where they are faced with solving the same seemingly simple task: navigate to a destination, given a starting point. The ultimate goal is for both agents to learn to solve the task successfully, followed by analyzing the data logged during training to reach a conclusion about offline reinforcement learning used in autonomous maritime navigation.

3.2 The dataset

The dataset contains ship traffic data collected from a realistic simulator during the training of students at Aboa Mare. In its raw format, the data came in Automatic Identification System (AIS) format encoded by the National Marine Electronics Association (NMEA) standard. Richard Nyberg and Tatjana Cucic extracted the raw data from the simulator at Aboa Mare and converted it into Comma Separated Value (CSV) files. Compared to raw data, CSV files are much easier to work with as they contain distinct rows and columns with informative headers. CSV files can also be visually inspected in Microsoft Excel, as it is a supported file format. The CSV files are referred to as "the dataset," and they are the starting point of the offline reinforcement learning agent.

Before preprocessing, the dataset contains 54 columns of parameters that hold information about the ship, such as the location, the speed, the direction, and the weather. However, only a fraction of the parameters are relevant for a proof-of-concept offline reinforcement agent, and the data preprocessing is discussed in the next chapter. Each CSV file holds information about the trajectory of one ship. Each row in the files corresponds to a second, and the state space is naturally discretized into one-second time steps. The dataset contains recorded behavior from four different areas: The North Sea, the Channel to Felixstowe, Dover Strait, and Rotterdam. The traffic recorded in the Rotterdam area was selected as the best of these locations, because it contained the most concentrated coordinates. The traffic in the Rotterdam area involves simulation data from 102 separate ships. Importing the data from the Rotterdam area into a GPS visualizing tool [14] generated Figure 3. The tool draws a red dot on a nautical map for each coordinate contained in the dataset. As seen in the figure, the traffic is heavily concentrated in a specific area, which is beneficial, given the small dataset. An offline reinforcement learning agent with a starting point and a destination within the area of concentrated data points should have a good chance of learning to navigate to the destination successfully.

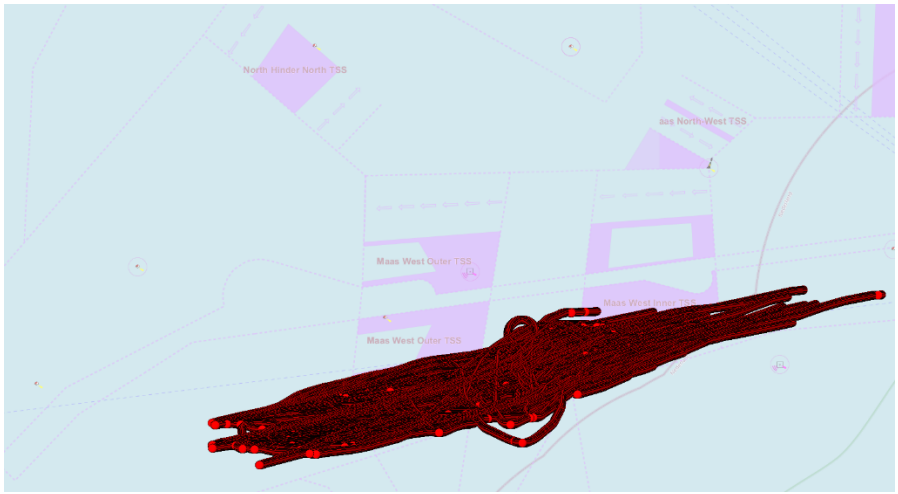


Figure 3: Data points from 102 separate ship simulation runs imported into a GPS visualizing tool [14].

A critical remark about the dataset is that it is recorded human behavior, which means that it may contain non-optimal behavior or errors that can affect the agent's learning. For this thesis, though, the quality of the data is not considered.

3.3 Ship terminologies

The names of the parameters in the dataset are not self-explanatory. Therefore, a subchapter explaining the relevant parameters and ship terminologies related to the parameters is necessary. A ship is affected by several external forces, which also affect the route it travels. The term "heading" is used to express the direction a ship is pointing. Still, because the ship can drift sideways due to environmental factors, such as the wind and the current, the term Course over Ground (COG) is used to express the ship's trajectory accurately. COG is the direction the boat travels relative to the bottom [13]. Similarly, Speed over Ground (SOG) tells the vessel's true speed, as it expresses the ship's relative to the ground.

The motion of a ship is expressed through six terms: surge, sway, yaw, heave, pitch, and roll. The SSS supports surge, sway, and yaw. For further simplification purposes, only surge and yaw are used in this implementation. Surge expresses the forward and backward movement of a ship, which is controlled by the throttle. Yaw describes the side-to-side movement of the stern and bow and is controlled by the rudder angle.

3.4 The ship simulator

The Simple Ship Simulator (SSS) is strictly software-based and must not be confused with Aboa Mare's simulator. Though the simulator at Aboa Mare is realistic and sophisticated, it is not well suited for testing and deploying custom machine learning solutions. SSS was developed to enable quick development and evaluation of machine learning algorithms of autonomous ships. The simulator is roughly divided into seven parts: the agent, the gym environment, the configuration file, the ship object, the simulation, the user interface, and the helper module [12].

The agent module contains all the code necessary for the reinforcement learning agent to learn to decide what actions to take. Suppose the agent is learning online, i.e., through trial-and-error. In that case, it will interact with a gym environment module that describes the agent's environment, and it gives rise to rewards each time the agent takes an action. Like real ships, the simulated ships also have many different properties, such as length and maximum engine RPMs. A configuration file is the starting point for every simulation. It contains all information needed to initialize a simulation, such as a map, a starting point, and a destination. The configuration file also contains initial values for the ship's properties, such as the initial speed and heading. The ship module contains all the properties of the ship object. The main properties are the x and y coordinates, the bearing, the speed, and the rudder angle. These five main properties are used in several other parts of the simulator, such as when calculating the distance between the ship and the destination. The simulation module transforms the states and actions into actual movement in the simulator. An important part of the simulator is the user interface (UI) seen in Figure 4. Using a UI is not mandatory, but it is greatly beneficial for the development process of reinforcement learning agents because trained agents can be imported into the simulator, and their behavior can be studied through the UI. As seen in the figure, the UI displays relevant information about the vessels, and it also contains several buttons for controlling the simulator. Lastly is the helpers module, which was made to reduce code duplication. It contains frequently used functions that can be imported instead of re-written.

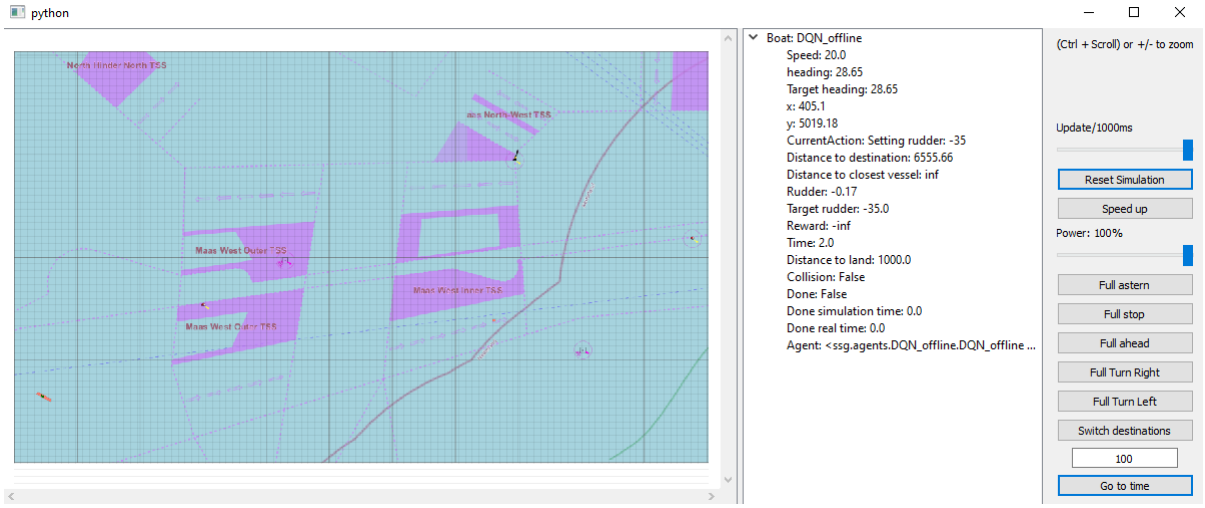


Figure 4: UI of SimpleShipSimulator

4. Implementation

When implementing a reinforcement learning solution, one can choose between two approaches. The implementation can be done either by developing a custom algorithm or by using an already implemented algorithm. For this thesis, the latter option was chosen because it is within the scope of the thesis, and suitable libraries that cover both online and offline reinforcement learning are available.

This chapter is dedicated to explaining the libraries used in the implementation, followed by developing an online and an offline agent.

4.1 Libraries

Libraries are pieces of reusable code that can be easily imported and used in a project. The massive benefit of using popular libraries is that they are thoroughly tested and often broadly applicable. Finding relevant libraries can be of great help, as they enable the developer to focus on the essential tasks instead of developing code that eventually leads up to the task. Usually, a large portion of programming is spent on troubleshooting faulty behavior caused by a bug in the code. Using time-tested libraries minimizes the risk of running into time-consuming errors and significantly speeds up the development process.

4.1.1 TensorFlow

TensorFlow is one of the most prominent frameworks for developing, training, and deploying machine learning systems. It was developed by Google and released as open-source in 2015, along with the paper "*TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*" [15]. Computation in TensorFlow is represented by data flow graphs that define the order in which the computations must be performed. A node in the graph represents a mathematical operation, such as add, subtract or divide. Each edge in the graphs is a multidimensional array called a *tensor*, on which the mathematical operations are done [15].

TensorFlow can be used independently for developing custom machine learning solutions, but it can also be used as the backend for other libraries.

4.1.2 Keras and Keras-rl

Keras is a deep learning Application Program Interface (API) developed in Python and running on a Tensorflow backend. As stated on the website, "Being able to go from idea to result as fast as possible is key to doing good research," and the focus of the library is to enable fast experimentation [16]. Essentially, Keras is a library that makes machine learning more approachable through its high-level API.

Keras-rl is a library containing state-of-the-art deep reinforcement learning algorithms and was built as an extension to the Keras library [17]. Developing and evaluating a reinforcement learning agent with Keras-rl is convenient, as functionality with OpenAI Gym is already integrated. Additionally, it includes several valuable callbacks and metrics to assess the performance of the agent.

Keras-rl was used for developing the online reinforcement learning agent because the SimpleShipSimulator included a collision avoidance agent implemented with this library. Thus, the collision avoidance agent provided a great starting point for developing an online RL agent for maritime navigation.

4.1.3 PyTorch

Like TensorFlow, PyTorch is a popular machine learning framework that has gained traction in recent years. It was developed by Facebook and released as open-source in 2017, along with the paper "*Automatic differentiation in PyTorch*" [18]. PyTorch is based on Torch, a tensor computation library, and the calculations are similar to TensorFlow. PyTorch and TensorFlow's critical difference is that PyTorch uses dynamic data flow graphs, while TensorFlow uses static data flow graphs. Dynamic data flow graphs add flexibility, as they are generated when the program executes, eliminating the need for defining graphs before execution [18]. Additionally, PyTorch's documentation is comprehensive and its API is easy to use. The use of PyTorch has grown exponentially recently, and it is showing great potential in becoming an industry-standard machine learning framework.

4.1.4 d3rlpy

The library used for developing an offline reinforcement learning agent is d3rlpy. It is one of the very few deep reinforcement libraries that provide both online and offline reinforcement learning algorithms. D3rlpy uses PyTorch for the backend and provides an impressive number of implemented algorithms and features, with more updates constantly being added. As stated by the developer, Takuma Seno, the library is the first of its kind and not only meant for researchers but also for real-world applications [19].

D3rlpy was used for the offline RL agent because of its easy-to-use API and extensive documentation. In addition to the algorithms, the library also includes functionality for creating MDPDatasets, which are the datasets used as input for the offline reinforcement learning agents. The online agent could also be implemented with the d3rlpy library, enabling setting up comparable scenarios and quick experimentation with different algorithms. Unfortunately, converting the online agent from using keras-rl to d3rlpy is time-consuming and could not be prioritized.

4.1.5 NumPy

NumPy is an essential library for scientific computing. It provides multidimensional array objects, known as *ND Arrays*, and routines for performing fast operations on arrays, such as mathematical operations or shape manipulation [21]. As previously discussed, tensors are essentially multidimensional arrays, which makes NumPy central in machine learning systems. In addition to the framework libraries TensorFlow and PyTorch using NumPy in core calculations, NumPy is also used for creating the dataset for the offline reinforcement learning agent, later described in detail.

4.1.6 Pandas

Pandas is a lightweight but powerful library built on the Numpy library and is frequently used in machine learning projects. Pandas can read various data formats, such as CSV, and convert them into *data frame* objects. A data frame is a 2-dimensional data structure with rows and columns, similar to a spreadsheet [22]. Data frames are used for analyzing and manipulating data in different ways. In this project, Pandas is used in the data preprocessing stage for reading, manipulating, and saving the CSV files containing ship log data.

4.1.7 Gym

Gym is a library for developing and evaluating reinforcement learning algorithms. It is a very versatile toolkit that is compatible with both TensorFlow and PyTorch [20]. The library includes several pre-built environments for testing reinforcement learning agents, e.g., Atari games and robotics tasks. Arguably the most crucial feature of Gym is that all environments share the same structure. This means that developers can build completely custom environments for testing new ideas in reinforcement learning. Because the SimpleShipSimulator is a custom implementation, it also requires a custom-made Gym environment to enable benchmarking RL agents.

4.2 Setting up a scenario

The map which defines the navigatable space in the SimpleShipSimulator was created using OpenSeaMap [31]. Based on the minimum and maximum coordinates in the dataset, an area shown in Figure 5 was selected. An essential step in making the map is taking notes of the longitude and latitude coordinates of the corners of the map. These coordinates are used for converting the format of the dataset coordinates, as explained later in the chapter.

In addition to the map being created, a training scenario must also be chosen carefully since the offline agent learns solely on the dataset. The starting point and destination must thus be contained in the area of traffic. Based on the traffic seen by visualizing the data points in the dataset, a starting point (square) and destination (circle) seen in Figure 5 were chosen. The same scenario can be used for training both agents.

Figure 5: SSS map showing starting point (square) and destination (circle).

4.3 Online agent design

Before implementing an offline reinforcement learning agent, a decision was made first to develop an online RL agent that learns entirely through trial and error. The decision was made mainly because of two reasons. Firstly, the Simple Ship Simulator contains an online RL agent used for collision avoidance, which could be used as a starting point for a navigation agent. Secondly, an online RL agent's performance can potentially be compared to an offline RL agent's performance. The collision avoidance agent implemented by Sebastian Penttinen [12] was an excellent way to learn about the SSS and reinforcement learning in general. Though there is an evident difference in the functionality of a collision-avoidance agent and a navigation agent, the same code structure can be used for both agents. Penttinen's collision avoidance agent provided the structure for an online RL agent seen in Figure 6. The structure is made of nine functions that together fully describe the parts of the agent. As a start, the *train* function defines the algorithm to be used while training, but first, it calls the *init_model* function to initialize a neural network. The *get_action_space* and *get_observation_space* return the action space and the state space, and they are called when initializing the neural network. In the *step* function, the current state is fetched through the *get_state* function, and the neural network model that was initialized at the beginning of the training predicts the best action to take in the current state. With a predicted action, the *take_action* function executes the action, and the *get_reward* function is called to deliver a reward based on the action made. With the

code structure stated, the functions are ready to be implemented to fulfill the functionality described by the comments in Figure 6.

```
class DQN:
    def load_from_file():
            # Loads a trained model

    def init_model():
            # Returns the neural network model

    def train():
            # Defines the algorithm and initializes training

    def get_action_space():
            # Returns the action space

    def get_observation_space():
            # Returns  the state space

    def get_state():
            # Returns the current state of the ship in the environment

    def step():
            # Make a prediction of best action to take in the current state

    def take_action():
            # Set rudder angle according to prediction

    def get_reward():
            # Returns the reward of an action taken in a time step
```

Figure 6: Code structure of online reinforcement learning agent.

The first and foremost things to define are the action and state spaces. In this case, the agent is designed to predict what the heading should be. The rudder angle will then be adjusted to move the ship towards the predicted heading. Experiments with predicting the rudder angle were conducted, but the approach of predicting the heading proved to be more successful. This might be because setting a rudder angle is not instantaneous. If the agent predicts to set a certain rudder angle, the simulator will execute setting that rudder angle, but the full effect comes with a delay because the rudder turns gradually to a set angle. Therefore, the agent will make false associations between an action and its effect, and the agent will consequently learn a counterproductive action-value function.

The predicted heading is a number between 0 and 359 degrees, with 1-degree increments. Thus, the action space is 360. The observation space, or state space, contains minimum and maximum values for the coordinates, the heading, the speed, and the rudder angle. In this case, the x

coordinates are between 0 and 10150, and the y coordinates are between 0 and 6000. This is because an image with 1015x600 resolution was used with a 0.1 scale. The image scale is set in the configuration file, which loads when initializing a simulation. The heading can be between 0 and 359 degrees, the speed is set to be between 0 and 22 knots, and lastly, the rudder angle can take values between -35 degrees and 35 degrees. A neural network can be initialized with the action space and state space defined. The action space is used to express the input layer's size, and the state space is used for expressing the output layer's size. In this case, four hidden layers were used, with 32,64,128 and 256 tensors.

Rewards are received at every time step through the *reward function* seen in Figure 7. First, the Boolean parameters outOfBounds and destinationReached are checked. If the ship is out of bounds, a negative reward will be distributed based on how far from the goal the agent is at the time step when going outside the map. The Pythagorean theorem can be written as $d = \sqrt{((x\_2 - x\_1)^2 + (y\_2 - y\_1)^2)}$ [29]. It is used in the *distanceTo* function for calculating the distance between the ship and the destination. If the ship has reached the destination, the destinationReached parameter will be true, and a positive reward will be distributed. The agent will also receive rewards at each time step based on the direction and distance to the destination. Calculating the angle between the ship's location and the destination is done with the *math.atan2* function [30]. It enables giving rewards to the agent when it is heading in the right direction. A small positive reward is given if the agent's Course over Ground is within 15 degrees of the direction of the destination. Otherwise, a small negative reward is given. Additionally, the agent also receives a reward based on the distance to the destination, and this reward is always negative. This reward is negative because the agent will try to maximize the total rewards received, and it must thus minimize the distance to the destination. If this were a positive reward instead, the agent would try to move as far away as possible from the destination.

```
def get_reward(self, ship, state, world, outOfBounds, destinationReached):
        """Returns the reward for a taken action """
        reward = 0

        # Ship is outside map
        if outOfBounds:
            reward += -0.02 * (distanceTo(ship.x, ship.target_destination_x, ship.y,
             ship.target_destination_y))

        if destinationReached:
            self.nb_completed += 1
            reward += 10

        slope = helpers.theta(ship.y, ship.target_destination_y, ship.x,
                ship.target_destination_x)

        # Give a small positive reward if ship is heading towards the destination
        if (-15 <= slope <= 15):
            reward += 0.01

        else:
            reward += -0.01

        # Reward for getting closer to the destination
        reward += (
             -0.00001 * (distanceTo(ship.x, ship.target_destination_x, ship.y,
            ship.target_destination_y))
            )

        ship.set_reward(reward)

        return reward
```

Figure 7: Online agent's reward function

Before beginning the training, a *train* function shown in Figure 8 is defined. At the beginning of the function, the custom gym environment is called, and the neural network is initialized. Experience replay is introduced through a replay memory, storing the observations, actions, rewards, and terminal flags of the latest 50 000 time steps. The policy used is a ε-greedy (epsilon greedy) policy linearly annealed over 100 000 time steps. With the ε-greedy policy, the agent will choose the action with the maximum expected reward, but with probability $1 - \varepsilon$, the agent will choose a completely random action [1]. A target network is defined to increase the stability of the training, as suggested by Minh et al. [7]. With these parameters defined, the online DQN agent is initialized using the Keras-rl library [17]. The agent is compiled with the Adam optimized using a learning rate of 0.0001 and mean absolute error metrics. Lastly, the training is started with the *fit* function. After training, the weights and the model are saved. The saved weights are used for building the neural network after training to visualize the behavior in the simulator. The saved model can be used to continue training the agent where the last

session ended. However, any changes made to the parameters mean that the saved model cannot be used for further training, and the training must start over.

```
def train(self, numberOfSteps, warmup=10000):
      env = gym_env.ShipGym(self)
      self.init_model()
      model = DQN.model
      memory = SequentialMemory(limit=50000, window_length=1)
      policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1.0,
                value_min=0.1, value_test=0.05, nb_steps=100000)
      dqn = DQNAgent(
            model=model,
            nb_actions=self.get_action_space().n,
            memory=memory,
            nb_steps_warmup=warmup,
            target_model_update=10000,
            policy=policy,
       )

       # Callback to get loss of each step
       loggedMetrics = Metrics(dqn)

       dqn.compile(Adam(lr=0.0001), metrics=['mae'])

       hist = dqn.fit(env, nb_steps=numberOfSteps, nb_max_episode_steps = 800,
       visualize=False, verbose=2, callbacks=[loggedMetrics, es])
       dqn.save_weights(
            "Weights/DQN/Navigation/LinearAnnealedPolicyWeights" +
             str(numberOfSteps) + ".h5f", overwrite=True
       )
       model.save("Model/latest")
```

Figure 8: Online agents train function

4.4 Offline agent design

Though the algorithm used for the online and the offline agent is the same, the agents are entirely different from each other. The online agent estimates the Q function based on trial-and-error by exploring the environment. In contrast, the offline agent estimates the Q function solely on historical data without any exploration. Therefore, arguably the most critical part of an offline reinforcement learning agent is the data.

Similar to the previous chapter, this chapter will explain the implementation of the offline RL agent. However, since the offline RL agent relies on the dataset containing relevant information and being in the correct format, the chapter will begin with reviewing how the data was interpreted and preprocessed.

Historical AIS data collected from ships can be used for training an offline reinforcement learning agent, but first, it must be cleaned and edited. This stage is also called *data preprocessing* and is necessary because an unprocessed dataset often contains missing values that must be added or irrelevant values that must be removed. Data preprocessing is considered a crucial part of a data-based machine learning implementation and can significantly affect the results.

As previously discussed, the AIS dataset originally contained 54 columns of data. A large portion of the columns are irrelevant for a proof-of-concept agent, and only the minimal number of columns needed to express the ship's state was selected. The state-space of the online reinforcement learning agent was used for reference when determining the relevant columns. For a proof-of-concept offline agent, the parameters needed are latitude, longitude, Speed over Ground, Course over Ground, and the rudder angle. Selecting only the relevant columns from the CSV files was straightforward with the pandas library.

After irrelevant columns have been removed, the remaining data must be edited. More specifically, the format of the coordinates must be modified. There are two reasons for the modifications. Firstly, with the changes, it is possible to evaluate the agent with the SimpleShipSimulator. Secondly, with x and y coordinates, it is possible to introduce rewards to the dataset with the reward function from the online agent. The SSS uses an image of a map, and a location in the simulator is expressed with x and y coordinates according to the resolution and scaling of the image. Therefore, original longitude and latitude coordinates in the dataset must be converted to x and y coordinates. The conversions require knowledge about the image resolution, the scaling, and the minimum and maximum values for the longitude and latitude of the area in the image. With these values, the conversions are done with the two formulas in Figure 9. The *longitude_range* and *latitude_range* are the differences between the maximum and minimum coordinates:

```
# Conversion from longitude to x
x = ((lon * res_width) - (min_longitude * res_width))/longitude_range

# Conversion from latitude to y
y = - (((lat * res_height) - (max_latitude * res_height))/latitude_range)
```

Figure 9: Formulas for converting longitude and latitude to x and y

The formulas in Figure 9 were derived from the reverse conversion done by Richard Nyberg, who developed the formulas seen in Figure 10. Nyberg's formulas were created with the logic that the image's lowest longitude is when $x = 0$, and the highest when $x = res\_width * scaling$. The same logic applies for y , but in reverse, the image's maximum latitude value is when $y = 0$, and the lowest when $y = res\_height * scaling$.

```
# x to longitude
long = min_long + (x/res_width * lon_range)

# y to latitude
lat = max_lat - (y/res_height * lat_range)
```

Figure 10: Richard Nyberg's formulas for converting x and y coordinates to longitude and latitude

Rewards are an essential part of reinforcement learning, and as mentioned, the dataset does not contain rewards initially. Therefore, rewards must be introduced manually. The same reward function used for the online agent was also used for introducing rewards to the dataset. A column with terminal flags was also added to the dataset. A terminal flag is a Boolean value that indicates whether an episode is running or if it has terminated. A terminal flag value of 0 indicates that the episode is running, and a value of 1 indicates that the episode has ended. The format of the dataset at this stage is seen in Figure 11.

	A	B	C	D	E	F	G	H
1	Time	x	y	SOG	COG	Port rudder angle	Reward	Terminal
2	00:00:00	368.66	5207.51	21.02.00	76.29	0.0	-0.16759012693855518	0
3	00:00:01	370.09	5207.17	21.02.00	76.29	0.0	-0.16757543933782454	0
4	00:00:02	371.77	5206.75	21.02.00	76.29	0.0	-0.1675581442132627	0
5	00:00:03	373.45	5206.34	21.02.00	76.3	0.0	-0.1675408684413371	0
6	00:00:04	375.13	5205.93	21.01.00	76.3	0.0	-0.16752359267826322	0
7	00:00:05	376.81	5205.51	21.01.00	76.32	0.0	-0.16750629758489205	0
8	00:00:06	378.49	5205.1	21.01.00	76.26	0.0	-0.16748902184066683	0
9	00:00:07	380.18	5204.7	21.02.00	76.37	0.0	-0.1674716673296281	0
10	00:00:08	381.85	5204.29	20.09.00	76.33	0.0	-0.16745448971462168	0

Figure 11: Dataset before MDPDataset conversion.

Lastly, the dataset was converted into batches of tuples containing state, action, next reward, next state, and terminal flag using the d3rlpy MDPDataset class [19]. One tuple in the MDPDataset corresponds to one row in the dataset in Figure 11.

The *train* function for the offline agent, shown in Figure 11, is similar to the training function of the online agent seen previously in Figure 8 but simpler. The dataset is first loaded and split into a training set and a testing set, with a training set size of 80 percent and a testing set size of 20 percent. The agent is trained with the training set and evaluated with the testing set. Typically in offline reinforcement learning, a gym environment is not used, but d3rlpy includes a convenient function for evaluating the agent using the environment. The offline agent trains without an environment, but after each episode, the evaluate_on_environment function runs. The function tests the agent's model in the environment by selecting actions and returns the rewards collected during that episode.

```
def train(self):
    dataset = MDPDataset.load('E:/Data/MDPDatasets/dataset.h5')
    env = gym_env.ShipGym(self)

    # split into train and test episodes
    train_episodes, test_episodes = train_test_split(dataset, test_size=0.2)

    dqn = DoubleDQN(use_gpu=True)

    # start training
    dqn.fit(train_episodes,
            eval_episodes=test_episodes,
            n_epochs=100,
            scorers={
                'environment': evaluate_on_environment(env),
                'td_error': td_error_scorer,
                'value_scale': average_value_estimation_scorer
            })
```

Figure 11: Offline agents train function

A significant advantage with the offline agent is that the model is saved after each episode during the training. This means that when the training has ended, the metrics can be assessed, and the best-performing model can be used. The saved models can also be used for training an agent further if the training was cut short.

4.5 Changes during implementation

During the implementation of the agents, a couple of noteworthy changes were made. Initially, the agent predicted the rudder angle directly, and the ship's rudder angle was set according to the prediction. This approach may seem logical, but the problem is that changing the rudder angle takes some time, and the agent is therefore at risk of making false correlations between actions and their effects. An example would be when the agent sets the rudder angle to 25 degrees, the rudder angle turns to 25 degrees gradually, not instantly. This could lead to the agent making incorrect correlations between setting the rudder angle to 25 degrees and the state it ends up in because of that action. The agent was changed to predict a desirable heading instead of directly predicting the rudder angle to prevent false correlations. The ship's rudder angle is then calculated and set in the take_action function seen in Figure 12. The difference between the predicted heading and the ships heading is calculated, and the rudder angle is set based on that difference.

```
def take_action(self, action, ship: Ship):

    assert action >= 0 and action <= 359

    heading = ship.heading
    predicted_heading = action
    diff = predicted_heading - heading

    if diff > 180:
        diff = heading - predicted_heading

    # Rudder angle to steer towards the predicted heading
    rudder = min(max(-35, diff), 35)

    ship.set_target_rudder(rudder)
    ship.set_action("Setting rudder: "+str(r))
```

Figure 12: Take action according to predicted heading

The reward function also underwent several changes during the implementation before arriving at the final version. Initially, a straightforward version was made that defined rewards for closing the distance to the destination, reaching the destination, and going out of the map's bounds. Adding out-of-bounds penalties based on the distance to the destination and rewards for heading towards the destination seemed to improve the agent's learning.

5. Results

This chapter will review the results of training the two agents in the area outside Rotterdam. A majority of the chapter is dedicated to the offline agent's results, but first, the results of the online agent are presented briefly.

Evaluation of the agents was done using two approaches: visually inspecting the behavior of the trained agent in the simulator and interpretation of the metrics saved during training. Primarily, the metrics are the total rewards per episode and loss. The loss tells how accurate the agent's predictions are, and a value close to zero is desirable.

5.1 Online agent

The online agent learned to navigate from a starting point to a destination. However, the training was unstable and inconsistent. Attempts of stabilizing the training by tuning the parameters, also known as *hyperparameter tuning*, were made but only stabilized the training marginally. Hyperparameter tuning means finding more optimal values for the parameters, such as the learning rate, discount factor, and target network update interval. Finding suitable values for the hyperparameters is time-consuming because a small change to one parameter can make a huge difference. Therefore, only one parameter can be adjusted at a time, and after each adjustment, the agent must be trained again. Additionally, finding the correct values is done through trial and error, similar to the training of an online agent.

The agent showed clear signs of learning but would suddenly "forget" all training and return to performing actions leading to undesirable scenarios, such as going outside the boundaries or getting stuck doing circles. In reinforcement learning, this is referred to as "catastrophic forgetting" and can be combatted by introducing experience replay [32]. However, as seen in Figure 13, the agent continued performing inconsistently even after introducing a replay memory. The upper graph visualizes the cumulated rewards of all episodes and should ideally show a clear upwards trend as the agent learns to navigate to the destination and receives higher rewards. However, as seen in the graph, the highest amount of rewards are gathered in the middle of the training. The lower graph in Figure 13 shows the loss och each time step, and as seen, the training becomes unstable towards the end. Because of this, the training was ended prematurely at around episode 2800 while the loss was low and rewards were high.

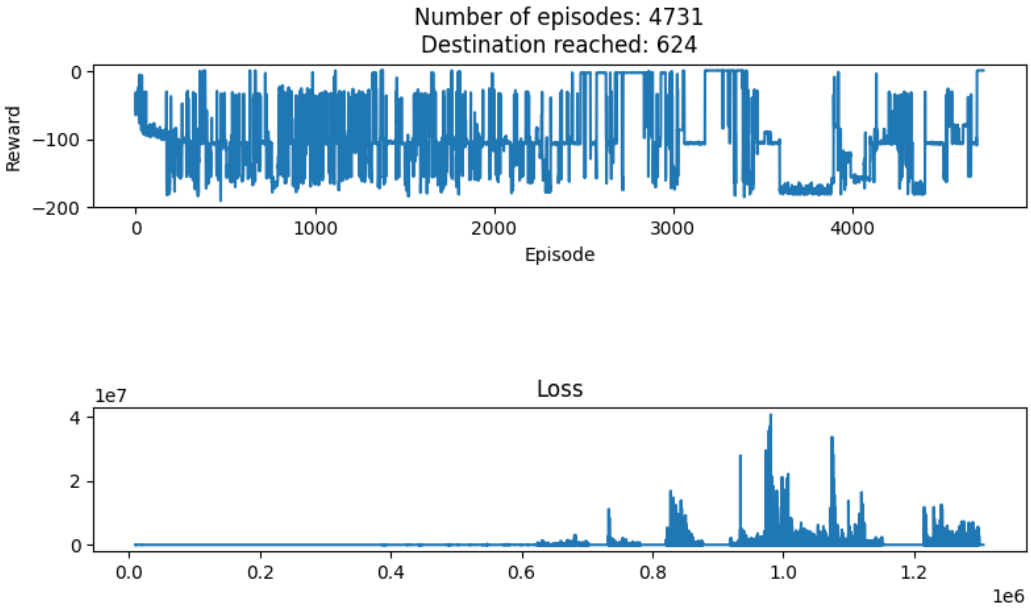


Figure 13: Online agent metrics

Loading the weights of the trained online agent into the simulator shows the behavior visualized in Figure 14. The green line shows the path taken by the agent. In the beginning, the agent has learned that it must turn left to avoid receiving a penalty for going outside the map. The trajectory also shows that the agent has learned that heading towards the destination is rewarding. After the sharp left turn, the agent corrects its course towards the destination with very slight oscillation before reaching the goal.

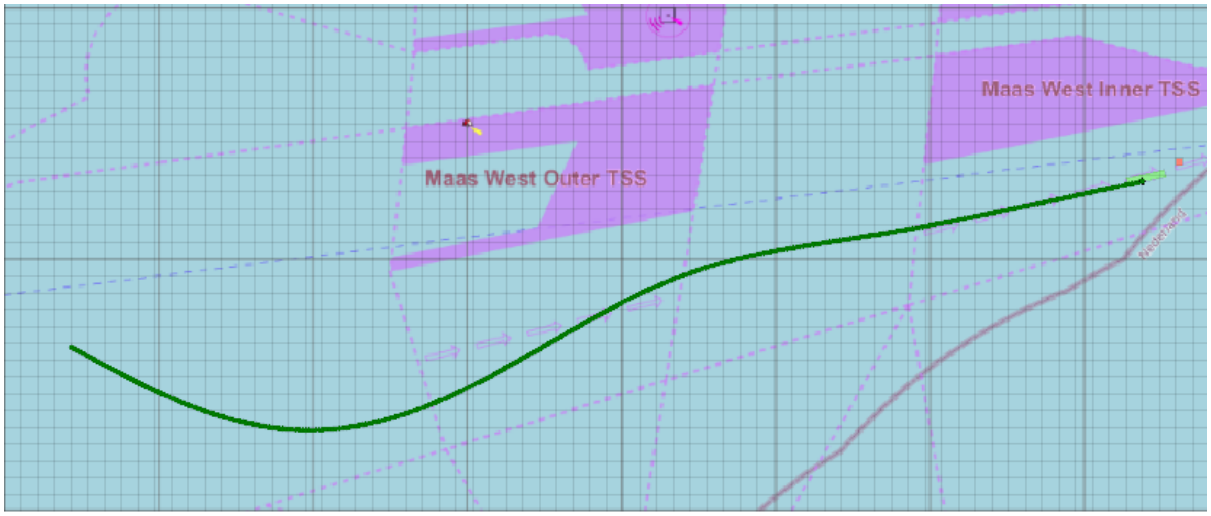


Figure 14: Behavior of trained online agent visualized

5.2 Offline agent

While the training of the online agent was unstable, the offline agent performed surprisingly well, given the small dataset. The offline agent reached the goal within 90 episodes with converging loss and temporal difference (TD) error, as seen in Figure 15. Low TD error is desirable, as large values indicate that the Q function overfits the training set [19]. With both the loss and TD error converging, the training is stable. As seen in the upper graph, the agent reaches the goal three times, indicated by the peaks in the graph. As discussed earlier, the dataset is scarce, and these graphs perfectly reflect that more data is needed for more effective learning. The graphs indicate stable training, but the agent does not have access to enough data to learn to reach the destination consistently. Although the agent struggles to learn to navigate to the destination, the results are useful, and the behavior can be visualized. Because the agents model is saved after each training episode, the best-performing model can be selected. As seen in Figure 15, the agent reached the destination at episode 86 with minimal loss and TD error.

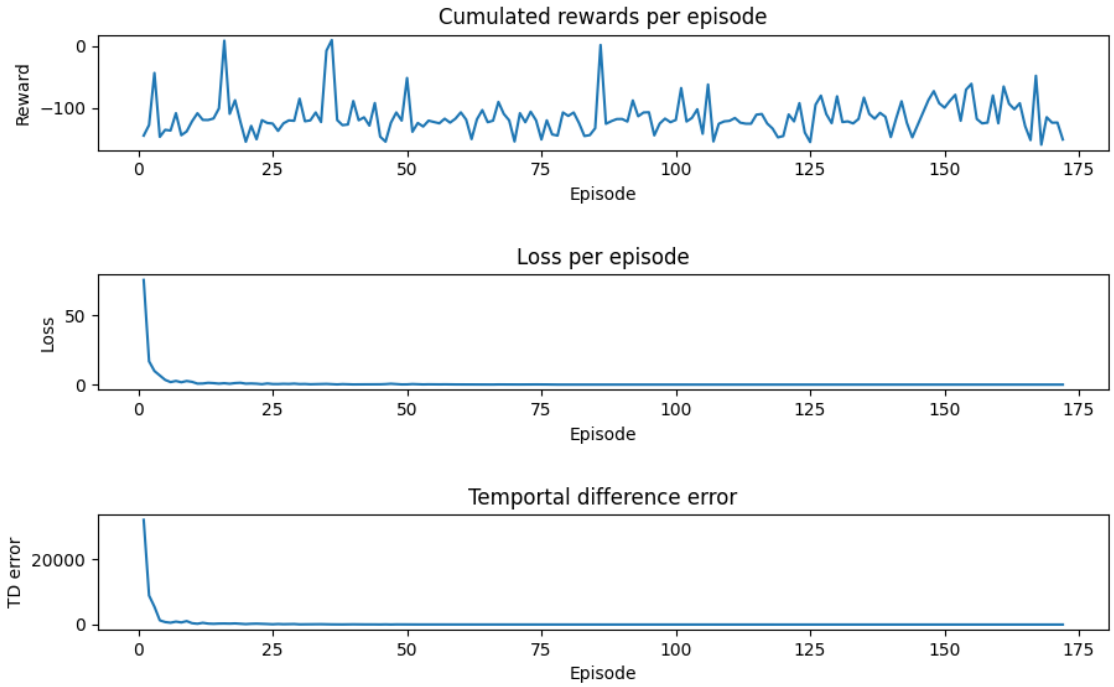


Figure 15: Offline agent metrics

Loading the agent's model from episode 86 into the simulator yields the behavior shown in Figure 16. The trajectory, marked by the dotted green line, is almost identical to the path learned by the online agent. A slight difference between the two trajectories is expected since the reward function is not very specific to speed up training, and both paths yield almost the same rewards.

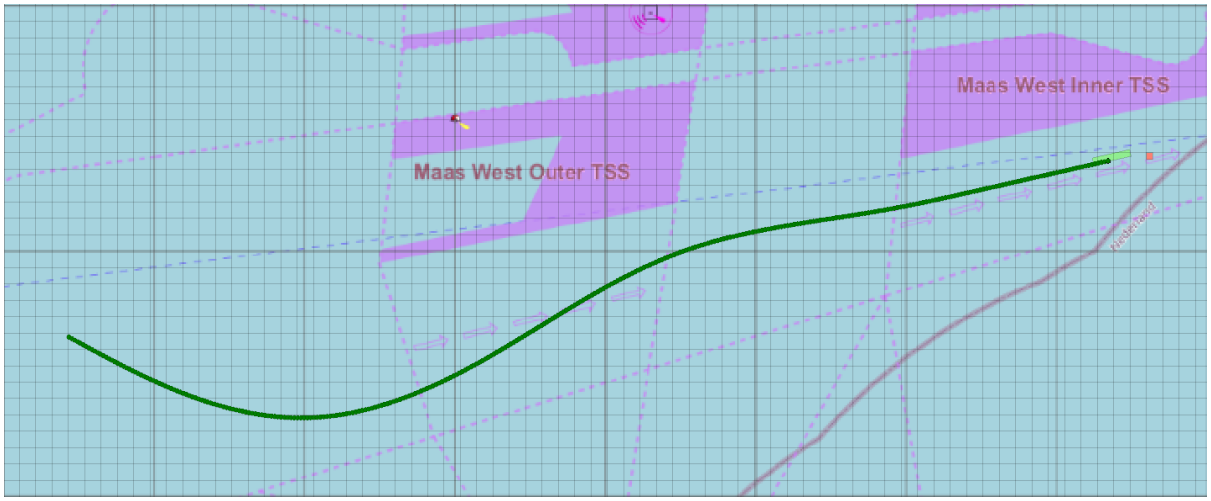
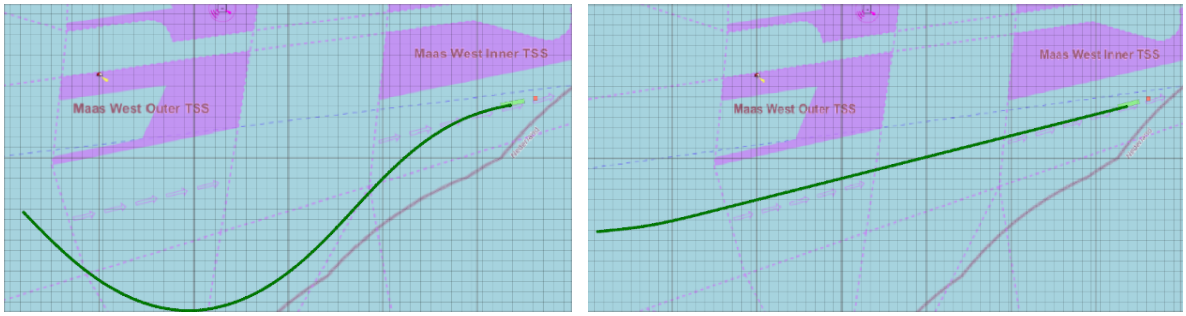


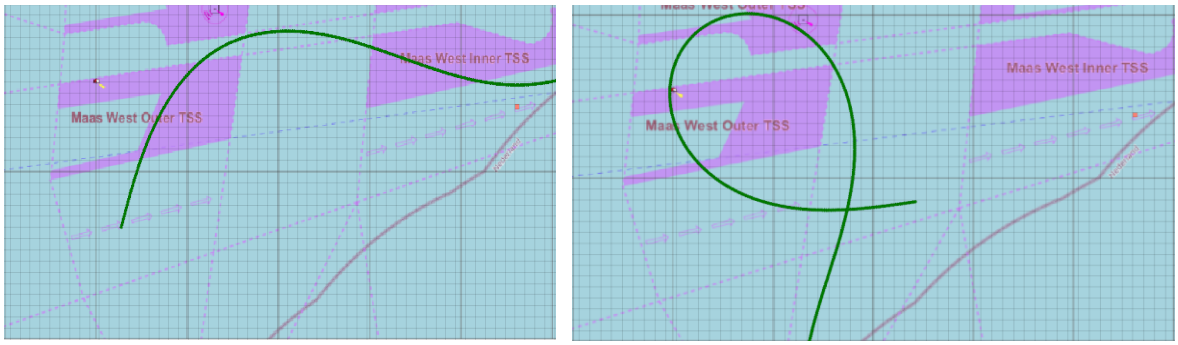
Figure 16: Offline agent behavior visualized

The trained offline agent's behavior was further tested by setting a few different starting points and starting directions. A successfully trained agent is expected to know how to navigate to the destination if the starting point is set within the trained area. The results of several different starting scenarios within the area are visualized in Figure 17. In Figure 17 a), the agent's starting point is set closer to the destination, and the agent's heading is set farther away from the direction of the destination. To correct the heading, the agent turns left until the course is straightened towards the destination. In Figure 17 b), the agent starts with a heading almost in course with the destination, and as expected, the agent only alters the course slightly before heading towards the destination. In Figures 17 c) and d), the agent is presented with scenarios that it could not solve, and the destination was missed. In Figure 17 c), the agent is set to start navigating south. Still, it nearly manages to correct the course and only barely misses the destination. In Figure 17 d), the agent's heading is set in the opposite direction of the destination. In this case, the agent starts with a sharp turn right to steer the ship in the opposite direction but does not manage to straighten out the course.

The agent performed poorly in the last two cases, probably caused by the narrow distribution of the ship traffic in the dataset. The traffic is very concentrated, and all traffic is headed in the same direction, as seen earlier in Figure 3, Chapter 3. Levine et al. [5] stated that in offline reinforcement learning, a narrow distribution of state-action pairs usually results in a brittle and non-generalizable solution, explaining the behavior in Figure 17 c) and d). When the agent was set with such a heading that it was forced outside the trained area, it encountered unfamiliar states and could not take the correct actions.



a) Steeper heading *b) Heading towards the destination*



c) Heading south *d) Heading opposite of destination*

Figure 17: Trained offline agent tested in four different scenarios in the simulator.

The agent's ability to generalize, i.e., to perform in scenarios outside the trained area, was further tested, but the agent's performance was deficient. Poor performance outside the trained area was expected, as an offline reinforcement learning agent cannot explore the environment, and consequently, it cannot learn to generalize.

Overall, the offline agent performed well. However, the metrics visualized earlier in Figure 14 clearly show that the agent has trouble learning the desired behavior. This can indicate that the agent is not well optimized and could benefit from hyperparameter tuning. Nevertheless, since the training is stable, it most likely suggests that the amount of data is insufficient. The lack of data was recognized as an upcoming challenge in this project and is discussed in Chapter 7.

6. Discussion

The findings of this thesis suggest that AIS data can be used for teaching an offline reinforcement learning agent maritime navigation. As presented in the previous chapter, the offline reinforcement learning agent learned to navigate from a starting point to a destination relatively quickly despite the small dataset. However, the results strongly suggest that the learning of the offline agent is limited by the size and diversity of the dataset. This was expected, as it is known that a small dataset may result in a brittle solution. The results also indicate that compared to an online agent, the offline agent learns faster and is less computationally costly. Still, because the agents are implemented using two different libraries, they are not entirely comparable, and therefore a thorough comparison would not be meaningful at this time. Additionally, it is believed that the learning of both agents is limited by a false state representation, further explained in the next chapter.

Implementation-wise, the agents are very different. The offline agent requires a relatively small amount of code, and the majority of time spent developing the offline agent was during data preprocessing. An interesting remark about the offline agent is that it does not require an environment for training. In this case, an environment was used for evaluation since it is difficult to know what the agent has learned without it. Still, the agent is fully capable of learning entirely without an environment. This is a significant advantage if the agent is known to learn the desired behavior successfully because not using an environment speeds up the training.

Another interesting remark is the agents' abilities to learn to solve the task at all. Despite the flawed state representation, the agents understood the ship's steering dynamics in some cases and found the way to the destination. More successful results can be expected with the effects of actions being expressed correctly through changes in the state representation.

As earlier discussed, hyperparameter tuning can be tedious as it can be done only one parameter at a time. Compared to an online agent, an offline agent benefits from faster training iterations and fewer hyperparameters. Therefore, optimizing an offline agent can be quicker and easier. Hyperparameter tuning was not relevant due to the limited results caused by false state representation and small dataset size but could be pursued once those issues are addressed.

Developing an autonomous navigation system involves breaking the system down into smaller subproblems that can be solved with different methods. Offline reinforcement learning shows potential and can likely be used in a larger autonomous path planning system when combined

with other machine learning methods. The need to utilize existing AIS data is evident as large amounts of valuable historical data are available. Furthermore, expressing a reward function that teaches an agent to navigate a ship in all possible real-world scenarios correctly is unreasonable. Meanwhile, that behavior may lie hidden in historical AIS data, with the possibility of being exploited. However, further work is required to make conclusions about the effectiveness of offline reinforcement learning.

7. Future work

Several aspects of ship navigation were simplified or overlooked during the implementation of these proof-of-concept agents due to lack of time. These aspects must be addressed in the future if these approaches are to be used in more realistic automated navigation systems. This chapter is meant to provide a baseline for future research. It is dedicated to discussing the possible improvements left untested due to late discoveries and time constraints.

7.1 Online agent

To pinpoint the exact reason for the online agent's stability issues is not easy. As earlier discussed, the primary and most probable cause is the delayed effect of the actions. A possible solution would be to change the state representation to capture the effect of an action fully. The current implementation represents the agent's current state with the coordinates, heading, speed, and rudder angle but does not consider the turning rate of the rudder. Without the knowledge of the turning rate, the agent is at risk of making false assumptions about the effect of setting a certain rudder angle.

One approach to introduce knowledge about the turning rate is the *Constant Delayed Markov Decision Process* (CDMDP), as proposed by Walsh et al. [33]. The CDMDP is represented by the 6-tuple (S, A, P, R, γ, k). The 6-tuple is an extension of the 5-tuple MDP discussed in Chapter 3 but with the addition of k, which represents the delay. The value of k tells how many time steps there are between taking an action and the effect of the action. In this case, k would be the number of time steps between setting a rudder angle and reaching the desired rudder angle. The value of k can be calculated by multiplying the turning rate of the rudder with the absolute value of the difference between the target rudder angle and the current rudder angle, according to the formula:

$$rudder_{rate} * |target_{rudder} - current_{rudder}|$$

The rudder turning rate is specific for each ship and is usually expressed in rad/s. However, it is unclear if CDMDP is applicable because the rudder turning rate is not delayed but *gradual*, meaning that the effect of an action occurs gradually rather than with a delay. The ship is affected immediately in the following time step after the agent sets a rudder angle, but the full effect does not occur until k time steps. Additionally, traditional MDP algorithms, such as DQN, cannot be used directly with a CDMDP because the MDP formulation relies on the

assumption that an action is executed immediately. Therefore, the MDP must be modeled, which can be done in several ways. One method is to augment the state-space, though this approach may yield limited results for large state-spaces [33].

A more promising approach would be to define the ship steering dynamics according to the first-order Nomoto model developed by Nomoto et al. in 1957 [35]. The Nomoto model defines the relationship between the rudder angle and ship motion. It takes a rudder angle as input and outputs the ship motion change. In their paper *A knowledge-free path planning approach for smart ships based on reinforcement learning*, Chen et al. use the Nomoto model to define the state of their agent [36]. The first-order Nomoto equation can be expressed as

$$T\dot{r} + r = K\delta \tag{1}$$

with the notation $\dot{\psi} = r$, where ψ is the heading of the ship. Equation 1 can be written as

$$T\ddot{\psi} + \dot{\psi} = K\delta \tag{2}$$

where T is the turning lag coefficient, K is the turning ability coefficient, r is the yaw rate, and δ is the rudder angle. The yaw rate r at time t can be calculated with the equation

$$r = K\delta_0\left(1 - e^{-\frac{t}{T}}\right) \tag{3}$$

The yaw rate r is the time derivative of the ship heading ψ. Therefore, ψ can be calculated with

$$\psi = K\delta_0\left(t - T + T * e^{-\frac{t}{T}}\right) \tag{4}$$

Lastly, as explained by Chen et al., these formulas coupled with a ship motion coordinate system can be used to express Equation 5, which can be used for calculating the position and heading of a vessel at any time:

$$\begin{cases} x(t) = x(0) + \int v \sin\psi \, dt \\ y(t) = y(0) + \int v \cos\psi \, dt \end{cases} \tag{5}$$

where $x(0)$ and $y(0)$ are the initial coordinates, v is the speed, and ψ is the heading of the ship. Using these formulas for state representation should give the agent adequate knowledge about the ship steering dynamics. With this knowledge, the agent has a much better chance of learning the desired behavior.

In addition to the state representation improvements, the online agent would also benefit from being implemented with d3rlpy, the same library used for implementing the offline agent. D3rlpy is frequently updated with fixes and new methods, and it provides valuable metrics for evaluating and comparing agents. Additionally, d3rlpy includes several new and powerful reinforcement learning algorithms, such as *Batch Constrained Q-learning* (BCQ) [37] and *Conservative Q Learning* (CQL) [38], which could be evaluated.

7.2 Offline agent

Currently, the biggest obstacle with the offline agent is the limited size and diversity of the dataset. The agent clearly showed that it is capable of learning navigation based on ship traffic but could not learn beyond a certain point because the data was not enough. The dataset could be enlarged by either further collecting AIS data or by data augmentation. Still, both approaches come with new challenges.

A technique used widely for increasing the size of datasets is data augmentation. It involves different ways of slightly modifying existing data to create new and valuable data for training. Image data is relatively easily augmented through techniques such as flipping, rotating, or cropping images. The problem with AIS data is that it is not easily augmented, as it requires extensive knowledge about navigation and the parameters within the dataset. For example, modifying coordinates to create new paths is possible. Still, it must be done precisely to achieve useful ship trajectories and avoid trajectories in certain areas, such as in restricted areas or on land. Trajectories can be modified by, e.g., flipping the starting point and destination or introducing a slight deviation to the heading of a ship to create a new trajectory. These methods of slightly modifying trajectories can result in a much larger dataset but at the cost of low

diversity. Methods for augmenting diverse and realistic ship trajectories must still be developed to increase AIS datasets' size successfully.

Collecting additional AIS data from the simulator at Aboa Mare is possible, but the process is tedious and involves many stages. Additionally, the question arises if the available data at Aboa Mare is enough in size and diversity. A straightforward solution would be to buy a large amount of AIS data from a data selling platform. The downside of buying AIS data is that it may prove costly. Additionally, data augmentation might still be necessary despite acquiring a relatively large and diverse dataset since it is unclear how much data is needed for effective learning.

Another essential aspect of offline reinforcement learning is the rewards and datasets collected from real-world applications seldomly contain rewards initially, which also is the case with AIS data. Rewards are the core stimulus to a reinforcement learning agent. Therefore, if the rewards are missing, they must be added to the dataset during the preprocessing stage. As discussed in Chapter 4, the rewards in the AIS dataset used for this offline agent were added using the reward function of the online agent. This approach is not ideal because the reward function is relatively simple and only meant for proof-of-concept. The reward function could be further developed to account for more advanced navigational scenarios, such as sea channels and sea marks. However, a human-made reward function can be wrong, resulting in the agent learning undesirable behavior.

A method for avoiding a lousy reward function could be to use *Inverse Reinforcement Learning* (IRL) to find the optimal reward function. The idea of IRL is to use an expert's demonstrations regarded as the optimal policy, which has been derived according to some reward function R, and the goal of IRL is to learn this unknown reward function [34]. When using AIS data, the expert's demonstrations would be the trajectories within the dataset. After an optimal reward function has been learned, it could be used for introducing rewards to the dataset. Moreover, the optimal reward function could also be used for training an online reinforcement learning agent.

Though it seems like the offline agent is not as sensitive as the online agent regarding delayed effects of actions, it would undoubtedly also benefit from an improved state representation using the first-order Nomoto model to express the ship steering dynamics.

7.3 General

On a general note, there are several opportunities for further research on topics closely related to this thesis. Given that both the online and the offline agents are implemented comparably and trained successfully, it would be interesting to see the differences in training times, performance, and computational requirements. Optimization through hyperparameter tuning would be required before such comparisons can be made. Prerequisites to hyperparameter tuning are correct state representation for both agents and a more prominent and diverse dataset for the offline agent. A comparison of different reinforcement learning algorithms would also be intriguing and highly relevant, as new and powerful algorithms, such as BCQ and CQL, are rising in popularity.

8. Conclusion

In this thesis, the possibility of using historical AIS data with offline reinforcement learning for autonomous ship navigation was researched, implemented, and evaluated. The project was done as part of a larger research project led by the MAST! Institute. The data used was previously collected from a realistic ship simulator at Aboa Mare. The reinforcement learning agents were implemented, tested, and evaluated in a separate simulator, the SimpleShipSimulator. The evaluation was done by visually inspecting the agents' behavior in SSS and analyzing metrics collected during the training.

An online RL agent that learns by exploring the environment was implemented first. The online agent was implemented as an introduction to the SimpleShipSimulator and for comparison purposes. The online agent provided limited results with unstable training. The unstable training was determined to be caused by the delayed effect of actions. Specifications about the false state representation were discussed, and improvement ideas were formalized.

An offline RL agent learning based on AIS data was also implemented and was the core of the thesis project. The AIS data was preprocessed into an MDPDataset and used as input when training the agent. The offline agent proved to be stable during training and successfully learned to solve the task of navigating from a starting point to a destination. Still, the agent had trouble solving the task consistently. Undoubtedly the offline agent suffered from false state representation caused by the delayed effect of actions. However, stable training also indicates that the offline agent's trouble learning is caused by the dataset's small size and variance. Ideas for tackling the limited size and distribution of the dataset were presented.

In conclusion, autonomous navigation with offline reinforcement learning based on AIS data is possible but requires more work. The project unveiled both expected and unexpected challenges. Some of the challenges were overcome, while the remaining discovered challenges were left for future research, along with suggested approaches.

9. Swedish summary

Autonom sjöfartsnavigering med fartygsdata och datadriven förstärkt inlärning

9.1 Introduktion

Autonoma system har utvecklats drastiskt under det senaste årtiondet, och alltmer avancerade system appliceras inom alla möjliga områden. Inom transport läggs en stor vikt på att automatisera fordon, exempelvis bilar, flygplan och fartyg. Företag som Tesla och Google är bland de ledande utvecklarna av autonoma system för självkörande bilar, och deras delvis självkörande system finns redan i vissa nya bilar [23]. Inom flygtransport används autonoma system bland annat i obemannade luftfarkoster som transporterar varor [24]. Den största delen av varor transporteras dock via sjötransport och autonoma system utvecklas även inom detta område. En undersökning visar att över 80 procent av den globala handeln sker via sjötransport [25]. En annan undersökning visar även att fartyg är det enda rimliga alternativet för att frakta varor i stor volym, eftersom det är överlägset mest kostnadseffektivt [26].

Autonoma fartyg kan innebära stora besparingar, eftersom obemannade fartyg kan konstrueras utan kommandobrygga och bostadsutrymmen för besättning. Detta ger möjligheten att bygga fartyg med mer utrymme för last och med förbättrad aerodynamik, vilket reducerar bränsleförbrukningen. Utöver förminskade kostnader förknippade med underhåll och bemanning har autonoma system även potential att minska antalet olyckor, eftersom uppskattningsvis 75 procent av alla sjöfartsolyckor är orsakade av mänskliga fel [27]. Autonom sjöfart är således ett område av stort och aktuellt intresse. Denna avhandling undersöker möjligheten att använda förstärkt inlärning baserat på historiska fartygsdata för att automatisera navigeringen av ett fartyg.

9.2 Förstärkt inlärning

Förstärkt inlärning är ett område inom maskininlärning som fokuserar på att automatisera uppgifter genom att definiera ett mål och regler som måste följas för att uppnå målet. Skillnaden mellan förstärkt inlärning och andra metoder inom maskininlärning är att förstärkt inlärning är baserat på belöningar och straff som ges till följd av handlingar. Handlingar utförs av en *agent* och den interagerar med en *miljö* [1]. Traditionell förstärkt inlärning är baserad på att agenten utforskar miljön genom att välja en handling enligt den situation agenten befinner sig i och utdelas en positiv eller negativ förstärkning, beroende på om handlingen är önskvärd eller ej. En handling ger upphov till en förstärkning och en ny situation, varpå agenten igen väljer en handling. På detta vis fortsätter interaktionen mellan agenten och miljön tills agenten nått målet eller tills en definierad gräns uppnåtts. Träningen av en förstärkt inlärningsagent är typiskt indelat i *episoder*. En episod är över när agenten antingen når målet eller när den når en definierad gräns, oftast uttryckt som en tidsgräns. Målet med förstärkt inlärning är att agenten ska lära sig en optimal regel som anger hur agenten ska handla för att få maximal belöning. En optimal regel definierar vilken handling som leder till mest belöning, i alla möjliga tillstånd [1]. En väsentlig del av ett förstärkt inlärningssystem är belöningsfunktionen, som måste definieras av utvecklaren. Belöningsfunktionen består av matematiskt formulerade uttryck som beskriver belöningarnas storlek och vilka situationer som ger upphov till belöningar. Traditionell förstärkt inlärning kan användas när en miljö finns tillgänglig för agenten att utforska, och dess styrka ligger i att agenten kan tränas trots att data saknas.

Förutom traditionell förstärkt inlärning finns också en relativt ny metod, nämligen *datadriven förstärkt inlärning*, vilket också är fokuset för denna avhandling. Inom datadriven förstärkt inlärning lär sig agenten en regel genom att enbart utnyttja en datamängd, istället för att utforska en miljö. Fördelen med datadriven förstärkt inlärning är att agenten kan lära sig utan att använda en miljö. I många verkliga fall, exempelvis inom sjukvård och robotik, saknas träningsmiljöer och det kan vara för dyrt eller rent av farligt att låta en traditionell förstärkt inlärningsagent testa sig fram utan en träningsmiljö. Ifall relevant data har blivit insamlat över en längre tid finns det dock potential att använda datamängden för att träna en agent utan någon miljö.

Den matematiska grunden till all förstärkt inlärning är en *Markovian Decision Process* (MDP). En MDP består av en samling matematiska formler som bland annat gör det möjligt att förutspå nästa tillstånd och nästa belöning, givet ett tillstånd och en handling [1]. En MDP uttrycks som en 5-tupel: (S, A, P, R, γ), där S är en samling tillstånd, A är en samling handlingar, P är en

samling sannolikheter för tillståndsövergångar, R är en samling belöningar och γ är en diskonteringsfaktor.

9.3 Projekt

Projektet i denna avhandling är en del av ett större forskningsprojekt mellan Åbo Akademi och Yrkeshögskolan Novia. Samarbetet kallas ”The Institute of Maritime Software Technology” (MAST! Institute) och forskningen berör digitalisering av sjöfart och autonoma fartyg [10].

Vid Novias utbildningslinje för sjöfart, Aboa Mare, används en realistisk fartygssimulator vid skolning av studeranden. Målet med denna avhandling är att använda historiska data från Aboa Mares fartygssimulator med datadriven förstärkt inlärning för att uppnå autonom sjöfartsnavigering. Att utvärdera en datadriven förstärkt inlärningsagent kan vara svårt utan referens. Därför implementeras även en agent med traditionell förstärkt inlärning, som lär sig genom att testa sig fram i miljön. Träning av den traditionella agenten och evaluering av båda agenterna sker i en separat simulator, *SimpleShipSimulator* (SSS), implementerad av Kim Hupponen [11] och Sebastian Penttinen [12]. Simulatorn blev implementerad som ett tidigare MAST! Institute projekt och är gjord för att utveckla och utvärdera maskininlärningsalgoritmer.

9.4 Implementation

Implementationen av den traditionella agenten gjordes med hjälp av *Keras-rl* biblioteket [17] och den datadrivna agenten gjordes med hjälp av *d3rlpy* [19] biblioteket. Biblioteken består av välutvecklad och återanvändbar kod som täcker grundläggande matematiska beräkningar involverade i en MDP. Biblioteken användes för att möjliggöra ett större projekt och för att försnabba utvecklingsprocessen. Båda agenterna använder *Deep Q-Network* (DQN) algoritmen, som utnyttjar ett neuralt nätverk för att estimera värden för olika handlingar i olika tillstånd [7].

Den traditionella agentens belöningsfunktion definierades så att agenten ständigt får en liten negativ bestraffning baserat på avståndet mellan agenten och målet. Agenten försöker då att minimera bestraffningen och därmed närmar den sig mål. Dessutom får agenten belöning om riktningen mot mål är inom 15 grader från agentens kurs och en större belöning utdelas när agenten når destinationen. Utgående från belöningsfunktionen lär sig agenten att förutspå

vilken handling som lönar sig i vilket tillstånd. Tanken är att agenten lär sig att förutspå vilken riktning den bör ha för att ha kurs mot destinationen och baserat på detta justeras fartygets roder.

Implementationen av den datadrivna agenten gick till stor del åt att förbehandla datamängden före den kunde användas för inlärning. Ur den ursprungliga datamängden valdes endast relevanta kolumner som behövdes för att uttrycka fartygets tillstånd. Ytterligare ändrades formatet av koordinaterna för att stämma överens med SSS. Datamängden saknade även belöningar och för att introducera belöningar användes samma belöningsfunktion som för traditionella agenten. Slutligen konverterades datamängden till satser av tupler för att motsvara formatet av en MDP. Likt den traditionella agenten lär sig den datadrivna agenten att förutspå hur kursen ska justeras för att ha riktningen mot destinationen.

9.5 Resultat

Båda agenterna tränades på ett område utanför Rotterdam med samma startpunkt och destination. Området valdes eftersom datamängden innehöll tät trafik i det området, vilket är en förutsättning för att den datadrivna agenten ska lära sig. Den traditionella agenten lärde sig navigera till destinationen, men träningen var inkonsekvent och varierade kraftigt. Försök till att stabilisera träningen gjordes men utan större framgång. Däremot visade den datadrivna agenten större potential och den lärde sig att navigera till mål snabbare. Trots goda resultat visar den datadrivna agenten tydliga tecken på att datamängden inte är tillräckligt stor. En klar nackdel med den datadrivna agenten är också att den inte kan generalisera, vilket betyder att den inte klarar av att navigera på områden utanför datamängden. Detta innebär att en stor mängd data krävs för att få ett bra resultat.

För stabilare och framgångsrikare resultat för båda agenterna krävs att styrningen av fartyget uttrycks matematiskt. I nuläget är inte agenterna medventa om att rodret svängs gradvis, utan de antar att rodret kan svängas till en viss vinkel omedelbart. Detta leder till att agenterna kan göra falska antaganden om hur fartyget påverkas av att svänga rodret.

9.6 Slutsats

En traditionell förstärkt inlärningsagent och en datadriven agent implementerades. De utvärderades baserat på utvärderingsvärden och på deras beteenden visualiserade i SimpleShipSimulator. Resultatet för den datadrivna agenten är hoppingivande och tyder på att insamlad fartygsdata och datadriven förstärkt inlärning kan användas för autonom sjöfartsnavigering eller som en del av ett större system. Resultaten begränsas dock av den minimala datamängden och är inte tillräckligt för att dra slutsatser om hur effektivt datadriven förstärkt inlärning för sjöfartsnavigering är. En större datamängd krävs för att möjliggöra mer omfattande testning och utvärdering. Datamängden kunde utökas genom dataförstoring eller genom ytterligare insamling av fartygsdata. Förutom diskussioner kring förstoring av data föreslogs även metoder för att förbättra båda agenternas inlärningsförmågor. Det mest centrala förbättringsförslaget är att uttrycka fartygets styrning enligt första ordningens Nomoto-modellen.

References

[1] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge: The MIT Press, 2015.

[2] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to algorithms*, 3rd ed. Cambridge: The MIT Press, 2009, pp. 359-414.

[3] D. Marinescu, Cloud computing, 2nd ed. Cambridge: Elsevier, 2017, pp. 482-487.

[4] L. Green and J. Myerson, "A Discounting Framework for Choice With Delayed and Probabilistic Rewards.", *Psychological Bulletin*, vol. 130, no. 5, pp. 769-792, 2004. Available: 10.1037/0033-2909.130.5.769 [Accessed 6 March 2021].

[5] S. Levine, A. Kumar, G. Tucker and J. Fu, "Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems", *arXiv.org*, 2020. [Online]. Available: https://arxiv.org/abs/2005.01643v3. [Accessed: 13 March 2021].

[6] C. Watkins and P. Dayan, "Q-learning", *Machine Learning*, vol. 8, no. 3-4, pp. 279-292, 1992. Available: 10.1007/bf00992698 [Accessed 24 March 2021].

[7] V. Mnih et al., "Human-level control through deep reinforcement learning", *Nature*, vol. 518, no. 7540, pp. 529-533, 2015. Available: 10.1038/nature14236 [Accessed 24 March 2021].

[8] N. Deshpande and A. Spalanzani, "Deep Reinforcement Learning based Vehicle Navigation amongst pedestrians using a Grid-based state representation*", *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, 2019. Available: https://hal.inria.fr/hal-02409042. [Accessed 24 March 2021].

[9] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks", *arXiv.org*, 2021. [Online]. Available: https://arxiv.org/abs/1511.08458v2. [Accessed: 29 March 2021].

[10] "MAST Institute!", [Online]. Available: https://mastinstitute.fouprojekt.novia.fi/. [Accessed: 29 March 2021].

[11] K. Hupponen, "A simulator for evaluating machine-learning algorithms for autonomous ships", *Doria.fi*, 2020. [Online]. Available: https://www.doria.fi/handle/10024/177441. [Accessed: 30 March 2021].

[12] S. Penttinen, "COLREG compliant collision avoidance using reinforcement learning", *Doria.fi*, 2020. [Online]. Available: https://www.doria.fi/handle/10024/177467. [Accessed: 30 March 2021].

[13] W. Porter, "How To Calculate Course Over Ground (Illustrated Guide)", *ImproveSailing*. [Online]. Available: https://improvesailing.com/guides/how-to-calculate-course-over-ground-illustrated-guide. [Accessed: 30 March 2021].

[14] "GPS Visualizer", *Gpsvisualizer.com*. [Online]. Available: https://www.gpsvisualizer.com/.

[15] M. Abadi et al., "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems", *arXiv.org*, 2015. [Online]. Available: https://arxiv.org/abs/1603.04467. [Accessed: 31 March 2021].

[16] "Keras: the Python deep learning API", *Keras.io*, 2021. [Online]. Available: https://keras.io/. [Accessed: 31 March 2021].

[17] M. Plappert, *Keras-rl*, 2016. [Online]. Available: https://github.com/keras-rl/keras-rl. [Accessed: 31 March 2021].

[18] A. Paszke et al., "Automatic differentiation in PyTorch", *OpenReview*, 2017. [Online]. Available: https://openreview.net/forum?id=BJJsrmfCZ. [Accessed: 05 April 2021].

[19] T. Seno, "d3rlpy: An offline deep reinforcement library", 2020. [Online]. Available: https://github.com/takuseno/d3rlpy. [Accessed: 05 April 2021].

[20] "Gym: A toolkit for developing and comparing reinforcement learning algorithms", *Gym.openai.com*. [Online]. Available: https://gym.openai.com/. [Accessed: 06 April 2021].

[21] "NumPy", *Numpy.org*. [Online]. Available: https://numpy.org/. [Accessed: 06 April 2021].

[22] "pandas - Python Data Analysis Library", *Pandas.pydata.org*, 2021. [Online]. Available: https://pandas.pydata.org/. [Accessed: 06 April 2021].

[23] Y. Gu, J. Goez, M. Guajardo and S. Wallace, "Autonomous vessels: state of the art and potential opportunities in logistics", *International Transactions in Operational Research*, vol.

28, no. 4, pp. 1706-1739, 2020. Available: https://onlinelibrary.wiley.com/doi/10.1111/itor.12785. [Accessed 7 April 2021].

[24] *Wing*. [Online]. Available: https://wing.com/fi_fi/. [Accessed: 23 March 2021].

[25] UNITED NATIONS CONFERENCE ON TRADE AND DEVELOPMENT., *REVIEW OF MARITIME TRANSPORT 2020*. [s.l.]: UNITED NATIONS, 2020.

[26] Y. Gu, S. Wallace and X. Wang, "Integrated maritime fuel management with stochastic fuel prices and new emission regulations", *Journal of the Operational Research Society*, vol. 70, no. 5, pp. 707-725, 2018. Available: https://www.researchgate.net/publication/322267651_Integrated_maritime_fuel_management_with_stochastic_fuel_prices_and_new_emission_regulations. [Accessed 7 April 2021].

[27] Allianz, "SAFETY AND SHIPPING REVIEW 2020", 2020. [Online]. Available: https://www.agcs.allianz.com/news-and-insights/reports/shipping-safety.html. [Accessed: 07 April 2021].

[28] *Consolidated text of The 1974 SOLAS Convention, The 1978 SOLAS Protocol, The 1981 and 1983 SOLAS Amendments*. London: Imo, 1986.

[29] "Distance in the Coordinate Plane", *Courses.lumenlearning.com*. [Online]. Available: https://courses.lumenlearning.com/waymakercollegealgebra/chapter/distance-in-the-plane/#:~:text=Derived%20from%20the%20Pythagorean%20Theorem,the%20length%20of%20the%20hypotenuse. [Accessed: 09 April 2021].

[30] "math — Mathematical functions — Python 3.9.5 documentation", *Docs.python.org*. [Online]. Available: https://docs.python.org/3/library/math.html. [Accessed: 09- Apr- 2021].

[31] "OpenSeaMap: Startseite", *Openseamap.org*. [Online]. Available: http://openseamap.org/. [Accessed: 09 April 2021].

[32] M. Rostami, S. Kolouri and P. Pilly, "Complementary Learning for Overcoming Catastrophic Forgetting Using Experience Replay", *arXiv.org*, 2019. [Online]. Available: https://arxiv.org/abs/1903.04566. [Accessed: 14 April 2021].

[33] T. Walsh, A. Nouri, L. Li and M. Littman, "Learning and planning in environments with delayed feedback", *Autonomous Agents and Multi-Agent Systems*, vol. 18, no. 1, pp. 83-105, 2008. Available: https://link.springer.com/article/10.1007%2Fs10458-008-9056-7. [Accessed 14 April 2021].

[34] S. Zhifei and E. Meng Joo, "A survey of inverse reinforcement learning techniques", *International Journal of Intelligent Computing and Cybernetics*, vol. 5, no. 3, pp. 293-311, 2012. Available: 10.1108/17563781211255862 [Accessed 14 April 2021].

[35] K. Nomoto, T. Taguchi, K. Honda and S. Hirano, "On the steering qualities of ships", *International Shipbuilding Progress*, vol. 4, no. 35, pp. 354-370, 1957. Available: 10.3233/isp-1957-43504.

[36] C. Chen, X. Chen, F. Ma, X. Zeng and J. Wang, "A knowledge-free path planning approach for smart ships based on reinforcement learning", *Ocean Engineering*, vol. 189, p. 106299, 2019. Available: https://www.sciencedirect.com/science/article/abs/pii/S0029801819304706. [Accessed 3 May 2021].

[37] S. Fujimoto, D. Meger and D. Precup, "Off-Policy Deep Reinforcement Learning without Exploration", *arXiv.org*, 2018. [Online]. Available: https://arxiv.org/abs/1812.02900. [Accessed: 05 May 2021].

[38] A. Kumar, A. Zhou, G. Tucker and S. Levine, "Conservative Q-Learning for Offline Reinforcement Learning", *arXiv.org*, 2020. [Online]. Available: https://arxiv.org/abs/2006.04779. [Accessed: 05 May 2021].