Engineering Cybernetics
TTK4550 Specialization Project

Investigating performance of Deep Reinforcement Learning algorithms for Path-Following and Collision Avoidance in Autonomous Vessels

Halvor Ødegård Teigen

Supervisor:
Professor Adil Rasheed

Trondheim, June 7th, 2021

Faculty of Information Technology and Electrical Engineering
DEPARTMENT OF ENGINEERING CYBERNETICS

NTNU
Norwegian University of
Science and Technology

Preface

This report is written as a part of TTK4550 Specialization Project at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology (NTNU). The project started as a wide literature review of reinforcement learning (RL) and the use of multi-agent RL for autonomous vessels. This was later narrowed down to exploring algorithms and investigating their performance for this application.

I am grateful to Eivind Meyer for providing an excellent software framework and for his help regarding this. Additionally, I would like to thank Torkel Laache and Thomas Larsen for being great discussion partners throughout the work. Finally, I would like to thank Professor Adil Rasheed for always being available for questions and for his guidance and supervision during this project.

Trondheim, 7.6.2021
Halvor Ødegård Teigen

Contents

List of Figures

List of Tables

Nomenclature

Abbreviations

A2C	Advantage Actor Critic
ACKTR	Actor Critic using Kronecker-Factored Trust Region
AI	Artificial Intelligence
CTE	Cross-Track Error
DDPG	Deep Deterministic Policy Gradient
DL	Deep Learning
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
GAIL	Generative Adversarial Imitation Learning
LSTM	Long Short-Term Memory
M	Million, ex: 1.5M
MADRL	Multi-Agent Deep Reinforcement Learning
MDP	Markov Decision Process
ML	Machine Learning
NED	North-East-Down
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
SAC	Soft Actor-Critic
TD	Temporal Difference
TD3	Twin Delayed DDPG

Other

$\bar{P}$	Normalized Performance
$\bar{U}$	Normalized Usability

Abstract

In this project, we explore various Deep Reinforcement Learning algorithms and investigate their performance in the application of path-following and obstacle-avoidance for autonomous vessels. This is done through training of multiple agents for each algorithm as well as extensive generalization testing. A custom *performance function* is developed to create a quantitative measure of performance for comparison of the selected algorithms. A *usability function* that takes both performance and training time into consideration is also developed.

The results show that PPO and ACKTR clearly outperform the other algorithms, both in terms of performance on the training scenario and generalization performance in real-world scenarios. PPO stands out in terms of usability and outperforms all algorithms across all tests. This proves PPO to be the preferred algorithm in this application.

Sammendrag

I dette prosjektet utforsker vi et utvalg Deep Reinforcement Learning algoritmer og ser på hvor egnet hver av disse er for trening av styringssystemer for autonome fartøy. Dette inkluderer både følging av en forhåndsbestemt rute og unngåelse av hindringer på veien. Dette gjøres gjennom trening av flere agenter for hver algoritme, samt omfattende testing av generaliseringsevne. En spesialtilpasset ytelsesfunksjon utvikles for å skape et kvantitativt mål på ytelse og muliggjøre sammenligning av algoritmene. En brukbarhetsfunksjon som tar høyde for både ytelse og treningstid utvikles også.

Resultatene viser at PPO og ACKTR er klart bedre enn alle de andre algoritmene som ble testet, både når det gjelder ytelse på trenings-scenarioet og evne til å generalisere styringsstrategien til realistiske scenarioer. PPO skiller seg ut når det gjelder brukbarhet og overgår samtlige algoritmer over samtlige tester. Dette viser at PPO er den foretrukne algoritmen for dette problemet.

Chapter 1

Introduction

In the universe of Reinforcement Learning (RL), there is a multitude of algorithms to choose from, each with its own advantages and disadvantages. The optimal choice of algorithm is highly dependent on the problem formulation and has to be explored through testing and gathering of statistical evidence. We extend the work of (Meyer, 2020) and investigate the performance of various RL algorithms in the problem of path-following and obstacle avoidance for autonomous ships.

1.1 Motivation and Background

Autonomous vessels

In recent years, the development and interest in autonomous shipping has increased significantly. According to the *International Chamber of Shipping* (2020), around 90% of the world's trade is transported via shipping. This is a testament to the high demand for development in this area. Both the Norwegian government and EU has provided a large amount of support for research within this field. Examples are, the European Union's Horizon 2020 research and innovation program giving 20M Euros to the AUTOSHIP project (*AUTOSHIP Project*, 2020), and The Norwegian Ministry of Transport and Communications giving NOK 12.5M to the county of "Møre og Romsdal" for the development of new autonomous passenger ferries and vessels. The Norwegian government has also made the Trondheim Fjord into a test bed for autonomous ship trials, and established the *Norwegian Forum for Autonomous Ships (NFAS)* consisting of some of the biggest actors within the maritime industry.

According to (de la Campa Portela, 2005), it is commonly accepted that around 80% of maritime accidents are due to human error. This is a significant amount and there is great potential for improvement with the use of autonomy. With autonomy comes other challenges but there is still a significant possibility to improve overall safety for autonomous ships compared to manned (Hoem et al., 2019). Autonomous ships also have the possibility to improve working conditions, lower damage-related and crew costs, and improve the ship's environmental performance (*Norwegian Forum for Autonomous Ships (NFAS)*,

2020). However, this requires vessels capable of acting on their own and able to handle unexpected changes in the environment.

Deep Reinforcement Learning

The requirement for robustness and ability to handle challenging, and infinitely many, situations makes the task of developing an autonomous vessel extremely challenging. Autopilot design for path following is a well-known discipline and has robust solutions using traditional methods. The big challenges appear when we are to combine this with situational awareness and obstacle avoidance. With such a large space of possible actions and strategies, explicitly programming the behavior is near infeasible and far from practical. This, in combination with the need for complex models of ship dynamics, leads us towards a different approach. Using model-free Reinforcement Learning (RL) removes the need for complex models and explicit behavioral programming. The RL agent learns the end-to-end connection between observations and actions through the principle of trial and error, which has shown great results in applications such as games (Silver et al., 2016), robotics (Niroui et al., 2019), and natural language processing (He et al., 2016).

Although RL as a concept has proven powerful for a variety of problems, the choice of RL algorithm is still highly application dependent. (Meyer, 2020) presents a solution for the problem of obstacle avoidance and path-following for an autonomous ship using the PPO algorithm. The motivation behind this project is to back up the findings of (Meyer, 2020) and provide statistical evidence for the performance of a selection of RL algorithms in this application.

1.1.1 State of the art

Deep Reinforcement Learning is a rapidly developing field with the state of the art constantly evolving. There are a number of toolkits, frameworks and libraries available for implementing, testing and comparing reinforcement learning algorithms in various applications.

The OpenAI Gym toolkit (Brockman et al., 2016) has quickly become a state-of-the-art framework for RL applications and is widely used in research within this field. Its popularity reflects its ease of use, flexibility, and powerful capabilities. The toolkit is made for developing and comparing RL algorithms and has a number of predefined benchmark environments for this.

Stable Baselines (Hill et al., 2018) is a set of improved implementations of reinforcement learning algorithms based on OpenAI Baselines. It has en easy to use interface and many state-of-the-art algorithms implemented, such as

PPO (Schulman, Wolski, Dhariwal, Radford and Klimov, 2017), DDPG (Lillicrap et al., 2015), and A2C (Mnih et al., 2016). Deep Q-Networks (DQN) (Mnih et al., 2013), an extension of Q-learning, is also widely used in state-of-the-art applications.

For the application of autonomous vessels, (Meyer, 2020) presents a state-of-the-art approach using the OpenAI Gym toolkit (Brockman et al., 2016), Stable Baselines (Hill et al., 2018), and the PPO RL algorithm implemented there. It is explained that PPO gave the best performance but not much evidence for this is presented. To the best of our knowledge, no one has provided a justification of this choice through comprehensive exploration and comparison of algorithms.

1.2 Research Objectives and research questions

1.2.1 Objectives

Our primary objective is to investigate and provide statistical evidence for the performance of a selection of RL algorithms applied to path-following and obstacle avoidance for an autonomous vessel.

The secondary objectives are stated as:

- Do a systematic quantitative testing of relevant algorithms applied to this problem.
- Provide a custom performance metric to enable a quantitative comparison of the algorithms.
- Show the generalization performance of the RL agents in realistic scenarios.

1.2.2 Research Questions

To the best of our knowledge there is currently no published work on comparing algorithms in a systematic and quantitative way for this application. To this end, the guiding questions governing the research can be stated as:

- Which RL algorithms show the best results for path-following and obstacle-avoidance for autonomous vessels?
- How does the performance of different algorithms compare quantitatively?
- How do these RL agents generalize to realistic scenarios?

1.3 Outline of Report

The thesis comprises of the following sections and content: **chapter 2** explains the fundamental theory behind the work in this project; **chapter 3** dissects the concrete methods and setup used; **chapter 4** presents the results and a discussion around them. The thesis concluded in **chapter 5**, where suggestions for future work are also presented.

Chapter 2

Theory

In this chapter, we present the relevant theory for the work done in this project. Namely a brief introduction to the dynamics of marine vessels and a dive into Deep Reinforcement Learning and a selection of relevant RL algorithms.

2.1 Dynamics of marine vessels

In order to develop a realistic environment for the agent to train in, we need to model the vessel dynamics. In this section, we provide a brief overview of these dynamics. For a more comprehensive derivation, the reader is directed to (Fossen, 2021).

To represent the position and orientation of the vessel, we define two coordinate frames, NED and BODY. These are shown in Figure 2.1.1.

NED (North-East-Down) is a reference frame with the origin fixed to the Earth's surface and the following orientation:

- x_n - axis points towards north
- y_n - axis points towards east
- z_n - axis points downwards, normal to the Earth's surface

BODY is a reference frame with the origin fixed to the vessel. The position and orientation of the vessel are described relative to the inertial reference frame, NED. The orientation of the axes is defined as follows:

- x_b - longitudinal axis (directed from aft to fore)
- y_b - transversal axis (directed to starboard)
- z_b - normal axis (directed from top to bottom)

Assumption 1 (3 DoF restriction)**.** *The vessel is always located on the surface, with zero pitch and roll angle.*

Including Assumption 1, the only parameters needed are position, heading, and their corresponding velocities. Descriptions of these are shown in Table 2.1.1. Our simulated vessel is modeled with an aft thruster providing a thrust of T_u in the x_b direction, and a rudder providing a moment about the z_b-axis of T_r.

Symbol	Description
x^n	Position in NED, along x_n-axis.
y^n	Position in NED, along y_n-axis.
ψ	Rotation about the z_b-axis (heading).
u	Velocity in BODY, along x_b-axis.
v	Velocity in BODY, along y_b-axis..
r	Angular rate about the z_b-axis (heading rate).

Table 2.1.1: SNAME notation for the relevant vessel coordinates.

Assumption 2 (No disturbances)**.** *There are no external disturbances to the vessel such as wind, ocean currents or waves.*

Using Assumption 2, the vessel dynamics (derived in Fossen (2021)) can be expressed as

$$\begin{aligned} \dot{\boldsymbol{\eta}} &= \mathbf{R}_{z,\psi}(\boldsymbol{\eta})\boldsymbol{\nu} \\ \mathbf{M}\dot{\boldsymbol{\nu}} + \mathbf{N}(\boldsymbol{\nu})\boldsymbol{\nu} &= \mathbf{B}\mathbf{f} \end{aligned} \tag{2.1.1}$$

where

- $\boldsymbol{\eta} = [x^n, y^n, \psi]^T$
- $\boldsymbol{\nu} = [u, v, r]^T$
- $\mathbf{f} = [T_u, T_r]$ is the control input vector
- $\mathbf{R}_{z,\psi}$ is the rotation matrix for a rotation of ψ about the z_n-axis
- $\mathbf{M}$ is the mass matrix of the vessel
- $\mathbf{N}$ incorporates the Coriolis and damping effects
- $\mathbf{B}$ is the actuator configuration matrix

Cross-track error

For the path-following part of the problem, a measurement of how well the vessel is actually following the path is needed. This is where the *cross-track error* (CTE) is used. Formally, the cross-track error is a measure of the closest Euclidean distance from the path to the vessel and is commonly used as a measure of path-following performance. Figure 2.1.1 explains the cross-track error graphically. The path is generated as a continuous, smooth, parameterized curve using 1D Piecewise Cubic Hermite Interpolator (PCHIP), which is provided by the Python library SciPy (Virtanen et al., 2019).

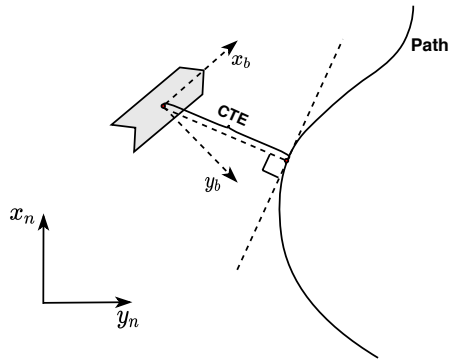


Figure 2.1.1: Definition of cross-track error displayed graphically. The coordinate frames NED (x_n, y_n) and BODY (x_b, y_b) are also shown.

2.2 Deep Reinforcement Learning

In this section, we introduce the relevant theory within the field of Deep Reinforcement Learning (DRL). For a more comprehensive read, the reader is directed to (Sutton and Barto, 2018) and (Li, 2018).

In the realm of Artificial Intelligence (AI), Machine Learning (ML) has been the most popular approach in recent years. We usually categorize ML by supervised, unsupervised, and reinforcement learning. In supervised learning, the desired output needs to be known in order to train the model, i.e. labeled data is needed. This can be used in applications like regression and classification. Unsupervised learning seeks to find patterns and relevant information within unlabeled data. Reinforcement learning (RL) takes an altogether different approach that uses the principle of trial and error to extract an optimal strategy to solve a problem. In this section, we will dive deeper into the topic of DRL. The term *deep* refers to the use of deep neural networks in machine learning approaches. This can be incorporated into all of the above-mentioned categories.

An overview of different approaches to RL is presented in Figure 2.2.1. As shown, we will focus on the model-free RL approach. On the one hand, we have value-based methods based on Temporal Difference (TD) Learning, while on the other we have policy gradient methods based on policy optimization. The algorithms explored in this project all adopt the actor-critic framework which combines functionality from both of these methods and is extensively used for continuous control applications. The upcoming parts of this section will introduce the reinforcement learning framework, explain the most important terms needed to understand the RL algorithms and their differences, explain the two main approaches to RL, and eventually lead to the actor-critic framework and why it is used.

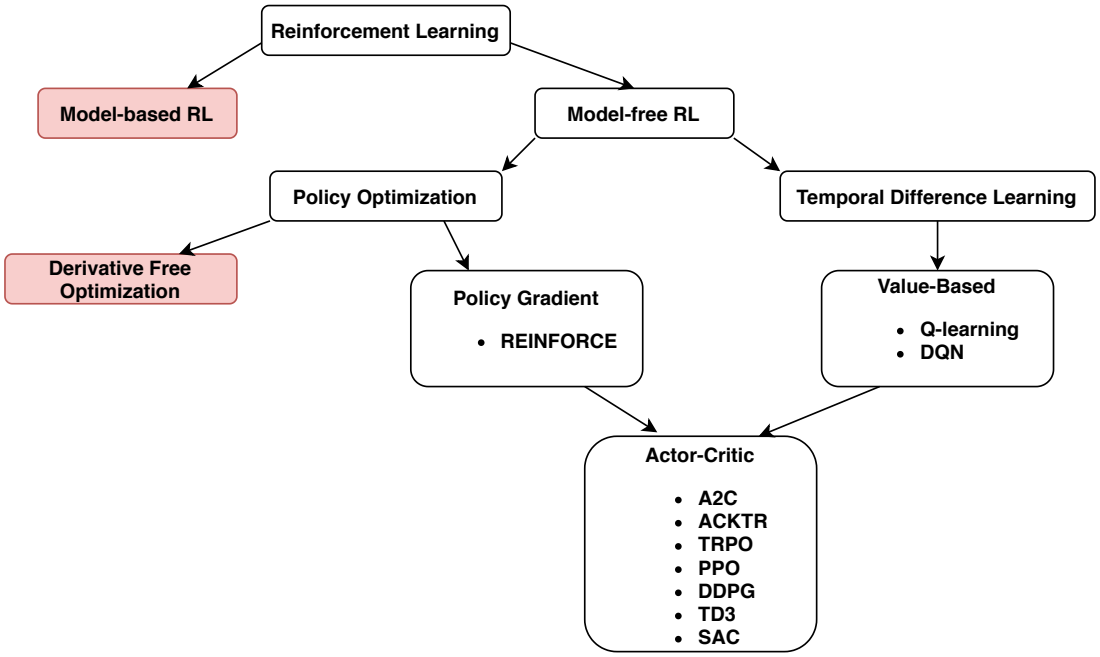


Figure 2.2.1: Overview of the most common Reinforcement Learning approaches. The red-colored topic-boxes are not discussed in this report.

2.2.1 Preliminaries

The two main components of reinforcement learning are the environment and the agent. These entities interact through actions, rewards, and states or observations. From a high-level perspective, the flow can be explained as follows: The agent performs an action a_t on the environment which changes the state from s_t to s_{t+1}. The new state s_{t+1}, or a partial observation of it, is received by the agent along with a reward r_t indicating how good the action was. This reward is then used to improve the policy $\pi(a_t|s_t)$, which is a set of rules that the agent follows to decide which action to take next. A graphical representation of this flow can be found in Figure 2.2.2.

An assumption made in many RL algorithms is that the problem can be formulated as a Markov Decision Process (MDP). A key attribute of this is that the future states depend only on the current state and action, not the past. The *model* of a MDP is defined by a transition function T giving the probability of moving to a state s' given a state s and an action a, and a reward function R giving the reward. If the model of the MDP is known, traditional optimization techniques can be used to find the optimal policy. This is often not the case and an approach like RL is needed to solve this.

The goal for the RL algorithm is to find the optimal policy, and it does this

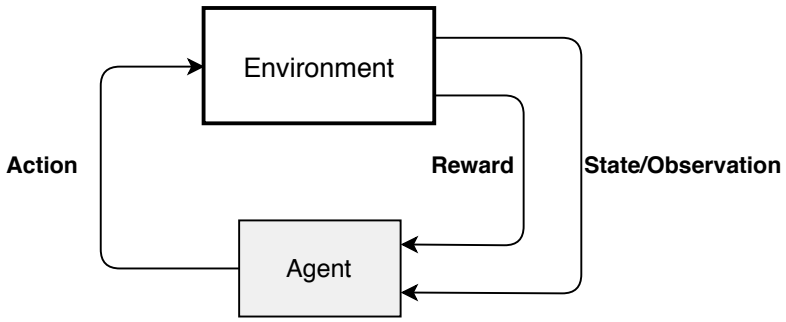


Figure 2.2.2: Overview of the Reinforcement Learning framework.

by maximizing the cumulative reward

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \tag{2.2.1}$$

This contains a discount factor $\gamma \in (0, 1]$ which dictates how much the agent cares about future rewards. A γ of 1 gives equal weight to all rewards, regardless of temporal conditions, while a smaller γ results in the short term rewards getting weighted higher than the long term. A $\gamma < 1$ is usually preferred because, in general, a good action (thus a large reward) is worth more now than far into the future.

Sequential vs Episodic

A problem can be formulated as either sequential or episodic. In a sequential formulation the task is never-ending, which means there is no end state and the reward must be assigned during the execution of the task. In an episodic formulation the task at hand can be viewed as a finite time problem that has one or more terminal states. This introduces the possibility of a reward being assigned both during and at the end of each episode. Our problem is formulated as episodic.

Exploration vs Exploitation

An important consideration in RL is the trade-off between exploration and exploitation. On the one hand, we need the agent to explore the environment and evaluate as many different strategies as possible and stop it from converging to a local optimum with sub-optimal performance. On the other hand, we want the agent to exploit the information in the current policy and not have completely random behavior. So how much of this exploration should the agent do, and how should it do it? This is a big question within RL and comes down to tuning for the respective algorithm and application.

On-policy vs Off-policy

An important thing to understand in differentiating RL algorithms is the difference between on-policy and off-policy approaches.

In summary, *On-policy methods attempt to evaluate or improve the policy that is used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data.* (Sutton and Barto, 2018)

In off-policy algorithms, the policy that is improved, called the *target policy*, is often an approximation to the optimal policy, which is typically deterministic, whereas the data generating *behavior policy* is often stochastic, exploring all possible actions in each state as part of finding the optimal policy (Maei et al., 2010). Advantages of separating the behavior policy from the target policy are that it allows more freedom for exploration, enables learning from data generated by a human (imitation learning), and learning from previously generated data for better sample efficiency. Downsides to off-policy training are difficulty getting reliable estimation, as it suffers from either high bias or high variance (Tosatto, 2020), and the tendency to be less stable than on-policy algorithms. Examples of off-policy algorithms are Q-learning, DQN, DDPG, TD3 and SAC.

In on-policy algorithms, the target and behavior policy are the same. This means that the policy π is updated with data collected by π itself. As (Sutton and Barto, 2018) states: *focusing on the on-policy distribution could be beneficial because it causes vast, uninteresting parts of the space to be ignored.* This may be beneficial in our case because exploring states far from the path is undesirable. A downside to this is that it may get trapped in local optima resulting in a suboptimal policy. Examples of on-policy algorithms are PPO, ACKTR, and A2C. (Chilamkurthy, 2020)

Temporal Difference (TD) Learning

As we will see, TD learning an important concept in RL. It is based on updating estimates based in part on other learned estimates, which is called bootstrapping. This tends to reduce variance and accelerate learning (Sutton and Barto, 2018). More importantly, TD learning has the ability to learn directly, and iteratively, from raw experience *without a model of the environment*. As discussed in section 1.1, this is essential. A conceptual update rule for TD learning can be expressed as:

$$\text{NewEstimate} = \text{OldEstimate} + \text{StepSize}(\text{Target} - \text{OldEstimate}) \tag{2.2.2}$$

This update rule is a core component in value-based RL methods, and algorithms like Q-learning and SARSA are based on it.

2.2.2 Value-based methods

To explain value-based methods we introduce two functions, the value function and the action-value function. The value function estimates how good it is for an agent to be in a given state. Formally, it is an expectation of the future reward given the state, and is expressed as

$$V_\pi(s) = E[R_t|s_t = s] \tag{2.2.3}$$

The value function can then, through its recursive properties, be expressed as

$$V_\pi(s) = \sum_{s',r} T(s,a,s')(R(s,a,s') + \gamma V^\pi(s')) \tag{2.2.4}$$

where $T(s,a,s')$ is the transition probability of moving to a state s' given a state s and an action a, and R is the reward. This shows that the expected value of the state is defined in terms of the immediate reward, through R, and values of possible next states, through γV^π, weighted by their transition probabilities.

The action-value function, or Q-value function, is defined as

$$Q_\pi(s,a) = E[R_t|s_t = s, a_t = a] \tag{2.2.5}$$

In addition to the state, this takes into account the action, i.e. it gives the expected future reward for a given *state-action pair*.

Finding the optimal policy π^* is a matter of finding the action that maximizes the expected reward. Using Equation 2.2.4 we get

$$V^*(s) = \max_a \sum_{s',r} T(s,a,s')(R(s,a,s') + \gamma V^*(s')) \tag{2.2.6}$$

This is known as the Bellman optimality equation and gives us a recursive formula for calculating the optimal value function. This can be solved with traditional optimization techniques when T and R are known. Analogous to Equation 2.2.6 the optimal action-value can be expressed as

$$Q^*(s,a) = \sum_{s',r} T(s,a,s')(R(s,a,s') + \gamma \max_{a'} Q^*(s',a')) \tag{2.2.7}$$

The relation between Q^* and V^* is then given by

$$\begin{aligned} V^*(s) &= \max_a Q^*(s,a) \\ Q^*(s,a) &= \sum_{s',r} T(s,a,s')(R(s,a,s') + \gamma V^*(s')) \end{aligned} \tag{2.2.8}$$

and finally, the optimal action is given by

$$\pi^*(s) = a^* = \arg\max_a Q^*(s, a) \tag{2.2.9}$$

The challenge in model-free approaches is that the transition and reward models T and R are unknown, which means that Equation 2.2.9 cannot be solved analytically. This creates a need to sample the MDP to gather statistical knowledge about this unknown model. As we know, TD learning introduces this ability. In Q-learning an update rule based on Equation 2.2.2 and 2.2.7 is used to find $Q^*(s, a)$. Pseudocode for the Q-learning algorithm is presented in algorithm 1 where this update rule is shown. A popular extension of this is Deep Q-learning which uses a neural network to represent this action-value function instead of a table. The reader is directed to (Mnih et al., 2013) for a comprehensive look at this approach.

Although value-based methods have some great properties and features like good sample efficiency and fast learning, an important thing to note is that they, in general, do not scale well to continuous action spaces. Imagine the action-value function $Q(s, a)$ as a table of values for states and actions. For a continuous action space, there are infinitely many possible action values, which in turn means that the table will get infinitely large. Equation 2.2.9 shows that the maximum of this table has to be found, and for an infinite set of values this becomes very difficult and computationally expencive. Due to this weakness, value-based methods are often combined with policy-based in what we call an actor-critic framework when applied to continuous problems.

Algorithm 1: Q-learning, from (Sutton and Barto, 2018)

initialize Q arbitrarily
foreach *episode* **do**
  s is initialized as the starting state
  repeat
    choose an action $a \in \mathcal{A}$ based on Q and an exploration strategy
    perform action a
    observe the new state s' and received reward r
    $Q(s, a) = Q(s, a) + \alpha \left(r + \gamma \cdot max_{a' \in A(s')} Q(s', a') - Q(s, a)\right)$
    $s = s'$
  until s' *is a goal state*;

2.2.3 Policy-based methods

While value-based methods optimize the policy through a value function, policy-based methods have a different, and more direct, approach. The policy

$\pi(a|s;\boldsymbol{\theta})$, parameterized with parameters $\boldsymbol{\theta}$, is optimized directly through gradient ascent on the expected reward. The parameters $\boldsymbol{\theta}$ can for instance be the weights of a deep neural network.

This approach comes with several advantages. Policy gradient methods have the ability to find stochastic optimal policies, something that action-value methods do not have (Sutton and Barto, 2018). In many problems, the best policy may not be deterministic and the optimal action has to be represented with probabilities. An example of this is in card games with imperfect information such as poker. The policy itself may also be a simpler function to approximate than the action-value function. In addition to this, policy parameterization is a good way to introduce prior knowledge of the problem (Sutton and Barto, 2018), which is very useful from an engineering perspective. Earlier, we mentioned that value-based methods scale poorly to growing action spaces. This problem is not as prominent in policy-based methods because instead of computing learned probabilities for each of the actions, it learns statistics of the probability distribution (Sutton and Barto, 2018). There are of course drawbacks to policy gradient methods, with the most significant being sample inefficiency and high variance.

REINFORCE (Williams, 1992) is a widely used policy gradient method that many algorithms build upon. This algorithm is commonly used with a baseline estimate of the reward to reduce variance. The baseline can be chosen arbitrarily as long as it does not vary with the action taken (Sutton and Barto, 2018). A good choice for this function is the advantage function δ = *discounted reward - estimated value*. The advantage function can be thought of as a measure of how much better the action that the agent took was compared to the expectation of what would happen in that state.

Algorithm 2: REINFORCE with baseline (episodic), from (Sutton and Barto, 2018)

Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathbf{w})$
Input: algorithm parameters: step size $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathbf{w}} > 0$
Initialize policy parameter $\boldsymbol{\theta}$ and state-value weights $\mathbf{w}$ (e.g. to $\mathbf{0}$)
foreach *episode* **do**
  Generate an episode $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$ following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
  foreach *step of episode* $t = 0, 1, \ldots, T-1$ **do**
    $G = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\delta = G - \hat{v}(S_t,\mathbf{w})$
    $\mathbf{w} = \mathbf{w} + \alpha^{\mathbf{w}} \delta \nabla \hat{v}(S_t,\mathbf{w})$
    $\boldsymbol{\theta} = \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} \gamma^t \delta \nabla \log \pi(A_t|S_t,\boldsymbol{\theta})$

2.2.4 Actor-Critic methods

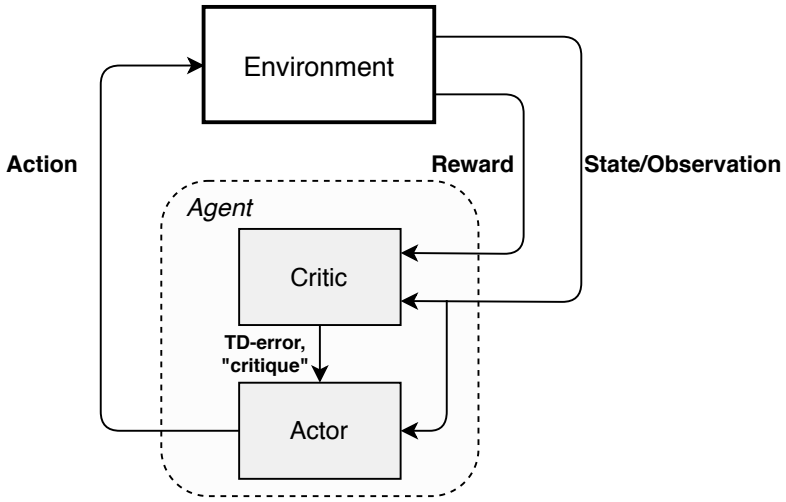


Figure 2.2.3: Actor-Critic framework for RL

The most desirable approach would be to combine the advantages of both value and policy-based methods, or at least mitigate some of the drawbacks of one by leveraging the other. The actor-critic method does exactly this and can be seen as a kind of hybrid approach. As seen in Figure 2.2.3 it uses both a parameterized policy, in the actor, and a value function in the critic. The actor calculates which action to take in a given state while the critic evaluates the action taken and gives a *critique*, in form of an error based on the value function, to the actor in order to improve the policy further. This brings the benefits from value-based methods, like better sample efficiency, together with the advantages of policy-based methods, like the ability to handle large and continuous state and action spaces.

2.2.5 DRL Algorithms

Our problem consists of a continuous action-space, thus we focus on the algorithms that support this. Discretizing the action space could be an alternative for a single-actuator system but the configuration of this problem requires multiple control inputs, i.e. a multi-discrete action space. This is not supported for the relevant algorithms in the current algorithm framework, Stable Baselines.

The following gives a quick introduction to the principles of each algorithm used in this project. The focus is put on the high-level understanding of how the algorithms compare to each other, and we do not derive the technical details behind each of them. For this, the reader is advised to have a look at the respective papers cited. Several of the papers compare the results to other algorithms

in some of the most common OpenAI gym (Brockman et al., 2016) continuous control benchmark environments, like Hopper-v1, Walker2d-v1, HalfCheetah-v1, Ant-v1, and Humanoid-v1.

- **DDPG (Deep Deterministic Policy Gradient):** DDPG builds on the Deep Q-Learning algorithm and extends it to the continuous action domain. It is an off-policy actor-critic algorithm based on the deterministic policy gradient. In terms of performance, the original paper (Lillicrap et al., 2015) states that fewer steps of experience was used by DDPG than by DQN-learning to find solutions in the Atari domain. It is also presented that it robustly solves more than 20 simulated physics tasks, including continuous control problems, but still requires a large number of training episodes to find solutions.
- **TD3 (Twin Delayed DDPG):** TD3 is an off-policy algorithm that builds on DDPG and introduces some critical changes to address a common problem with DDPG; The learned Q-function often overestimates the Q-values, which leads to the policy exploiting these errors and eventually breaking. The changes made are (1) learning two Q-functions and using the smaller Q-value in the Bellman error loss function, (2) updating the policy less frequently than the Q-function, (3) adding noise to the action making it harder for the policy to exploit the errors in the Q-function. The original paper (Fujimoto et al., 2018) presents results that show TD3 outperforming several other algorithms such as DDPG, PPO and ACKTR in a selection of OpenAI gym continuous control tasks.
- **PPO (Proximal Policy Optimization):** Is an on-policy method that uses an actor-critic architecture. PPO is based on the principle of a trust region, i.e. improving the policy as much as possible without going too far from where we are and breaking the policy. PPO implements this in a simple and less computationally demanding way compared to other trust-region methods. Instead of having to solve a constrained optimization problem like in TRPO (Schulman, Levine, Moritz, Jordan and Abbeel, 2017), PPO uses a clipping function that is both easier to implement and computationally less expensive. The original paper (Schulman, Wolski, Dhariwal, Radford and Klimov, 2017) presents results where PPO outperforms both A2C and A2C + Trust Region in several continuous control tasks, most of which are the same tasks as shown in the TD3 paper (Fujimoto et al., 2018).

  The PPO algorithm can also be used with a recurrent Long Short-Term Memory (LSTM) (Hochreiter and Schmidhuber, 1997) policy that introduces the concept of memory for the agent. This is included in the project as it was suggested by (Meyer, 2020) and was believed to improve performance with moving obstacles.

- **A2C (Advantage Actor Critic):** Advantage Actor Critic has two main variants: Asynchronous Advantage Actor Critic (A3C) described in (Mnih et al., 2016) and Advantage Actor Critic (A2C). In short, A2C is a synchronous variant of the A3C algorithm. The term asynchronous refers to the property of parallel training, where multiple agents in parallel environments independently update a global value function. The term advantage refers to the advantage function like the one discussed in subsection 2.2.3. A2C is an on-policy algorithm, like PPO, and has been found to give equal performance to A3C (Wu, Mansimov, Liao, Radford and Schulman, 2017). In (Mnih et al., 2016), A3C shows good results compared to DQN in Atari environments but no comparison for continuous control applications is provided.
- **ACKTR (Actor Critic using Kronecker-Factored Trust Region):** ACKTR is an on-policy algorithm that combines three popular techniques: actor-critic methods, trust region optimization, and distributed Kronecker factorization (Ba et al., 2017). According to (Wu, Mansimov, Liao, Grosse and Ba, 2017), ACKTR improves sample efficiency and scalability over algorithms like A2C and TRPO, and performs similarly or better than the these algorithms in several continuous control environments.
- **SAC (Soft Actor-Critic):** SAC is an off-policy, actor-critic RL algorithm. It is based on the maximum entropy reinforcement learning framework where the actor aims to complete the task at hand while acting as randomly as possible. Formally this translates to maximizing expected reward while also maximizing entropy. SAC has proven to outperform both DDPG, TD3 and PPO in some of the continuous control benchmark environments. (Haarnoja et al., 2018)

Chapter 3

Method and set-up

In this chapter, we explain the specifics of how the work was conducted. This includes presenting the setup for training and testing, as well as the method for comparing the agents' performance.

3.1 Set-up

This project builds upon the work of (Meyer, 2020) and uses the software framework provided there. The vessel dynamics are based on the CyberShip II (Skjetne et al., 2004), a 1:70 scale replica of a supply ship which has a length of 1.255 m and mass of 23.8 kg. For training, a randomly generated artificial environment, later referred to as MovingObstacles-v0, is used. This scenario consists of a random path of length 8000 m, 17 moving obstacles with random size, heading and speed representing other ships, and 11 static obstacles of varying size. Two examples from this scenario are shown in Figure 3.1.1.

The two real-world scenarios used for generalization testing are Trondheim-v0 and Agdenes-v0, displayed in Figure 3.1.2a and 3.1.2b respectively. *Traffic (i.e. other vessels) is sampled as a random subset of the total recorded AIS data in the area. This allows for quantitative statistical testing through repeated trials* (Meyer, 2020).

The algorithms are all trained and tested on the same hardware, a Dell OptiPlex 7060 computer with an Intel Core i7-8700 CPU, 32 GB RAM, and Intel UHD Graphics 630.

Simulation parameters

The hyperparameters for each algorithm are listed in Appendix A. All parameters and values not listed in these tables, are set as default from Stable Baselines. A separate training run with hyperparameters inspired by RL Baselines Zoo (RAFFIN, 2020) was also conducted to explore if this would yield different results. RL Baselines Zoo is a collection of pre-trained RL agents together with tuned hyperparameters for several control problems.

Remark: *The default distance unit used in this package is 10 m (decameters). Return values, such as those returned by length and position attributes, must be*

multiplied by 10 to obtain the values in meters. This choice was made out of convenience, given that the terrain data has a 10x10 metric resolution (Meyer, 2020). The distance data is converted and presented in meters in this report.

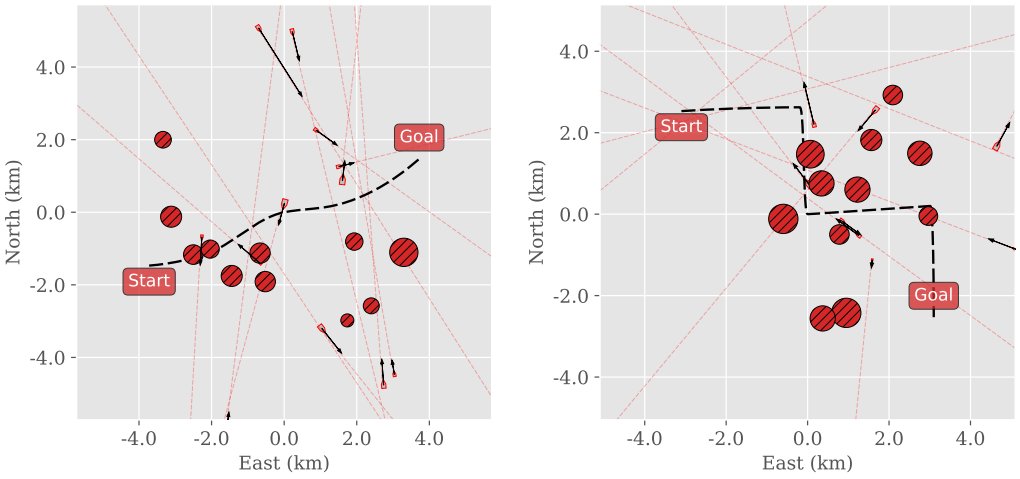
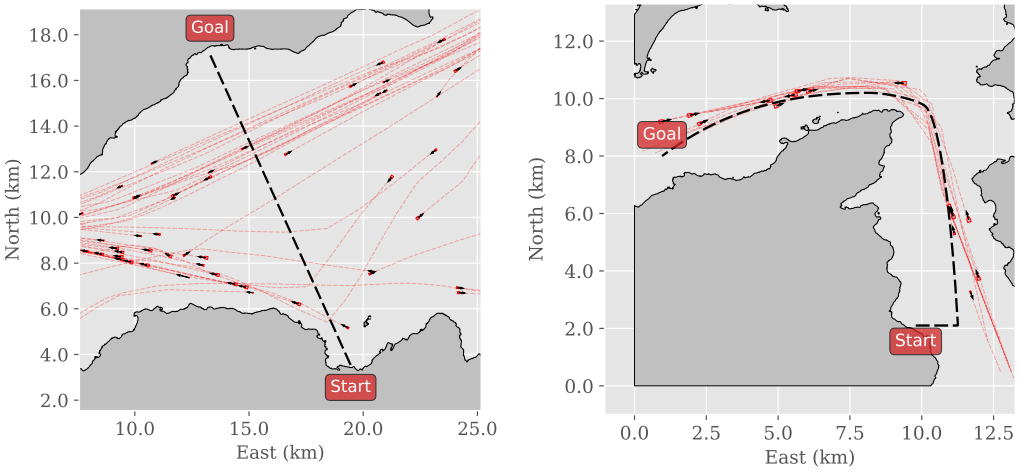


Figure 3.1.1: Maps of two different MovingObstacles-v0 scenarios with traffic and obstacles. The path, traffic and obstacles are generated randomly for each episode. The black dashed line is the desired path and the red lines are trajectories of obstacle vessels.



(a) Trondheim-v0 scenario with traffic.

(b) Agdenes-v0 scenario with traffic.

Figure 3.1.2: Map of the two real-world scenarios. Traffic is sampled randomly from AIS data for each episode. The black dashed line is the desired path and the red lines are trajectories of obstacle vessels.

3.2 Method

3.2.1 Installation and setup of simulator

The simulator provided by (Meyer, 2020) has served as a great tool and framework for testing the algorithms. However, getting this setup up and running was not as trivial as first assumed. It was clear that the work on this software had been iterative and new packages were installed as needed. This lead to multiple dependencies of both packages and their versions, and it was a significant amount of work to set it up correctly on a new system.

A list of installed packages was provided by Eivind Meyer, however this included *all* Anaconda and Python packages on his system, not just the ones needed for this project. The initial work then consisted of debugging to find out which packages were needed and which versions were compatible with each other. Cases where one package required version $> x.y.z$ of package a while another package required version $< x.y.z$ of package a were also encountered. All packages were downgraded to the appropriate version in order to make the software run. In addition, it was discovered that several resource files needed for execution, like the data for real-world scenarios, were missing from the GitHub repository, and that other programs like *Microsoft Visual C++ 14.0* and *Microsoft MPI* were required. After retrieveing these files and a significant amount of time spent on debugging the installation and fixing bugs in the code itself, everything was set up correctly.

To make it easier for future work with this code a setup file and an installation guide (for use with windows and anaconda) was developed. This simplifies the installation on new computers and handles the python dependencies, hopefully preventing others from having to repeat the work described above. The guide, along with all the necessary code, is provided in a GitHub repository (Ødegård Teigen, 2020). Note that GitHub does not support files larger than 100MB and Git LFS has to be used for tracking large files such as the terrain data.

3.2.2 Execution of experiment

First, two agents for each algorithm are trained and the resulting metrics are averaged between the two. The *resulting metrics* refer to the metrics averaged over the last 100 episodes of training for each agent. We acknowledge that the average between just two agents will not necessarily be the true mean but it is used to reduce the probability of comparing outliers. Each agent is trained for 1.5M timesteps. This number is chosen as a compromise between reaching a

satisfactory performance level for comparing the algorithms while not making the training time unreasonably long. As the training time varies significantly between algorithms the number of training timesteps could also be varied but emphasis is put on keeping this parameter equal for all algorithms to strengthen the comparability.

An agent for each algorithm is also tested in the two real-world scenarios to examine the generalization performance. In this case, the test is run for 100 episodes and the average for each metric is taken. The number of episodes is chosen as a compromise between computational time and statistical significance.

3.2.3 Measuring performance

To compare the algorithms a comparable performance metric is needed. We design this as a combination of important metrics weighted through custom functions. So which metrics are considered important?

Average reward from the reward function is important because it is used in optimization during training and should indicate overall performance. The progress metric measures how far the vessel was able to get along the path from start to finish, given as a percentage. The episode is terminated either by reaching the goal (100% progress), a collision, or a time-out in form of a maximum number of timesteps elapsed without reaching the goal (This time-out is implemented in training mode only). Progress is considered one of the most important metrics because it says something about the main objective, getting to the goal position without colliding. Average collisions are included because it is a measurement of the obstacle avoidance performance, while cross-track error is is included because of its connection to the path-following performance.

Even though the average reward should be a good indicator of performance, it can be optimistically biased because it is the actual function being optimized. As stated in (Henderson et al., 2019), local optima can make learning curves indicate successful optimization of the policy, when in reality the policy is not qualitatively representative of the desired behavior. A custom performance function using additional metrics will hopefully give a better picture of the performance. This function should also be constrained, for instance to an interval of $\langle 0, 100 \rangle$, to make it more intuitive. We would like the performance function to have the general form

$$\bar{P} = w_1 \cdot f_1(\bar{R}) + w_2 \cdot f_2(\overline{PR}) + w_3 \cdot f_3(\bar{C}) + w_4 \cdot f_4(\overline{CTR}) \tag{3.2.1}$$

where

- w_i - Weight for the respective metric where $\sum_{i=1}^{4} w_i = 100$
- f_i - Contribution curve for the respective metric, bounded in $\langle 0, 1 \rangle$
- $\bar{R}$ - Avg. reward
- $\overline{PR}$ - Avg. progress, as a percentage (0-100).
- $\bar{C}$ - Avg. collisions, as a percentage (0-100). Defined as how many of the episodes were terminated due to a collision.
- $\overline{CTR}$ - Avg. Cross-Track Error, in meters.

The weights control the importance of the metrics relative to each other, e.g. how much we care about the collision rate compared to the progress, while the contribution curves control the importance of the values themselves relative to each other, e.g how how important is it that the progress value is large. Choosing equal weights for all terms will be sufficient in our case as the metrics can be said to have approximately equal importance. Disregarding the criteria for the weights and setting all weights equal to 1, we design the un-normalized performance to be

$$P_{un} = \frac{1}{1 + 1.1^{-0.02\bar{R}-50}} + \frac{1}{1 + e^{8-0.15\overline{PR}}} + 1.1^{-\bar{C}} + 1.05^{-0.04\overline{CTR}} \tag{3.2.2}$$

Note that each term is bounded for the valid values of the metrics. Normalizing Equation 3.2.2 to the interval $\langle 0, 100 \rangle$ by multiplying by 100 and dividing by the number of metrics used gives the *Performance*

$$\bar{P} = 100 \cdot \frac{P_{un}}{4} \tag{3.2.3}$$

Taking training time into consideration to get a good picture of the algorithms usability in this problem, we define the un-normalized usability to be

$$U_{un} = P_{un} + \frac{1}{1 + e^{0.5T-4}} \tag{3.2.4}$$

where T is the training time for 1.5M timesteps, in hours. Again, we normalize by multiplying by 100 and dividing by the total number of metrics used, giving the *Usability*

$$\bar{U} = 100 \cdot \frac{U_{un}}{5} \tag{3.2.5}$$

$\bar{P}$ and $\bar{U}$ will be in the interval $\langle 0, 100 \rangle$ where all metrics will have equal weight.

Explanation and justification of function terms

Each term in the performance and usability functions is designed to have a reasonable contribution curve with regards to what values of the metric are considered acceptable and not. Figure 3.2.1 shows plots of each terms value contribution to the functions.

Figure 3.2.1a shows the contribution curve for the reward. From experience with training, it is observed that none the algorithms get a positive cumulative reward. This is the reason behind giving a term value of 100 from a reward value of 0 and above. We then use a sigmoid function that decays appropriately with decreasing reward. From around -3500 and below, there is minimal term value given as this is an indication of a poorly performing agent. In the case of the reward being -2000 or higher, we indicate that the agent is performing relatively well and give a large term contribution.

Figure 3.2.1b shows the contribution curve for the average progress made the episodes. We assume that an agent that, on average, completes less than 70% of the objective on average is considered poor. Being able to complete more than 80% is considered acceptable and gives a large term contribution.

Figure 3.2.1c shows the contribution curve from the collision rate. We claim that an agent with a collision rate of over 20% is dangerous and close to useless, and choose a rapidly decaying exponential that has low term contribution for these values. Assuming a collision rate of up to 10% is regarded as acceptable performance, we give large contributions for values lower than this.

Figure 3.2.1d shows the contribution curve for the average cross-track error of the vessel. We allow some deviation from the path for obstacle avoidance and choose a slower decaying exponential with the approximate acceptable range from 0 to around 500 m.

Figure 3.2.1e shows the contribution curve related to the training time of the agents. We allow some minimum training time with a flat curve for small values. This gives a large contribution for agents with 5 hours or less of training time. Assuming a training time of over 10 hours is considered poor, the curve rapidly decreases close to this time. The threshold for a *reasonable* training time is set by comparing the best and worst algorithms w.r.t. this metric. We assume a training time over 3x as long as the shortest training time is considered poor.

Resource demand

The work done in this project demands large amounts of computational resources, with regards to both time and power. With an average training time of around 15 hours (excluding the estimated training time for SAC), the time

demanded to get statistically robust results is significant. The demand for constant monitoring in case of program crashes and need for manual continuation is also an important factor.

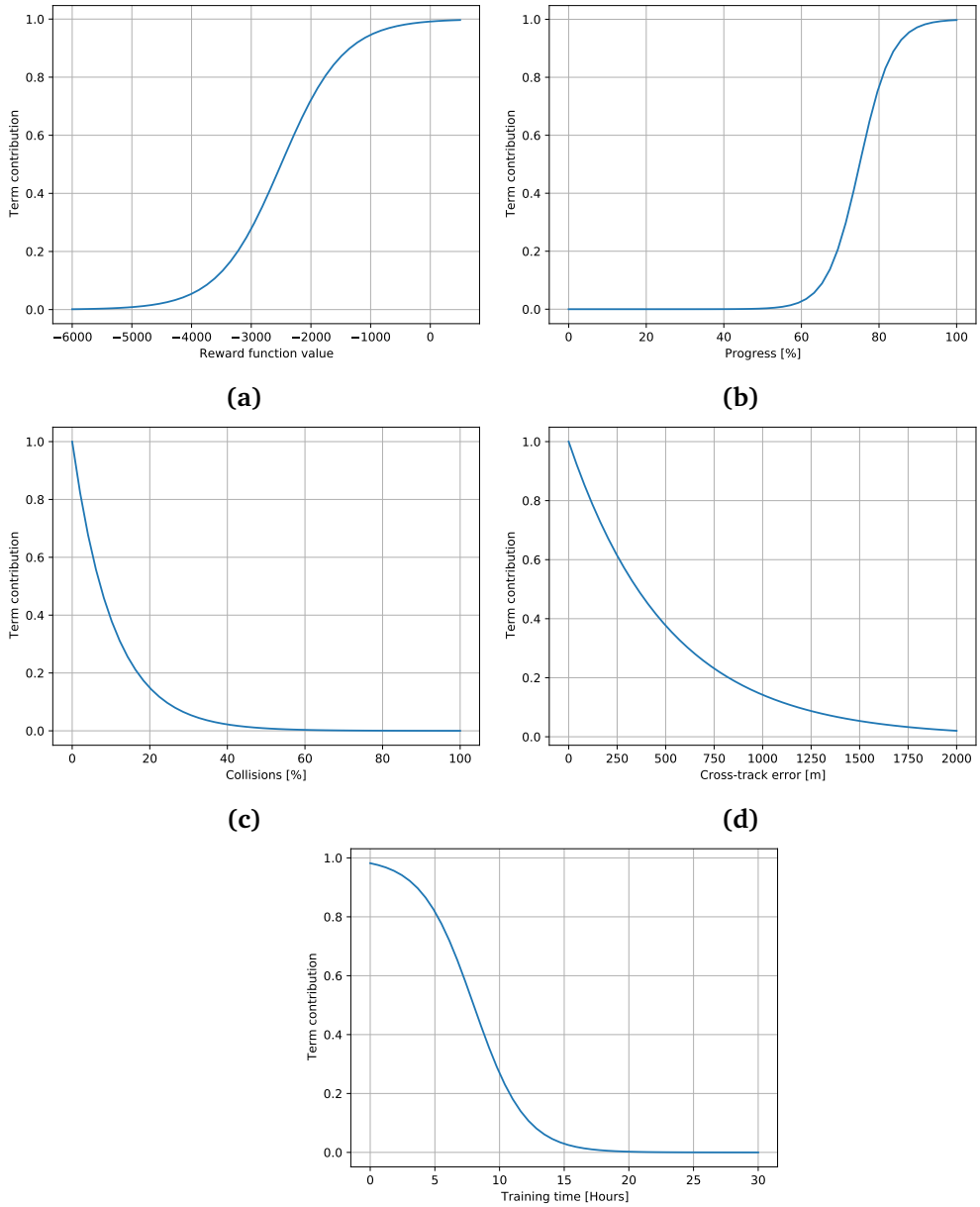


Figure 3.2.1: Plots of each metrics contribution curve in the performance and usability functions.

Chapter 4

Results and Discussions

In this chapter, we present and discuss the results. This includes results from both training and generalization testing of the agents.

4.1 Results

Remark: *Note that the maximum value for Progress is ≈99% as being within a small area around the goal position is considered finishing the episode.*

The results from training of the agents are presented in Table 4.1.1 and Figure 4.1.1. The numeric values for the performance $\bar{P}$ and usability $\bar{U}$ are also included. As mentioned in chapter 3 the metrics presented are averages from the last 100 episodes of each agents training. This is then averaged over two separate agents for each algorithm.

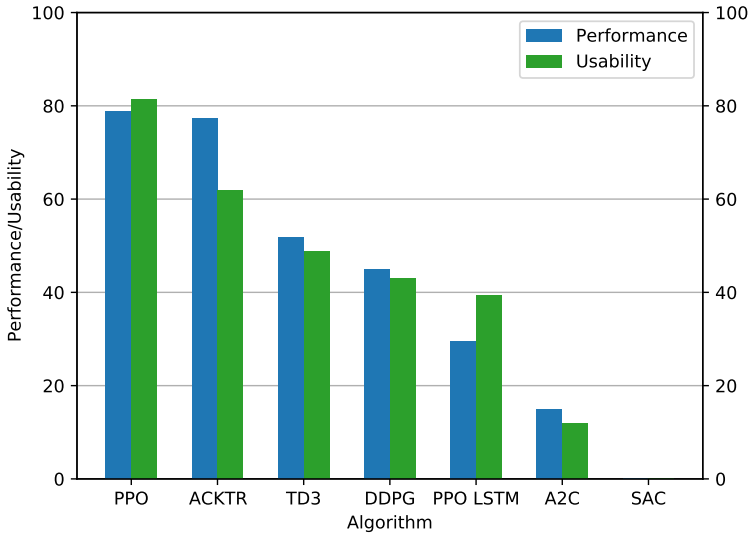


Figure 4.1.1: Plot of performance and usability for each algorithm. SAC is set to zero because training was not completed.

The algorithms were also tested on two real-world scenarios, Agdenes-v0 and Trondheim-v0. A summarizing plot of the generalization performance for each algorithm is presented in Figure 4.1.2. The specific results from the Agdenes tests are displayed in Table 4.1.2 while the results from Trondheim are

in Table 4.1.3.

The values for DDPG and TD3 are blank because the vessel controlled by these agents never reached a terminal state and were not able to complete an episode. This was discovered after testing the TD3 agent for 859500 timesteps and only getting 2.95% progress. A similar observation was made for DDPG. The reason they are still present in the training results in Table 4.1.1 is due to two factors; (1) The training mode includes a terminal state when reaching a maximum number of timesteps which is not present when testing, and (2) the training scenario is more complex to navigate in resulting in a higher chance of collision, another terminal state.

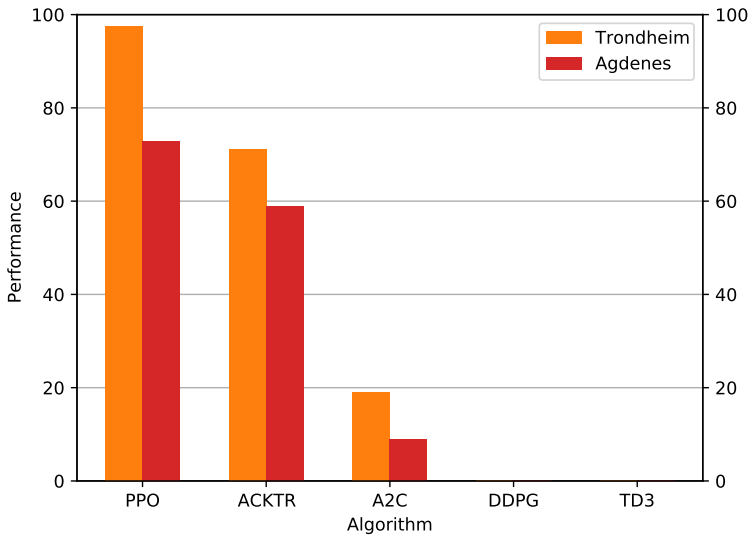


Figure 4.1.2: Plot of the generalizing performance for each algorithm in the Trondheim and Agdenes scenarios. DDPG and TD3 Performance is set to zero because they were not able to finish the objective.

Figure 4.1.3 and 4.1.4 show examples of trajectories taken for each agent in real-world environments. The agents are simulated for up to 10000 timesteps. The PPO and ACKTR agents are able to finish the objective and reach the goal position. The A2C agent gets about halfway, colliding in Agdenes, while the DDPG and TD3 agents barely have any progress and are not able to come close to reaching the objective.

The behavior of DDPG and TD3 agents was also investigated by running scenarios with each agent as "autopilot" and visually inspecting the behavior in real-time. This showed the agents making little to no progress towards the goal, including standing still at the initial position, spinning in a seemingly random manner, and stopping when met with an obstacle, depending on the scenario and agent. Figure 4.1.3 and 4.1.4 show the same undesirable behavior for both

DDPG and TD3.

The behavior of the A2C agent was investigated in the same way as DDPG and TD3, but in the MovingObstacles scenario, and showed that the agent seemed to disregard obstacles and collide often. Path-following performance was adequate but it did not succeed in the obstacle avoidance part of the problem. This is confirmed by the high collision rate and relatively low cross-track error for A2C in Table 4.1.1, and is backed by Figure 4.1.3d where the vessel collides with the shoreline. Note that both the A2C, PPO LSTM and ACKTR algorithms made the program crash during training.

Results after a round of hyperparameter tuning is presented in Table 4.1.4 along with a comment on how this compared to the previous tuning.

Table 4.1.1: Performance metrics for the RL algorithms. In MovingObstaclesNoRules-v0. The metrics are averaged between two separate agents trained with the same algorithm in the same environment for the same amount of timesteps. The exception to this is PPO LSTM and SAC. For PPO LSTM, only one agent was trained. SAC was only partially trained due long training time. Values that stand out as particularly good or bad are marked in green or red respectively.

Algorithm	**Reward**	**Progress**	**Collision**	**CTE**	**Training**	$\bar{P}$	$\bar{U}$
PPO	-775.2	97.3 %	2.0 %	547 m	3.25 h	78.8	81.3
DDPG	-2211.9	4.1 %	5.5 %	315 m	9.17 h	45.0	43.1
TD3	-2021.4	4.9 %	0.5 %	471 m	9.05 h	51.7	48.8
A2C	-3875.7	48.7 %	45.5 %	345 m	34.50 h	14.8	11.9
ACKTR	-977.8	94.8 %	5.0 %	330 m	28.50 h	77.2	61.8
PPO LSTM	-2338.9	10.7 %	8.0 %	1015 m	5.33 h	29.5	39.4
SAC	-	-	-	-	-	-	-

Table 4.1.2: Performance metrics for the RL algorithms. In Agdenes-v0. The agent trained first for each algorithm was used to prevent the introduction of a bias. DDPG and TD3 are blank because they were not able to finish the objective. Values that stand out as particularly good or bad are marked in green or red respectively.

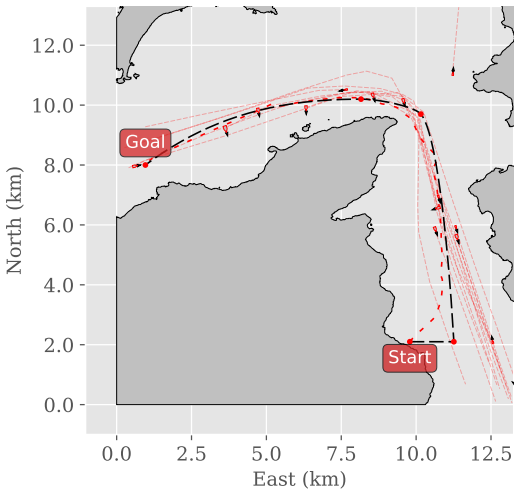
Algorithm	**Reward**	**Progress**	**Collisions**	**CTE**	$\bar{P}$
PPO	-1626.84	93.59 %	10.0 %	184.9 m	72.8
DDPG	-	-	-	-	
TD3	-	-	-	-	
A2C	-17190.84	51.03 %	100 %	531.5 m	8.9
ACKTR	-2557.04	88.9 %	16 %	182.4 m	58.9

Table 4.1.3: Performance metrics for the RL algorithms. In Trondheim-v0. The agent trained first for each algorithm was used to prevent the introduction of a bias. DDPG and TD3 are blank because they were not able to finish the objective. Values that stand out as particularly good or bad are marked in green or red respectively.

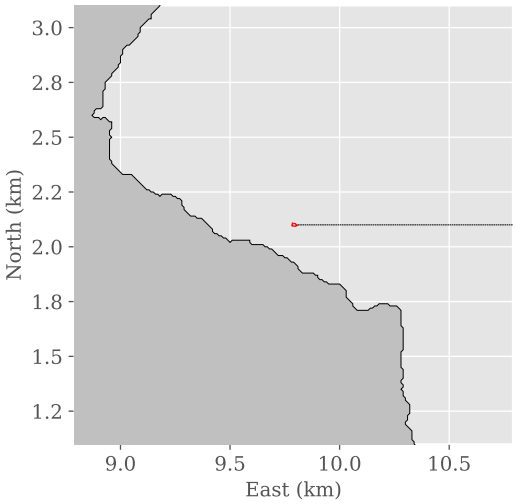
Algorithm	**Reward**	**Progress**	**Collisions**	**CTE**	$\bar{P}$
PPO	1649.75	99.01 %	0.0 %	51.7 m	97.5
DDPG	-	-	-	-	
TD3	-	-	-	-	
A2C	-12851.36	61.22 %	79.0 %	164.7 m	19.0
ACKTR	-2250.20	97.20 %	5.0 %	252.4 m	71.1

Table 4.1.4: Results for the separate training with tuned hyperparameters. In MovingObstaclesNoRules-v0.

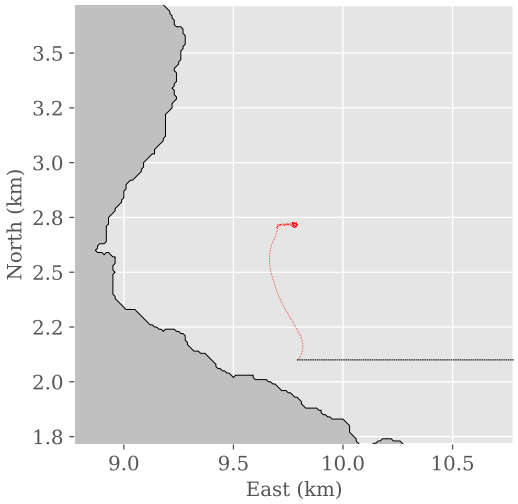
Algorithm	**Note**	$\bar{P}$	$\bar{U}$
PPO	Parameters unchanged, no tuned training performed	78.8	81.3
DDPG	Very similar results as with previous parameters	41.2	39.4
TD3	Worse results than with previous parameters	30.7	34.6
A2C	Training not finished, estimated training time over 100 h	-	-
ACKTR	Similar but slightly worse results, still crashed during training	65.7	52.5
SAC	Training not finished, estimated training time over 350 h	-	-



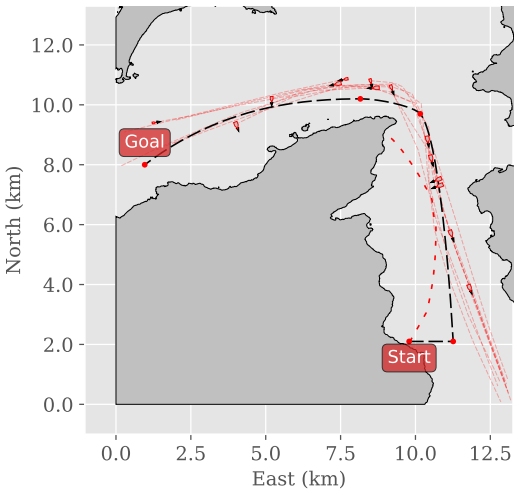
(a) PPO



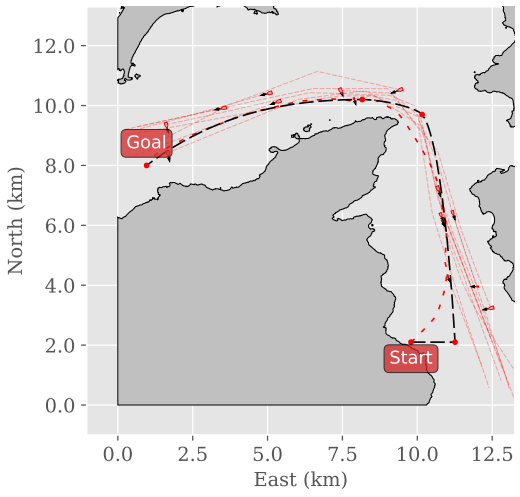
(b) DDPG zoomed in



(c) TD3 zoomed in

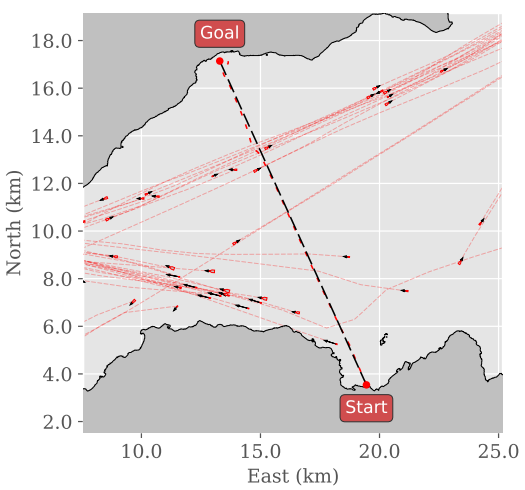


(d) A2C

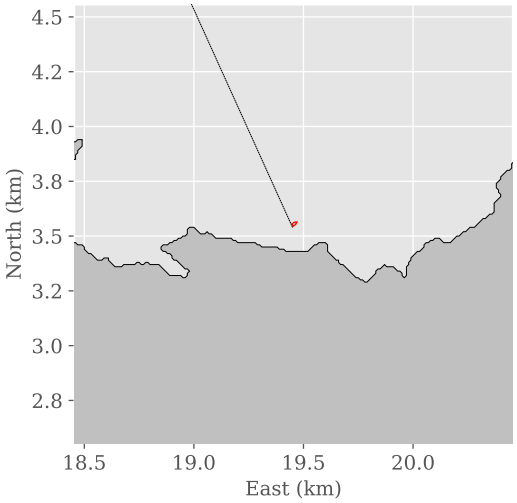


(e) ACKTR

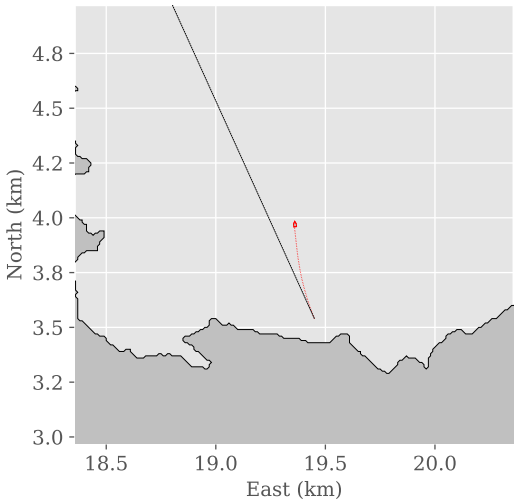
Figure 4.1.3: Plots of each agents trajectory in the Agdenes scenario. The thickest red dashed line is the vessels trajectory and the black line is the desired path.



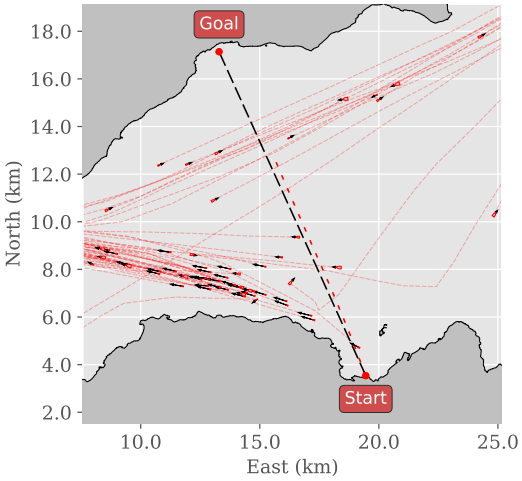
(a) PPO



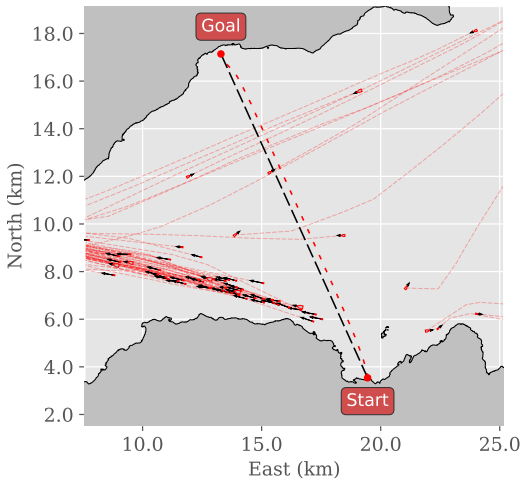
(b) DDPG zoomed in



(c) TD3 zoomed in



(d) A2C



(e) ACKTR

Figure 4.1.4: Plots of each agents trajectory in the Trondheim scenario. The thickest red dashed line is the vessels trajectory and the black line is the desired path.

4.2 Discussion

From Figure 4.1.1 and 4.1.2 it is clear that PPO is the overall best algorithm for this problem, and the claim from (Meyer, 2020) is confirmed. The PPO agent has the shortest training time, and the best performance and usability in all cases. This is also strengthened by the results in (Schulman, Wolski, Dhariwal, Radford and Klimov, 2017) where PPO outperformed both A2C and A2C with trust region (similar to ACKTR) in several continuous control environments. The performance for ACKTR is also good, but we need to take in to account that the training time is over 8x as long and that the algorithm crashes during training. This brings the usability to a significantly lower level than the performance.

The rest of the algorithms show overall poor performance and are not considered usable for this application. This is justified by the DDPG and TD3 agents not being able to complete the real-world scenarios, the poor scores for A2C and PPO LSTM, along with the unreasonable training time for SAC.

Figure 4.1.3 and 4.1.4 show that the qualitative behavior of the agents reflect the quantitative results from the performance metric.

Training

The results discussed in this section are displayed in Table 4.1.1 and Figure 4.1.1

Training varied significantly between the algorithms. Some had reasonable training times and completed without problems while others had longer training times and even crashed during execution. This required the training to be manually continued from the last saved agent. The reason for this crashing is not known with certainty but our hypothesis is that the computer ran out of memory as the error that occurs is of type "MemoryError". This is backed by the fact that it happens after several hours of training which reduces the chance of it being implementation error.

The training with PPO was fast and gave excellent results. It had the overall best performance and usability and is the preferable algorithm for this application. If training time is taken into consideration, its lead increases over the other algorithms.

ACKTR crashed once during training and had a total training time of 30 hours. Although the performance is good and almost on par with PPO, the crashing is considered unacceptable and together with the long training time, the conclusion is that ACKTR is not the preferred algorithms for this application.

Training with A2C also crashed, up to three times, and had a total training time of 37 hours. This is considered unacceptable and together with the overall poor performance it makes the algorithm unusable in this application.

DDPG displayed reasonable overall performance in training but we notice a particularly low value for the progress. This is an indicator that the vessel with this agent may be standing still or at least not completing the episodes. This was confirmed by the investigation explained in section 4.1. Training time is relatively long but still within what we regard as acceptable. While the overall performance shows that DDPG may be applicable to this problem, the low progress rate is alarming and it comes down to the generalization performance.

TD3 had very similar results to DDPG with the main deviation being a low collision rate. Regardless, TD3 suffers from the same problem of a low average progress and the same conclusion is made.

The PPO LSTM approach gave very similar results to DDPG and TD3 with a low average progress. In addition, the algorithm crashed during training. Due to these factors, we did not find it reasonable to use more resources in training multiple agents or testing in real-world environments with this algorithm. It also lead to the conclusion that it is not a good approach for this application.

SAC was initially believed to give good results as it was shown to outperform PPO, DDPG, and TD3 in several continuous control benchmarks (Haarnoja et al., 2018), but this was contradicted after training started. It was discovered that the training time would be unrealistically long. After 20 hours of training, the algorithm had completed approximately 78 000 of 1.5M timesteps. This lead to an estimated training time of 380 hours, or almost 16 days, which we find to be unreasonable. Because of this, it was decided to disregard further training and testing with this algorithm. It is also the reason for the lack of results for SAC in Table 4.1.1.

In addition to the final results, it is interesting to look at the training history to see the improvement over time with regards to the number of timesteps. This is shown in Appendix B and C. Figure B.0.1 clearly shows that PPO and ACKTR are the only agents with significant reward increase over time. This is confirmed by the progress plots in Figure C.0.1 where PPO and ACKTR are the only ones with increasing progress. All this indicates that the other algorithms do not converge to a satisfactory optimum.

Generalization

As Figure 4.1.2 shows, the generalization performance of the agents is varying. Comparing the results in Figure 4.1.1 and Figure 4.1.2, the results for training and generalization testing are similar for PPO, ACKTR and A2C. This

indicates that these agents are generalizing well and not overfitting to, or exploiting weaknesses in, the training scenario. Even though good generalization is indicated for the A2C agent, we argue that the generalization performance is inconclusive. This is because it already had poor performance in the training scenario, and a similar poor performance in real-world scenarios can then both mean that it performs *as good* as in training and generalizes well, or that it simply does not generalize well at all. As shown in Figure 3.1.2b and Figure 3.1.2a the Agdenes scenario is more difficult to navigate in than Trondheim which explains the consistently lower performance scores for Agdenes. We would like to emphasize that a collision rate of over 10% for both ACKTR and PPO in Agdenes is still significant and shows that these agents might not be good enough for actual real-world implementation yet.

The DDPG and TD3 agents were not able to complete any episodes in real-world scenarios and one could assume that they do not generalize well. Although bringing the training results into consideration, we see that the progress they make is below 5% which is consistent with the behavior in the real-world scenarios. It can then be argued that the same conclusion as for A2C can be made, and that the poor performance in real-world scenarios is just a consequence of an overall poor agent. Examples of this behavior are shown in Figure 4.1.3 and 4.1.4. As mentioned in previously, generalization testing was not performed for SAC and PPO LSTM.

Hyperparameter tuning

The performance of algorithms is known to vary with different choices of hyperparameters. A separate round of tuning, with inspiration from the RL Baselines Zoo (RAFFIN, 2020), was performed to see if the results were drastically different than the ones already presented. As shown in Table 4.1.4 it turned out that the performance and usability of the tuned algorithms was similar, or worse in some cases, than previously. In any case, it did not change the overall results of our comparison. This strengthens our findings by indicating that the poor performance of some agents was not a consequence of bad tuning.

Why do some algorithms fail in this application?

Looking at the overall results, taking the behavior itself into consideration, it seems like the on-policy algorithms perform better than the off-policy ones. This might be related to a statement made in (Sutton and Barto, 2018): *In on-policy methods, the agent commits to always exploring and tries to find the best policy that still explores [...they estimate the value of a policy while using it for control...]. In off-policy methods, the agent also explores, but learns a deterministic optimal policy that may be unrelated to the policy followed.* We see indications of

this in the progress metric and in Figure 4.1.3 and 4.1.4 where the off-policy algorithms DDPG and TD3 seem to have learned a target policy that is far from optimal. Because it follows one policy μ and optimizes another π it seems like the optimized *target* policy π does not have as optimal behavior as the updates from following μ might indicate.

This is of course just one hypothesis and it does not explain the poor performance of A2C. There may be a number of reasons why the specific algorithms perform the way they do. One of the disadvantages with AI and machine learning methods is that the inner workings are complex and often lack explainability. This is especially the case in RL and in methods where Deep Neural Networks are used.

Performance and usability functions

The performance and usability functions are shaped to compare the specific algorithms used in this specific problem, and are not made to be general measures for all algorithms in all applications. Which values are considered good or acceptable for each metric is highly subjective and while we have tried to be as objective as possible, some data from the trainings had to be used in shaping the contribution curves. This is especially true for the reward function value and training time, where reference values are needed in order to determine how good or bad a value is. Metrics like the progress and collision rate are more intuitive and their contribution functions can be set independent from the results in this implementation. The cross-track error is highly dependent on the environment and is hard to quantify as small or large for a general function used across multiple scenarios.

Using values from experience with training as reference values comes with some drawbacks that we are aware of. This way of designing a function will naturally give a bias towards differentiating the algorithms rather than creating an objective metric. This is not seen as a major issue here, as the intention behind creating this function is in fact differentiating the algorithms used. A problem could occur if new algorithms were to be introduced and these perform either better than the best or worse than the worst. The way the function is now there would be little to no difference between the former best/worst and this new algorithm.

A question that was considered when designing these functions was whether to reward wanted behavior, punish unwanted behavior, or a combination of both. We decided to reward wanted behavior to avoid negative values and make it possible to limit the metric to an interval of $\langle 0, 100 \rangle$. This is an intuitive way of looking at performance as it resembles a percentage measure.

While we acknowledge their weaknesses, the performance and usability measures have served as good indicators of how the algorithms compare in this application. The measures clearly separate the algorithms and have served their purpose well.

Chapter 5

Conclusion and future work

In this chapter, we conclude the report, reflect on the work done, and outline some alternatives for future work within this application.

5.1 Conclusions

The major conclusions of the thesis are:

- We have proved that the PPO RL algorithm is the best out of the selected algorithms in the application of path-following and obstacle-avoidance for autonomous vessels. The ACKTR agent also performed well in both training and generalization scenarios but because of the algorithm crashing during training and a significantly longer training time, we conclude with PPO being the best choice.
- This is justified using the custom performance and usability measures and it was shown that PPO outperformed the other algorithms by a significant margin with only ACKTR coming close. DDPG, TD3, A2C, PPO LSTM, and SAC proved to perform poorly and are all considered inapplicable for this problem.
- For the algorithms performing well, namely PPO and ACKTR, the agents generalize well to real-world scenarios and show promising results for realization on real vessels. We have also shown that the algorithms' performance stays consistent, relative to each other, for both training and real-world scenarios. The generalization performance of the DDPG, TD3, A2C, PPO LSTM and SAC agents stay inconclusive as these agents were not able to perform well, even in the training scenario.

In doing so we answer all the research questions mentioned in subsection 1.2.2 thereby realizing all primary and secondary objectives.

Reflecting on the work done in this report, there are of course improvements that could have been made. First of all, a selection of algorithms was chosen based on a combination of availability from Stable Baselines and research in algorithms that have been applied to similar problems previously. An option for even more rigorous testing is to implement custom algorithms, either based on Stable Baselines or through the introduction of new frameworks.

Another aspect is the tuning of hyperparameters. We present two different tunings of the parameters but a more systematic and rigorous tuning is of course preferred. We discuss this further in section 5.2.

In the initial work a lot of time was spent on setting up and understanding the current implementation. This is of course a prerequisite for being able to improve and update the code but thinking back, it would most likely have been beneficial to spend time updating the algorithm implementation framework to Stable Baselines3 (Raffin et al., 2019) or Ray RLlib(Liang et al., 2018). This would have reduced the work related to package dependencies as there is more support for later versions in these frameworks. More on this in section 5.2.

5.2 Future Work

5.2.1 Update frameworks

Due to the rapid development in the field of RL, implementations quickly become outdated and need continuous updates. The current implementation has many deprecation warnings due to the fact that it uses Stable Baselines, which does not support Tensorflow 2. A suggestion is to update the code to use Stable Baselines3 which is the successor to Stable Baselines that uses PyTorch. The modular design of the software should make this possible. Stable Baselines3 is currently in beta.

In terms of computational resource demand, it could be advantageous to update the software to leverage cloud computing. An initial exploration of this lead to the discovery of Ray RLlib which is compatible with Microsoft Azure. RLlib is also an alternative to Stable Baselines3 and provides much of the same functionality, in addition to a lot more.

We believe that an update to Stable Baselines3 would be the easiest but the added functionality of RLlib could aid further development and is viewed as the preferred and more future-proof alternative.

5.2.2 Multi-Agent Deep Reinforcement Learning (MADRL)

An illustration of the difference between a single-agent and multi-agent setup is shown in Figure 5.2.1. The current implementation is made from a single-agent perspective. Extending this to multi-agent environments is not trivial and comes with both mathematical and software related challenges. (Kapoor, 2018) lists and discusses some of these challenges. It is important to note that *multi-agent* refers to more than just inserting multiple agents in an

environment. The multi-agent domain introduces the concept of collaboration, in the form of both cooperative and competitive settings.

Another aspect is explained in Neto (2005):

> *Learning in multi-agent systems, however, poses the problem of non-stationarity due to interactions with other agents. In fact, the RL methods for the single agent domain assume stationarity of the environment and cannot be applied directly.*

In other words, the stationarity assumption of MDP is broken in the multi-agent case and we have to generalize it to a Markov Stochastic Game. A challenge is that a MDP has at least one optimal policy and of the given optimal policies at least one is stationary and deterministic. However, for many Markov games, there is no deterministic optimal policy that is undominated because it critically depends on the the behavior of the opponent. (Kapoor, 2018)

Other challenges include difficulty specifying a global learning goal, the curse of dimensionality, and instability of the learning dynamics. As displayed in Figure 5.2.1, multiple agents can share a policy. From a software perspective this introduction of *shared resources* also proposes a challenge.

Stable Baselines does not currently have support for multi-agent RL algorithms and software has to be revised implementing a MADRL-supported platform. The OpenAI Gym framework is capable of multi-agent implementations and has been used for this in research (OpenAI, 2019). A suggestion is to investigate the possibility of using Ray RLlib which has support for scalable multi-agent algorithms and environments. From a software architecture point of view, the code is made in a modular way to facilitate extensions and prevent the need to change all parts of the code.

A selection of relevant resources for multi-agent reinforcement learning is: (Kapoor, 2018), (Buşoniu et al., 2010), and (Liang and Liaw, 2018).

5.2.3 Inverse RL

An advantage of RL is its possibility to learn from demonstration, not only the policy but the reward function itself. We propose an extension of this project where the concept of inverse RL is used. This is an approach where the reward function is learned implicitly from demonstration. Reward function design is one of the biggest challenges in RL and is crucial for the performance of the agent. Being able to learn the reward function from demonstration would eliminate this step and significantly simplify the RL design process. (Hwang et al., 2019) shows successful steps towards realizing this which were validated in

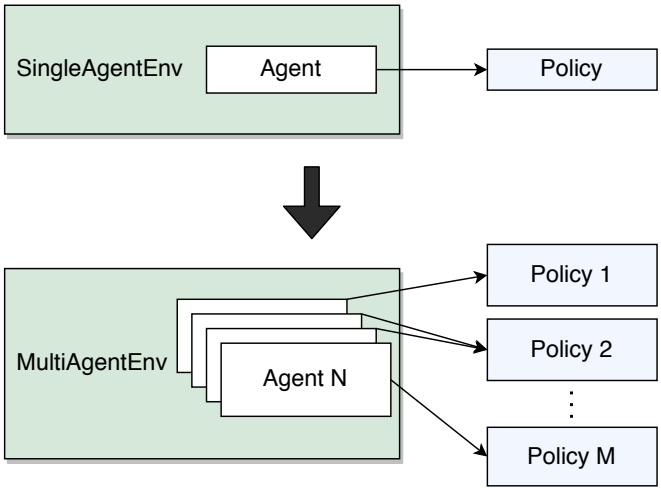


Figure 5.2.1: Illustration of the difference between a single-agent and multi-agent setup.

several classical benchmark domains, but there is still need to experiment with more complex environments.

Another similar approach is to learn the policy by having an *expert* (human) control the vessel for some episodes first to get expert trajectories and pre-train the agent using this. This might improve the convergence rate significantly as the initial random exploration is reduced. An implementation of this using PPO and GAIL (Hill et al., 2018) was attempted but did not yield great results and the implementation was concluded to be unsuccessful. Further research into this is suggested.

5.2.4 Hyperparameter tuning

Tuning the hyperparameters for each algorithm is an important factor in the resulting performance. Due to the high demand for computational time and power for testing each configuration, this is a demanding task and could be explored systematically in further research. In this project, we have experimented with two different configurations, one of which is based on (RAFFIN, 2020). A better approach would be to implement some sort of grid search or Bayesian optimization for hyperparameter tuning. RLlib and Ray Tune (Liaw et al., 2018) include options for this.

Bibliography

AUTOSHIP Project (2020), https://www.autoship-project.eu/. accessed: 2020-10-22.

Ba, J., Grosse, R. and Martens, J. (2017), 'Distributed second-order optimization using kronecker-factored approximations'.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. and Zaremba, W. (2016), 'Openai gym'.

Buşoniu, L., Babuška, R. and De Schutter, B. (2010), Multi-agent reinforcement learning: An overview, *in* 'Innovations in multi-agent systems and applications-1', Springer, pp. 183–221.

Chilamkurthy, K. (2020), 'Off-policy vs on-policy vs offline reinforcement learning demystified!', https://towardsdatascience.com/off-policy-vs-on-policy-vs-offline-reinforcement-learning-demystified-f7f87e275b accessed: 2020-11-09.

de la Campa Portela, R. (2005), 'Maritime casualties analysis as atool to improve research abouthuman factors on maritimeenvironment.'.

Fossen, T. (2021), *Handbook of Marine Craft Hydrodynamics and Motion Control*, John Wiley & Sons.

Fujimoto, S., van Hoof, H. and Meger, D. (2018), 'Addressing function approximation error in actor-critic methods'.

Haarnoja, T., Zhou, A., Abbeel, P. and Levine, S. (2018), 'Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor'.

He, J., Chen, J., He, X., Gao, J., Li, L., Deng, L. and Ostendorf, M. (2016), 'Deep reinforcement learning with a natural language action space'.

Henderson, P., Islam, R., Bachman, P., Pineau, J., Precup, D. and Meger, D. (2019), 'Deep reinforcement learning that matters'.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S. and Wu, Y. (2018), 'Stable baselines', https://github.com/hill-a/stable-baselines.

Hochreiter, S. and Schmidhuber, J. (1997), 'Long short-term memory'.

Hoem, Å., Fjørtoft, K. and Rødseth, Ø. (2019), 'Addressing the accidental risks of maritime transportation: Could autonomous shipping technology improve the statistics?'.

Hwang, R., Lee, H. and Hwang, H. J. (2019), 'Option compatible reward inverse reinforcement learning'.

International Chamber of Shipping (2020), https://www.ics-shipping.org/shipping-facts/shipping-and-world-trade. accessed: 2020-10-22.

Kapoor, S. (2018), 'Multi-agent reinforcement learning: A report on challenges and approaches', *arXiv preprint arXiv:1807.09427* .

Li, Y. (2018), 'Deep reinforcement learning: An overview'.

Liang, E. and Liaw, R. (2018), 'Scaling multi-agent reinforcement learning', https://bair.berkeley.edu/blog/2018/12/12/rllib/?fbclid=IwAR0v2zesfYwYmLpxCJ0aeUOSqiSFpgnZY04MGIOQBLt5SQ59r-SakEGckhQ&utm_campaign=Artificial%2BIntelligence%2Band%2BDeep%2BLearning%2BWeekly&utm_medium=web&utm_source=Artificial_Intelligence_and_Deep_Learning_Weekly_84.

Liang, E., Liaw, R., Moritz, P., Nishihara, R., Fox, R., Goldberg, K., Gonzalez, J. E., Jordan, M. I. and Stoica, I. (2018), 'Rllib: Abstractions for distributed reinforcement learning'.

Liaw, R., Liang, E., Nishihara, R., Moritz, P., Gonzalez, J. E. and Stoica, I. (2018), 'Tune: A research platform for distributed model selection and training', *arXiv preprint arXiv:1807.05118* .

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D. (2015), 'Continuous control with deep reinforcement learning'.

Maei, H. R., Szepesvári, C., Bhatnagar, S. and Sutton, R. S. (2010), 'Toward off-policy learning control with function approximation'.

Meyer, E. (2020), 'On course towards model-free guidance, a self-learning approach to dynamic collision avoidance for autonomous surface vehicles'.

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D. and Kavukcuoglu, K. (2016), 'Asynchronous methods for deep reinforcement learning'.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M. (2013), 'Playing atari with deep reinforcement learning'.

Neto, G. (2005), 'From single-agent to multi-agent reinforcement learning: Foundational concepts and methods, learning theory course'.

Niroui, F., Zhang, K., Kashino, Z. and Nejat, G. (2019), 'Deep reinforcement learning robot for search and rescue applications: Exploration in unknown cluttered environments'.

Norwegian Forum for Autonomous Ships (NFAS) (2020), `http://nfas.autonomous-ship.org/`. Accessed: 2020-10-22.

OpenAI (2019), 'Emergent tool use from multi-agent interaction', `https://openai.com/blog/emergent-tool-use/`.

RAFFIN, A. (2020), 'Rl baselines zoo: a collection of pre-trained reinforcement learning agents', `https://github.com/araffin/rl-baselines-zoo`.

Raffin, A., Hill, A., Ernestus, M., Gleave, A., Kanervisto, A. and Dormann, N. (2019), 'Stable baselines3', `https://github.com/DLR-RM/stable-baselines3`.

Schulman, J., Levine, S., Moritz, P., Jordan, M. I. and Abbeel, P. (2017), 'Trust region policy optimization'.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O. (2017), 'Proximal policy optimization algorithms'.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T. and Hassabis, D. (2016), 'Mastering the game of go with deep neural networks and tree search.'.

Skjetne, R., Smogeli, Ø. N. and Fossen, T. I. (2004), 'A nonlinear ship manoeuvering model: Identification and adaptive control with experiments for a model ship'.

Sutton, R. S. and Barto, A. G. (2018), *Reinforcement Learning: An Introduction, second ed.*, The MIT Press.

Tosatto, S. (2020), 'Enhancing sample efficiency in reinforcement learning with nonparametric methods', https://developer.nvidia.com/blog/enhancing-sample-efficiency-in-reinforcement-learning-with-nonparametric-methods/, accessed: 2020-11-09.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C., İlhan Polat, Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P. and Contributors, S. . (2019), 'Scipy 1.0–fundamental algorithms for scientific computing in python'.

Williams, R. J. (1992), 'Simple statistical gradient-following algorithms for connectionist reinforcement learning'.

Wu, Y., Mansimov, E., Liao, S., Grosse, R. and Ba, J. (2017), 'Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation'.

Wu, Y., Mansimov, E., Liao, S., Radford, A. and Schulman, J. (2017), 'Openai baselines: Acktr & a2c'.

Ødegård Teigen, H. (2020), 'Gym-auv github repository', https://github.com/halvorot/gym-auv.

Appendix A

Hyperparameters

Parameter	Value	Alt. value
Number of steps to run for each environment per update	1024	1024
Number of training minibatches per update	32	32
Factor for trade-off of bias vs variance λ	0.98	0.98
Discount factor γ	0.999	0.999
Number of epoch when optimizing the surrogate	4	4
Entropy coefficient for the loss calculation	0.01	0.01
Learning rate	2e-4	2e-4

Table A.0.1: Hyperparameter values for PPO algorithm. **Value** is the value used for main trainings while **Alt. value** is the value used in the tuned training.

Parameter	Value	Alt. value
Number of steps to run for each environment per update	1024	1024
Number of training minibatches per update	1	1
Factor for trade-off of bias vs variance λ	0.98	0.98
Discount factor γ	0.999	0.999
Number of epoch when optimizing the surrogate	4	4
Entropy coefficient for the loss calculation	0.01	0.01
Learning rate	2e-4	2e-4

Table A.0.2: Hyperparameter values for PPO LSTM algorithm. **Value** is the value used for main trainings while **Alt. value** is the value used in the tuned training.

Parameter	Value	Alt. value
Number of steps to run for each environment	-	16
Discount factor γ	-	0.99
Entropy coefficient for the loss calculation	-	0.0
Learning rate	-	0.06
The type of scheduler for the learning rate update	-	Constant

Table A.0.3: Hyperparameter values for ACKTR algorithm. **Value** is the value used for main trainings while **Alt. value** is the value used in the tuned training.

Parameter	Value	Alt. value
Size of the replay buffer	1000000	50000
Normalize observations	True	True
Normalize returns	False	False
Discount factor γ	0.98	0.98
Actor learning rate	0.00156	0.00156
Critic learning rate	0.00156	0.00156
Batch size	256	256
Parameter noise type	AdaptiveParamNoise stddev 0.287	AdaptiveParamNoise stddev 0.1

Table A.0.4: Hyperparameter values for DDPG algorithm. **Value** is the value used for main trainings while **Alt. value** is the value used in the tuned training.

Parameter	Value	Alt. value
Size of the replay buffer	1000000	50000
Update the model every "Value" steps.	1000	-
Gradient updates after each step	1000	-
Steps before learning starts	10000	1000
Batch size	-	256
Action noise type	NormalActionNoise mean 0, stddev 0.1	NormalActionNoise mean 0, stddev 0.1

Table A.0.5: Hyperparameter values for TD3 algorithm. **Value** is the value used for main trainings while **Alt. value** is the value used in the tuned training.

Parameter	Value	Alt. value
Number of steps to run for each environment per update	16	5
Discount factor γ	0.99	0.995
Entropy coefficient for the loss calculation	0.001	0.00001
Learning rate	2e-4	0.00083
The type of scheduler for the learning rate update	Linear	Linear

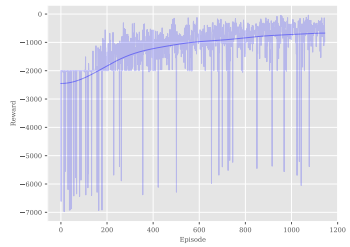
Table A.0.6: Hyperparameter values for A2C algorithm. **Value** is the value used for main trainings while **Alt. value** is the value used in the tuned training.

Parameter	Value	Alt. value
Batch size	-	256
Number of steps before learning starts	-	1000

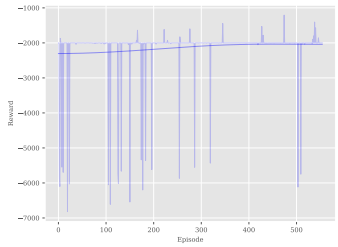
Table A.0.7: Hyperparameter values for SAC algorithm. **Value** is the value used for main trainings while **Alt. value** is the value used in the tuned training.

Appendix B

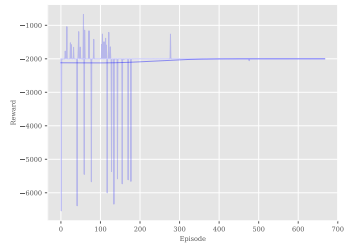
Reward plots



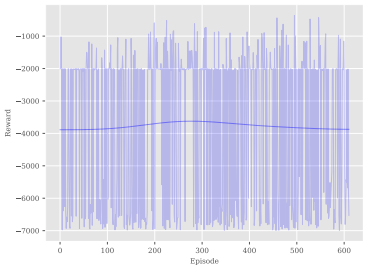
(a) PPO, 1.5M timesteps of training equal to approximately 1150 episodes



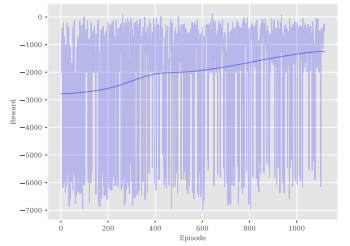
(b) DDPG, 1.5M timesteps of training equal to approximately 550 episodes



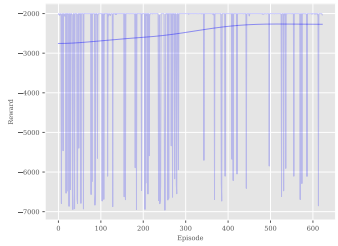
(c) TD3, 1.5M timesteps of training equal to approximately 650 episodes



(d) A2C, last 528 000 timesteps of training equal to approximately 600 episodes



(e) ACKTR, last 831 000 timesteps of training equal to approximately 1100 episodes

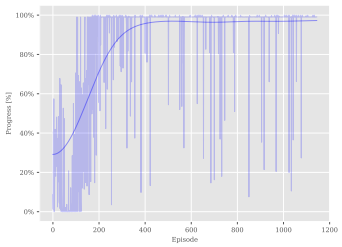


(f) PPO LSTM, first 1.27M timesteps of training equal to approximately 600 episodes

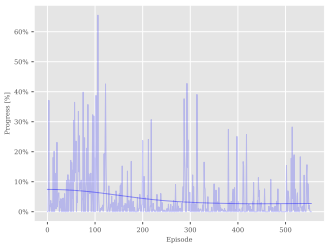
Figure B.0.1: Plots of the reward gathered in each episode during training for each of the algorithms. The moving average of the reward is displayed as a solid line.

Appendix C

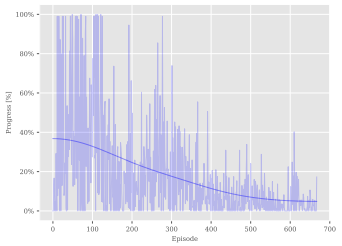
Progress plots



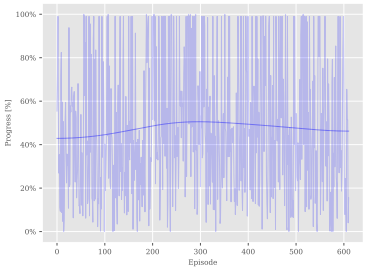
(a) PPO, 1.5M timesteps of training equal to approximately 1150 episodes



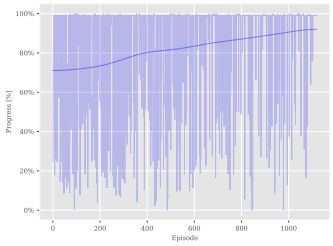
(b) DDPG, 1.5M timesteps of training equal to approximately 550 episodes



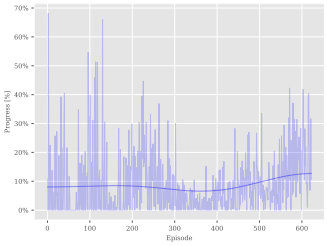
(c) TD3, 1.5M timesteps of training equal to approximately 650 episodes



(d) A2C, last 528 000 timesteps of training equal to approximately 600 episodes



(e) ACKTR, last 831 000 timesteps of training equal to approximately 1100 episodes



(f) PPO LSTM, first 1.27M timesteps of training equal to approximately 600 episodes

Figure C.0.1: Plots of the progress made by the agent in each episode during training for each of the algorithms. The moving average of the progress is displayed as a solid line.