Path-following for a fully-actuated marine vessel with reinforcement learning

Ella-Lovise Hammervold Rørvik

Specialization assignment for the Department of Engineering Cybernetics

Course: TTK4550

Deadline: June 18. 2019

Supervisor: Anastasios Lekkas

Problem description

The main objective of this project is to develop a path-following system for a marine vessel, using deep reinforcement learning (DRL). DRL has shown promising results in continuous control problems, including within the control of marine vessels. The path-following problem consists of different objectives, including both forward speed control and convergence to desired path. These problems will be formulated as Markov Decision Processes (MDPs), and solved by DRL-algorithms in a progressive (stepwise) manner. Both training and testing conditions will be investigated for a theoretical marine vessel model, including how to design the reward functions. Given the continuous state space and action space nature of the problem, emphasis will be given on gradient-based algorithms such as proximal policy optimization, deep deterministic policy gradients, and guided policy search.

Foreword

This project is written as a part of the course TTK4550 at the Norwegian University of Science and Technology (NTNU). The project was carried out in the spring of 2019, under the supervision of associate professor Anastasios Lekkas. The main contributions of this project is a progressive methodology solving the path-following problem, using deep reinforcement learning (DRL). The steps solve different aspects of the path-following problem for a marine vessel, which combined solves the task of path-following.

The task of path-following for a marine vessel has previously been solved using reinforcement learning (RL), for an underactuated marine vessel, i.e., with fewer control settings than degrees of freedom. The fully-actuated marine vessel used in this project was originally presented in [1], used to develop an algorithm of autonomous docking. This project extends the number of actuators used compared to previous research, adding the problem of thruster allocation.

DRL can solve several complex continuous problems. It entails extensive computations, and several libraries are therefore used to save computation time. The methods are therefore implemented in the open source programming language Python, providing libraries with efficient handling of such tasks. The most used libraries in this project are Tensorflow [2], Numpy [3], Matplotlib [4], and Pandas [5], all available as open source code. The deep reinforcement learning algorithm was based on an open source implementation from the organization Spinning Up [6], based on work by Lillecap et al. [7]. The software was run on standard PCs from Lenovo and Dell, without GPU, using the Linux operating system.

The report is written in Latex, and most visualizations are produced using library Matplotlib and the webpage Draw Io [8].

Acknowledgment

A wholehearted thank you to my supervisor, associate professor Anastasios Lekkas. You provided new insights and creative solutions to any and all challenges during the project in the spring of 2019. To my fellow students, thank you for caring for me during this work. Last, but not least, a warm hug to my familiy for sticking out with me during long hours of machine learning and writing.

Trondheim, 18. June 2019.

Abstract

Traditionally the task of path-following is in classical control theory solved by using multiple cooperating controllers. The controllers collectively perform the task of convergence towards the desired path and desired constant forward velocity. Often the problem is solved using cascading controllers, utilizing models of the system dynamics. A vast literature documents these methods. The performance of the system depends on the performance of each controller and uncertainties in the model of the marine vessel. To achieve good results, expert knowledge of both physics and control theory is required when, using these traditional methods.

In recent years, a data-based field of study called deep reinforcement learning (DRL) has successfully been applied to numerous continuous control problems, including the path-following problem. Deep reinforcement learning (DRL) optimizes decision-making problems through exploring actions in an environment, and receiving feedback of performance. Recent developments in DRL has led to successful solutions of previously unsolved tasks by data-based approaches. DRL is, for instance, able to utilize sensor information to find functional control laws, performing end-to-end learning. Therefore, using DRL to solve the path-following problem, uncertainties of the marine vessel becomes less, and direct sensor information might be used.

The contribution of this project is a methodology for developing a DRL-agent that can solve the path-following problem in a progressive/stepwise manner, for a fully-actuated marine vessel. Each step builds on the previous step, where the steps are:

1. Forward velocity control.
2. Forward velocity control & course stabilization.
3. Heading & forward velocity control.
4. path-following system.

The DRL-algorithm is able to perform each step, controlling both thruster angles and force. The DRL created control law is, therefore, able to both replace thruster allocation and the traditional controllers. The degrees of success depend on the complexity of the task and design of the system.

This project shows that DRL can solve several aspects of the path-following problem, comparable to the behavior of traditional controllers. Even though the path-following system may not yet perform as well as more traditional methods, the DRL method can solve the task without explicit knowledge of dynamics of the marine vessel and thruster allocation, or the need of several cascading controllers.

Acronyms

AI	Artificial intelligence
ANN	Artificial neural network
DDPG	Deep deterministic policy gradient
DL	Deep learning
DOF	Degree of freedom
DRL	Deep reinforcement learning
MDP	Markov decision process
ML	Machine learning
RL	Reinforcement learning
SGD	Stochastic gradient descent
DP	Dynamic programming
LP1:Vel	Learning phase 1: Forward velocity control
LP1:Vel agent	Agent trained in LP1:Vel
LP2:VelCourse	Learning phase 2: Forward velocity control and course stablization
LP2:VelCourse agent	Agent trained in LP2:VelCourse
LP3:HeadVel	Learning phase 3: Heading and forward velocity control
LP3:HeadVel agent	Agent trained in LP3:HeadVel
LP4:Path	Learning phase 4: Path-following
LP4:Path agent	Agent trained in LP4:Path

Symbols

α	Learning rate
$\boldsymbol{\eta}$	Position vector for marine vessel
$\boldsymbol{\nu}$	Velocity vector for marine vessel
u	Surge, forward velocity
v	Sway
ψ	Heading, yaw
$\mathbf{w}$	Weight vector
$\mathbf{b}$	Bias vector
$J(\boldsymbol{\theta})$	Loss/objective function of parameter vector $\boldsymbol{\theta}$
$Q(s, a)$	Action-value function
$V(s)$	State-value function
$R(s)$	Reward-function given by state s
S	State-space
A	Action-space
R	Reward-space

Contents

Chapter 1

Introduction

1.1 Background and motivation

Autonomous marine vessels are on the rise, due to increased interest in several fields, such as marine biology, environmental monitoring, seafloor mapping, oceanography, and military use, to name a few. One of the primary tasks of autonomous marine vessels is to follow a predefined path in the presence of unknown environmental forces [9, 10]. The path-following problem has been a subject of extensive research and has led to a plethora of corresponding literature. There has also been developed several prototypes of autonomous marine vessels, such as Falco, [11] and Yara Birkeland [12]. Falco is the world's first fully-autonomous ferry, and Yara Birkeland is the world's first zero-emission autonomous bulk freight ship.

Even though there has been progress in the development of autonomous marine vessels, the problem of being able to steer accurately and reliably in a harsh marine environment is still a challenging task. Motion control systems must yield good performance in the presence of external disturbances, model uncertainty, and with respect to change of objectives during a mission. This is especially relevant in the case of marine vehicles operating in harsh environments [13]. Ocean currents, wind, and waves may induce large deviations from the desired position on the assigned path if not counteracted by the control system.

path-following is the task of converging to and staying on a predefined path without temporal constraints, [10, p. 254]. The problem is challenging due to its highly nonlinear nature, and the high uncertainties in both models of marine vessel and environment. One standard approach to achieve the control objective is to use models representing vessel dynamics and kinematics. These are used to develop suitable kinematic (i.e., guidance, the perturbing system) and dynamic (i.e., control, the perturbed system). Traditional systems often follow a hierarchical cascading architecture, where the guidance system uses lower level controls to achieve the objective in path-following. This results in easier stability analysis, described in a vast literature [14–17]. The cascade architecture is used in such a way that the vehicle's forward velocity tracks the desired speed profile, while guidance acts on the orientation of the vehicle to drive it to the path. Tracking of desired forward velocity is an underlying assumption in path-following [18]. The motion control system also needs to consider uncertainty factors and environmental forces. The cancellation of effects induced by known forces is often assigned to the guidance system since it is responsible for generating reference trajectories to be fed to the underlying control system [9, 19–21].

Several objectives are desired in path-following. There are several fields performing optimization, for instance, reinforcement learning (RL). RL algorithms are capable of tackling a wide range of control problems, ranging from robotics and healthcare to self-driving cars and finance. All these problems are called sequential decision-making tasks, and consists of finding a sequence of actions to be performed in an uncertain environment to achieve predefined goals. From the control theory, the decisions correspond to the control actions that need to be selected to achieve the control objective. RL proposes a formal framework for solving sequential decision-making tasks.

Reinforcement learning (RL) is also known as neuro-dynamic programming or approximate dynamic programming. It is a theory developed by the artificial intelligence (AI) community for reaching optimal performance of the system and for the given uncertainties of the environment [22]. The main idea in RL is that an agent learns, by interacting with the environment, how to optimize some objective given in the form of cumulative rewards. The environment may be stochastic, and the agent may even only perceive partial information about the current state. Making the best possible decision according to some desired set of criteria is often challenging, and even more so when there are time constraints and uncertainty in the system model.

RL has close ties to optimal control theory, where both aims at maximizing an objective function over time [23, p. 30]. The main difference between optimal control theory and RL is that RL gives evaluative feedback, rather than instructive feedback. This means that RL evaluates the actions taken rather than instructing by giving correct actions [22, p.25]. The evaluative feedback in RL is typically given through an engineered reward function, using a scalar depending on whether an action improves on the given objective or not in a specific state. The RL algorithm uses this evaluative feedback to find a mapping from state to action, called policy. A policy is the equivalent of a control law in traditional control theory, and solves the goal/objective in the best possible way [24].

The interest in RL increased when it promised to deal with the curse of dimensionality and modeling [24]. The curse of dimensionality concerns the explosion of computational cost as the number of states increases, while the curse of modeling concerns how to model the reality accurately. Handling these challenges represented a breakthrough in the practical application of RL to complex problems. Some of the achievements were reached by combining RL with deep learning techniques [25–27]. Deep RL (DRL) is most advantageous in problems with high dimensional state-spaces and helps in tasks with lower prior knowledge because of its ability to implicitly learning different levels of abstractions from data [28]. Through the development in DRL, machines have attained a super-human level in playing Atari based on pixel information [29] and mastering Go [30]. Through impressive results on sequential decision-making problems in computer games, researchers started using DRL for real-world applications such as robotics [31], self-driving cars [32] and finance [33]. Even though DRL increased applicability of RL for continuous control, problems with exploring the environment efficiently are still challenging, being able to generalize a good behavior in a slightly different context, etc. For this reason, several algorithms have been proposed in the framework of DRL, depending on the objective of the sequential decision-making tasks.

There already exists a good system performing path-following, so why try to solve the task using DRL? Summarized the reasons are:

- Data-based approaches (such as DRL) might help towards a more general approach and less time spent on engineering a solution for each time, compared to more traditional methods.
- DRL has also already been applied to control marine vessels with successful results [34–36].
- Marine vessels are characterized by highly non-linear the dynamics and uncertainty in the modeling of the vessel. This often complicates and the narrows the region of application in traditional methods.
- DRL is also well suited due to the possibility of using multiple sources of measurements.

This project is inspired by the work of Andreas Bell Martinsen and Anastasios Lekkas [34, 36], and will look into the path-following problem for a 3-degrees-of-freedom (DOF) fully-actuated marine vessel. The key distinguishing factors compared to previous works are:

- The marine vessel is fully-actuated, using two separately controlled actuators, meaning the DRL-algorithm needs to understand both thruster allocation and dynamics of the marine vessel.
- The DRL-algorithm needs to solve simultaneously both objectives of following a

constant desired forward velocity and desired path. In previous works, the DRL-algorithm controlled did not have to solve the task of forward velocity control.

The path-following problem is solved in a progressive/stepwise manner. Some parts of the system, can potentially be transferred to more complex problems, such as docking, collision avoidance, etc. These tasks have less previous research due to the high complexity of the tasks.

1.2 Research hypothesis and contributions

The main objective of this thesis is to investigate how to apply DRL to make a path-following system of an fully-actuated marine vessel, using two separately controlled actuators. The marine vessel will need to meet several objectives to solve the task regarding both forward velocity and path convergence. Consequently, the following research hypothesis was formulated for this project:

- Can DRL solve multilayer problem successfully, such as the task of path-following for a fully-actuated vessel?
- Is it possible to improve learning performance by defining subproblems, and solving the task by solving these subproblems in a progressive manner?
- Is the behavior of the DRL created controllers different from classic controllers?

The contribution of this project is a methodology for developing a DRL-agent that can solve the path-following problem in a progressive/stepwise manner. It starts at a lower level solving control of forward velocity, which also has thruster allocation as a prerequisite. It further more builds on a series of steps to achieve course stability, and applying a number of steps eventually convergence to the desired path and constant forward velocity. The steps involved are:

- Forward velocity control.
- Forward velocity control & course stabilization.
- Forward velocity & heading control.
- Path-following.

The different learning problems consists of two main components:

1. **An environment**: The environment will return observations of state and measurements of performance for doing action $\mathbf{a}$ in state $\mathbf{x}$. In reinforcement learning the environment also consists of the dynamics of the marine vessel.
2. **A DRL agent**: The DRL agent consists of a policy, which maps an observation of the state to next control action.

The agent learns the control policy, through exploration and exploitation of the actions in the environment. The policy is represented using Artificial Neural Networks (ANNs), learned through the DRL method, called Deep Deterministic Policy Gradient (DDPG).

1.3 Outline of report

The report is organized as follows:

- Chapter 2 introduces important concepts and notation of reinforcement learning, in particular deep learning and deep reinforcement learning, focusing on policy gradient methods.
- Chapter 3 discusses some of the main concepts of kinematics, motion control, guidance for marine vessels and path-following.
- Chapter 4 mentions the tools used, the simulated vessel, and path-following using deep reinforcement learning (DRL). This includes the use of state vectors, action vectors, reward function and transfer learning under different scenarios, before discussing steady-state error compensation and the DDPG-algorithm.
- Chapter 5 delves into the results.
- Chapter 6 is future work
- Chapter 7 is conclusion

Chapter 2

Reinforcement learning

AI is a large field, and has no generally accepted formal definition. There is however consensus that it is concerned with a *thought process*, *reasoning* and *behavior* [23]. The field of AI attempts to build intelligently behaving entities, dealing with a wide range of issues such as playing chess at master level and driving autonomously in a crowded street.

Machine learning (ML) is an important field within AI. ML addresses the question of how to build computer programs that improve their performance of some task through experience [37]. Machine learning uses ideas from a diverse set of disciplines, among others including control theory and psychology. ML algorithms have been applied in a variety of application domains and have shown to be of genuinely practical value.

Commonly ML algorithms are divided into four categories, based on their purpose and data [23]. The four main categories are:

- **Supervised learning**: The task of supervised learning is given a training set of examples of input-output pairs, approximating the relationship between input and output. This relationship may be used to predict the output values for new data, based on learned relationships.
- **Unsupervised learning**: In unsupervised learning, the program learns patterns/rules based on the input data even though no explicit feedback is supplied, meaning there does not exist any explicit input-output pairs.
- **Semi-supervised learning**: In semi-supervised learning, the program learns patterns/rules based on a given mixture of examples of input-output pairs and input without output examples. Noise/oracular truths might corrupt the input and output data, and both the noise and lack of explicit output data makes this problem separate from supervised and unsupervised learning.

- **Reinforcement learning**: In reinforcement learning (RL) the program learns by receiving a series of reinforcements, called rewards. The program needs to analyze and learn how to act to to receive the highest cumulative reward.

Deep reinforcement learning (DRL) is a subfield within RL, using a technique called deep learning for function approximation. Deep learning is important for efficiently using RL in continuous control input and state space.

The subjects of the next sections give an introduction to fields related to reinforcement learning, and gives a introduction to specific areas and methods used in this project. Introductions provided are:

- Section 2.1: Artificial neural networks (ANNs), used for function approximation.
- Section 2.2: Reinforcement learning fundamentals.
- Section 2.3: Learning methods for estimating value functions.
- Section 2.4: Classes or RL algorithms.
- Section 2.5: Deep reinforcement learning (DRL).
- Section 2.6: Policy gradient methods.
- Section 2.7:Transfer learning.

2.1 Neural networks for function approximation

Function approximation is an instance of supervised learning, where the mathematical definition is [23]:

Given a training set of n example input-output pairs $(x_1, y_1), (x_2, y_2), ...(x_N, y_N)$ *where each* y_j *was generated by an unknown function* $y = f(x)$ *discover a function* $\hat{f}$ *that approximates the true function.*

The set $(x_1, y_1), (x_2, y_2), ...(x_N, y_N)$ is called a training set.

The approximated function can be thought of as a way of mapping of input-output pairs. The function approximation $\hat{f}$ is found through searching the space of possible function approximations, for one who performs well on the training set. A hypothesis performs well if it correctly predict the value of y for a sample not present in the training set. If so, this means the algorithm is applicable to other relevant datasets of the domain in question, and generalizes well.

Different techniques exists, based on the type of input-output data (continuous, discrete,etc.) and complexity of function (linear, nonlinear, etc.). In recent years DL has improved the learning from high-dimensional data such as time series, images and videos [28]. Artificial neural networks (ANN) and DL are considered well suited when working on continuous input-output pairs, and high function complexity. The combination of ANN and DL is called Deep neural networks (DNN), and is often used in Deep Reinforcement Learning (DRL). To give a short introduction into this filed, this section will briefly present:

- Artificial neural networks (ANNs).
- Deep neural networks (DNNs).
- Training of neural networks.

2.1.1 Artificial neural networks

Artificial neural networks (ANNs) is data-driven computing, inspired by the human brain. The brain is a highly complex organ, capable of performing advanced computing with large amounts of data. ANN is in the same manner well suited for finding relations in problems with huge amounts of data, without prior explicit knowledge of underlying relationship between the measured inputs and the observed outputs [22, 38].

The objective of ANN is to map an input into a desired output, analogous to a mathematical function. The internal function structure of ANNs is patterned after the interconnections between neurons found in biological systems. The ANNs principles are however a result of manmade inventions, that bear little resemblance to the biological systems. ANNs have a long history, and the first generation of neural network was implemented in 1957 by Frank Rosenblatt [39]. It was called the "perceptron", and learned automatically how to classify simple geometrical objects. Neural networks have evolved, and today ANNs are, among others, used for nonlinear functions approximation [22].

An ANN is a network of interconnected units with similar properties as neurons in the brain [22, 23, 25]. Neurons are the main component of the nervous systems. A unit consists of inputs and one output. The output is found by first calculating a weighted sum of the input signal $\mathbf{x}$ plus bias b, meaning $z = \sum x_i w_i + b$. The output a is computed by applying the weighted sum to a activation function $f(z)$, giving output $a = f(\sum_i x_i w_i + b)$. Figure 2.1 shows the relationship between the input and output, of one neuron.

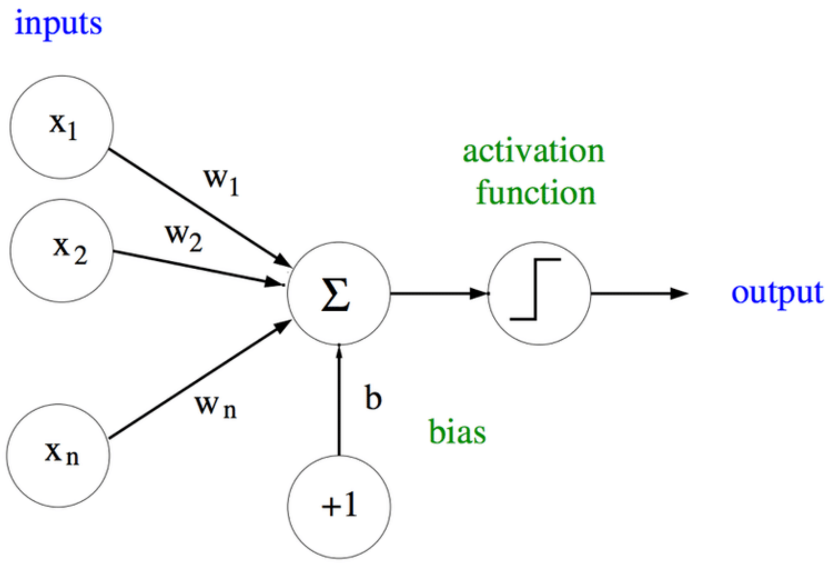


Figure 2.1: Model of an artificial neuron. Illustration from [40].

Each link to the neuron has its individual weight. The individual weights multiplied with the respective inputs gives the strength of the signal into the unit. The units are connected through those links, where a link from unit i to j propagate the activation a_i from i to j, with a numeric weight $w_{i,j}$, visualized in Figure 2.2. The weights are the parameters of the function represented by the neural network.

The activation function produces the unit's output, from the weighted sum z, and is typically a nonlinear function. Some of the more commonly used activation functions are sigmoid (2.1a), rectifier nonlinearity (ReLU) (2.1b), hyperbolic tangent (tanh) (2.1c), or no activation at all (2.1d):

$$f(z) = \frac{1}{1 + e^{-z}}, \tag{2.1a}$$

$$f(z) = \max(0, z), \tag{2.1b}$$

$$f(z) = \tanh(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}}, \tag{2.1c}$$

$$f(z) = z. \tag{2.1d}$$

The units of an NN can be connected in different manners, using different topologies. There are two fundamentally distinct ways of connecting units: *Feed-forward networks* and *recurrent networks*. Feed-forward networks only have connections in one direction, making a directed acyclic graph. A recurrent network, on the other hand, sends at least some of the outputs back into its inputs. By feeding back previous outputs, it is possible to gain short-term

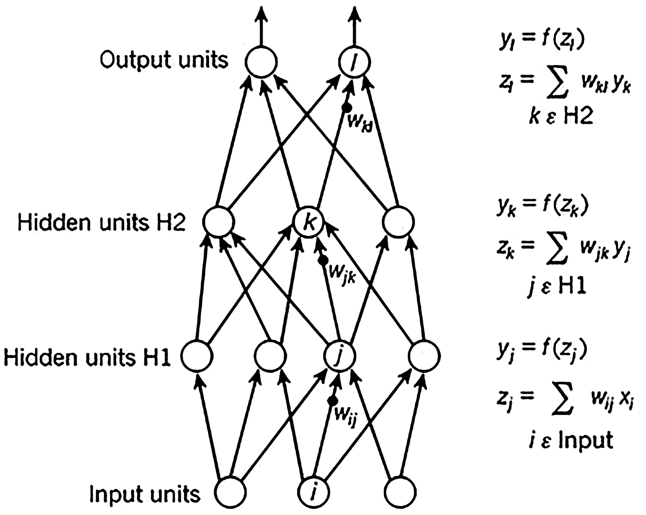


Figure 2.2: A generic feedforward ANN with four input units, two output units, and two hidden layers. Note that bias is not included in this particular figure. Illustration from [25].

memory.

Feed-forward networks are usually arranged in layers, with each unit in one layer receiving input only from units from within the immediately preceding layer. There are different types of layers, where the three main categories are:

- Input layer: The main assignment of the input layer is to feed the input to the first hidden layer.
- Hidden layer: A hidden layer is neither directly connected to the input nor the output.
- Output layer: The output layer presents the output from the neural network.

An example of a generic feedforward ANN is shown in figure 2.2. This ANN consists of four-layers, with one input-layer consisting of three units, two hidden layers each consisting of three and four units and one output-layer consisting of two output unit. The figure includes the equations used for computing the forward pass in the neural network.

The number of layers represent the depth of a network, and an ANN can consist of as little as no hidden layers. A network consisting of no hidden layers can represent only a small fraction of the possible input-output functions. An ANN with one single hidden layer, can approximate any continuous function to any degree of accuracy, as long as it contains a large enough finite number of sigmoid units. This theorem is called the universal approximation theorem. It is also true with other nonlinear activation functions, under certian conditions.

The ANNs properties are decided by both how the units are connected (topology) and the properties of the units. To approximate a desired function, both topology, weights and activation function can be adjusted to reach a desired objective.

2.1.2 Deep neural networks

Experience and theory shows that the task of approximating complex functions are made easier through the use of several hidden layers, in so-called deeper neural networks (DNNs) [22, 23, 25]. DNNs are characterized by a succession of multiple processing layers, where the sequence of these transformations lead to learning different levels of abstraction. An example of a simple deep feed forward neural network is shown in Figure 2.2.

Representation learning are methods that discover the representations needed for detection or classification from raw data [25]. Deep learning methods are representation learning methods with multiple levels of representation. Each layer consists of a distinct representation of the input, and for each hidden layer a new more abstract representation of the input is found. Very complex functions can be approximated through a sufficiently large number of hidden layers.

An image can serve as an example, where the first layer might transform an array of pixel values into a representation of presence or absence of edges for certain locations and orientation. The second layer may be representing patterns looking at edges, regardless of small variations in the edge position. Learning from data using a "general-purpose learning procedure", without human engineers, is one of the key features of deep learning.

Deep learning relies on a function $f : X \leftarrow Y$ parameterized with $\boldsymbol{\theta} \in \mathbb{R}^{n_\theta}$ [28]:

$$y = f(x; \boldsymbol{\theta}). \tag{2.2}$$

2.1.3 Training neural networks

Training of neural networks (NN) is the process of adjusting weights, to improve the networks overall performance as measured by an objective function $J(\boldsymbol{\theta})$, with weight vector $\boldsymbol{\theta}$ [22, 23, 25]. The objective function $J(\boldsymbol{\theta})$, also called loss-function, measures the error between the output from the NN and the correct output in the training set (called label). The objective will depend on the desired properties of the system.

The partial derivative of the objective function with respect to the weights , vector $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$, can be used to properly adjust the weight vector. The partial derivative indicates how the error from the network would increase or decrease if the weights were increased by an infinite small amount, given current values of all the network's weights. The weights can then be adjusted to decrease the error. A popular method to adjust the weights using the gradient of the objective is stochastic gradient descent (SGD).

SGD-methods are called "gradient descent" methods since the step $\boldsymbol{\theta}$ is proportional to the negative gradient of the objective function $J(\boldsymbol{\theta})$. This is the direction in which the error falls most rapidly. Gradient descent methods are called stochastic when the update is performed, on a single stochastically selected example.

SGDs try to minimize errors on a single example by adjusting the weight vectors after each example, by a small amount in the direction that would give the highest error reduction on that sample:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}), \tag{2.3}$$

with $\alpha > 0$ as step-size parameter, also called learning rate.

The gradient $\nabla J(\boldsymbol{\theta})$ is simpler to calculate in networks with no hidden layers, as the training data provides an easy way to calculate the error, and only one layer of weights causing the error. Updating several layers of weights, with only performance measure available in one layer, is a more challenging task. To solve this problem, the error can be back-propagated from the output layer to the hidden layers. Backpropagation enables DNNs to change the weights to differentiate the representation in each layer from the representation of the previous layer.

The backpropagation algorithm consists of alternating forward and backward passes through the network, calculating the partial derivatives of the error with respect to the weights for each layer. The idea is that hidden node j is "responsible" for some fraction of the error of the output nodes connected to it. It is therefore necessary to divide the error between the different hidden nodes. This is done according to the strength of the connection between the hidden node and the output node, and propagated back to the specific hidden layer. To propagate the error back, the algorithm utilizes that the gradient of the objective function with respect to the weights of a DNN can be found by applying the chain rule for derivatives. This is illustrated in Figure 2.3, showing a computational graph from input x to output y. Each node is a function of the previous one, with $x = f(w), y = f(x), z = f(y)$. To compute $\frac{\partial z}{\partial w}$, the chain rule gives:

$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial z}\frac{\partial x}{\partial w}. \tag{2.4}$$

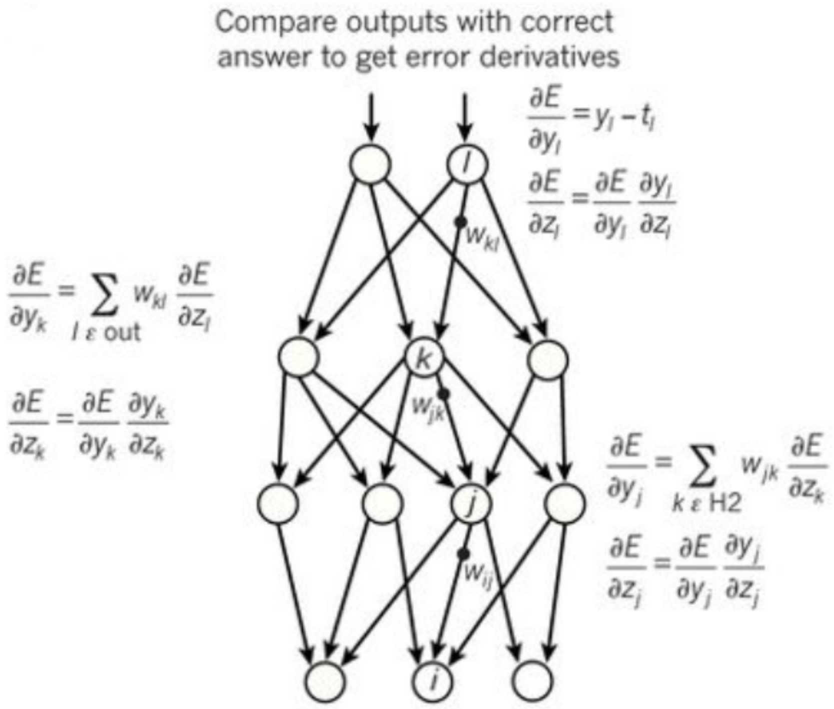

Compare outputs with correct answer to get error derivatives
$\frac{\partial E}{\partial y_l} = y_l - t_l$
$\frac{\partial E}{\partial z_l} = \frac{\partial E}{\partial y_l} \frac{\partial y_l}{\partial z_l}$
l
w_{kl}
$\frac{\partial E}{\partial y_k} = \sum_{l \varepsilon \text{out}} w_{kl} \frac{\partial E}{\partial z_l}$
$\frac{\partial E}{\partial z_k} = \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial z_k}$
k
w_{jk}
$\frac{\partial E}{\partial y_j} = \sum_{k \varepsilon \text{H2}} w_{jk} \frac{\partial E}{\partial z_k}$
$\frac{\partial E}{\partial z_j} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial z_j}$
j
w_{ij}
i

Figure 2.3: Illustration of backpropagation [25]

The forward passes compute the activation of each unit using the current activation of the network's input units. After each forward pass, a backward pass efficiently computes a partial derivative for each weight, as an estimate of the true gradient. Through backpropagation the partial derivative of the error with respect to the input is calculated, by simply passing gradients back through the network in reverse order, and multiplying it with the gradients of the previous layers.

The deeper the neural network is, the harder it can be to train. The backpropagation algorithm can produce good results for networks with few layers, typically with 1 to 2 hidden layers. There are several reasons for this. Deeper network have more weights, and it is harder to generalize correctly (called overfitting). Another problem is that the partial derivatives computed by backward passes either decay rapidly toward the input side of the network, or grow rapidly towards the input side of the network. This makes it harder to learn. To solve problems learning deep neural networks, several techniques have been developed, such as dropout [41], batch normalization [42] and deep residual learning [43].

2.2 Reinforcement learning fundamentals

Reinforcement learning (RL) is the area of machine learning that deals with sequential decision-making [17]. In RL an agent needs to make decisions in an environment, learning how to behave such that it optimizes a given objective. The agent acts according to a policy. A scalar reward is given as feedback to the agent based on behavior. This feedback is used to change its policy. The goal of the agent is to maximize the expected cumulative reward.

The agent may have some information about the environment (a priori) or it needs to gain this of the environment through interaction with the environment. It learns through exploring the environment, using trail-and-error, and collect information about the environment.

The following introduction consists of:

- A problem formulation in RL called Markov decision process (MDP) in section 2.2.1.
- A discussion of the role of a policy in section 2.2.2.
- The principles of how to calculate the expected cumulative reward in section 2.2.3.
- The principles of how to evaluate the value of being in a certain state s in section 2.2.4.
- The principles of optimal behaviour and value in section 2.2.5.

2.2.1 Markov decision process

Markov decision processes (MDPs) is a class of sequential decision problems [23]. Sequential decision problems incorporate utilities (performance), uncertainty and perception. The utility of a system depends on a sequence of actions performed in an uncertain environment, and may be positive or negative but must be bound. MDP is an intuitive and fundamental formalism used in multiple disciplines such as (stochastic) planning, robot control, and game playing problems [44]. RL solves MDPs, finding a policy which maximizes the expected cumulative reward.

In MDP an agent interacts with an environment in discrete time steps [22, 23]. The environment is modeled as a set of states s, where an agent performs actions a to control/influence the environment. The agent is expected to operate autonomously, being able to perceive an environment, adapt to changes and pursue goals. It selects actions based on its perception of the environment. After performing an action, the environment gives the agent a new state s' and receives feedback through a scalar reward signal r. The interaction between the environment and the agent is illustrated in Figure 2.4.

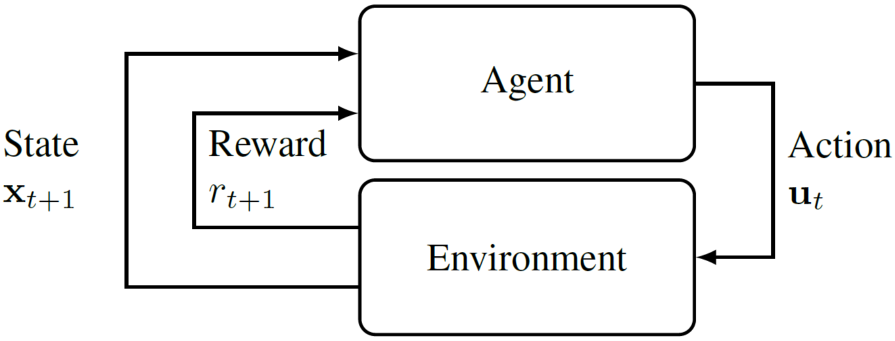


Figure 2.4: Visualization of the interaction between the environment and the agent. Illustration from [36].

The reward signals to the agent what to achieve, but not how. The reward is given by a reward function $R(\cdot)$, which defines the goal of a reinforcement learning problem. It can depend on the current state of the world, the action just taken, the next state of the world, or any combination of the three.

The reward function R is critically important in reinforcement learning. It is the primary basis for altering the policy, such that the agent maximizes the expected total reward it receives. If an action given by the policy achieves a low reward, the policy may be changed during the learning process to select another action in the same situation in the future. It is thus critical that the reward-function is designed to truly reflect the problem to be solved.

An MDP is defined as a 5 tuple $< S, A, R, P, \gamma >$, where:

- S is is the set of all valid states, called observation/state space. A MDP needs to have a set of fully observable states, with an initial state s_0. Fully observable means the observations are the same as the states of the environment.
- A is the action space. $A(s)$ is the set of all valid actions for state s, called action space. Action space can be discrete or continuous.
- $R : S \times A \times S \longrightarrow \mathbb{R}$ is the reward function, with $r_t = R(s_t, a_t, s_{t+1}) \in R$.
- $P : S \times A \times S \longrightarrow [0, 1]$ is the Markovian transition probability function $P(s'|s, a)$.
- $\gamma \in [0, 1)$ is the discount factor.

A stochastic transition model $P(s'|s, a)$ denotes the probability of reaching state s' given action a is performed in state s. For a Markovian transition model, the probability of reaching

s' from s depends only on s and not on the history of earlier states. This is called the Markov property, and means that the future of the process only depends on the current observation, and the agent has no interest in looking at the full history.

2.2.2 Policy

A policy is a mapping from states to actions and gives the action recommended in state s, and is a solution to the MDP [22]. If the agent has a complete policy, the agent will always know what to do next from any state at any given time.

A policy can either be deterministic or stochastic:

- A stochastic policy is denoted $\pi(s, a)$, with $a \sim \pi(\cdot|s_t)$, and gives the probability of selecting each possible action for a given state s.
- A determinisitc policy, gives $\pi(s) : S \rightarrow A$.

The quality of a policy is measured by the expected utility of the possible environment histories generated by the policy, due to the stochastic nature of the environment. An optimal policy yields the highest expected utility, and is further explained in section 2.2.5. RL methods specify how its experience should change the agent's policy, in order to achieve higher expected cumulative reward.

2.2.3 Goal, reward and return

The reward signal is a way of formalizing the purpose or goal of the agent, and is a distinctive features of RL [22]. The reward signal r given after performing action a_t signals the immediate and intrinsic desirability of environmental states. Maximizing based on immediate rewards does not necessarily lead to maximizing the total amount of reward it receives in the long run. For example, a state might give a low reward in itself, but can be followed by states that give rewards resulting in a high cumulative reward. The agent wants to maximize the cumulative reward, making the best decision in the long run, as this is closer to achieving the goal/objective of the agent . This idea is stated in the reward hypothesis [22, p. 53]:

That all of what we mean by goals and purpose can be well though of as the maximization of the expected value of the cumulative sum of a received scalar signal (called reward).

During the creation of the reward function it is crucial that the agent receives rewards aiding the agent to achieve the desired goal [22]. The design of the reward function is a part of the design of the environment, and is a crucial part. The reward function should only indicate the desired objective, and not how this should be achieved. The agent should be free to solve the assignment within the constraints, hopefully providing new and smart (or at lest creative) ways of solving the task.

The return is denoted G_t and is defined as some specific function of the reward sequence, meaning some notion of the cumulative reward. Two ways of defining the return is finite-horizon undiscounted return, (2.5a), and infinite-horizon discounted return, (2.5b):

$$G_t = \Sigma_{k=0}^{T} r_{t+k+1}, \tag{2.5a}$$

$$G_t := \Sigma_{k=0}^{\infty} \gamma^k r_{t+k+1}, \tag{2.5b}$$

where the sequence of rewards received after time step t is denoted $r_{t+1}, r_{t+2}, r_{t+3}, \ldots$.

Finite-horizon undiscounted return is the sum of rewards obtained in a fixed number of steps. Examples of where finite-horizon undiscounted return might be more appealing is in episodic tasks. In an episodic task, there is a natural notion of the final time step, meaning the agent-environment interaction breaks naturally into subsequences, called episodes, with a final time step T. The end state of an episodes is called the terminal state. The terminal state may have different outcomes (rewards). An example of how the same terminal state of a system can give different outcomes is found when playing games. The player either win or lose, when in terminal state, but by how much may vary.

A problem with the finite-horizon undiscounted return, is when the final time step $T \longrightarrow \infty$, leading to a possible infinite cumulative reward. Since the agent maximizes the return, this is not a desired scenario. The cases where $T \longrightarrow \infty$ are called continuing tasks. These are cases where the agent-environment interactions do not break naturally into identifiable episodes, but goes on forever. Examples of such processes are on-going process-control tasks or an robot performing an assignment with a long life span. Infinite-horizon discounted return is a way of handling these tasks. With the infinite-horizon discounted return, the return is the sum of all rewards received k time steps in the future discounted by γ^{k-1}. γ is the discount rate, where $0 \leq \gamma \leq 1$, and determines the present value of future rewards. The closer γ gets to one the agent becomes more farsighted, meaning it takes the future rewards more strongly into account. Similarly, if $\gamma = 0$, the agent is only concerned with maximizing immediate rewards.

Both finite-horizon undiscounted and infinite-horizon discounted return, meets the consistency condition. This means that the returns at successive time steps are related, giving:

$$G_t := r_{t+1} + \gamma G_{t+1}. \tag{2.6}$$

This condition makes it easier to calculate returns from reward sequences, and is used in developing algorithms in RL. It works for all time steps t < T, even if termination occurs at t+1, as long as the termination reward $G_T = 0$.

2.2.4 Value functions

A value function seeks to estimate the expected return [22]. It "indicates the long-term desirability of states after taking into account the states that are likely to follow based on the policy π", and the rewards available in those states [22]. Since the value function seeks to estimate the expected return, the value function is more critical for action choices, than the immediate reward. The value function is used to select the actions giving the highest amount of reward in the long run, and thereby optimizing the agent goal/objective.

There are two main types of value-functions:

- **State-value function** gives the expected return when the agent follows policy π after timestep t, starting in state s. It is denoted $v_\pi(s)$.
- **Action-value function** gives the expected return for taking action a in state s under policy π. It is denoted $q_\pi(s, a)$.

The state-value function $v_\pi(s)$ when the agent follows policy π after timestep t, starting in state s is:

$$v_\pi(s) := \mathbb{E}_\pi[G_t|S_t = S] = \mathbb{E}_\pi[\Sigma_{k=0}^{\infty}\gamma^k R_{t+1+1}|S_t = s], \text{for all } s \in S, \tag{2.7}$$

where $\mathbb{E}_\pi(\cdot)$ denotes the expected value of the return given state s.

The action-value function $q_\pi(s, a)$ starting from state s, taking the action a under policy π is:

$$q_\pi(s, a) := \mathbb{E}[G_t|S_t = s, A_t = a] = \mathbb{E}[\Sigma_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s, A_t = a]. \tag{2.8}$$

The state-value and action-value function can be estimated based on experiences received over the agent's lifetime. How this estimation is performed is an important part of several reinforcement algorithms.

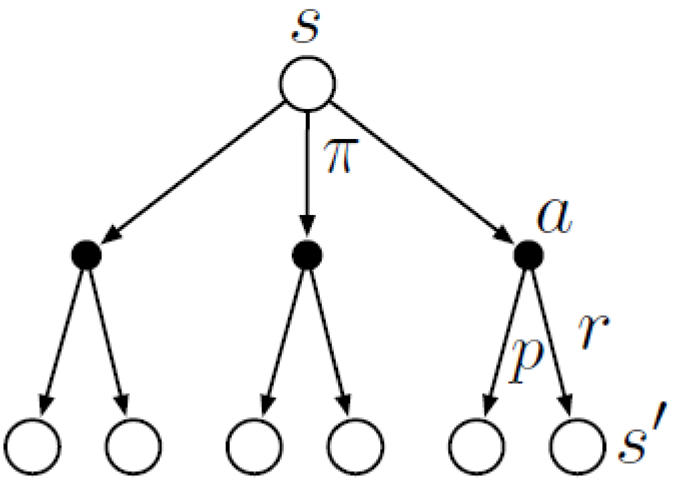


Figure 2.5: Backup diagram illustrating different paths from initial state s (root node), based on performed action a from policy π and possible next states s' based on transition model p. The agent receives reward r for performing action a in state s [22]

The consistency condition of the return impacts the value-function to also have it. This means for any policy, π and state s a consistency condition holding between the values of successive states s and s':

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s] = \Sigma_a \pi(a|s) \Sigma_{s',r} p(s', r|s, a)[R + \gamma v_\pi(s')], \tag{2.9a}$$

$$q_\pi(s, a) = \mathbb{E}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] = \Sigma_{s'} p(s'|s, a)[R + \gamma \Sigma_{a'} \pi(s', a') q_\pi(s', a'), \tag{2.9b}$$

for all $s, s' \in S$, $r \in R$ and $a \in A(s)$. These equations are the Bellman equations for v_π and q_π, acting on-policy. On-policy means that the agent is always acting according to the current policy π.

The Bellman equation for v_π gives the sum over all values of the three variables a, s' and r. It is illustrated in Figure 2.5. The open circles represent a state, and the solid circles represent a state-action pair. The root node at the top represents the starting state s. The agent can take any actions available for a certain state, based on its policy π. Performing a certain action will lead to a new state s', along with a received reward r, depending on the dynamics given by probability p. The Bellman equation averages over all possibilities, weighting each by its probability of occurring.

2.2.5 Optimal policies and optimal value functions

The goal in RL is to find a policy that achieves the highest expected return, where the optimal policy performs better or equal to all other policies [22]. The optimal (or near optimal) policy for the environment is found through interaction/experimentation of the agent in the environment (i.e., the MDP), gaining knowledge about how to optimize its behavior through the rewards received after performing an action in a specific state. An optimal policy is reached when it maximizes the expected total reward.

A policy π with expected return greater than or equal to that of π' for all states, is defined to be better than or equal to policy π'. This means that $\pi \leq \pi'$ if and only if $v_\pi(s) \leq v_{pi'}(s)$ for all $s \in S$. The optimal policy is denoted π_*. It may be more than one optimal policy, but they all share the same optimal value-function. The optimal action-value q_* gives the expected return for taking action a in state s and after that following the optimal policy, while the optimal state-values v_* gives the expected return for state s and thereafter following the optimal policy. These are defined by the Bellman optimality equations:

$$v_* = \max_\pi v_\pi(s), \text{ for all} s \in S, \tag{2.10a}$$

$$q_*(s,a) = \max_\pi q_\pi(s,a), \text{ for all} s \in S \text{ and } a \in A(s). \tag{2.10b}$$

The Bellman optimality equation (2.10a) is a system of equations, one for each state. If there are n states, there exist n equations with n unknowns. For finite MDPs (finite state-, reward- and action-space) a unique solution to the optimal state-value function v_* exists, if the dynamics of the environment p are known. Then v_* may be found by applying traditional methods for solving nonlinear equations. The same method may be used to solve q_*. Dynamic programming (DP) is algorithms that uses a perfect model of the environment to compute optimal policies.

After calculating v_* and q_*, finding the optimal actions is relatively straightforward. The optimal actions found through v_* are the actions leading to the maximum obtained by the Bellman optimality equation for v_*. For the optimal action-value function q_* the optimal policy is found immediately by looking up the actions for a specific state s, giving the maximum value. The optimal action-value function gives the optimal actions without knowledge of the dynamics of the environment.

2.3 Learning methods for estimating value functions

Different techniques can be used to find the optimal policy. When the dynamics of environment is known, dynamic programming (DP) may be used, as described in the previous section. When the environment is unknown, estimates of value function are typically needed [22]. This may be done using either:

- Monte Carlo methods.
- Temporal-difference learning.

In the following sections, the techniques will be explained through estimation of state-value function $V(S_t)$, but these methods can be also used to approximate other functions.

2.3.1 Monte Carlo methods

In RL, Monte Carlo (MC) methods are capable of estimating the return, by averaging sample rewards. This can be performed by using samples of sequences of states, actions and rewards from actual or simulated interactions with an environment. It is thus not needing explicit knowledge of the environment dynamics, but rather depending on experience. Using only experience the method is theoretically, able to attain optimal behaviour.

Monte Carlo methods estimate the return, by simply averaging the returns observed after visits to that state. Monte Carlo methods must therefore wait until the return of a certain state s is known, and thereafter use the return as a target for state-value function updates. The average converges towards the expected value, as more returns are observed.

There are different methods to update the average return. One technique is called the every-visit MC method, which estimates $v_\pi(s)$ as the average following all visits to state s. A simple every-visit Monte Carlo method is:

$$V(S_{t+1}) = V(S_t) + \alpha[G_t - V(S_t)], \tag{2.11}$$

where G_t is the actual return following time t, and α is the step-size parameter.

MC-methods gives an unbiased but noisy estimator of the expected return.

2.3.2 Temporal-difference learning

Temporal-difference (TD) learning combines ideas from Monte Carlo and dynamic programming (DP). TD uses experience, in the same manner as Monte Carlo methods, and has no knowledge of the dynamics of the environment per se. The difference is that TD methods update estimates based partly on other learned estimates, without waiting for a final outcome (bootstrapping). This means TD will only have to wait until the next time step, whereas Monte Carlo methods have to wait until the end of an episode to update the estimate (receiving G_t). The simplest TD method at time $t + 1$ updates value-function estimation based on reward R_{t+1} and the estimated $V(S_{t+1})$, giving:

$$V(S_t) = V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \tag{2.12}$$

This is called TD(0), or one step TD, looking ahead on step at a time.

TD learning, DP and Monte Carlo methods estimates different aspects of $V_\pi(s)$:

- Roughly speaking, the target of Monte Carlo methods is to use an estimate of $v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s]$, averaging samples of returns.
- DP methods use an estimate of $v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1} | S_t = s)]$, where $V(S_{t+1})$ is an estimate even though the dynamics of the environment are known.
- The TD target is an estimate of $\mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1} | S_t = s]$, sampling the expected values and using the current estimate of V instead of the true v.

TD error is defined in (2.13), and is the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_{t+1})$. γ_t is the error in $V(S_t)$, available at time $t + 1$:

$$\delta_t := R_{t+1} + \gamma V(S_{t+1}) - V(S_t). \tag{2.13}$$

2.4 Classes of RL algorithms

There are different components used to learn an agent how to maximize the expected reward [17]. An RL agent will include one or more of the following components:

- A representation of a value function.
- A representation of the policy π.

- Model of the environment, with estimated transition- and/or reward function.

When the method uses a model of the environment, the method is called model-based RL, otherwise the method is called model-free. The model-free methods are typically split into value-based, policy-based and actor-critic methods [17, 45, 46], based on which components the agent consists of:

- Value-based: Methods based on value functions.
- Policy-based: Methods based on policy functions.
- Actor-critic methods: A hybrid, based on both value and policy functions.

2.4.1 Value-based methods

The value-based methods aims at learning a value function that estimates the expected return of being in a given state,which subsequently is used to define a policy [17]. The optimal policy is found by selecting the action that maximizes the learned value function. For continuous control the problem of finding an action is not trivial, since choosing an action requires solving a maximization problem. Extensions are made for value-based methods to handle continuous controls, but they are often restrictive [47, 48]. Examples of value-based methods are Q-learning [49] and deep Q-network [29].

2.4.2 Policy-based methods

Policy-based methods, or policy search methods, learn directly a parameterized policy [45]. The methods search directly for an optimal policy $\pi*$, but without learning the value function. A parameterized policy π_θ is typically used, with parameters updated to maximize the expected return $\mathbb{E}[R|\theta]$, using either gradient-based or gradient-free optimization . Most methods are using gradient-based training, due to being more sample-efficient for policies with a large number of parameters. An example of a policy gradient method is REINFORCE [50]. Policy gradient methods are explained in more details in section 2.6.

2.4.3 Actor-critic methods

Actor-critic methods learn both a value function and an explicit representation of the policy, merging the advantages of value-based and policy search methods [45]. The methods consists of an actor and a critic, where the actor learns by using feedback from the critic. The goal of these methods is to learn an actor that maximizes the critic. The actor is the parameterized policy and the critic is a low variance estimator of the expected return (value function). Actor-critic methods therefore often converge much faster than policy search methods with high variance. A trade off is higher bias than policy search methods. Examples of actor-critic methods are actor-critic [51, 52], natural actor-critic [53], trust-region policy optimization [54], proximal policy optimization algorithms [55] and deep deterministic policy gradient [7].

2.5 Deep reinforcement learning

Explicitly solving the Bellman optimality equation (2.10) is one way finding an optimal policy, and thereby solving the reinforcement learning problem [22]. However, this is a method seldom used for real world problems, for different reasons. Problems include the need to know accurately the dynamics of the environment, the computational resources needed to solve the solution and the Markov property. The computational resources and memory constraints arise in problems with high state- and action-space, leading to the need for solving several optimal Bellman equations and requiring sufficient space to store these solutions.

To avoid the need of value-functions for each state or state-action pair, the idea is to approximate a good general policy or value-function, which can be used in the entire state-space. The approximated functions use some sort of compact parameterized representation. The online nature of RL makes it possible to put more effort into learning to make good action selections for frequently encountered states. This is at the expense of putting less effort into infrequently encountered states. This means the approximated functions are able to make decisions in states which it has not experienced. This feature distinguishes RL from other approaches approximately solving MDPs.

In theory function approximation in RL can be performed using all methods from supervised learning, but not all methods are equally suitable. Methods that are more suitable for RL are methods with the following properties:

- The learning can be performed online, meaning the function approximation are performed while the agent interacts with its environment or with a model of its environment. This requires the method to learn efficiently from incrementally acquired data.
- The method is able to handle target function or values changing over time, called non-stationary target functions and values. The first might come from approximating

the value function q_π while π are still changing, while the second might come from approximated return from bootstrapping methods like TD learning.

The function approximations may be made through deep learning, giving deep reinforcement learning. Neural networks are for instance well suited for dealing with high-dimensional sensory inputs. Often function approximation are used on value-functions $Q(s_t, a_t)$ or $V(s_t)$, or policy $\pi(a_t|s_t)$, with approximations $Q(a_t|s_t, \mathbf{w})$ and $V(s_t, \mathbf{w})$, with parameter vector $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and$\gtrapprox \in R^d$. The parameterized policy $\pi(a|s, \boldsymbol{\theta})$ gives the probability of taking action a at time t given the environment state s with parameter $\boldsymbol{\theta}$, giving $\pi(a|s, \boldsymbol{\theta}) = P(A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta})$. The parameterized state-value function gives the expected return for state s_t at time t with parameter $\mathbf{w}$, while action-value function gives the expected return given state s_t and action a_t at time t with parameter $\mathbf{w}$.

2.6 Policy gradient methods for DRL

Policy gradient methods is a class of methods which uses stochastic gradient ascent with respect to the policy parameters to optimize the expected cumulative reward to find a good policy[17, 22]. They belong to the broader class of policy-based methods, since it directly learns a parameterized policy.

The policy is parameterized by parameter vector $\boldsymbol{\theta} \in \mathbb{R}^{d'}$. The parameterized policy $\pi(a|s, \boldsymbol{\theta})$ gives the probability of taking action a at time t given the environment state s with parameter $\boldsymbol{\theta}$, giving $\pi(a|s, \boldsymbol{\theta}) = P(A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta})$. The method used to parametrize the policy, is not important in policy gradient methods as long as the policy is differentiable with respect to its parameters. This means $\nabla \pi(a|s, \boldsymbol{\theta})$ exists and is finite for all $s \in S, a \in A(s)$, and $\theta \in R^{d'}$. This parameterization can be performed using neural network.

The parameter vector needs to be the changed to improve the performance measure, where the goal is to estimate the optimal policy $\pi_{\boldsymbol{\theta}} = \pi_*$. Stochastic gradient ascent is used to update the policy parameter $\boldsymbol{\theta}$ to maximize the performance measure $J(\boldsymbol{\theta})$, giving:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla \hat{J}(\theta_t), \tag{2.14}$$

where $\nabla \hat{J}(\theta_t) \in \mathbb{R}^{d'}$ is an approximation of the gradient of J with respect to the policy parameter $\boldsymbol{\theta}_t$. It can be approximated using the policy gradient theorem, explained in section 2.6.1. Stochastic gradient ascent is a similar technique as, stochastic gradient descent, but it maximizes the objective, and takes a step in the positive direction of the gradient.

The performance measure $J(\boldsymbol{\theta})$ is different for episodic and continuous cases, where in this rapport we will focus on the first. For an episodic case the performance measure is the value of the start state for the episode, meaning $J(\boldsymbol{\theta}) = v_{\pi_\theta}$. v_{π_θ} is the true value function for the policy π_θ determined by $\boldsymbol{\theta}$.

2.6.1 Policy gradient theorem

The policy gradient theorem provides an analytical expression for the gradient of the performance measure $J(\boldsymbol{\theta})$ with respect to the policy parameter $\boldsymbol{\theta}$ [22]. The gradient of the performance measure can be estimated in different ways, but using the policy gradient theorem gives a representation which is independent of the action selection and state distribution. This is a good feature given that the environment is generally unknown, making it difficult to estimate the effect of a policy update on the state distribution. State distribution is a the distribution of how often the states occur following policy.

The policy gradient theorem represents the gradient of J as a column vector of partial derivatives with respect to the components of $\boldsymbol{\theta}$. The policy gradient theorem for an episodic case for an on-policy algorithm is:

$$\nabla J(\boldsymbol{\theta}) \propto \mathbb{E}_\pi[G_t \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})], \tag{2.15}$$

with $\frac{\nabla \pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})} = \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$.

The gradient vector describes the direction in parameter space that most likely increases the probability of repeating the action A_t on future visits to state S_t. Using this gradient in the stochastic gradient ascent algorithm, helps moving in the direction that favor actions that gives the highest return.

The policy gradient theorem makes it possible to sample the return on each time step, which expectation is equal to the gradient. The update of the parameter vector $\boldsymbol{\theta}$ using (2.15), yields the REINFORCE update:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha q(A_t, S_t) \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}), \tag{2.16}$$

with learning rate α, policy $\pi(a|s, \boldsymbol{\theta}$ and action-value function $q(a, s)$.

The policy gradient theorem lays the theoretical foundation for several policy gradient algorithms. The algorithm called REINFORCE uses the update of the same name, sporting good theoretical convergence properties. This happens at the cost of high variance using a Monte Carlo method to approximate the policy gradient (2.15), also leading to slow learning.

One way of reducing the variance is by including a comparison of the action value to an arbitrary baseline b(s), giving the approximated policy gradient:

$$\nabla J(\boldsymbol{\theta}) \propto \Sigma_s \mu(s) \Sigma_a (q_\pi(s,a) - b(s)) \nabla(a|s,\boldsymbol{\theta}), \tag{2.17}$$

The only restriction on baseline $b(s)$ is that it cannot depend on action a. The policy gradient theorem with baseline is used to give a version of REINFORCE where the parameter vector is updated according to:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta} + \alpha(G_t - b(S_t)) \frac{\nabla \pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})}, \tag{2.18}$$

This is a strict generalization of REINFORCE, since the baseline could be uniformly zero. The baseline in general does not affect the expected value of the update, but it can improve its variance.

One intuitive choice for baseline is an estimate of the state value, $b(s) = \hat{v}(S_t, \mathbf{w})$, where $\mathbf{w} \in R^d$ is a weight vector in the approximated state-value function. The state-value weights $\mathbf{w}$ may be learned using Monte Carlo methods.

The REINFORECE algorithm with baseline, is represented in algorithm 1. The baseline here is an approximated state-value function, $b(s) = \hat{v}(S_t, \mathbf{w})$. This algorithm has two step sizes, denoted $\alpha^{\boldsymbol{\theta}}$ and α^{w}.

Algorithm 1 REINFORCE: with Baseline (episodic), for estimating $\pi_{\boldsymbol{\theta}} = \pi_*$

```
Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$ and state-value function parameterization $\hat{v}(s,\mathbf{w})$
Algorithm parameter: step sizes $\alpha^{\boldsymbol{\theta}} > 0$ and $\alpha^{\mathrm{w}} > 0$.
Initalize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and $\mathbf{w} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)
for each episode do
    Generate an episode $S_0, A_0, R_1, ...S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot,\boldsymbol{\theta})$
    Loop for each step of the episode t = 0,1,..., T-1
    $G \leftarrow \Sigma_{k=t+1}^{T} \gamma^{k-t-1} R_k$
    $\delta \leftarrow G - \hat{v}(S_t, \mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathrm{w}} \delta \nabla \hat{v}(S_t, \mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t \delta \nabla ln \pi(A_t|S_t,\boldsymbol{\theta})$
end for
```

2.6.2 Actor-critic

Methods that learn approximations of both policy and value functions are called actor-critic methods [56]. These methods consist of an actor and a critic. The critic continuously evaluates the performance of the actor by providing the value of being in a certain state when acting according to the policy (value function). The actor selects the action to be performed in a certain state (policy), and is it uses feedback from the critic to improve its policy.

Actor-critic methods are hybrids of value-based methods and policy-based methods. The fundamental difference between actor-critic methods and policy gradients is the use of a learned value function as baseline.

In the previous section, the algorithm REINFORCE with baseline was presented, see algorithm 1, using a learned value-function and policy, but this algorithm is not an actor-critic method [22]. This is because the estimation is not updated based on estimated values of subsequent states (bootstrapping), and as such the function is not used as a critic. The value-function in REINFORCE is used as a baseline for the state which estimate is being updated, and does not criticize directly the progress of the development of policy function.

The algorithm REINFORCE with baseline are unbiased and will converge asymptotically to a local minimum. Using Monte Carlo methods it tends to learn slowly and produce estimates of high variance. It is also inconvenient to implement online or for continuing problems, due to the need to wait for the return after an episode. Through TD methods these inconveniences can be eliminated, giving lower variance and an online-algorithm. Actor critic methods therefore use TD-learning. By using TD(0), the algorithm is a fully online and incremental algorithm. An online algorithm has the property of being able to collect labels and train itself while it is also executing as a working agent, constantly learning with every step.

Actor-critic methods approximates the policy $\pi(a|s, \boldsymbol{\theta})$ with parameter vector $\boldsymbol{\theta}$ and approximated state-value function $\hat{v}(S_t, \mathbf{w})$ by weight $\mathbf{w}$. The value-function is approximated through TD(0)-learning. The intuitive explanation, is that after the actor applies an action to the environment, the critic uses the TD error, (2.13), to evaluate if the return is higher or lower than expected. If the TD error is positive, the critic suggests that the tendency to select this action should be strengthened for the future, and weaker if the TD error is negative. The interaction between actor and critic is illustrated in Figure 2.6.

The desired objective is to get both TD error and the estimation error of the value-function converging towards zero. This means that the value-function estimator is a good approximation of the true value function V_π, meaning gaining an optimal policy. The TD error is minimized using stochastic gradient ascent for updating the weight $\mathbf{w}$ of the critic, giving

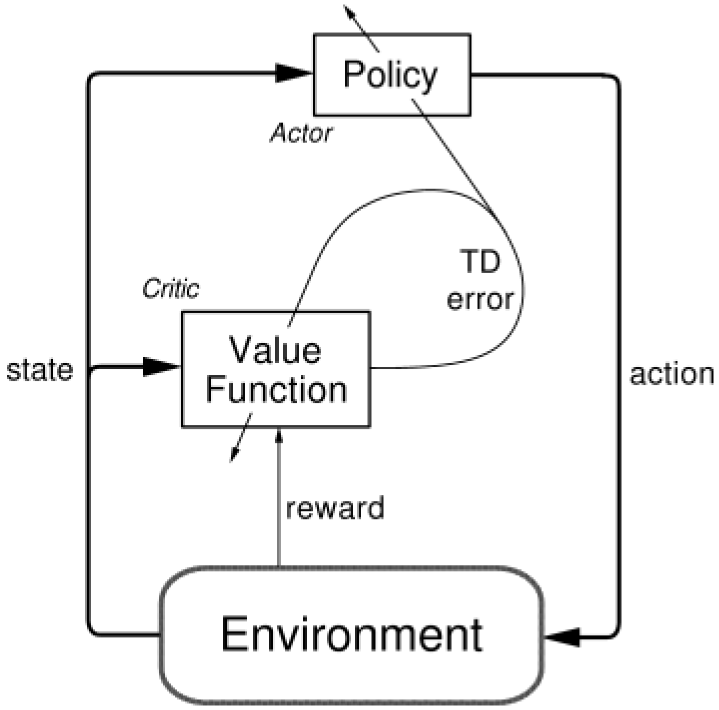


Figure 2.6: Figure showing the interaction between actor and critic in the actor-critic model. The critic gives the value of being in a certain state when acting according to the policy of the actor, while the actor gives the action to be performed in a certain state [22]

the update equation:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha_w \delta_t \nabla_{\mathbf{w}} V_{\mathbf{w}}(s). \tag{2.19}$$

The actor updates its policy based on the approximated policy gradient, using TD error given by the critic. The actor is updated in a similar manner as in REINFORCE, but the full-return is replaced with the one-step return $(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$ giving:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \delta_t \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta}). \tag{2.20}$$

An entire one-step actor-critic algorithm for estimating $\pi_\theta = \pi^*$, is presented in algorithm 2.

Algorithm 2 One-step Actor-Critic (episodic), for estimating $\pi_{\boldsymbol{\theta}} = \pi_*$

```
Input: a differentiable policy parameterization $\pi(a|s, \boldsymbol{\theta})$ and state-value function parameterization $\hat{v}(s, \mathbf{w})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}} > 0$ and $\alpha^{\mathbf{w}} > 0$.
Initialize policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathbf{w} \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)
for each episode do
    Initialize S (first state of episode)
    $\mathbf{I} \leftarrow 1$
    Loop for each step of the episode t = 0,1,..., T-1
    $A \sim \pi(\cdot|S, \boldsymbol{\theta})$
    Take action A, observe S',R
    $\delta \leftarrow R + \gamma\hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha^{\mathbf{w}}\delta\nabla\hat{v}(S, \mathbf{w})$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\mathbf{I}\gamma^t\delta\nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$
    $\mathbf{I} \leftarrow \gamma\mathbf{I}$
    $S \leftarrow S'$
end for
```

2.6.3 Deep deterministic policy gradient

Deep deterministic policy gradient (DDPG) [7] is an actor-critic and model-free algorithm. It is able to operate in continuous state and action space . It approximates the policy (actor) and the value function (critic), and has a structure similar to the one-step episodic actor-critic Algorithm 2.

DDPG uses the action-value function $Q^{\pi}(s_t, a_t)$, instead of using the state-value function, as presented in the actor-critic Algorithm 2. The action-value function describes the expected return after taking an action a_t in state s_t and thereafter following policy π:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}[G_t|s_t, a_t], \tag{2.21}$$

with infinite-discounted future return R_t,and discounting factor γ.

DDPG uses a deterministic policy, $\mu : S \longrightarrow A$. Using the Bellmann equation (2.9) the action-value can be described as:

$$Q^{\mu} = \mathbb{E}[r(s_t, a_t) + \gamma Q^{\mu}(s_{t+1}, \mu(s_{t+1}))], \tag{2.22}$$

Since DDPG uses the action-value, the algorithm does not depend on explicitly knowing the dynamics of the environment, and thus the algorithm can learn Q^{μ} off-policy. This means the algorithm can use transitions generated from a different stochastic behavior policy β

during learning.

The action-value function is parameterized by parameter θ^Q, and is:

$$J(\boldsymbol{\theta}^Q) = \mathbb{E}[(Q(s_t, a_t|\boldsymbol{\theta}^Q) - y_t)^2], \tag{2.23a}$$

$$y_t = r(s_t, a_t) + \gamma Q(s_{t+1}, \mu(s_{t+1})|\boldsymbol{\theta}^Q). \tag{2.23b}$$

The loss function used to approximate the action-value function is the squared TD error for action-value, meaning $L = Q(s_t, a_t|\boldsymbol{\theta}^Q) - y_t$.

The deterministic policy is parameterized with parameter vector $\boldsymbol{\theta}^\mu$, giving a deterministic mapping from states to a specific action. The Q-function is updated by taking one step of gradient descent using $\nabla_{\boldsymbol{\theta}^Q} L(\boldsymbol{\theta}^Q)$:

$$\boldsymbol{\theta}^Q \leftarrow \boldsymbol{\theta}^Q + \alpha^{\boldsymbol{\theta}^Q} \nabla_{\boldsymbol{\theta}^Q} L(\boldsymbol{\theta}^Q). \tag{2.24}$$

The actors parameter $\boldsymbol{\theta}^\mu$ is updated by calculating the gradient of the loss function, with respect to the actor parameters, from state-distribution β giving the policy gradient :

$$\nabla_{\boldsymbol{\theta}^\mu} J \sim \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_{\boldsymbol{\theta}^Q} Q(s, a|\boldsymbol{\theta}^\mu)|_{s=s_t, a=\mu(s_t|\boldsymbol{\theta}^\mu)}] = \mathbb{E}_{s_t \sim \rho^\beta}[\nabla_a Q(s, a|\boldsymbol{\theta}^Q)|_{s=s_t, a=\mu(s_t|\boldsymbol{\theta}^\mu)} \nabla_{\boldsymbol{\theta}^\mu} \mu(s|\boldsymbol{\theta}^\mu)|_{s=s_t}], \tag{2.25}$$

using the chain rule $\frac{\partial Q}{\partial \theta_i^\mu} = \frac{\partial Q}{\partial \mu(s_t|\theta_i^\mu)} \frac{\partial \mu(s_t|\theta_i^\mu)}{\partial \theta_i^\mu}$.

The policy is updated by using one step of gradient ascent giving:

$$\boldsymbol{\theta}^\mu \leftarrow \boldsymbol{\theta}^\mu + \alpha^{\boldsymbol{\theta}^\mu} \nabla_{\boldsymbol{\theta}^\mu} L(\boldsymbol{\theta}^Q). \tag{2.26}$$

Theoretical convergence is not guaranteed when introducing non-linear function approximators. These approximators are essential to be able to generalize when having large state and action spaces. DDPG solves this problem by using replay buffer, and a separate target network for calculating y_t.

The replay buffer is a finite buffer storing transitions (s_t, a_t, r_t, s_{t+1}) during exploration. The actor and critic are updated at each time step by a sampling a small set of transitions (minibatch) uniformly from the buffer. When the buffer is sufficiently large, DDPG is able to learn from a set of uncorrelated transitions. It therefore fulfills the assumptions (of many optimization algorithms) that the samples are independently and identically distributed. The replay buffer can be used due to DDPG being off-policy.

Soft target updates consists of creating a copy of the actor and critic networks, called target critic $Q'(s, a|\boldsymbol{\theta}^{Q'})$ and target actor $\mu'(s|\boldsymbol{\theta}^{\mu'})$. These networks are used for calculating the target values, where the target policy is the optimal policy giving $\mu(s) = \text{argmax}_a Q*(s, a)$ and target Q-function $Q*(s, a)$. The reason for using soft target update is that $Q(s, a|\boldsymbol{\theta}^{Q})$ is used in the loss function. This means the Q-update is prone to divergence, due to the loss function depends on the parameter being trained. The target-networks are therefore updated by factor $\tau << 1$, slowly tracking the learned networks:

$$\boldsymbol{\theta}^{Q'} \leftarrow \tau\boldsymbol{\theta}^{Q} + (1-\tau)\boldsymbol{\theta}^{Q'}, \tag{2.27a}$$

$$\boldsymbol{\theta}^{\mu'} \leftarrow \tau\boldsymbol{\theta}^{\mu} + (1-\tau)\boldsymbol{\mu}'. \tag{2.27b}$$

This update improves the stability of learning, but might learn slowly, due to soft target updates. In practice the improved stability outweighs slower learning.

Another problem is how to handle features with different scales, as this can lead to problems with too small or too big gradients during backpropogation. This is solved by using batch normalization [42], previously discussed in 2.1.

To find a good policy it is needed to explore the solution space. DDPG explores the solution space by adding noise samples from noise process N to the actor policy:

$$\mu'(s_t) = \mu(s_t|\boldsymbol{\theta}_t^{\mu}) + N. \tag{2.28}$$

Since DDPG is an off-policy algorith, the exploration can be treated independently from the learning algorithm.

A pseudocode of the DDPG-algorithm is in Algorithm .

2.7 Transfer learning

Reinforcement learning has been successfully applied to numerous learning tasks, while other tasks have more mixed results. Some of these problems have improved the learning performance by transferring experience gained in learning one task, to a different but similar task [57]. Transfer learning (TL) is transferring knowledge from one or more *source tasks* to

Algorithm 3 DDPG algorithm, for estimating $\pi_{\boldsymbol{\theta}} = \pi_*$

```
Input: a differentiable policy parameterization $\mu(s|\boldsymbol{\theta}^{\mu})$ and action-value function parameterization $Q(s, a|\boldsymbol{\theta}^{Q})$
Parameters: step sizes $\alpha^{\boldsymbol{\theta}^{Q}} > 0$ and $\alpha^{\boldsymbol{\theta}^{\mu}} > 0$.
Randomly initialize critic network $Q(s, a|\boldsymbol{\theta}^{Q})$ and actor $\mu(s|\boldsymbol{\theta}^{\mu})$, with parameter $\boldsymbol{\theta}^{Q}$ and $\boldsymbol{\theta}^{\mu}$.
Initialize target network $Q(s, \boldsymbol{\theta}^{Q'})$ and $\mu(a|s, \boldsymbol{\theta}^{\mu'})$, with weights $\boldsymbol{\theta}^{Q'} = \boldsymbol{\theta}^{Q}$ and $\boldsymbol{\theta}^{\mu'} = \boldsymbol{\theta}^{\mu}$.
Initialize replay buffer
for each episode do
    Initialize a random process $N$ for action exploration
    Initialize S (first state of episode)
    Loop for each step of the episode t = 0,1,..., T-1
    $A \sim \mu(S|\boldsymbol{\theta}^{\mu}) + N_t$
    Take action A, observe S',R
    Store transition (S,A,R,S') in replay buffer
    Sample a random minibatch of N transitions $(s_i, a_i, r_i, s_{i+1})$ from replay buffer
    Set $y_i = r_i + \gamma Q(s, \mu'(S|\boldsymbol{\theta}^{\mu'}|\boldsymbol{\theta}^{Q'})$ for $i \in 1 \ldots N$
    Update critic by minimizing the loss: $\frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\boldsymbol{\theta}^{Q}))^2$
    Update policy with $\frac{1}{N}\sum_i \nabla_a Q(s_i, \mu(s_i|\boldsymbol{\theta}^{\mu}|\boldsymbol{\theta}^{Q})\nabla_{\boldsymbol{\theta}^{\mu}}\mu(s_i|\boldsymbol{\theta}^{\mu}$
    Update critic target network: $\boldsymbol{\theta}^{Q'} \leftarrow \tau\boldsymbol{\theta}^{Q} + (1-\tau)\boldsymbol{\theta}^{Q'}$
    Update actor target network: $\boldsymbol{\theta}^{\mu'} \leftarrow \tau\boldsymbol{\theta}^{\mu} + (1-\tau)\boldsymbol{\mu'}$
end for
```

learn one or more *target tasks* faster than if transfer was not used. The insight behind TL is that "generalization may occur not only within tasks, but also across tasks" [57].

There are different groups of methods used to transfer learning, depending on attributes of the problems:

- Methods applied to tasks which have the same state variables and actions.
- Methods that transfer between tasks with different state variables and actions.
- Methods that learns a mapping between tasks with different state variables and actions.

Knowledge transferred will depend on the source and target task. Knowledge transferred in TL, with examples from RL, both low-level or high level, such as:

- Low-level knowledge, e.g. action-value function Q, a policy π.
- Higher level knowledge, e.g. what action to use in some situations (i.e. a subset of the full set of actions), rules or advice.

The type of knowledge transferred directly affects the type of learning applicable. For instance, a TL method that transfers an action-value function originally learned using TD would likely require the target task agent to use the same method to exploit the transferred knowledge.

Research shows that if two tasks are closely related the learned policy from a source task can be used to provide a good initial policy for the target task [58]. A task i consists of state-space S_i, action-space A_i, transition model T_i, reward space R_i and policy π. An example of TL is first training a policy on task 1 with $(S_1, A_1, T_1, R_1, \pi_0)$, providing a solution π_1. Thereafter the target task (task 2) is initialized with π_1, solving the task 2 with $(S_2, A_2, T_2, R_2, \pi_1)$, giving solution π_2. If $S_1 = S_2$ and $A_1 = A_2$, π_1 is a legitimate policy for task 2. If else a mapping of either or both action- and state-space is needed.

An example of TL used on a real project is first learning an agent to solve a 1-D pole balancing task with a shorter pole and decreasing mass [59]. This knowledge was then transferred to a new agent solving a problem with higher complexity, by using a longer pole. The new agent learned more quickly, due to TL.

Chapter 3

Guidance and control of marine vessels

The deep reinforcement learning (DRL) agent learns through interaction with the environment. In this project the environment consists of the dynamics of marine vessel, effected by a current. The environment is simulated using the equations of motion of a theoretical marine vessel. These dynamics are also often used in design of traditional path-following-systems. The knowledge of traditional systems for path-following can give useful heuristic which can simplify the learning process of an DRL-agent.

General information about the dynamics of a marine vessel will be presented in this chapter, as well as one traditional method of solving the straight-line path-following problem.

3.1 Kinetics and kinematics

A marine vessel are described by number of degrees of freedom (DOF), with maximum 6-DOF [10]. "The Society of Naval Architects and Marine Engineers" (SNAME) notation of 6-DOF are presented in Table 3.1. These can be used to determine the position and orientation of the marine vessel, and can be grouped as:

- Coordinates describing position and translation motion: The Cartesian coordinates describing position (x, y, z) and linear velocities (u, v, ω).
- Coordinates describing orientation and rotational motion: The Euler angles (ϕ, θ, ψ) and angular velocities (p, q, r).

These coordinates can be given in several geographic reference frames. Two common reference frames are:

Table 3.1: The notation of SNAME (1950) for marine vessels , [10, p. 16].

DOF	**Linear and angular velocite**	**Position and euler angle**	**Force and moment**
Motions in the x direction (surge)	u	x	X
Motions in the y direction (sway)	v	y	Y
Motions in the z direction (heave)	ω	z	Z
Rotation about the x axis (roll)	p	ϕ	K
Rotation about the y axis (pitch)	q	θ	M
Rotation about the z axis (yaw)	r	ψ	N

- **The North-East-Down (NED) coordinates system**: $\{n\} = \{x_n, y_n, z_n\}$, is usually defined as the tangent plane on the surface of the Earth moving with the craft. The x-axis points towards North, y-axis points towards east, and z-axis points downwards normal to the earths surface.

- **The body-fixed reference frame**: $b = \{x_b, y_b, z_b\}$ is a moving coordinate frame fixed to the craft. The body axes of a marine craft are chosen to coincide with the principal axes of inertia, giving x_b directed from aft to the fore, y_b directed to starboard, and z_b directed from top to bottom.

The dynamics of a system consists of kinematics and kinetics. Kinematics concerns the geometrical aspects of motion, while kinetics is concerned with motion caused by forces. The marine craft equations of motion gives a vectorial form of a rigid body dynamics, notation from [10], and are:

$$\dot{\boldsymbol{\eta}} = \mathbf{J}_{\Theta}(\boldsymbol{\eta})\boldsymbol{\nu}, \tag{3.1a}$$

$$\mathbf{M}_{RB}\dot{\boldsymbol{\nu}} + \mathbf{C}_{RB}(\boldsymbol{\nu})\nu = \boldsymbol{\tau}_{RB}, \tag{3.1b}$$

$$\boldsymbol{\tau}_{RB} = \boldsymbol{\tau}_{hyd} + \boldsymbol{\tau}_{hs} + \boldsymbol{\tau}_{wind} + \boldsymbol{\tau}_{wave} + \boldsymbol{\tau}_{control}, \tag{3.1c}$$

where the rigid-body forces are represented in (3.1c). The matrices consists of inertia matrix $\mathbb{M} \in \mathbb{R}^{6x6}$ and coriolis matrix $\mathbb{C}(\nu) \in \mathbb{R}^{6x6}$. $\boldsymbol{\nu} = [u, v, w, p, q, r]^T$ and and $\boldsymbol{\eta} = [x_n, y_n, z_n, \phi, \theta, \psi]$, consisting of:

- (x, y, z): The distance from NED to BODY expressed in NED coordinates.

- (ϕ, θ, ψ): The Euler angles, representing angles between n and b.

- (u, v, w, p, q, r): The linear and angular velocities in body-fixed reference frame.

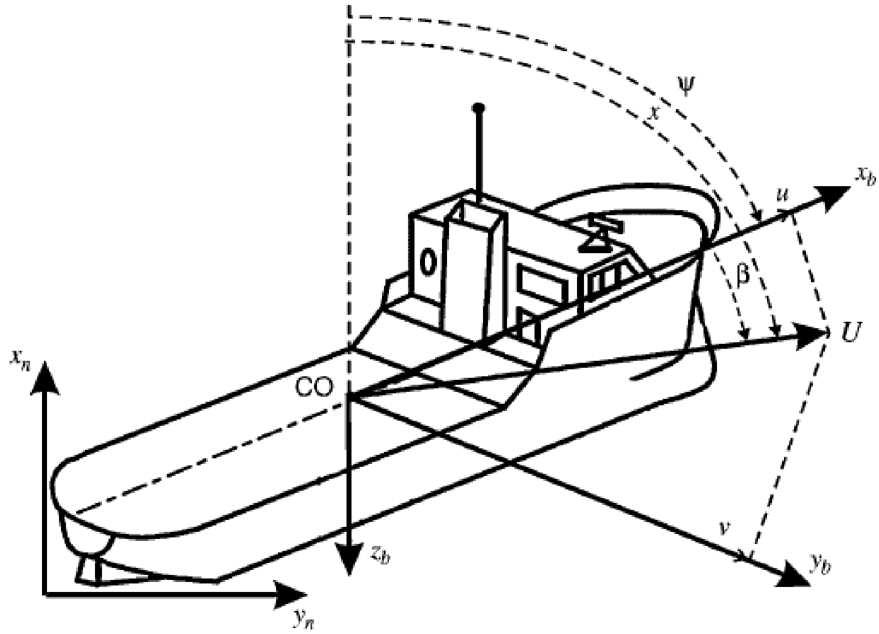


Figure 3.1: An illustration of a 3-DOF surface vehicles, with illustration of surge u, sway v, course χ, heading angle ψ and sideslip angle β, with body-fixed reference frame $\{b\} = \{x_b, y_b, z_b\}$ and Earth-fixed reference frame $\{n\} = \{x_n, y_n, z_n\}$. Illustration from [10, p. 40].

3.1.1 Maneuvering models for surface vessel

A frequently used simplification of a surface vessel is only using 3-DOF. The three remaning degrees of freedom are surge,sway and yaw, and are illustrated in Figure 3.1.

The 3-DOF horizontal plane models for maneuvering are based on (3.1b). The assumption for using 3-DOF model is that the hydrostatic forces $\boldsymbol{\tau}_{hs} = 0$ and small values of angular velocity ω and Euler angles ϕ and θ. These simplifications makes a good approximation for most conventional ships, and yields the equations of motion:

$$\dot{\boldsymbol{\eta}} = \mathrm{J}_{\Theta}(\boldsymbol{\eta})\boldsymbol{\nu}, \tag{3.2a}$$

$$\mathbf{M}_{RB}\dot{\boldsymbol{\nu}} + \mathbf{C}_{RB}(\boldsymbol{\nu})\nu = \boldsymbol{\tau}_{wind} + \boldsymbol{\tau}_{wave} + \boldsymbol{\tau}_{control}, \tag{3.2b}$$

where $\boldsymbol{\nu} = [u, v, r]^T$ and $\eta^n = [N, E, \psi]$. The rotation matrix $\mathbf{R}(\psi)$ is:

$$\mathbf{R}(\psi) = \begin{bmatrix} \cos(\psi) & -\sin(\psi) & 0 \\ \sin(\psi) & -\cos(\psi) & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.3}$$

If current is present, the hydrodynamic forces needs to be considered. The hydrodynamic forces can for a 3-DOF marine vessel can be modeled as:

$$\boldsymbol{\tau}_{hyd} = \mathbf{M}_A\dot{\boldsymbol{\nu}} + \mathbf{C}_A(\boldsymbol{\nu_r})\nu_r - \mathbf{D}(\boldsymbol{\nu_r})\nu_r, \tag{3.4}$$

where $\boldsymbol{\nu}_r$ is the relative velocity $\boldsymbol{\nu} - \boldsymbol{\nu}_c$, and $\boldsymbol{\nu}_c$ is the velocity of the ocean current.

If the ocean currents are constant and irrotational in n, and use the model of hydrodynamic forces from (3.4), the equations of motion be can reorganized to:

$$\mathbf{M}\dot{\boldsymbol{\nu}}_r + \mathbf{C}(\boldsymbol{\nu}_r)\boldsymbol{\nu}_r + \mathbf{D}(\boldsymbol{\nu}_r)\boldsymbol{\nu}_r = \boldsymbol{\tau}_{\text{wind}} + \boldsymbol{\tau}_{\text{wave}} + \boldsymbol{\tau}_{\text{control}} \tag{3.5a}$$

$$\mathbf{M} = \mathbf{M}_{\text{RB}} + \mathbf{M}_{\text{A}}, \tag{3.5b}$$

$$\mathbf{C}(\boldsymbol{\nu}_r) = \mathbf{C}_{RB}(\boldsymbol{\nu}_r) + \mathbf{C}_A(\boldsymbol{\nu}_r), \tag{3.5c}$$

with inertia matrix $\mathbb{M} \in \mathbb{R}^{3x3}$, Coriolis matrix $\mathbf{C}(\nu) \in \mathbb{R}^{3x3}$ and dampening matrix $\mathbf{M} \in \mathbb{R}^{3x3}$.

For maneuvering in the horizontal plane, the relationship between course, heading and sideslip are important. These angles are illustrated in Figure 3.1. The sideslip angle β , also called drift angle, is "the angle from the x_b axis of b to the velocity vector of the vehicle " [10, p. 40]. It is calculated as follows:

$$\beta = \arcsin(\frac{v}{U}), \tag{3.6}$$

where U is the speed of the marine craft:

$$U = \sqrt{u^2 + v^2}. \tag{3.7}$$

The course angle χ is "the angle from x_n axis of n to the velocity vector of the craft" [10, p. 41]. It is calculated as follows:

$$\chi = \psi + \beta, \tag{3.8}$$

where ψ is the heading (yaw) angle, and is "the angle from x_n axis of n to the x_b axis of b" [10, p. 41].

3.1.2 Ocean current forces and moments

One simple model of a ocean current for a 3-DOF marine vessel model is a 2-D irrotational ocean current model [10]. The model only simulates motion in the horizontal plane, with:

$$\boldsymbol{\nu}_c^n = \begin{bmatrix} V_c \cos(\beta_c) \\ V_c \sin(\beta_c) \\ 0 \end{bmatrix}, \tag{3.9}$$

where β_c is the angle of the current and V_c the absolute value of the ocean current velocities.

The ocean current can be transformed to reference-frame body $\{b\}$ by using the rotation matrix $\mathbf{R}(\psi)$ from (3.3), leading to:

$$\boldsymbol{\nu}_c^b = R(\psi)^\top \boldsymbol{\nu}_c. \tag{3.10}$$

3.1.3 Control allocation

The generalized control forces $\boldsymbol{\tau}_{\text{controll}} \in \mathbb{R}^n$ to the actuators, needs to be expressed in terms of control inputs $\mathbf{u} \in \mathbb{R}^r$ [10]. When $r >= n$ it is called an fully-actuated control problem, and if $r < n$ it is referred to as an underactuated control problem.

The relationship between actuator forces, moments and control forces can be specified by the thrust configuration matrix $\mathbf{T}(\boldsymbol{\alpha}) \in \mathbb{R}^3$. For a 3-DOF model it maps the thrust force f and body-frame angles α from each thruster into the surge, sway and yaw forces and moments in the body frame. The control force is calculated as follows:

$$\boldsymbol{\tau} = \mathbf{T}(\boldsymbol{\alpha})\mathbf{f}, \tag{3.11}$$

where each column in $\mathbf{T}(\boldsymbol{\alpha})$ gives the thruster configuration of thruster i, called $T_i(\alpha_i)$. Thruster configuration of thruster i is:

$$T_i(\alpha_i)f_i = \begin{bmatrix} F_x \\ F_y \\ F_y l_x - F_x l_y \end{bmatrix} = \begin{bmatrix} f_i \cos(\alpha_i) \\ f_i \sin(\alpha_i) \\ f_i(l_x \sin(\alpha_i) - l_y \cos(\alpha_i)) \end{bmatrix}, \tag{3.12}$$

where l_x and l_y are the moment arms.

The thruster allocation problem is selecting thruster angles $\boldsymbol{\alpha}$ and forces $\mathbf{f}$ to achieve the desired force $\boldsymbol{\tau}$. It can be solved in numerous ways for a fully-actuated vessel. The problem can also include for instance limitations of input amplitude or rate saturation.

The thruster allocation problem depends on the thruster, and two examples of common actuators are:

- **Tunnel thruster** produces force F_y in the y-direction. It is only effective in low speeds, and is used in low-speed maneuvering and stationkeeping.
- **Azimuth thruster** produces two force components F_x, F_y in the horizontal plane, controlled by rotating an angle α about the z-axis and force F.

3.2 Path-following

Path-following is the task of following a predefined path and velocity, independent of time [10, 17]. A well known method used for path-following is line-of-sight (LOS) guidance. LOS guidance calculates a vector from the craft to a point on the path between two way points. The difference between this LOS vector and the prescribed path can be used as setpoint for a heading controller, and will force the craft to track the path. Guidance laws of speed comes in addition to LOS guidance. We will in this section focus on lookahead-based LOS-guidance, for a straight-line path-following.

The straight-line considered, is between the two waypoints $\mathbf{p}_k^n = [x_k, y_k]$ and $\mathbf{p}_{k+1}^n = [x_{k+1}, y_{k+1}]$. The path tangential angle describes the angle of the path, and is defined as:

$$\gamma_p = \operatorname{atan2}(y_{k+1} - y_k, x_{k+1} - x_k), \tag{3.13}$$

where atan2 is the four-quadrant version of $\arctan(y/x) \in [-\pi/2, \pi/2]$.

The coordinates of the craft in the path-fixed reference frame can be computed by:

$$\boldsymbol{\epsilon} = \begin{bmatrix} s(t) \\ y_e(t) \end{bmatrix} = \mathbf{R}_p(\alpha_k)^T(\mathbf{p}^n(t) - \mathbf{p}_k^n), \tag{3.14a}$$

$$y_e(t) = -(x(t) - x_k)\sin(\alpha_k) + (y(t) - y_k)\cos(\alpha_k), \tag{3.14b}$$

$$s(t) = (x(t) - x_k)\cos(\alpha_k) + (y(t) - y_k)\sin(\alpha_k), \tag{3.14c}$$

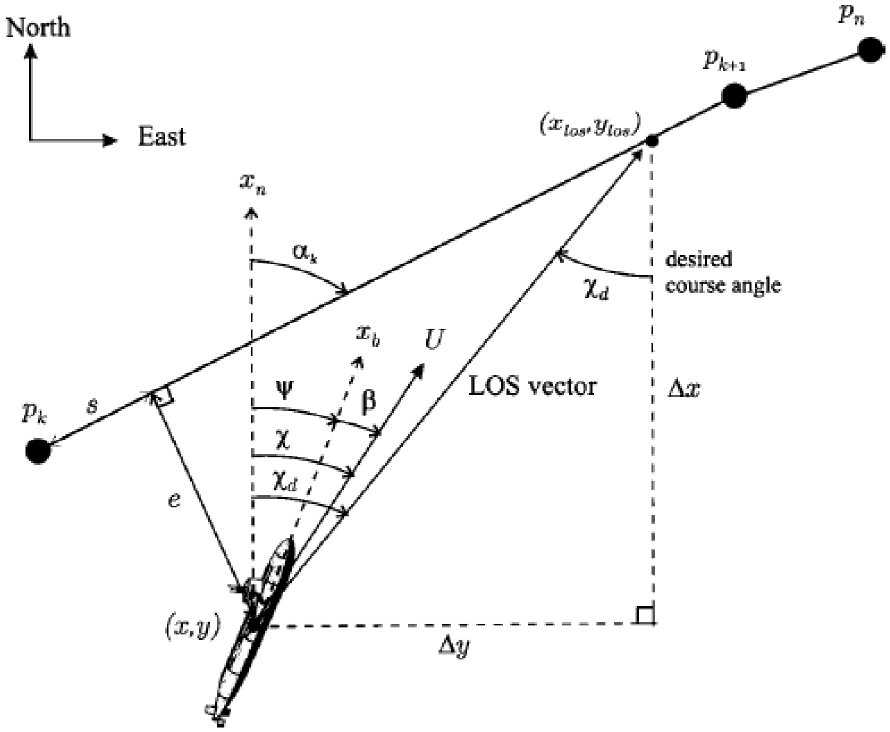


Figure 3.2: LOS steering law, with desired course angle χ_d pointed towards the desired LOS intersection point $x_{\text{LOS}}, y_{\text{LOS}}$. Illustration from [10, p. 259].

where $\mathbf{p}^n(t) = [x, y]$ is the position of the vessel in NED-reference frame, $y_e(t)$ is the cross-track error (normal to the path), and $s(t)$ is the along-track distance (tangential to the path). The cross-track error andalong-track distance illustrated in Figure 3.2.

For path-following the cross-track error should converge to zero, $e(t) \longrightarrow 0$, meaning the vessel have converged to the path.

LOS steering law achieves this by calculating the desired course angle:

$$\chi_d = \chi_p + \chi_r, \tag{3.15}$$

where $\chi_p = \gamma_p$ and the velocity-path relative angle χ_r is:

$$\chi_r := \arctan \frac{-e}{\Delta}. \tag{3.16}$$

The velocity-path relative angle ensure that the velocity is directed towards a point on the path at a certain distance ahead of the marine vessel, called the lookahead distance Δ. $\Delta > 0$ is a tunable parameter, and a rule of thumb suggest to start with $\delta = 3 * L$, where L is the length of the vessel. A small lookahead distance results in more aggressive steering, while a bigger will converge to teh path more slowly.

The overall performance of this type of path-following system will depend on the tunable parameters ,such as Δ, as well as the performance of the motion control systems.

Chapter 4

Design and implementation

Traditionally path-following has been solved using cascaded systems of kinematic and dynamic controllers. For instance, using one system for minimizing the cross-track error and one for reaching a constant forward velocity. The major contribution of this project is a progressive end-to-end DRL-methodology for solving the path-following problem. A global agent/policy has been developed solving the complete path-following problem, from thruster allocation to convergence towards the desired path and forward velocity. This methodology does not need explicit knowledge of the dynamics of the marine vessel, and does not depend on the performance of several subsystems, such as separate motion control and thruster allocation systems.

A DRL-agent will optimize the given objective by finding a policy that solves the Markov decision process (MDP) which models the path-following problem. The design of the MDP includes state space S, action space A, and reward function R, all parts of the environment. Since this project is using a theoretical vessel, the DRL-agent finds a proper policy through interacting with a simulated environment. The interaction between agent and environment is visualized in Figure 4.1. The DRL-algorithm used in this project is DDPG, and is customized to suit the environment in question.

In this chapter, the selected tools will be presented first, before going into the dynamics of the environment, the steps of the methodology to solve the path-following problem and how the DDPG-algorithm was implemented.

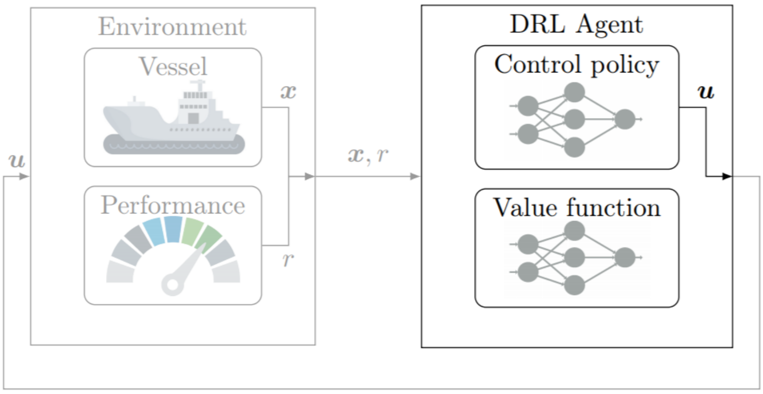


Figure 4.1: Illustration of the interaction between policy (control algorithm) and the vessel. The DRL-algorithm is highlighted above. Illustration from [36]

4.1 Tools

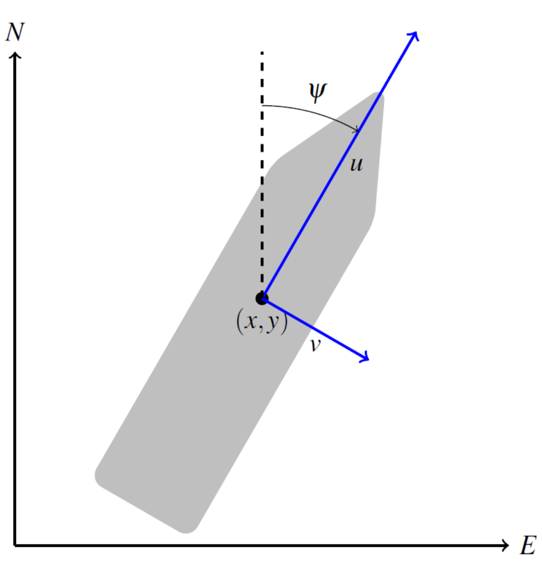
The implementation of the DRL algorithm and environment was implemented using the Python programming language, with a rich collection of machine learning libraries available.

The DDPG implementation in this project was based on implementation from Spinning Up [6]. Spinning Up is an educational resource produced by the organization OpenAI, with the mission to simplify learning of DRL. Their code philosophy is to produce code that is as simple as possible, highly consistent with other associated algorithms and exposing fundamental similarities between them. Due to this philosophy, the following design decisions were made in Spinning UP's implementation, either giving strong support to the original implementation of DDPG by Lillicrap et al.[7] or doing alterations in select areas:

- Including batch normalization, handling larger differences in the magnitude of observations. Batch normalization is included in the original implementation by Lillicrap.
- Changing transformation of action space, to handle action-space with non-zero means. This was implemented to handle the action-space of the marine vessel.
- Update the networks not at the end of the episode, but at each step, as presented in original implementation by Lillicrap et al.[7].

The DDPG-implementation uses the library Tensorflow , to implement function approximators of policy and value functions [2]. Tensorflow is an open source platform for machine learning, providing a library that eases the implementation of backpropagation, batch normalization etc.

Other libraries used are numerical libraries Pandas [5], Numpy [3], and visualization library Matplotlib [4].



(a) A 3-DOF marine vessel in a North-East-Down (NED) referance frame, centered at (x,y), heading ψ, and velocities surge u and sway v. Illustration from [1]

(b) The marine vessel consists of 3 thrusters. 1 and 2 are azimuth thrusters, and 3 is a tunnel thruster. In this project the tunnel thruster is turned off. Illustration from [1]

4.2 Marine vessel and environmental forces simulation

The marine vessel simulator applied in this project, is using a model of a container ship, 76.2 m long and $600 * 10^3$ tonnes dead weight, taken from [1]. The model has 3-DOF, with position vector $\eta = [x, y, \psi]^\top \in \mathbb{R}^2$, and velocity vector $\nu = [u, v, r]^\top \in \mathbb{R}^2$, with Cartesian coordinates (x, y), yaw angle ψ,linear velocities (u,v), and yaw rate r. These states are illustrated on a marine vessel in Figure 4.2a.

The equation of motion for the container ship consists of a rigid-body mass matrix **M**, constant damping matrix **D**, and no external forces is:

$$\dot{\boldsymbol{\eta}} = J_\theta \boldsymbol{\nu}, \tag{4.1a}$$

$$\mathbf{M}\dot{\boldsymbol{\nu}}_r + \mathbf{D}\boldsymbol{\nu}_r = \boldsymbol{\tau}_{\text{control}}. \tag{4.1b}$$

$\boldsymbol{\tau}_{\text{control}}$ is found using the thruster allocation equation (3.11). The vessel has two azimuth thrusters in the aft, and one tunnel thruster, visualized in Figure 4.2b. The tunnel thruster is turned off in this project. The parameters of the vessels are included in appendix .1.

A small current was added to assess the robustness of the path-following agent. The current was modeled as a 2-D irrotational ocean current model (3.9), with $V_c = 0.5$ m/s and $\beta_c = 45°$.

The agent interacts with the environment by setting thruster input $\mathbf{a} = [\alpha_1, \alpha_2, \delta_1, \delta_2]^\top$, and receiving measurements of the new state and reward after acting upon the environment. The new states are calculated by solving the 3-DOF equation of motion, using Euler's method:

$$\boldsymbol{\eta}_{t+1} = \mathrm{R}(\psi_t)\boldsymbol{\nu}_t h + \boldsymbol{\eta}_t, \tag{4.2a}$$

$$\boldsymbol{\nu}_{r,t+1} = (\mathbf{M}^{-1}(\boldsymbol{\tau_t}(\boldsymbol{\alpha_t}, \mathrm{f_t}) - \mathbf{D}\boldsymbol{\nu}_{r,t}))h + \boldsymbol{\nu}_{r,t}, \tag{4.2b}$$

with stepsize $h = 0.1$. Knowing new $\boldsymbol{\eta}$ and $\boldsymbol{\nu}$, the reward and new states can easily be calculated, such as crab angle β, cross track error y_e , angle of path γ_p, etc.

4.3 Path-following using DRL

To make an efficient path-following system for a marine vessel, two objectives need to be met simultaneously:

- Minimization of cross-track error y_e.
- Convergence to desired constant forward-velocity u_d.

In previous works, [34, 36], an end-to-end training for path-following of a marine vessel, using DRL is proposed. This DRL-agent controlled only thruster angle α, with a separate conventional non-AI forward velocity control to ensure a desired forward velocity. The vessel used in this project is fully-actuated, with two azimuth thrusters, with actuator forces $\mathbf{f} = [f_1, f_2]$ and angles $\boldsymbol{\alpha} = [\alpha_1, \alpha_2]$. A different approach is therefore needed also to make sure the vessel will also reach a desired forward velocity u_d. The requirement of holding a positive forward velocity comes from the fact that cross-track error could potentially be minimized through stopping at the desired path or circling the path.

Starting this project, it was first attempted to learn both objectives at the same time, with no prior knowledge. Using this formulation, the agent was not able to find a solution within a reasonable time. This might be caused by the high complexity of the assignment, the need for understanding both dynamics and thruster allocation of the vessel, and difficulty interpreting the reward. Due to this, it was decided to divide (and conquer) the assignment of path-following through transferring the knowledge from one sub-objective to the next.

The respective sub-objectives were divided into learning phases. The corresponding agent trained in learning phase X will be called "LPX:Y agent", where Y is a short abbreviation of the control task. The path-following agent are going to be tested on a spline, consisting of 4 waypoints. The learning phases are:

1. **LP1:Vel - Forward velocity controller**: An agent performing forward-velocity control, converging towards a constant desired forward velocity.
2. **LP2:VelCourse - Forward velocity & course stabilization**: Adding course stabilization, the agents learn how to drive in a straight line, while holding a desired constant forward velocity. This step was designed to find out if the agent was capable of driving in a straight line, a required quality.
3. **LP3:HeadVel - Heading & forward velocity controller**: An agent capable of following both the desired heading and forward velocity. This agent can utilize the knowledge from LP2:VelCourse, as initial weights for policy. Through learning how to control heading, the agent will understand the dynamics and thruster allocation, such that it can change the direction of the vessel.
4. **LP4:Path - Path-following**: An agent performing path-following, minimizing cross-track error and following a desired forward velocity control. This is done by transferring knowledge (policy) from the DRL-agent performing heading & forward-velocity.

The transfer learning performed between the different steps is illustrated in Figure 4.3.

Several of these learning phases consist of several objectives. In order to reach the highest possible expected return, we have to guide the agent to find out how to solve this multi-objective optimization problem. This is performed through the design of action-space, state-space, training scenarios and reward function.

The following sections discuss the state space, action space, design of reward-function, training scenarios, and the use of transfer learning.

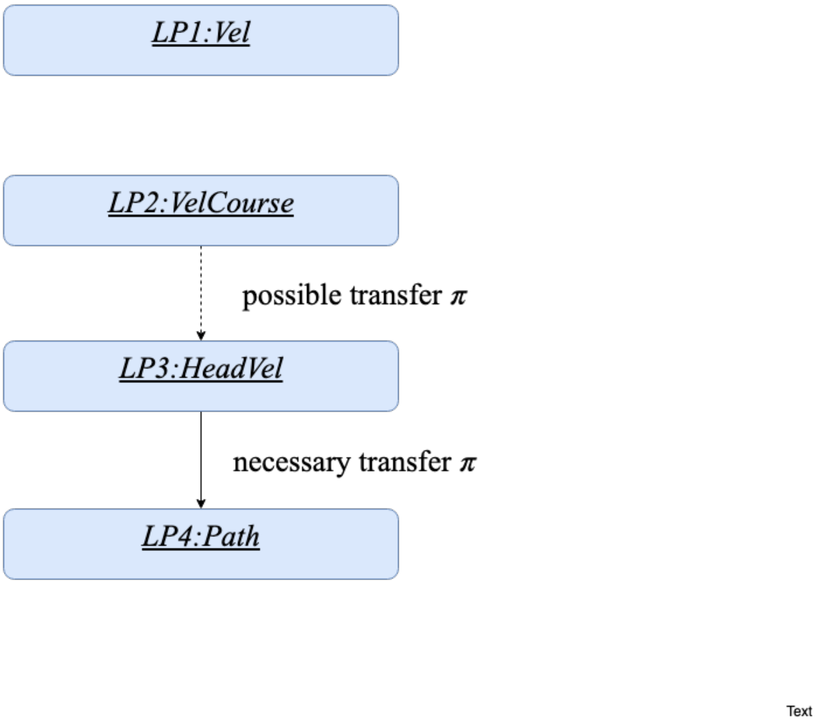


Figure 4.3: Illustration of transfer learning performed between the different steps.

4.3.1 State vector

The observation or state given to a DRL-agent is used to optimize expected return. For path-following, the state vector could simply hold $\boldsymbol{\eta}$ and $\boldsymbol{\nu}$. This would give basic information about the vessel, and could suffice to solve the task at hand. However, this the state vector depends on rotation and translation of the path, and none of them give direct information regarding convergence to the desired path, such as cross-track error. The non-linear properties of the neural network could, in principle, be able to learn transformations to handle these difficulties. However, this could mean longer and less stable training to achieve the desired objective, and might also be needing a larger neural network to achieve the transformations.

Cross-track error (3.14b) and difference between heading and path-tangential angle (3.13), $\tilde{\psi} = \psi - \gamma_p$ are states giving more direct knowledge towards the task of path-following. These are also invariant of the path placement.

The different learning phases have different needs regarding state vector due to different objectives. For this reason, two different state vectors where used, one for LP1-LP2 and one for LP3-LP4.

LP1:Vel and LP2:VelCourse

These learning phases has the smallest state vector being:

$$\mathbf{x} = \left[\tilde{u}, \dot{u}, v, \tilde{v}, r, \chi\right], \tag{4.3}$$

where $\tilde{u} = u - u_d$ gives information about the difference between desired velocity and actual velocity. $\dot{\tilde{u}}$ gives information about how fast the change is happening. Derivatives not given by the euqation of motion (4.1) were found through forward finite difference.

By also giving information about the linear velocities r and v, rotation velocity r, and course χ, the agent gets enough information to understand the vessels dynamics utilized in forward velocity control and course stabilization. This state vector is also invariant to the position of the vehicle.

For LP1:Vel and LP2:VelCourse, there are no need of knowing y_e or $\tilde{\psi}$, since the objectives are not dependent on a desired path.

LP3:HeadVel and LP4:Path

To perform path-following, the state vector was inspired by the state vector $\mathbf{x} = [y_e, \dot{y}_e, \psi - \gamma_p, r, u, v]^\top]$ from [34]. Since the agent does not perform forward velocity control in that paper, the states $\tilde{u}$ and $\dot{\tilde{u}}$ were added, inspired by state vector of the LP1-LP2. The extended state vector is invariant to the position of the path and heading of the vessel.

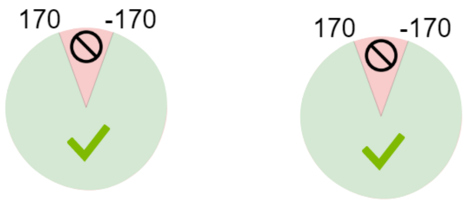
This produces the extended state vector:

$$\mathbf{x} = \left[y_e, \dot{y}_e, \chi - \gamma_p, r, u, v, \tilde{u}, \dot{u}\right]^\top. \tag{4.4}$$

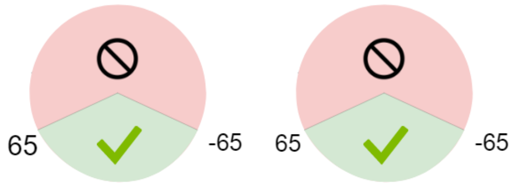
By using cross-track error, instead of the position error in NED-reference frame, the extended state vector is position invariant as well.

4.3.2 Action vector

The model of the container ship uses two actuators, exerting forces $\mathbf{f} = [f_1, f_2]^\top$ and angels $\boldsymbol{\alpha} = [\alpha_1, \alpha_2]^\top$, for steering and propulsion. The original action space was from $\boldsymbol{\alpha} \in [-170, 170]$ degrees and force $\mathbf{f} \in [0, 200]$ kN. Allowing for the use of the full action space, the DRL-agent had a problem finding an acceptable solution within a reasonable



(a) Illustration of action space of actuator angles, from $\boldsymbol{\alpha} \in [-170, 170]$.



(b) Illustration of action space of actuator angles, from $\boldsymbol{\alpha} \in [-65, 65]$.

learning time for LP3:HeadVel and LP4:Path. By narrowing the action space, the agents were able to find better solutions. This can be due to less need for exploration. Consequently, the following action spaces were used in the different learning phases:

1. LP1:Vel : $\boldsymbol{\alpha} \in [-170, 170]$ and force $\mathbf{f} \in [0, 200]$.
2. LP2:VelCourse: $\boldsymbol{\alpha} \in [-170, 170]$ and force $\mathbf{f} \in [0, 200]$.
3. LP3:HeadVel: $\boldsymbol{\alpha} \in [-65, 65]$ and force $\mathbf{f} \in [0, 200]$.
4. LP4:Path: $\boldsymbol{\alpha} \in [-65, 65]$ and force $\mathbf{f} \in [0, 200]$.

A somewhat smaller action space is called upon for the two latter tasks. This is probably due to the higher inherent complexity of the environments dynamics.

4.3.3 Reward function

The reward function gives feedback on the performance of the agent and is essential in reinforcement learning. The agent needs to learn the relation between an action performed in a specific state and the reward received, and thereby how it should behave to receive the highest return. In RL the design of the reward function can be more essential than tuning parameters of the DRL-algorithm.

There are many ways to design the reward function, and several methods may achieve the same result. Most important is giving the DRL-agent sufficient feedback, ensuring desired behavior. If the reward function is unfortunately designed, the agent might opt for local optima.

The development of the reward function for the different learning phases is presented in Figure 4.5. An introduction to the different reward functions for the different sub-objectives is given in the following sections.

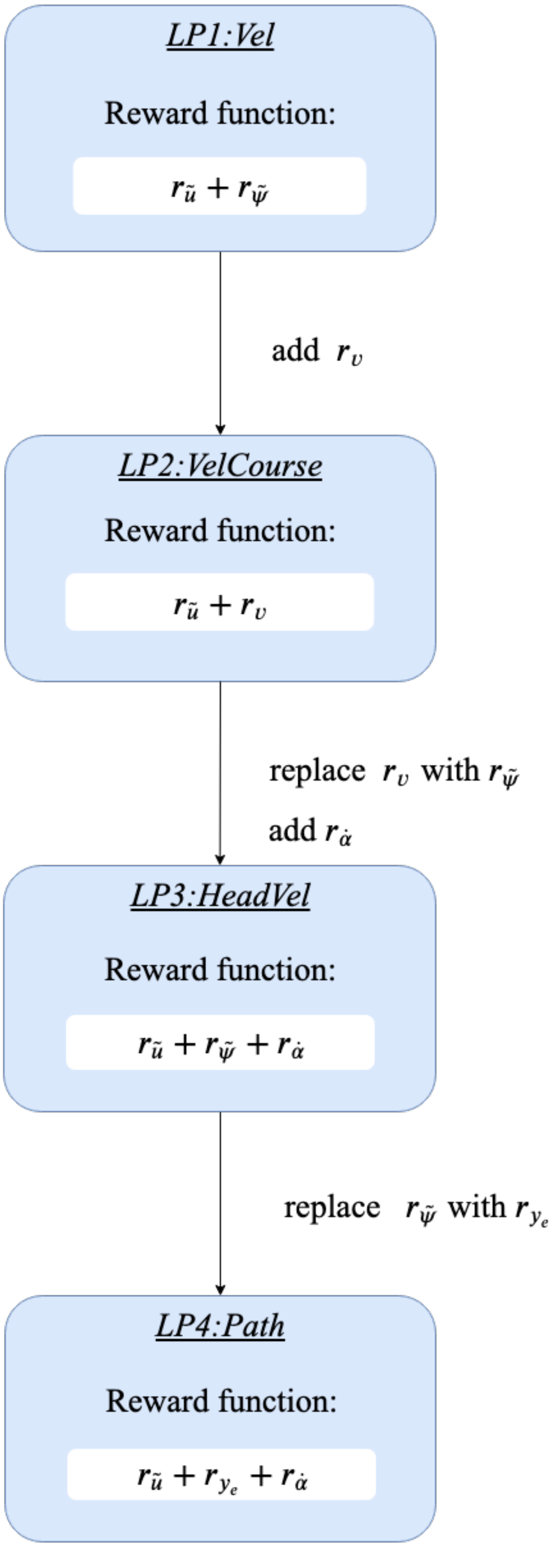

LP1:Vel
Reward function:
$r_{\tilde{u}} + r_{\tilde{\psi}}$
add r_{υ}
LP2:VelCourse
Reward function:
$r_{\tilde{u}} + r_{\upsilon}$
replace r_{υ} with $r_{\tilde{\psi}}$
add $r_{\dot{\alpha}}$
LP3:HeadVel
Reward function:
$r_{\tilde{u}} + r_{\tilde{\psi}} + r_{\dot{\alpha}}$
replace $r_{\tilde{\psi}}$ with r_{y_e}
LP4:Path
Reward function:
$r_{\tilde{u}} + r_{y_e} + r_{\dot{\alpha}}$

Figure 4.5: Development of the reward function for each learning phase.

LP1:Vel

The objective is to reach a desired forward velocity u_d, meaning $\tilde{u} = u - u_d \longrightarrow 0$. To reach this goal a Gaussian distributed reward function is proposed, inspired by [34]. The reward function is:

$$r(\tilde{u}) = e^{\frac{-\tilde{u}^2}{2\sigma_{\tilde{u}}^2}}. \tag{4.5}$$

with $\sigma = 0.5$. This gives a reward function with a maximum reward of 1, and a minimum reward of 0. The reward function is visualized in Figure 4.6.

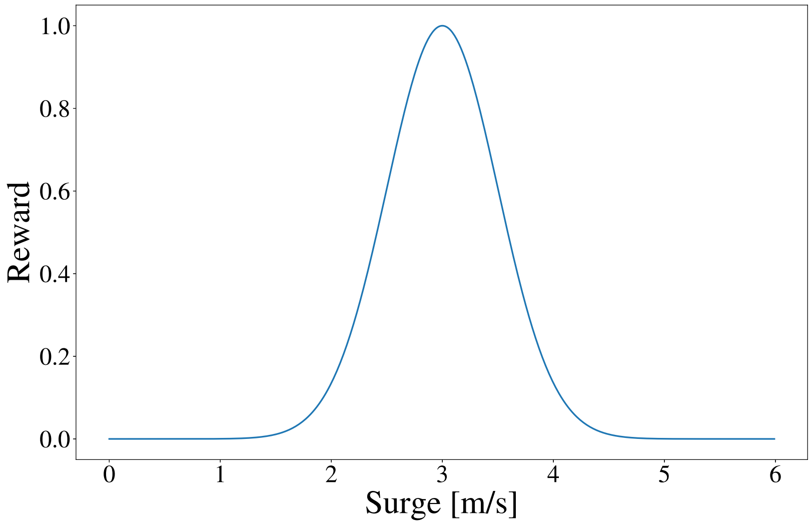


Figure 4.6: Illustration of reward-function for LP1:Vel, with $\sigma = 0.5$ and $u_d = 3$.

LP2:VelCourse

Here two objectives are to be optimized. Course stability means having $v \longrightarrow 0$, giving the two objectives $u \longrightarrow u_d$ and $v \longleftarrow 0$. An additional term was appended to the reward function of forward velocity controller, 4.5, to reach both of the objectives simultaneously:

$$r(\tilde{u}, v) = r_{\tilde{u}} + r_v, \tag{4.6a}$$

$$r_{\tilde{u}} = e^{\frac{-\tilde{u}^2}{2\sigma_{\tilde{u}}^2}}, \tag{4.6b}$$

$$r_v = -(1 - e^{\frac{-v^2}{2\sigma_v^2}}), \tag{4.6c}$$

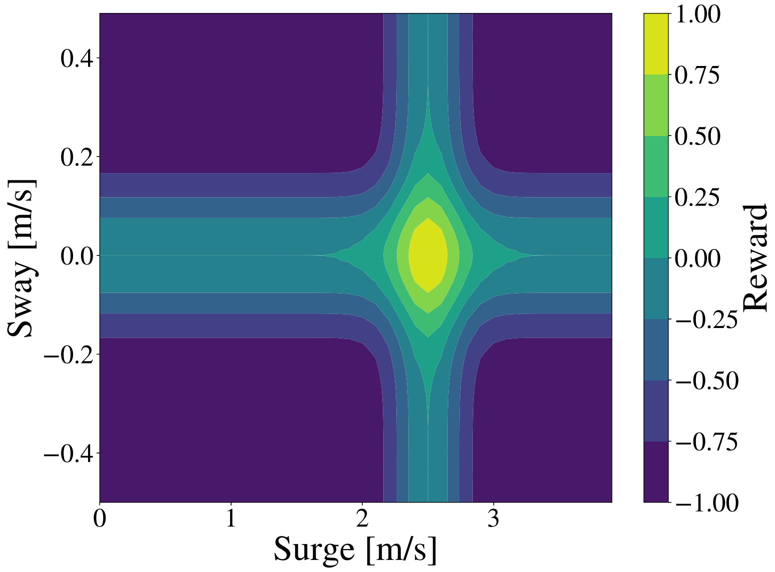


Figure 4.7: Illustration of reward-function for LP2:VelCourse, with $\sigma_v = 0.1$, $\sigma_{\tilde{u}} = 0.2$ and $u_d = 2$.

with $\sigma_{\tilde{u}} = 0.2$ and $\sigma_v = 0.1$. This reward function consists of two components, $r_{\tilde{u}}$ gives points when $u \longrightarrow u_d$, and reward r_v gives penalty when deviating from zero surge.

It was first suggested to use a reward-function with positive rewards when v converged towards 0, but this did not seem to be working. This might be caused by the agent getting ample reward for reaching one objective, and did not care to proceed. This led to the suggested reward function, where the agent does not receive a positive reward unless both objectives regarding forward velocity and cross-track error are solved. The reward will, for instance, be zero if the vessel has too high v, even when following the desired velocity. This way the agent understand that even when $\tilde{u} \longrightarrow 0$, to gain points it also needs $v \longrightarrow 0$. The relation between u, v and reward is visualized in Figure 4.7.

LP3:HeadVel

Here the objective is $\psi \longrightarrow \psi_d$ and $u \longrightarrow u_d$. The reward function is constructed as sum of rewards given to achieve the two objectives, $r_{\tilde{u}}$ and $r_{\tilde{\psi}}$, and a penalty of rough actuator use, $r_{\dot{\alpha}}$, giving:

$$r(\tilde{u}, v, \dot{\boldsymbol{\alpha}}) = r_{\tilde{u}} + r_{\tilde{\psi}} + r_{\dot{\alpha}}, \tag{4.7a}$$

$$r_{\tilde{u}} = \begin{cases} 3e^{\frac{-\tilde{u}^2}{2\sigma^2_{\tilde{u}}}}, & u > 0.1 \\ -3, & \text{otherwise} \end{cases}, \tag{4.7b}$$

$$r_{\tilde{\psi}} = \begin{cases} 3e^{\frac{-\tilde{\psi}^2}{2\sigma^2_{y_e}}}, & |\tilde{\psi}| \leq \frac{\pi}{2} \\ -5, & \text{otherwise} \end{cases}, \tag{4.7c}$$

$$r_{\dot{\boldsymbol{\alpha}}} = -\sum_{i==0}^{2} |\dot{\alpha}_i| c_i. \tag{4.7d}$$

The reward for $\tilde{\psi}$ $r_{\tilde{\psi}}$ consists of a Gaussian function, with $\sigma_{\tilde{\psi}} = 0.1$. When the reward function only consisted of positive rewards when reaching the desired heading, the vessel could end up driving in circles, collecting the reward for each round it passed the correct heading. The reward $r_{\tilde{\psi}}$ was therefore designed to give a large negative reward for each time it deviates more than 90° from the desired heading.

The reward for $\tilde{u}$ $r_{\tilde{u}}$ gives positive points for reaching the desired forward velocity u_d. It consists of a Gaussian function, with $\sigma_{\tilde{u}} = 0.2$, similar as in (4.5) and (4.6). In addition to awarding points when reaching desired forward velocity, it is giving negative reward when the vessel has a velocity below 0.1 m/s. This was implemented so that the total reward would be zero, when $\tilde{\psi} \longrightarrow 0$ but $u < 0.1$. This helped the agent towards opimizing both objectives.

To avoid frequent and large changes in actuator angle, a penalty on in actuator angle rate $\dot{\boldsymbol{\alpha}}$ can be used, giving reward $r_{\dot{\boldsymbol{\alpha}}}$. This penalty is the sum of changes scaled by the value of the difference between the maxima of the angular workspace. The agents did not use the actuators as roughly as actuator angles, som there was no need for penalizing the use of actuator force.

The weight given to each of the objectives is important, as the agent will only receive one scalar reward. The maximum and minimum rewards points given for $r_{\tilde{\psi}}$ are $(-5, 3)$, $(-3, 3)$ for $r_{\tilde{u}}$ and $(-1, 0)$ for $r_{\tilde{u}}$. Higher maximum values were given to reach desired forward velocity and heading, being the main objectives. At the same time, a small penalty is given for rough use of the actuator angles.

The reward function is visualized in Figure 4.8, showing the relationship between ψ, u and total reward.

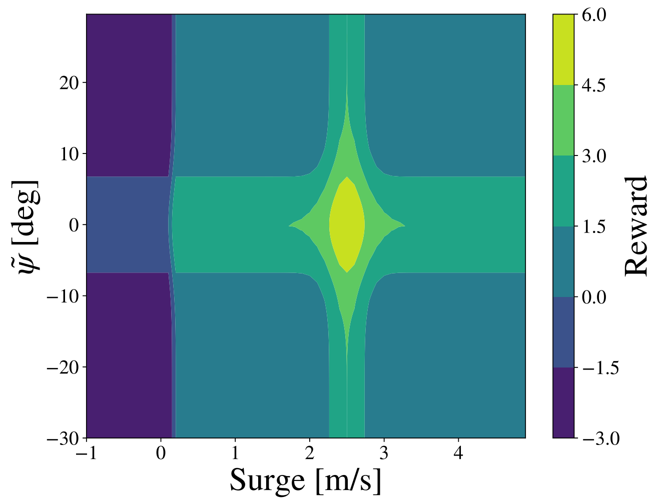


Figure 4.8: Illustration of reward-function for LP3:HeadVel,with $\sigma_{\tilde{\psi}} = 0.1$, $\sigma_{\tilde{u}} = 0.2$,$u_d = 2$ and $\dot{\alpha} = 0$.

LP4:Path

In this task, the two objectives are to minimize cross-track error y_e and follow a desired forward velocity u_d. The reward function for straight path-following was therefore designed with a similar structure as the heading & forward-speed controller, using the same max and min-values of reward-functions, but where $\tilde{\psi}$ is replaced by y_e:

$$r(\tilde{u}, v) = r_{\tilde{u}} + r_{y_e} + r_{\dot{\alpha}}, \tag{4.8a}$$

$$r_{\tilde{u}} = \begin{cases} 3e^{\frac{-\tilde{u}^2}{2\sigma_{\tilde{u}}^2}}, & u > 0.1 \\ -3, & \text{otherwise} \end{cases}, \tag{4.8b}$$

$$r_{y_e} = \begin{cases} 3e^{\frac{-\tilde{\psi}^2}{2\sigma_{y_e}^2}}, & |\tilde{\psi}| \leq \frac{\pi}{2} \\ -5, & \text{otherwise} \end{cases}, \tag{4.8c}$$

$$r_{\dot{\alpha}} = -\sum_{i=0}^{2} |\dot{\alpha}_i| c_i, \tag{4.8d}$$

with $\sigma_{y_e} = 100$ and $\sigma_{\tilde{u}} = 0.2$. The reward function is visualized in Figure 4.9, showing the relationship between y_e, u and total reward.

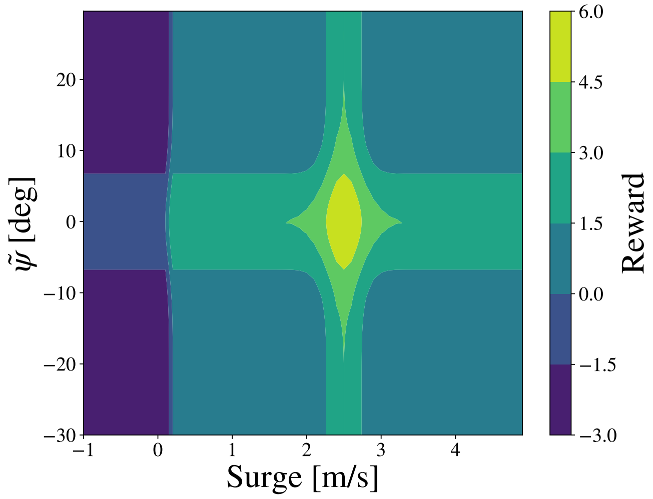


Figure 4.9: Illustration of reward-function for LP4:Path,with $\sigma_{y_e} = 100$, $\sigma_{\tilde{u}} = 0.2$, $u_d = 2.5$ and $\dot{\alpha} = 0$.

4.3.4 Transfer learning

Transfer learning was utilized to help the agent solve the complex assignment of path-following, satisfying requirements of forward velocity and path convergence simultaneously. Transfer learning was performed by initializing the parameters in the function approximations of policy π, from the agent of the previous learning phas. To perform transfer learning this way without mapping, the agents need to have the same state space S and action space A.

Transfer learning is performed in several training phases:

- The policy π from LP2:VelCourse was used as initialization for LP3:HeadVel.
- The policy π for LP3:HeadVel was used for initialization of LP4:Path.

These transfers are also visualized in Figure 4.3.

It was necessary to train a LP2:VelCourse with the same state and action space as in LP3:HeadVel. This was due to the fact that LP2:VelCourse was originally designed with other state-vector.

Transferring only the policy π and not the value-functions is due to the fact that the rewards are given quite differently. Therefore only the policy was transferred, giving knowledge of how to behave, without giving knowledge of how to evaluate good behavior via the policy. However, by transferring the policy of the previous learning phase provides a good starting point for the agent, that gives faster learning of state-action pair with high return.

4.3.5 Training scenario

Various training scenarios were required to train the different agents. Different scenarios provide the agent with the opportunity to explore the dynamics of the physical system (or simulator) and actions. The right quantity of exploration and exploitation helps the agent find the policy giving the highest expected return. For all training scenarios, the episodes are initialized randomly from a set of valid states, summarized in Table 4.1:

Table 4.1: Valid initialization states of an episode for specific variables for the different learning phases

Variable	**LP1:Vel and LP2:VelCourse**	**LP2:VelCourse- extended state-vector, LP3:HeadVel and LP4:Path**
x [m]	0	500 m radius from path
y [m]	0	$(0, 10000)$
ψ [deg]	0	$(-45, 45)$
u [m/s]	$(0, 5)$	$(0, 2)$
v [m/s]	$(-0.3, 0.3)$	$(-0.1, 0.1)$
r [m/s]	0	0
u_d [m/s]	$(2, 3)$	2.5

LP1:Vel and LP2:VelCourse

Here the vessel was started with random linear velocities, while fixed starting positions were given, as summarized in Table 4.1. The agent learned using a given constant desired forward velocity for each episode, randomly selected at the start of each episode within a valid set of values. The notion behind varying u_d was to let the agent gain a better understanding of the dynamics of the vessel regarding the control of forward velocity.

LP3:HeadVel and LP4:Path

It was not merely enough to train the agent on a straight path with different initializations for each episode. The agent was confirmed not to be able to follow a spline of two interconnected straight lines forming a wedge well. Using a path formed as a sinusoidal, the agent learned better how to follow the desired heading. An example of development in γ_p when path formed as a sinusoidal, is shown in Figure 4.10. This improvement might be caused by more exploration of the dynamics of the vessel and thruster allocation. It was subsequently confirmed that the agent was more capable of following a spline after training on a sinusoidal path.

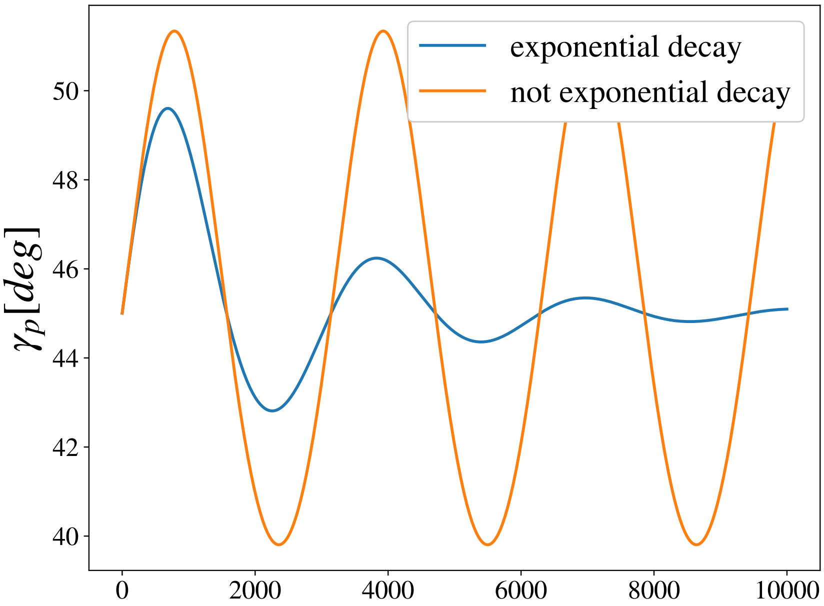


Figure 4.10: Development in γ_p, for sinusoidal path, with either constant amplitude, or exponentially decreasing amplitude.

It was necessary to train both LP3:HeadVel and LP4:Path on a sinusoidal path, even when using transfer learning. This agent seemed to "forget" the dynamics of how to control the direction of the vessel when training the LP4:Path agent on a straight path.

During testing, it was found that with an exponentially decreasing amplitude, the agent improved at following a straight path during testing. This might be caused by the agent gaining better knowledge of the dynamics of the vessel, given the opportunity to both explore sinusoidal and nearly straight path, given the exponentially decreasing sinusoidally shaped path. Examples of an exponentially decreasing path is shown in Figure 4.10.

The specification of the initialization is shown in Table 4.1. The vessel was initialized within a 500-meter radius of the sinusoidal path, with a 45-degree angle. The period was equal to a traveling distance of 500m and an amplitude of 100 meters. The linear velocities had a random initialization within a valid initialization set. This were somewhat smaller compared to the training of the forward velocity controller to further simplify the training-problem.

A random u_d was used at the start of each episode when training LP2:VelCourse agent with extended state vector. This was for the same reason as described earlier. Otherwise, the

Table 4.2: Parameters used for state augmentation.

State	k_i	**Anti-windup limits**
U	0.1	0.1 m/s
v	0.1	0.01 m/s
ψ	0.1	0.1

agent was trained with the same scenarios as LP3:HeadVel and LP4:Path. A different training scenario compared to when it has a smaller state vector is the needed to explore more states.

4.3.6 Steady-state-error compensation

The Gaussian reward function can lead to steady state error (SSE), as discussed in Martinsen et al. [34]. They proposed to estimate the SSE error through integral action of the error, and then compensate for it by augmenting the state accordingly. The SSE e_{ss} can be estimated as:

$$e_{ss,t+1} \longleftarrow e_{ss,t} + k_i h e_t, \tag{4.9}$$

where $e_{ss,0} = 0$ and k_i describes the integration rate. The factor k_i is a constant and needs to be tuned so that the estimator is slower than the system to avoid instability, meaning $k_i \in (0, 1]$. The steady-state compensation is not used during training, as e_{ss} is dependent on previous states (not only $e_{ss,t-1}$), making it a nonstatic Markov process.

To handle SSE, the state is augmented so that:

$$x = x + e_{ss}. \tag{4.10}$$

Anti-windup was also implemented to avoid overshoots. The anti-windup was implemented using saturation as follow:

$$e_{ss,t} = \begin{cases} e_{ss,\max} & , e_{ss,t} > e_{ss,\max} \\ e_{ss,\min} & , e_{ss,t} < e_{ss,\min} \\ e_{ss,t} & , \text{otherwise} \end{cases} . \tag{4.11}$$

Several agents used augmented states to minimize SSE, for instance $\tilde{u}$, v and $\tilde{\psi}$, using parameters represented in Table 4.2.

4.4 DDPG-algorithm

The selected DRL-algorithm is deep deterministic policy gradient (DDPG). There are numerous advantages of using this algorithm since it is:

- Model-free.
- Off-policy.
- Online.
- Handles continous state/action space.
- Applied before to control of marine vessel,[34, 36].

By being off-policy, the agent can learn from experience, performed by a different policy. This can be an important factor when applying DRL to real life marine vessels, as the agent might not be able to explore directly (at least not all the time) due to safety concerns. It can then be useful to use examples from a simulator,or by delving into real life experiences.

The policy $\pi(\mathbf{x})$ is responsible for generating a control action, meaning generating $\mathbf{a} = [f_1, f_2, \alpha_1, \alpha_2]^\top$. The value function $Q(x, u)$ estimates the cumulative discounted reward, for taking action $\Game$ in state $\mathbf{x}$.

The DDPG algorithm used in this project was implemented based on the original implementation of Lillicrap et. al [7], using the same set of hyperparameters and architecture as presented in the paper. Pseudocode of the algorithm is presented in algorithm 3.

The policy and value-function are approximated using neural networks, with the same architecture of the hidden layers. There are 2 hidden layers, consisting of 400 and 300 hidden units, respectively. The activation function between the hidden layers is ReLU, (2.1b), with batch normalization between each layer.

The policy network is given the state $\mathbf{x}$ as input the state, and the output is action $\mathbf{a} \in \mathbb{R}^4$. The activation function of the output layer is a hyperbolic tangent-function, (2.1c), with values in range $(-1, 1)$. This value needs to be scaled before being applied to the marine vessel. This leads to the following logic of the neural network:

$$
\begin{aligned}
\mathbf{h}_1 &= \mathrm{ReLU}(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1),\\
\mathbf{h}_2 &= \mathrm{ReLU}(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2),\\
\pi(\mathbf{x}) &= \tanh(\mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3)\mathbf{u}_{\mathrm{scale}} + \mathbf{u}_{\mathrm{mean}},
\end{aligned}
$$

with $\mathbf{h}_i$ representing the output from hidden layer i, the trainable parameters weight matrix $\mathbf{W}_i$ and bias vector $\mathbf{b}_i$.

The value-function has both $\mathbf{x}$ and action $\mathbf{a}$ as input, and output is a scalar. In Lillicraps implementation, the action was added as an input to the second hidden layer. During experimentation, it did not seem to matter if it was added at the input layer or at the second hidden layer. The output-layer has no activation, due to the nature of approximating a scalar value. This leads to the following logic of the neural network:

$$
\begin{aligned}
\mathbf{h}_1 &= \mathrm{ReLU}(\mathbf{W}_1(\mathbf{x} + \mathbf{a}) + \mathbf{b}_1),\\
\mathbf{h}_2 &= \mathrm{ReLU}(\mathbf{W}_2\mathbf{h}_1 + \mathbf{b}_2),\\
Q(\mathbf{x}, \mathbf{a}) &= \mathbf{W}_3\mathbf{h}_2 + \mathbf{b}_3,
\end{aligned}
$$

with $\mathbf{h}_i$ representing the output from hidden layer i, with the trainable parameters weight matrix $\mathbf{W}_i$ and bias vector $\mathbf{b}_i$.

Compared to the approach by Lillicrap et al. [7], the networks are trained using ADAM [60] optimizer, instead of SGD due Spinning Up using Adam. The parameters used relevant to the performance of the training is:

- Batch size: 64
- Replay buffer size: 10^6.
- Actor learning rate: 10^{-4}.
- Critic learning rate: 10^{-3}.
- Discount rate $\gamma = 0.99$.
- Target network update rate $\tau = 10^{-3}$.

The environment does not have a terminal case, for either learning phase. The DDPG-algorithm implemented from Spinning Up came with the option of a maximum episode length, which was set to 1000 timesteps. This means that all the episodes had a length of 1000 timesteps.

Chapter 5

Results and discussion

The main results of the DRL-agent performing the four learning phases are presented and discussed in this chapter. The four learning phases are:

1. **LP1:Vel**: Forward velocity controller
2. **LP2:VelCourse**: Forward velocity & course stabilization
3. **LP3:HeadVel**: Heading & forward velocity controller
4. **LP4:Path**: Path-following

The agents were tested on different initial states $\boldsymbol{\nu}$ and $\boldsymbol{\eta}$. For each learning phase, one scenario is illustrated. These scenarios are presented using time series plots and position plots:

- **Time series plots**: Showing actions performed at a given timestep, the next state, and the reward.
  - The state plots can include surge u, sway v, yaw ψ, yaw rate r and cross-track error y_e.
  - The reward plots include both total reward at a given time, and its respective components, if several objectives are optimized.
  - The actuator plots include actuator angles $[\alpha_1, \alpha_2]^\top$ and force $[f_1, f_2]^\top$.
- **Position plots** consist of the path of the vessel, given in NED referance frame.

These plots also include desired values, e.g., u_d, ψ_d, and highest possible reward.

The different training scenarios are compared based on:

- The cumulative rewards of a training scenario
- Mean absolute error of the objectives of the training scenario

5.1 Forward velocity control and course stability

State plots of the LP1:Vel agent performing a test scenario are presented in Figure 5.1, with initial states $\boldsymbol{\eta} = \mathbf{0}$ and $\boldsymbol{\nu} = \mathbf{0}$. The rewards received are visualized in Figure 5.3. In this case, the forward velocity converges towards the desired forward velocity $u_d = 3$ m/s, and the reward signal is close to the maximum value. The agent achieved this by driving in a circle, which is an easy arguably but surprising way to gain a constant forward velocity.

To force the agent to drive straight forward, a penalty on sway v was enforced. State plot of the LP2:VelCourse agent performing a test scenario is visualized with state plots in Figure 5.2 and reward plots in Figure 5.3. This test scenario started with the same initial states as the test scenario for LP1:Vel.

The testing scenario for both LP1:Vel and LP2:VelCourse agent shows the development of states as expected from traditional control theory. However, for both LP1:Vel and LP2:VelCourse, a steady state error is present, even with reward close to maximum. The Gaussian reward function might cause the steady-state error, where solutions close to the optimum ($u \rightarrow u_d$ and $v \rightarrow v_d$) give close to maximum reward. This is visualized at 200 s where it is a steady-state error, but the agent still receives almost maximum rewards, for state plots in Figure 5.3 and 5.2, and reward plots in Figure 5.4 5.3.

With a smaller σ of the Gaussian reward function, there is a smaller variance for giving higher rewards. A smaller σ could theoretically lead to smaller steady-state error. We experimented with different values of σ for all the learning phases agents, decreasing by 30 - 50 %. The reduction often leads to agents having problems finding solutions within 500 episodes. More experimentation could have been performed to decrease the values, but by decreasing it with 5 - 10 %, it would arguably not effect finding solutions, but would most likely not improve the steady-state error significantly.

The LP2:VelCourse agent was able to find good solutions with a lower $\sigma_{\tilde{u}}$ than LP1:Vel agent, but the steady-state error of $\tilde{u}$ is quite similar for the two agents. This shows that lower values of $\sigma_{\tilde{u}}$ do not necessarily yield better solutions. The reason for LP2:VelCourse being able to have lower σ might be due to receiving a penalty of movement of sway v, meaning the agent still receives non-zero feedback still in several cases.

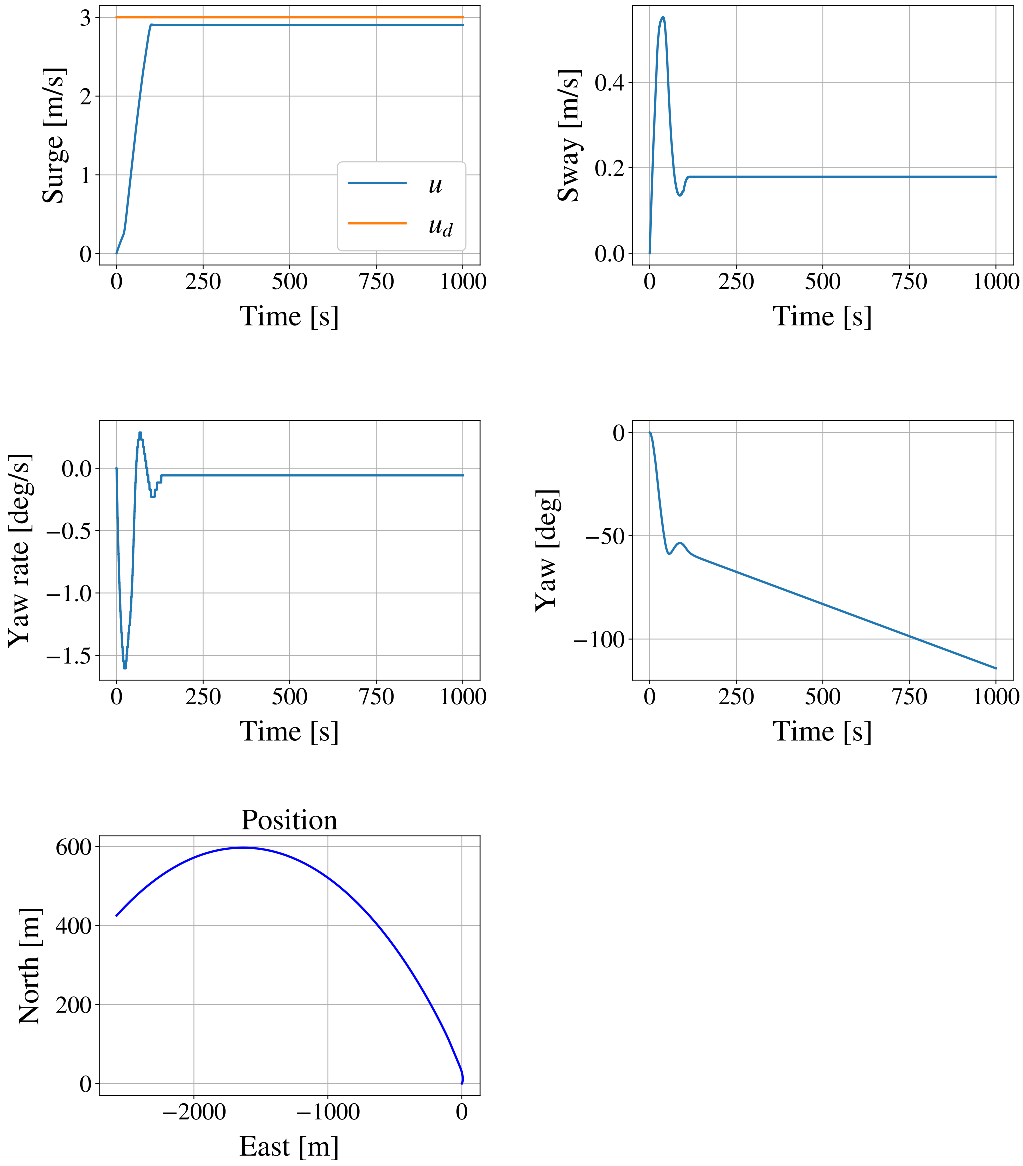


Figure 5.1: State plots for an LP1:Vel agent, without compensation of steady-state error. Test scenarios start with initial states $\boldsymbol{\eta} = \mathbf{0}$ and $\boldsymbol{\nu} = \mathbf{0}$.

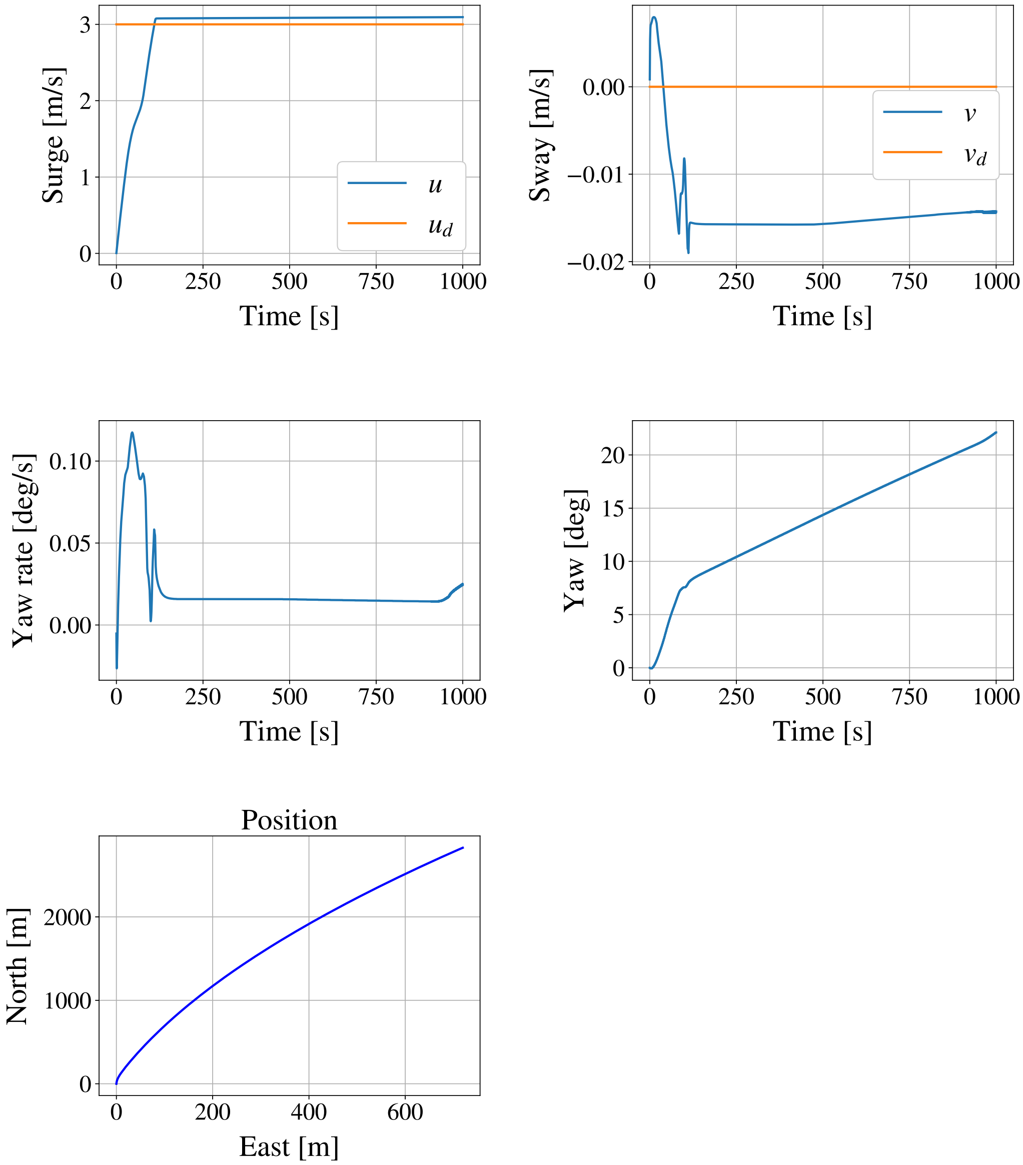


Figure 5.2: State plots for an LP2:CourseVel agent, without compensation of steady-state error. Test scenario starts with initial states $\boldsymbol{\eta} = \mathbf{0}$ and $\boldsymbol{\nu} = \mathbf{0}$.

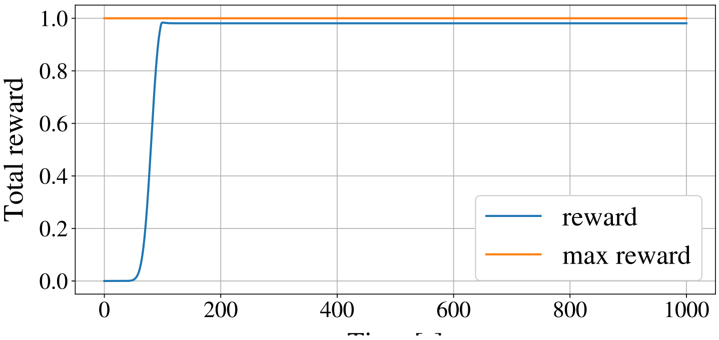


Figure 5.3: Reward plots for an LP1:Vel agent, without compensation of steady-state error.

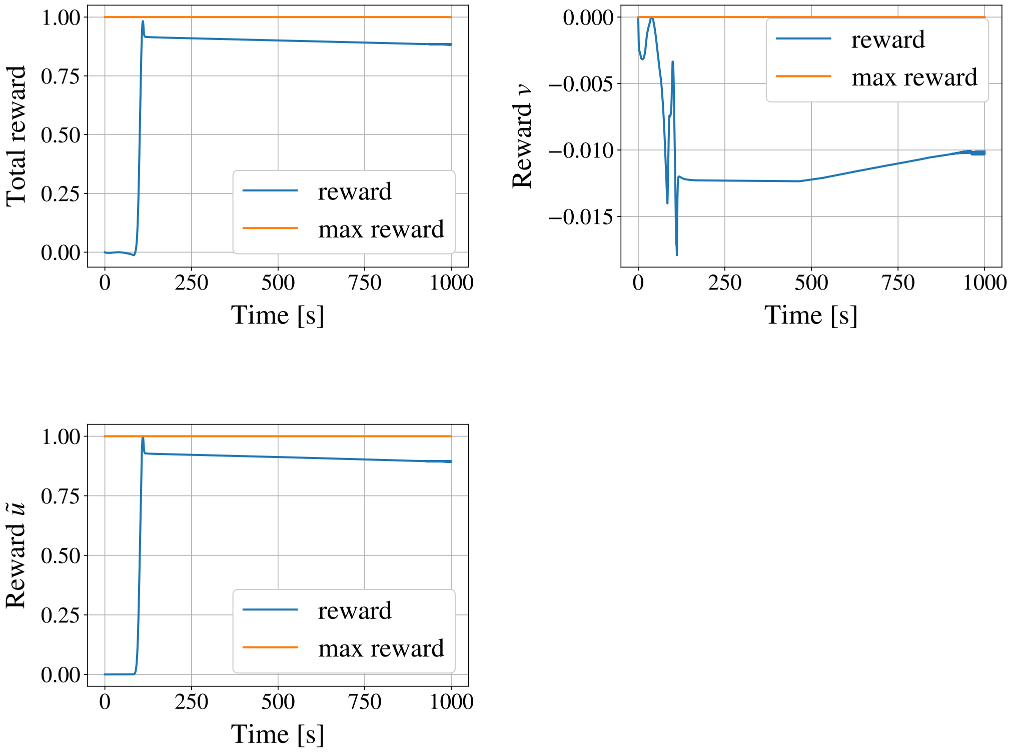


Figure 5.4: Reward plots for an LP2:CourseVel agent, without steady-state error compensation.

The steady state error of v and $\tilde{u}$ was handled by augmenting the states, according to (4.9). A state plot of LP2:VelCourse agent performing the same test scenario with steady-state compensation is shown in Figure 5.6. The figure shows that the steady-state compensation helped on the convergence of v towards 0, without significant impact on u.

Interestingly enough, steady-state compensation of v and $\tilde{u}$ worked for LP2:VelCourse agent with extended state vector. Stateplots of this agent is illustrated in Figure 5.5, showing $u \rightarrow u_d$ and $v \rightarrow 0$, for the same test scenario. One thing to notice here is that this specific agent has prioritized control over forward-speed rather than v, compared to Figure 5.5, meaning that the control law might be more susceptive to changes in $\tilde{u}$. The reward plot for LP2:VelCourse agent with extended state vector and steady-state compensation is shown in Figure 5.7. From the figure we can see the agents receiving close to maximum rewards for both $\tilde{u}$ and v.

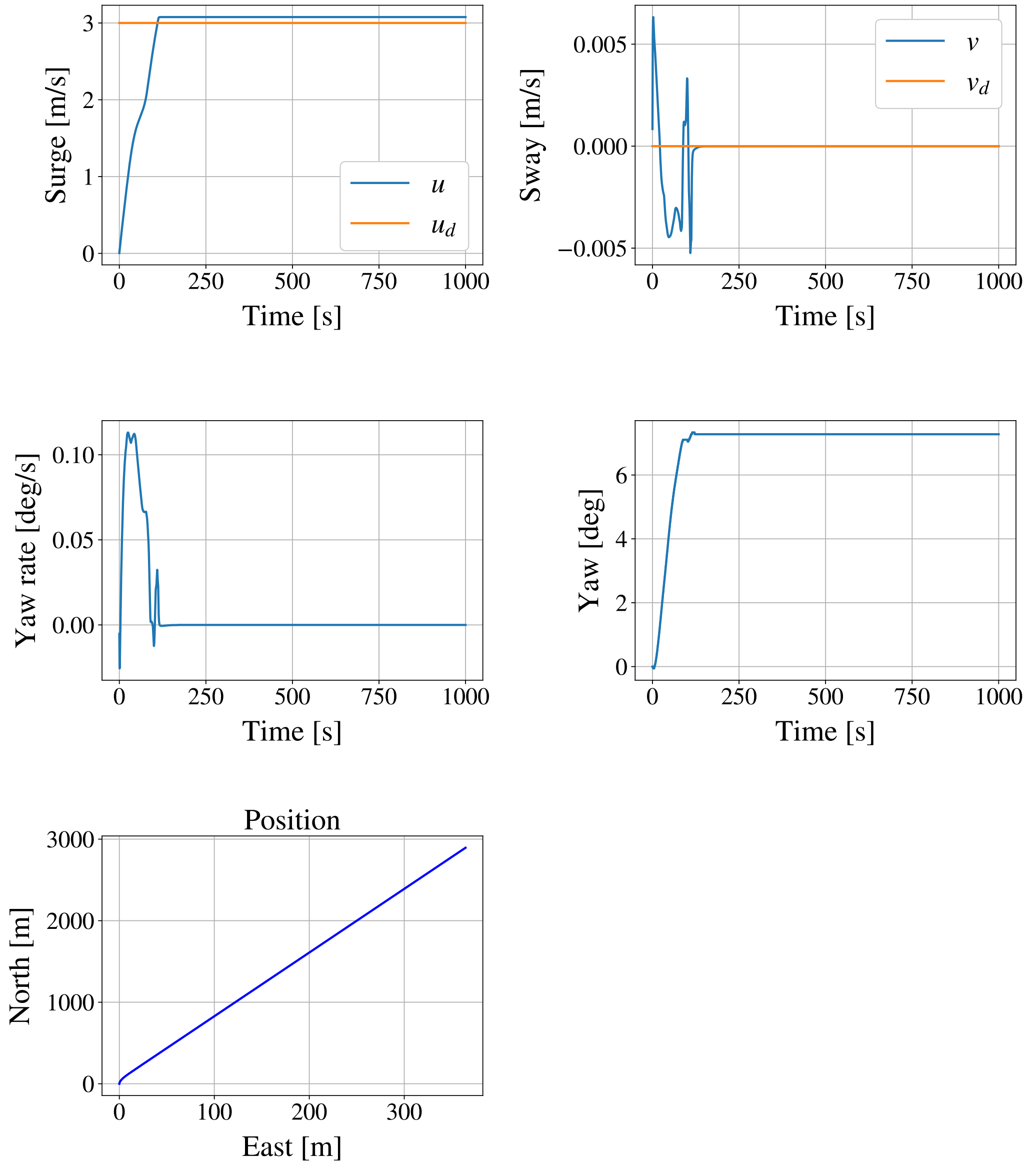


Figure 5.5: State plots for an LP2:VelCourse agent, with steady-state error compensation. Test scenario starts with initial states $\boldsymbol{\eta} = \mathbf{0}$ and $\boldsymbol{\nu} = \mathbf{0}$.

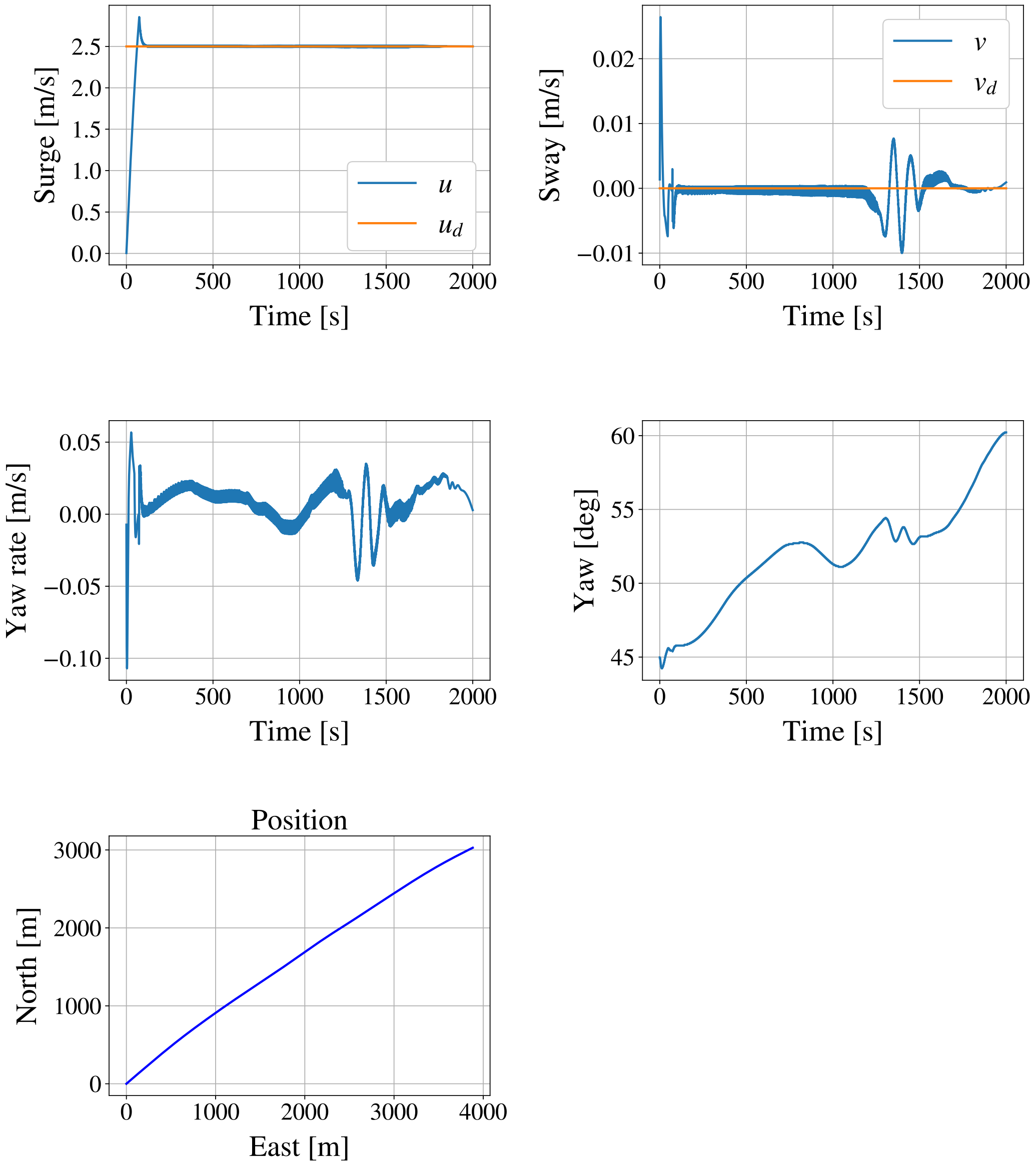


Figure 5.6: State plots for an LP2:VelCourse agent, with steady-state error compensation and extended state-vector. Test scenario starts with initial states $\boldsymbol{\eta} = \mathbf{0}$ and $\boldsymbol{\nu} = \mathbf{0}$.

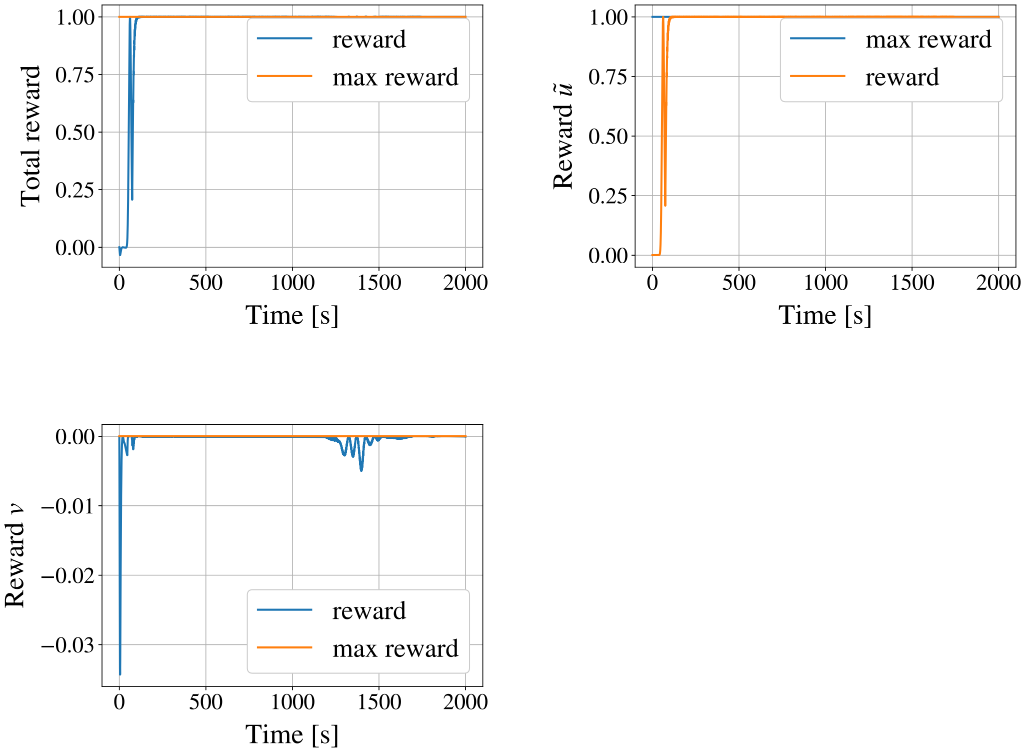


Figure 5.7: Reward plots for an LP2:VelCourse agent, with steady-state error compensation and extended state-vector.

The difference in behavior regarding steady-state error compensation might come from:

- The vessel is not being as sensitive for small changes in $\tilde{u}$ or v. This might be due to the challenges of multi-objective optimization, where agents receive a weighted sum of rewards from several objectives. The two objectives are "struggling" against each other, making it harder to achieve optimal control of both objectives simultaneously. Due to the random nature of exploration in DDPG, the agent might reach different priorities of whether to minimize v or $\tilde{u}$, as they are equally weighted the same.
- This steady-state compensation is inspired by the traditional integral control. Given the nature of neural networks, the agent might employ different reasoning to gain the result that minimizes the state $\tilde{u}$ and v. This untraditional reasoning might not cooperate well with steady-state compensation, or even counteract it.
- There are difference in elements of the state vector. The two agents underwent therefore, different test scenarios. The agent could, therefore, find different relations, leading to different performances.
- A combination of the factors mentioned.
- None of the above. It could simply be a problem of tuning for steady-state correction, k_i, and limits of anti-windup or the agent could eventually achieve better behavior if given the opportunity of longer training.

All the agents presented in this chapter was capable of achieving relatively good solutions for all the scenario presented. The testing of the LP2:VelCourse agent with extended state vector and steady-state compensation, is showed in Table 5.1. The table gives the mean absolute of $\tilde{u}$ and v, and the cumulative reward of 20000 timesteps. The results show that the LP2:VelCourse agent was capable of achieving favorableresults for all the test scenarios, converging towards desired forward velocity and sway.

Table 5.1: The results from LP2:VelCourse agent with extended state vector and steady-state compensation on test scenarios. The test scenarios are run for 20000 timesteps, $u_d = 3$, and have different inital values of $\boldsymbol{\eta}$ and $\boldsymbol{\nu}$.

Initial state: **(E,N,ψ,u,v,r)** **[m,m,rad,m/s,m/s,rad/s]**	**Mean absolute** $\tilde{u}$	**Mean absolute** v **[m/s]**	**Cumulative reward**
(0,0,$\pi/4$,0,0,0)	0.04229	0.00115	19350
(100,500,0,0,0,0)	0.05942	0.00078	19176
(100,500,0,5,0,0)	0.03902	0.00211	19397
(100,500,0,2,0.1,0)	0.00819	0.00078	19809
(100,500,0,2,0.5,0)	0.01210	0.00285	19530
(100,500,$\pi/5$,2,0.5,0)	0.01206	0.00388	19519
(100,500,$\pi/5$,2,0.5,0.1)	0.17423	0.02179	11837
(100,500,$\pi/5$,2,0.1,0.5)	0.01336	0.00183	19617

5.2 Heading and forward velocity control

The LP3:HeadVel agent was trained on a sinusoidal curve of ψ_d, with desired forward velocity $u_d = 2.5$. Steady-state compensation was performed on $\tilde{\psi}$ and $\tilde{u}$, according to (4.9). An test scenario performed by the LP3:HeadVel agent is presented in Figure 5.8 with corresponding reward plot in Figure 5.9, with initial states $\boldsymbol{\eta} = [0, 0, \pi/4]^{\top}$ and $\boldsymbol{\nu} = \mathbf{0}$. The agent is capable of following the sinusoidally-shaped desired heading ψ_d relatively well, while the forward velocity does not converge as good towards the desired forward velocity. The reward function in LP3:HeadVel consisted of two Gaussian functions, with equal weighting. From the resulting plot, the received rewards from the two objectives are similar, prioritizing those two objectives fairly.

Better convergence with regard to heading control might be due to more efficient exploration of the dynamics regarding heading, due to the sinusoidally-shaped path. An attempt was made to improve the tracking of forward velocity, by decreasing value of $\sigma_{\tilde{u}}$, tuning of steady-state compensation and stimulating exploration by varying u_d both within episodes and between episodes. These suggestions did not improve this and could be further researched. We see that the agent can solve both objectives quite well, considering that it solves both control and thruster allocation. It also converges relatively rapidly towards the desired values, even though it has some problems reaching the desired forward velocity.

The LP3:HeadVel agent trained on a sinusoidal path was tested on a constant ψ_d. Several of the states continued to have sinusoidal tendencies even when it was not needed. The training scenario was changed to a sinusoidal path, with exponentially decreasing amplitude.

This bridges, to a certain extent, the differences between a sinusoidal and a straight path, by the use of a diminishing sinusoidal curve. A test scenario performed by the LP3:HeadVel agent is shown in the state plots in Figure 5.10 and corresponding reward plot in Figure 5.11. The agent trained on an exponentially decreasing sinusoidal curve was also tested on a constant ψ_d, and showed less sinusoidal tendencies but is still capable of following the desired heading.

The LP3:HeadVel with steady state compensation was also tested on other scenarios, some of which are presented in Table 5.2. The agent worked well in all scenarios tested, being able to converge towards the desired heading and forward velocity. The difference in mean absolute error mainly comes from different initialization.

Table 5.2: The results from LP3:Head agent with exponentially decreasing sinusoidal ψ_d on test scenarios. The test scenarios are run for 30000 timesteps, $u_d = 2.5$, and have different initialization of $\boldsymbol{\eta}$ and $\boldsymbol{\nu}$.

Initial state: (E,N,ψ,u,v,r) [m,m,rad,m/s,m/s,rad/s]	**Mean absolute $\tilde{u}$ [m/s]**	**Mean absolute $\tilde{\psi}$ [deg]**	**Cumulative reward**
(0,0,$\pi/4$,0,0,0)	0.307	4.950	204640
(-100,0,$\pi/2$,0,0,0)	0.307	6.069	203410
(100,0,-$\pi/2$,0,0,0)	0.311	4.978	201738
(100,100,0,3,0,0)	0.276	4.627	205905
(200,400,$\pi/2$,5,0.1,0)	0.299	6.374	200510
(200,400,3$\pi/2$,5,0.1,0)	0.291	7.267	199723
(200,-400,-3$\pi/2$,5,0.1,0)	0.285	5.053	206757

To both see the performance of the LP3:HeadVel agent on a straight line, and to assess the generalizability of the agent, the agent was tested on a spline with the current present. The desired heading $\psi_d = \arctan -e/delta + \gamma_p$ was calculated using LOS steering law to converge towards the path. Δ was set to $3L$, with L being the length of the vessel. The state plot from following a spline is shown in Figure 5.12, and demonstrates that LP3:HeadVel is quite capable of following a straight path when given $\psi_d = \arctan -e/delta + \gamma_p$. From this, it seems as if the agent was capable of generalizing the knowledge of heading control.

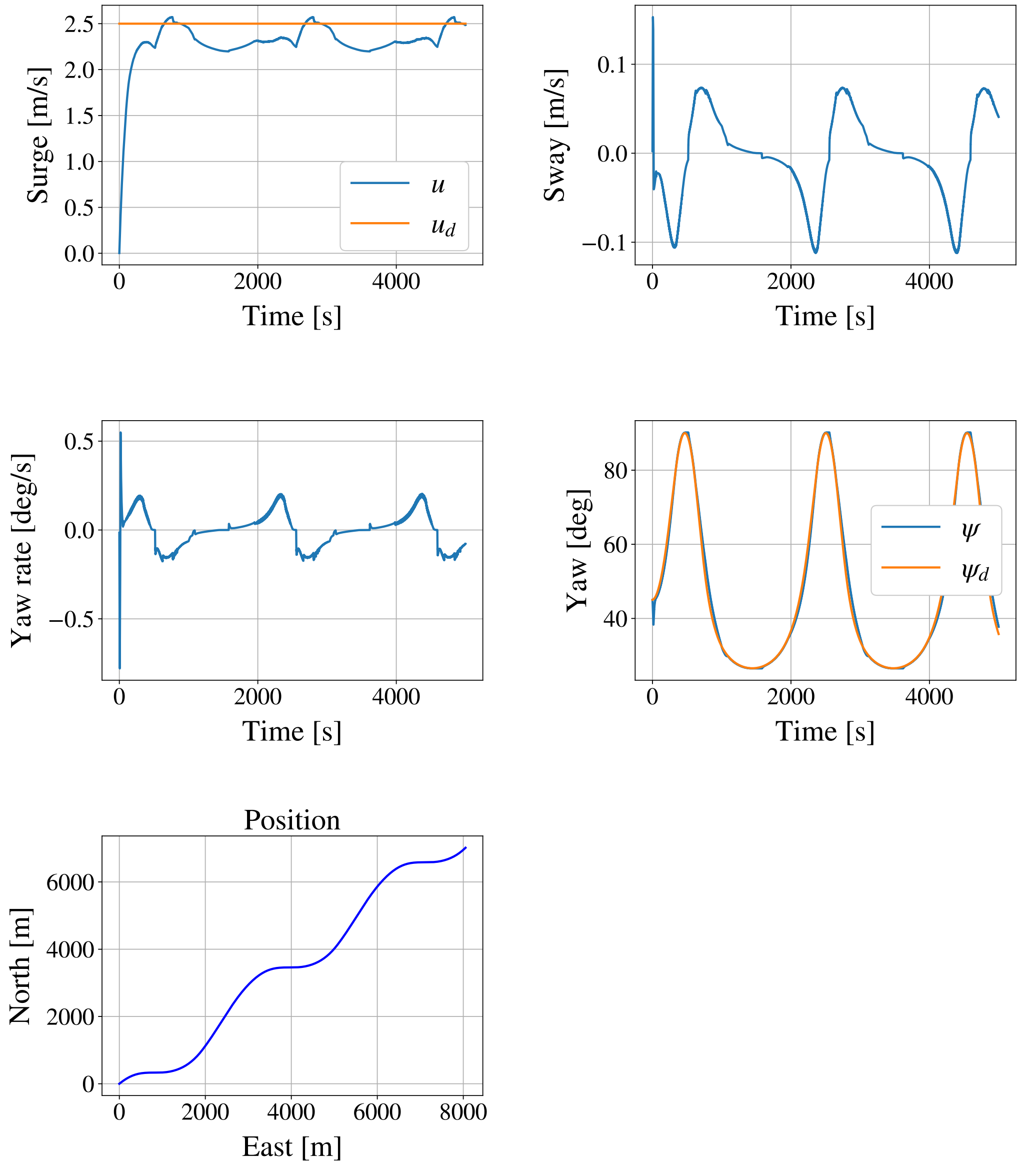


Figure 5.8: State plots of an LP3:HeadVel agent following a sinusoidal path, with compensation of steady-state error. Test scenario starts with initial states $\boldsymbol{\eta} = [0, 0, \pi/4]^\intercal$ and $\boldsymbol{\nu} = \mathbf{0}$.

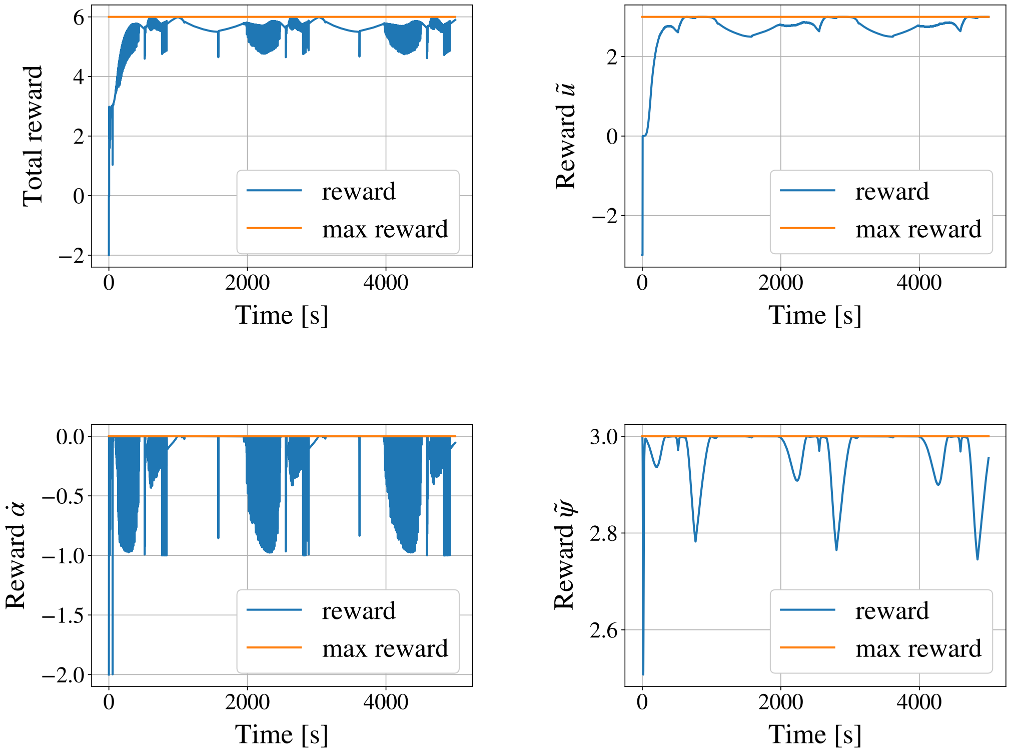


Figure 5.9: Reward plots of an LP3:HeadVel agent following a sinusoidal path, with compensation of steady-state error.

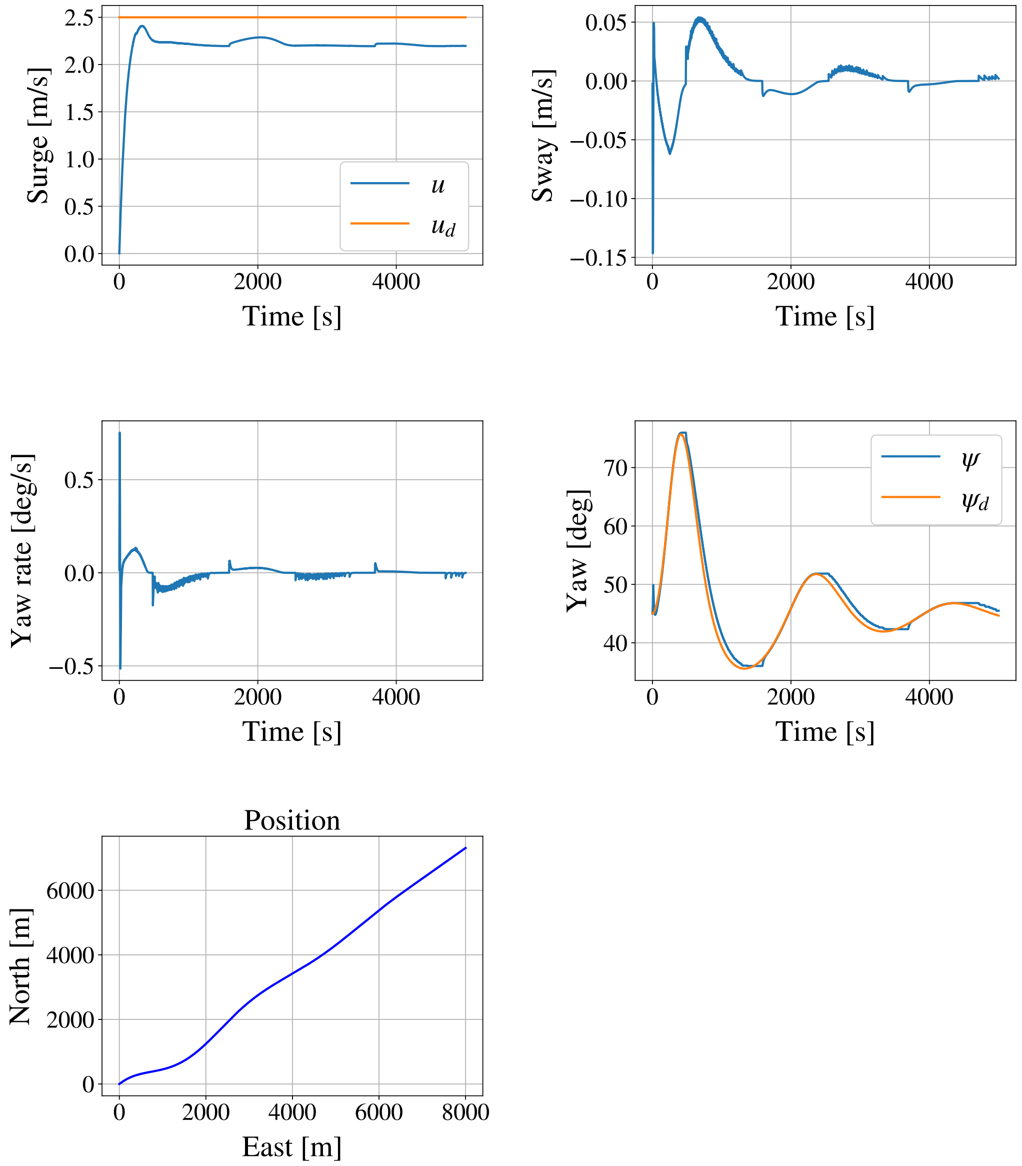


Figure 5.10: State plots of an LP3:HeadVel agent following a exponentially decreasing sinusoidal path, with compensation of steady-state error. Test scenario starts with initial states $\boldsymbol{\eta} = [0, 0, \pi/4]^\top$ and $\boldsymbol{\nu} = \mathbf{0}$.

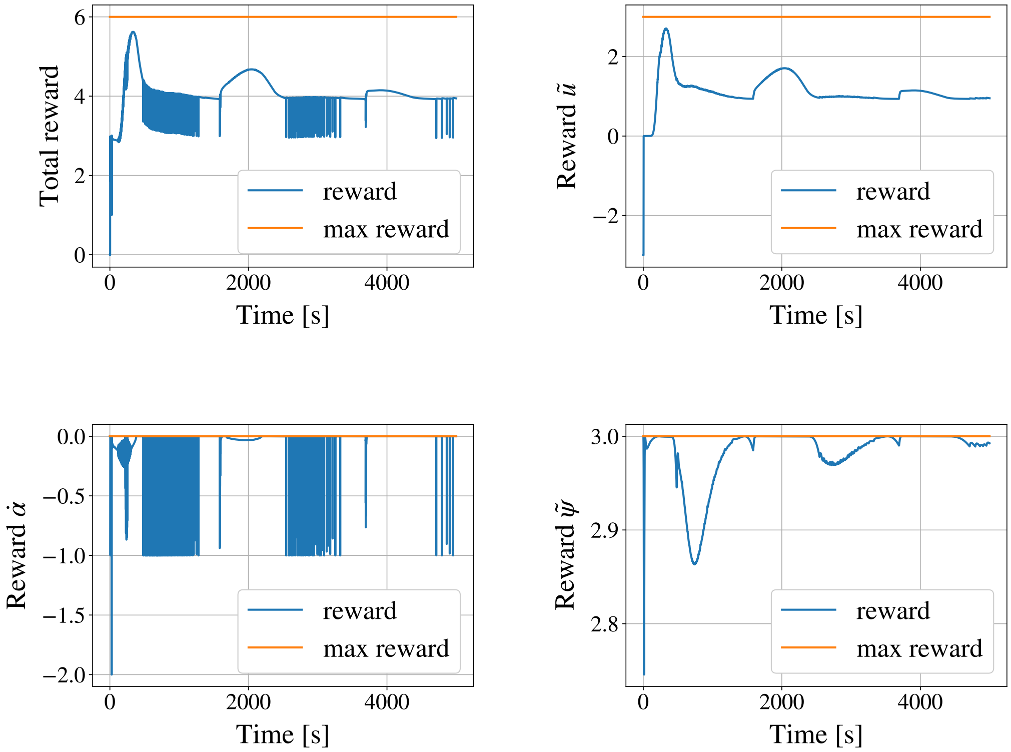


Figure 5.11: Reward plots of an LP3:HeadVel agent following an exponentially decreasing sinusoidal path, with compensation of steady-state error.

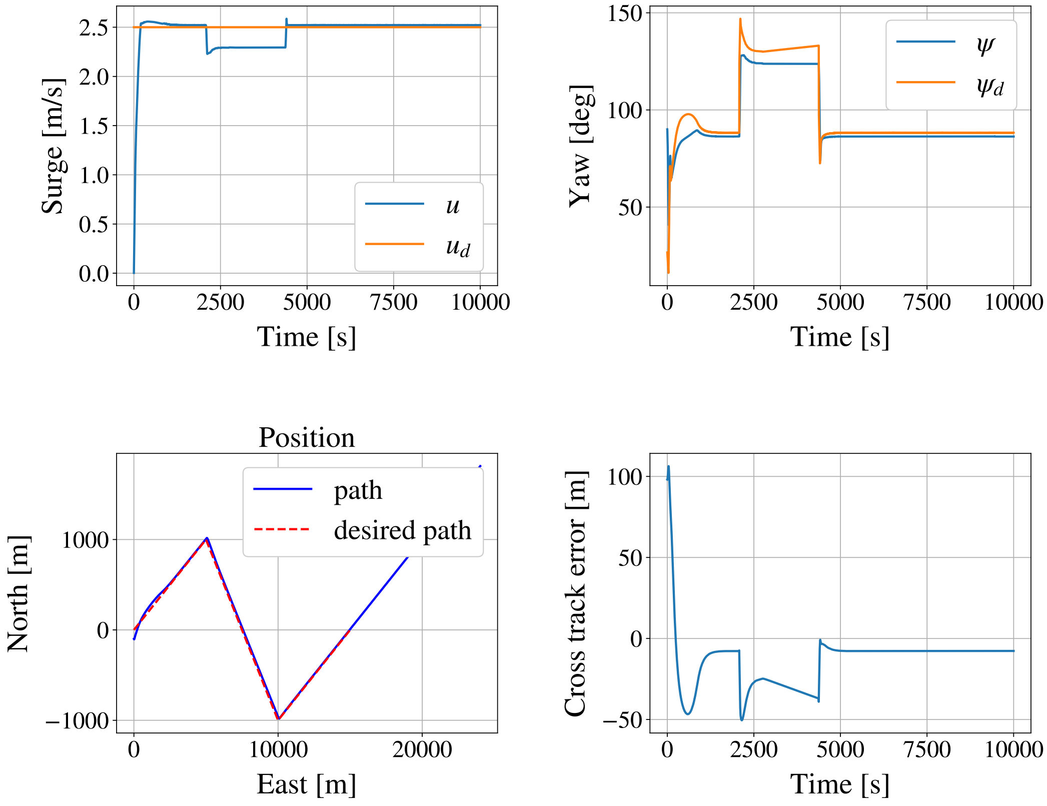


Figure 5.12: State plots of an LP3:HeadVel agent following $\psi_d = \arctan(-e/\delta) + \gamma_p$ to converge the vessel towards a spline, with compensation of steady-state error. The spline consists of the waypoints ([0,5000],[1000,5000],[-1000,10000],[0,15000]), and does not stop when reaching the last waypoint.

5.3 Path following

The LP4:Path agent both minimizes the cross-track error and follows a desired constant forward velocity. It was trained on an exponentially decreasing sinusoidal path. A test scenario of the LP4:Path agent following an exponentially decreasing sinusoidal path is visualized in Figure 5.13 and the corresponding rewards are visualized in Figure 5.14. The agent converges towards the desired path and forward velocity, with a mean absolute cross-track error of 28 m and mean absolute $\tilde{u}$ of 0.3 m/s.

The LP4:Path agent was tested on a spline, using the same initial states as of the test scenario of the LP3:HeadVel agent with LOS generated ψ_d. The state plots for this test scenario for LP4:Path agent are visualized in Figure 5.15 without ocean current, and with ocean current in Figure 5.16. These plots show that the agent can converge towards the desired path, but with a mean absolute error of 16 m without current and 59 m with current, as presented in Table 5.4. Performing new training on a situation with currents may produce better results when the ocen current is present, as presented in article [34].In LOS the desired heading is changed to compensate for ocean currents. In our approach, the ocean current is only available from $\chi = \psi - \beta$. It could be tried to add the sideslip angle to the state vector, to help the DRL-agent, in addition to more training.

The agent is having some difficulties reaching desired forward velocity both with and without current, in same manner as for LP3:HeadVel agent. One interesting fact is that the agent increased its velocity after reaching a turn. The agent might be doing this to reach the path faster. In such cases, the agent has learned how to change control of forward velocity to gain smaller cross-track error. In traditional systems, this is performed by a dedicated system. It might also be random, which we don't know due to the black-box element of the agent.

The LP4:Path agent was tested on different scenarios, where some cases are presented in Table 5.3. The agent converged towards desired velocity and path when the agent was within approximately 300 m of the path. As the distance to the path and $\tilde{\psi}$ increased, the agent had more problems converging towards the path. The interesting thing is that even though the agent had some problems reaching the path, the agent still performed similarly with regard to forward velocity control. This probably means the agent learned that even though it might not be able to reach the path, it still can receive points for reaching the desired velocity.

It was experimented with steady-state error compensation on y_e and $\tilde{u}$. No parameters were found to handle the cross-track error successfully. The reason why the agent did not always converge towards the path and steady-state compensation did not work, could be caused by a short learning process, with the agent only training for approximately 25 episodes. This probably means that the agent had insufficient time to understand the dynamics of how to minimize cross-track error fully, and might minimize the cross-track error based solely on cross-track error ,as in traditional thinking. This is discussed more in section 5.5.

Table 5.3: The results from LP4:Path agent on a spline in different test scenarios. The test scenarios are run for 100000 timesteps, $u_d = 2.5$, and have different initialization of $\boldsymbol{\eta}$ and $\boldsymbol{\nu}$.

Initial state: (E,N,ψ,u,v,r) [m,m,rad,m/s,m/s,rad/s]	**Mean absolute $\tilde{u}$ [m/s]**	**Mean absolute y_e [m]**	**Cumulative reward**	**Comment**
(100,0,$\pi/2$,0,0,0)	0.326	16.490	627961	
(100,0,-$\pi/2$,0,0,0)	0.296	15.197	645159	
(100,400,$\pi/2$,3,0,0)	0.285	14.5077	648381	
(200,400,$\pi/2$,5,0,0)	0.296	14.660	645940	
(100,400,$\pi/2$,3,0,0)	0.284	14.296	648887	
(400,100,$\pi/2$,3,0,0)	0.278	62.853	510220	
(400,100,-$\pi/2$,3,0,0)	0.302	491.701	-1303	Did not converge to path, runs in circles
(450,100,$\pi/2$,3,0,0)	0.301993054339	302.569702231	1698	Did not converge to path, runs in circles

The LP4:Path agent was compared to the LP3:HeadVel agent, using ψ_d from the LOS-guidance. The results are presented in Table 5.4. The results show that LP3:HeadVel agent with LOS generated ψ_d outperforms LP3:Path agent, with regard to mean absolute cross-track error and cumulative reward. The two agents perform quite similarly concerning the mean absolute error in forward velocity. This may mean that the agent could have learned more about how to minimize the cross-track error. Bear in mind that the LP4:Path agent is handling no less than three levels of abstraction simultaneously:

- Thruster allocation.
- Motion control.
- Guidance.

Table 5.4: Performance of LP3:HeadVel agent with LOS generated ψ_d and LP4:Path agent performing path-following on the previously mentioned spline.

Control law - agent	**Current active**	**Cumulativ reward**	**Mean absolute $\tilde{u}$ [m/s]**	**Mean absolute y_e [m]**
LP3:HeadVel agent with LOS generated ψ_d	Yes	677942	0.082	16.495
	No	642988	0.318	5.773
LP4:Path agent	Yes	609753	0.158	59.236
	No	627961	0.326	16.490

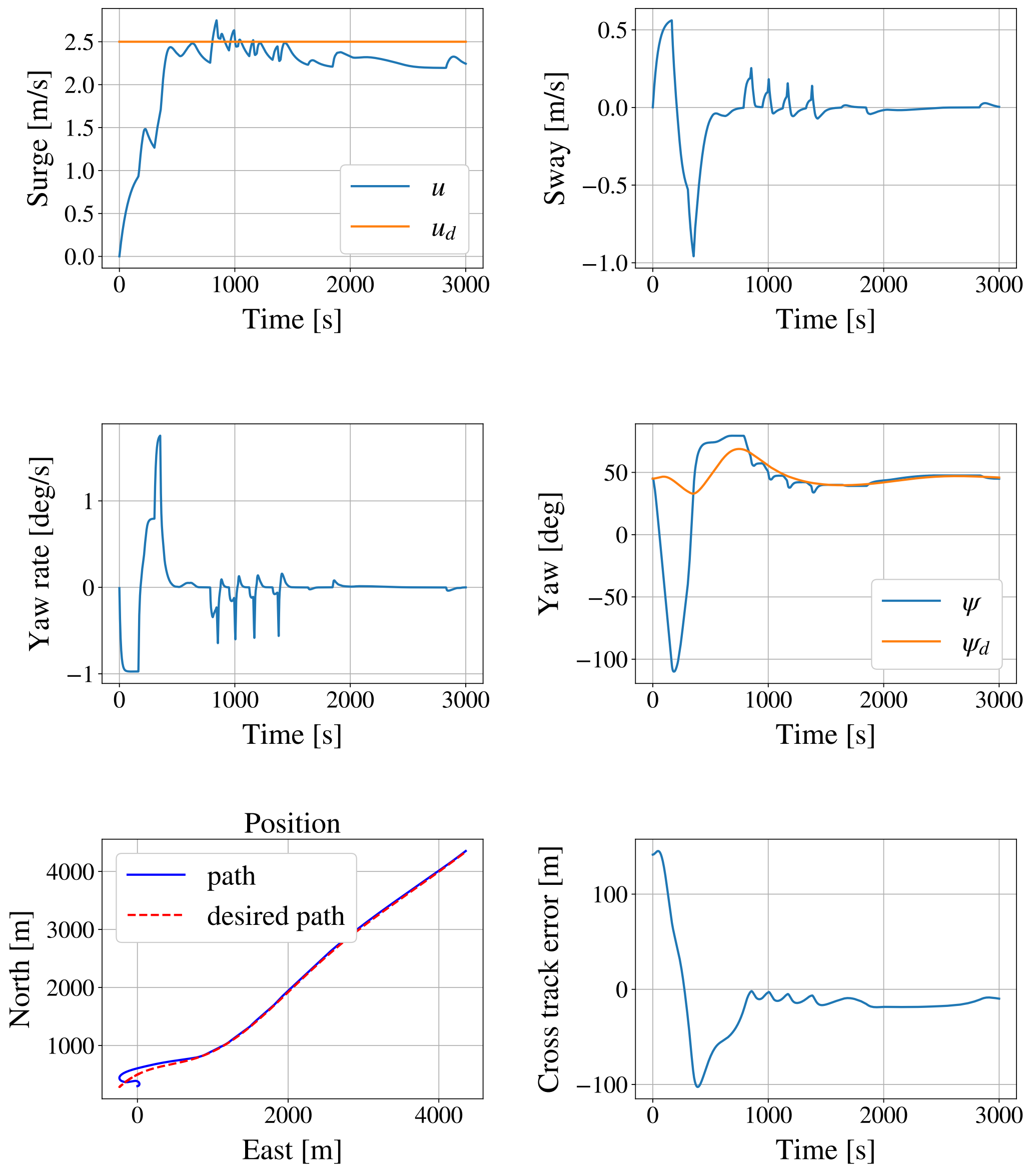


Figure 5.13: State plots of an LP4:Path agent following an exponentially decreasing sinusoidal path, without compensation of steady-state error. The initial states of the test scenario was $\boldsymbol{\eta} = [0, 0, \pi/4]^\top$ and $\boldsymbol{\nu} = \mathbf{0}$.

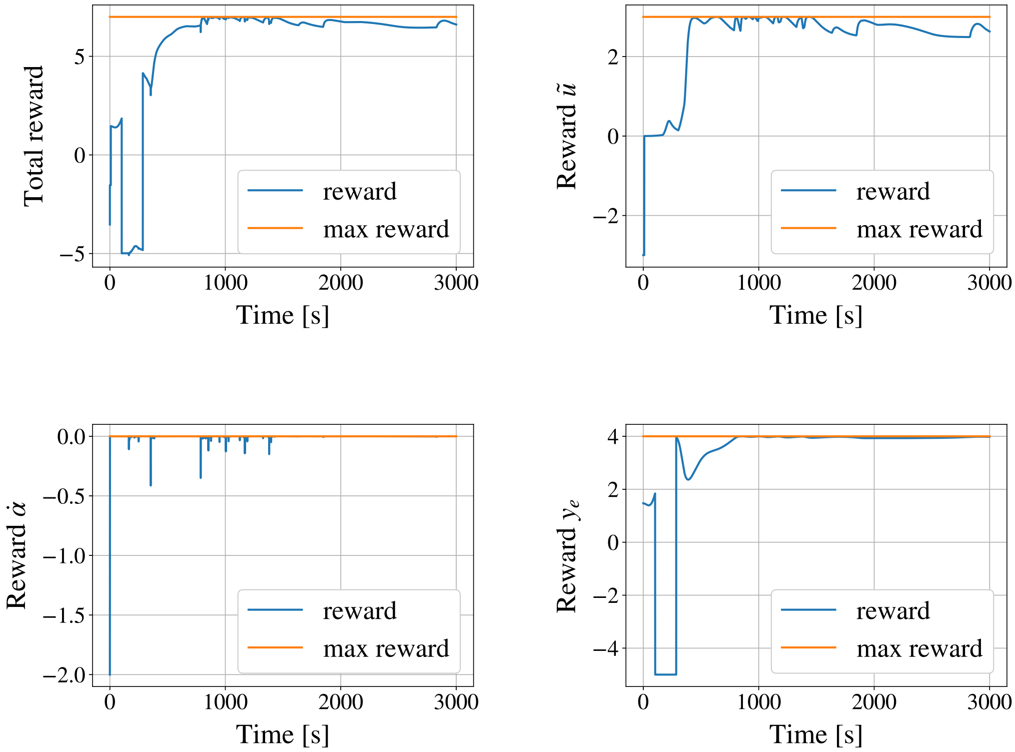


Figure 5.14: Reward plots of an LP4:Path agent following an exponentially decreasing sinusoidal path, without compensation of steady-state error.

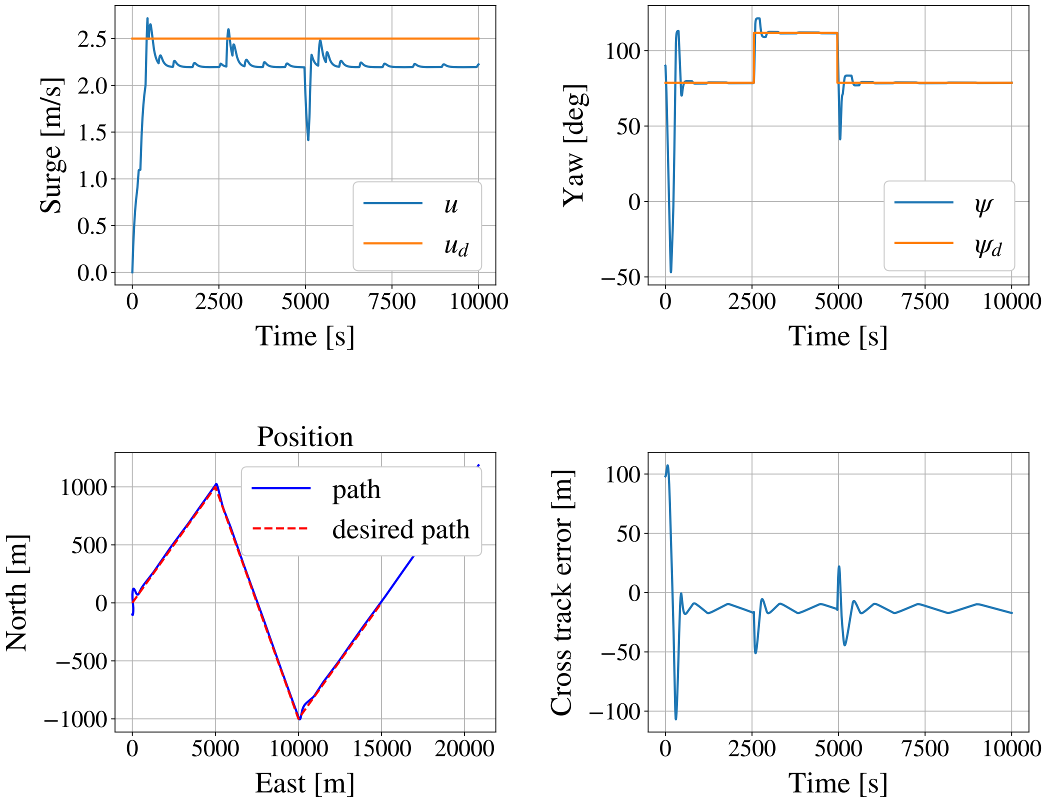


Figure 5.15: State plots of an LP4:Path agent following a spline without current present, without compensation of steady-state error. The initial states of the test scenario was $\boldsymbol{\eta} = [0, 0, \pi/4]^\top$ and $\boldsymbol{\nu} = \mathbf{0}$.The spline consists of the waypoints ([0,5000],[1000,5000],[-1000,10000],[0,15000]), and does not stop when reaching the last waypoint.

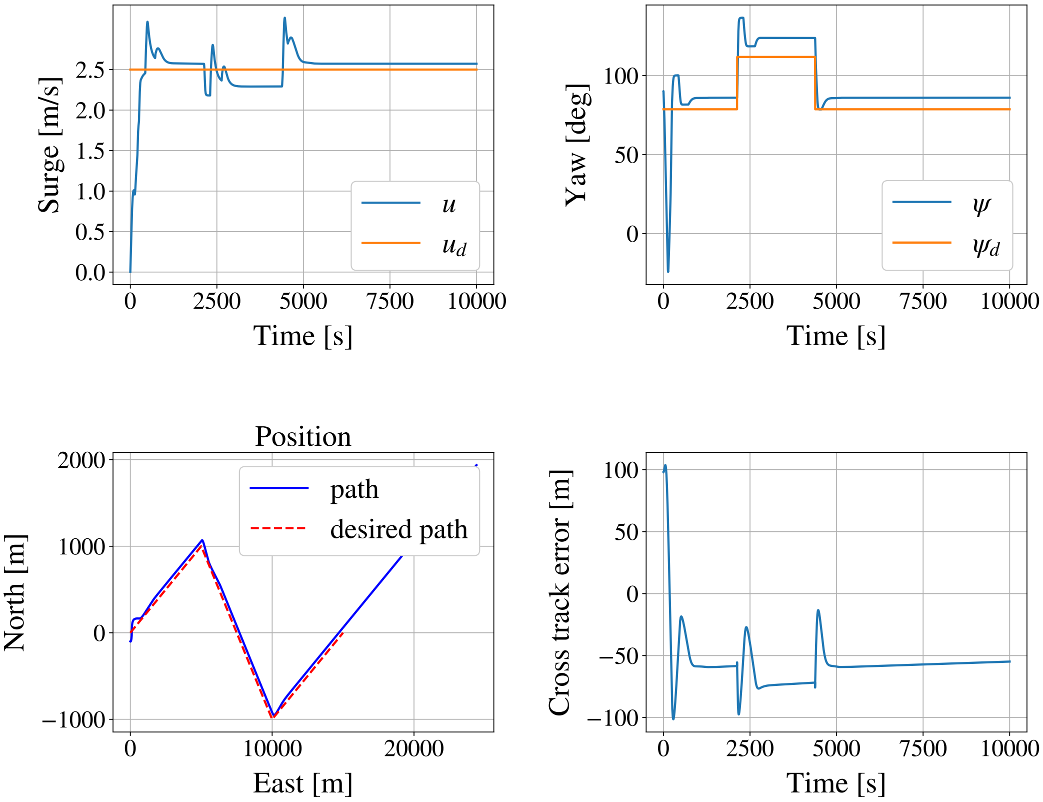


Figure 5.16: State plots of an LP4:Path agent following a spline with current present, without compensation of steady-state error. The initial states of the test scenario was $\boldsymbol{\eta} = [0, 0, \pi/4]^\top$ and $\boldsymbol{\nu} = \mathbf{0}$. The spline consists of the waypoints ([0,5000],[1000,5000],[-1000,10000],[0,15000]), and does not stop when reaching the last waypoint.

5.4 Actuator rate limiting

In real life control, rate of actuator use must be limited due to tear and wear. The use of the actuators can be quite aggressive if no penalty is added to limit the use, with optimal control. The use of actuator for LP1:Vel agent and LP2:VelCourse agent without the extended state vector, is modest, even without penalty of use. An example of nice actuator use with agent LP2:VelCourse is shown in Figure 5.17 , with corresponding state plots in Figure 5.5. One may speculate that the modest actuator uses is due to the inherent simplicity of the assignments.

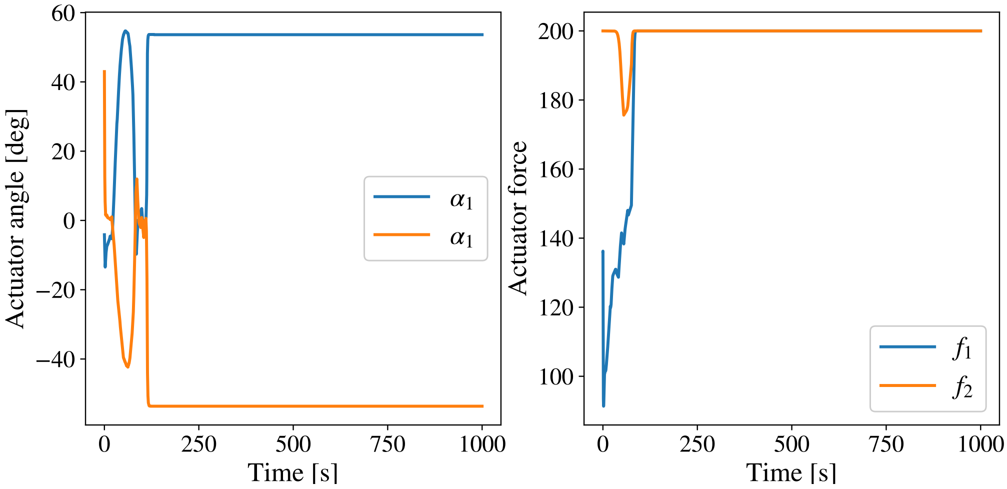


Figure 5.17: Actuator plots for an LP2:VelCourse agent, with compensation of steady-state error. The action space is $\mathbf{f} \in [0, 200]$ kN and $\boldsymbol{\alpha} \in [-170, 170]$ degrees.

The LP2:HeadVel agent with the extended state vector appears to be using the actuators more aggressively. This is illustrated in Figure 5.18, with corresponding state plots in Figure 5.6. Both the actuator angle and force changes quite frequently in this case. One may hypothesize that the state vector extension has led to higher strain on the reasoning in LP2:VelCourse, and might be due to the inclusion of states not relevant for the task of forward velocity control, e.g. cross-track error. This could potentially have been remedied to a certain extent, if trained for a longer period, but would probably have remained a challenge anyhow, given the challenge of actuator used in optimal control.

Due to the aggressive use of the actuator with LP3:VelCourse agent with extended state-vector, the addition of a penalty on actuator angle was proposed for LP3:HeadVel. From optimal control theory, rough use of the actuators is not unknown. An example of resulting actuator use for LP3:HeadVel agent is visualized in Figure 5.19, with corresponding state plots in Figure 5.13 and reward plots in Figure 5.14. The reward plots show that even though the agent receives a penalty for aggressive actuator use, the penalty is relatively small compared

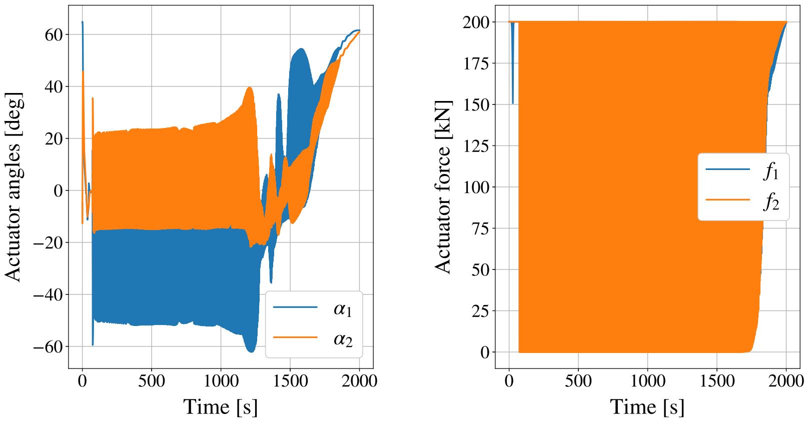


Figure 5.18: Actuator plots for an LP2:VelCourse agent, with compensation of steady-state error. The action space is $\mathbf{f} \in [0, 200]$ kN and $\boldsymbol{\alpha} \in [-65, 65]$ degrees.

to the reward received for better convergence towards the desired path and forward velocity. It was experimented with different weights of rewards, to reduce the rate of actuator angle, but weighing the different objectives is a challenging design task. Even though the agent does not seem to be prioritizing the assignment of reducing the actuator angle rate, it seems to be using the thruster less aggressively than for forward velocity and course stabilization with full state vector.

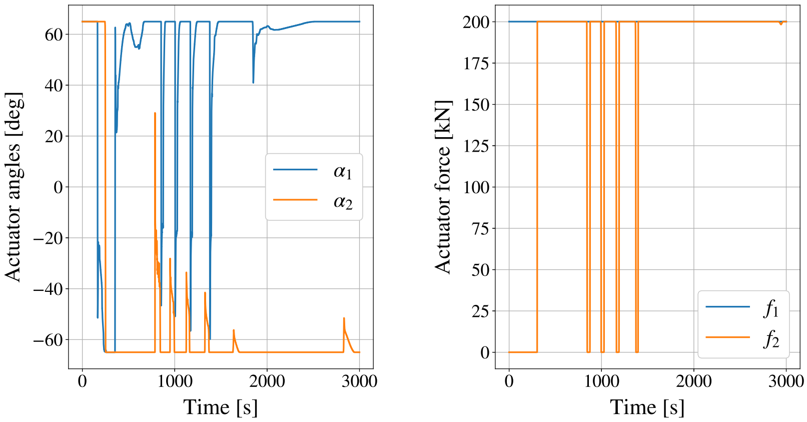


Figure 5.19: Actuator plots for an LP4:Path agent following an exponentially decreasing sinusoidal path, without compensation of steady-state error. The action space is $\mathbf{f} \in [0, 200]$ kN and $\boldsymbol{\alpha} \in [-65, 65]$ degrees.

One interesting fact is that the actuator angles have opposite values for LP2:VelCourse with extended state vector and LP3:HeadVel, meaning the two thrusters are to a certain degree working against each other. Due to this, full use of force only gives a forward velocity of $2 - 3m/s$. In future work, one could make constraints so that the actuators work more efficiently, and not against each other.

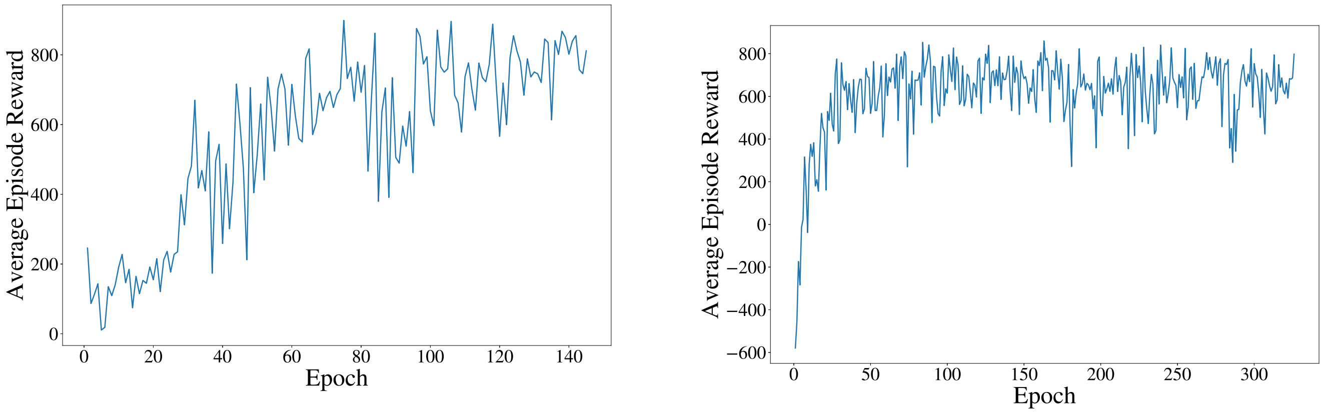
5.5 Training progress

The average episode reward is an interesting performance indicator of the agent during training, illustrated in Figure 5.20. These plots represent the average episode reward for each episode. The different learning phases have different reward-functions, with different magnitudes of average episode rewards. One exception is Figure 5.20b and 5.20c, where the state-vector is different, while the test scenarios are the same. The agent has typically converged to a good policy within 80 - 100 episodes for all learning phases.

The plots representing the learning process of LP2:VelCourse agent, Figure 5.20b and 5.20c, show that the learning process for the extended state vector takes longer to reach the same average episode reward. This might be due to the need for more exploration with bigger state vectors.

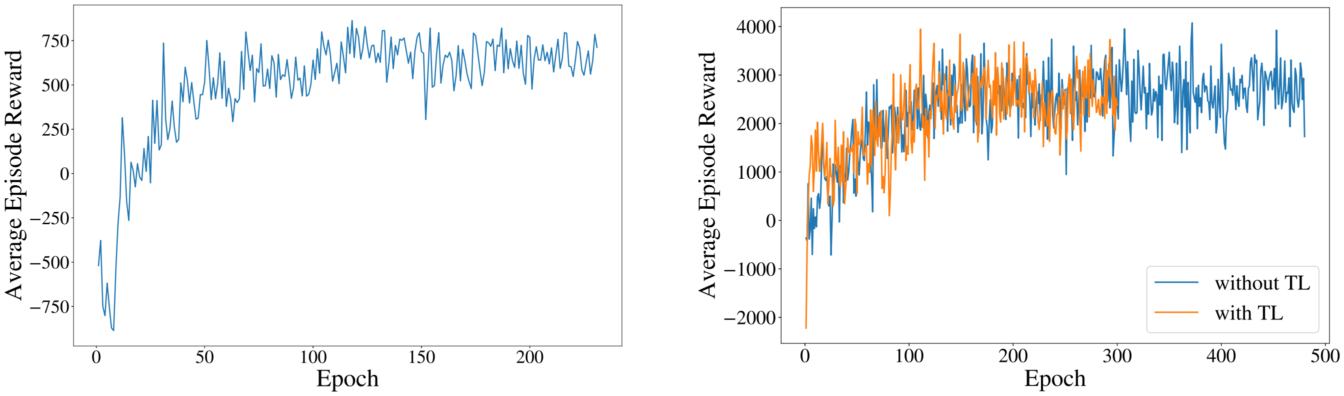
Transfer learning was performed in two steps: From LP2:VelCourse to LP3:HeadVel, and from LP3:HeadVel to LP4:Path. Transfer learning from LP3:HeadVel to LP4:Path was necessary; without it, the agent was not able to find a solution. Transfer learning was not imperative between LP2:VelCourse to LP3:HeadVel, but it increased the learning speed. This is illustrated in Figure 5.20d, where the agent using transfer learning gains quicker higher average episode rewards. Both converge towards the same value at the end.

The training process of the LP4:Path agent is different compared to the other agents, shown in Figure 5.20e. During the training of LP4:Path agent it seemed to perform better with less training, able to quickly elevate the knowledge from the source task to the new task. Continuing to train, the agent seemed to favor optimizing forward velocity too much. This may imply that the proposed reward function LP4:Path should be changed to avoid this. By changing the reward function, the LP4:Path agent might gain better results with more training.



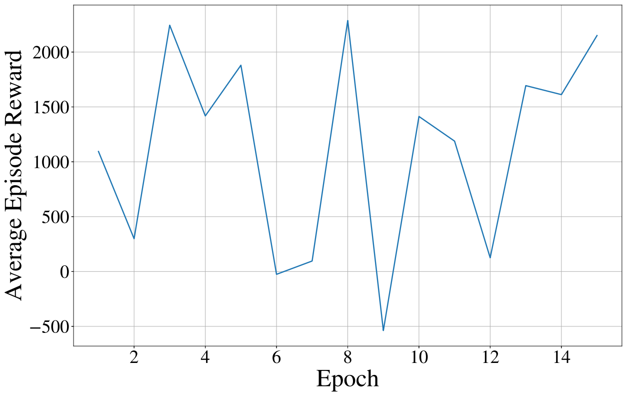
(a) LP1:Vel

(b) LP2:VelCourse



(c) LP2:VelCourse with extended state

(d) LP3:HeadVel



(e) LP4:Path

Figure 5.20: Plots showing the development of average episode reward during training.

Chapter 6

Future work

Even though the DRL-agent was capable of following a spline successfully, several aspects can be further investigated to increase performance. These include:

- **A different type of reward function**: Judging from the results, the Gaussian reward function seemed to be associated with steady state error. This is probably due to high rewards given in proximity of optimal values. Triangular reward functions were tested, but this gives the opposite problem; of giving too high rewards too early. One suggestion is testing a reward function which first would increase rewards gradually, and which becomes more "pointy" near the optimal states. This would differentiate rewards close to an optimal states, and could, for instance, be a combination of a Gaussian and triangular function.

- **Testing even more sets of states**: Several different states in the state-vector were tested. As an example, good results were achieved for LP1:Vel with $\mathbf{x} = [\tilde{u}, \dot{\tilde{u}}, v]^\top$. More experimentation could have been carried out for all the learning phases, in order to for instance excessive states, removing excessive states. For instance, when u_d is constant, it should not be necessary with both u and u_d present in the state-vector.

- **Problems with scalar reward**: Several of the learning phases where performing multi-objective optimization. The scalar reward-function represented the objectives as a weighted sum of partial objectives, such as path convergence and forward velocity control. Selecting the correct weight and avoid unhealthy competitions between objectives in objectives was a difficult design challenge. One should test alternatives to handling multi-objective problems such as multi objective reward functions [61] and hierarchical reinforcement learning [62].

- **Reformulating reward functions in LP4:Path**: As previously discussed, the learning process of LP4:Path was somewhat suboptimal. Looking at the converged policy, it seems to prioritize $u \rightarrow u_d$. This could, for instance, be solved by not giving reward when for instance $u \rightarrow u_d$ unless $y_e < 1000$ m. It was experimented with this but the time available did not offer sufficient opportunity to find a good solution.

- **Exploration of forward velocity**: As previously discussed the convergence of $u \rightarrow u_d$ of LP3:HeadVel and LP4:Path should be further investigated to improve simultaneous velocity and heading control/ cross-track error minimization.
- **Current exploration**: It would be interesting to train the agent more in the presence of ocean current, and see if it can perform better.
- **Docking and collision avoidance**: The next step may be to look at how these agents can be used to solve other more complex tasks, such as docking or collision avoidance. One might speculate that the use of multiple thrusters controlled by a single DRL control system could give an advantage using torque to rotate the vessel into position etc. Collision avoidance could potentially be trained to handle complex scenarios as well.

A long-term goal is to transfer the agent trained on a simulator of a marine vessel, to a real marine vessel. This may utilize transferred policies from training on suitable models, and subsequently perform a certain degree of re-training at sea. One might even envision a certain degree of adaptability in controlled re-learning in real life. Describing how to implement any of these, using DRL, still complying with safety regulations, would be a challenging and exciting future area of research and application.

Chapter 7

Conclusion

The contribution of this project is a methodology for developing a deep reinforcement learning (DRL) agent that can solve the path-following problem for a marine vessel in a progressive/stepwise manner. It begins at a lower level solving control of forward velocity, providing dual thruster allocation. It further adds a series of steps to achieve course stability, heading control and eventually convergence to path and desired forward velocity. Each learning phase represents a new objective. By defining subproblems and solving the task in a progressive manner, the learning performance was improved.

This methodology is model-free, where an agent learns control-laws (policies) by exploring the environment. The policy and value-function are both represented by artificial neural networks. These are trained using the DRL-algorithm Deep Deterministic Policy Gradient (DDPG). The environment in question consists of a theoretical model of a container ship, and equations of motion were used to calculate new states and rewards after the agent has performed an action.

For each learning phase, a reward function was created, together with a fitting state vector. Due to the multi-objective nature of the tasks, the reward function was constructed as a weighted sum of the objectives. Designing good scalar reward functions when multiple objectives are present is a challenging task, both regarding weighting and how to achieve optimization of several objectives simultaneously. The agent will implicitly prioritize some objectives, such as path-following, at the risk of starving others, such as speed control.

The performance of the different agents had mixed success, depending on the complexity of the task and design of the system. The behavior of the agent is similar to what you would expect from classical approaches, in most cases. The LP1-LP3 agents performed similarly to most motion controller works. The agents converge towards the desired values but experiences for instance a steady-state error in some cases. The agent experienced steady-state errors, most likely due to the use of a Gaussian-function in the reward leading to premature reward saturation. Steady-state error compensation took care of this in the motion control cases. The steady-state error compensation did not work on the agent performing path-following. Bear however in mind that the path-following problem is a complex assignment, where the agent solved three levels of abstraction. Ordinary steady-state error compensation might not work in this multi-level control law situation without, or further improvements of the path-following agent.

Even though the agents from the different learning phases had some problems achieving optimal results, the agents were able to reach relatively good results. The path-following agent was able to handle three levels of abstractions simultaneously; thruster allocation, motion control and path-following. The proposed progressive methodology using transfer learning from the lower levels of abstraction and elementary control towards path-following is showing promise, but needs more work to achieve better performance, increased stability, and robustness.

References

[1] Martinsen AB, Lekkas AM, Gros S. Autonomous docking using direct optimal control. In: Submitted to IFAC CAMS 2019;. .

[2] Abadi M, Agarwal A, Barham P, Brevdo E, Chen Z, Citro C, et al.. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems; 2015. Software available from tensorflow.org. Available from: `http://tensorflow.org/`.

[3] Oliphant T. NumPy: A guide to NumPy; 2006–. Accessed: 2019-05-30. USA: Trelgol Publishing. Available from: `http://www.numpy.org/`.

[4] Hunter JD. Matplotlib: A 2D graphics environment. Computing in Science & Engineering. 2007;9(3):90–95.

[5] McKinney W. Data Structures for Statistical Computing in Python. In: Proceedings of the 9th Python in Science Conference,; 2010. p. 51–56. Available from: `http://conference.scipy.org/proceedings/scipy2010/mckinney.html`.

[6] Alex Ray; Amanda Askell ; Ben Garfinkel ; Christy Dennison ; Coline Devin ; Daniel Zeigler; Dylan Hadfield-Menell; Ge Yang; Greg Khan; Jack Clark; Jonas Rothfuss; Larissa Schiavo ; Leandro Castelao ; Lilian Weng ; Maddie Hall ; Matthias Plappert; Miles Brundage; Peter Zokhov PA. AI O, editor. Spinning Up; 2018. Available from: `https://spinningup.openai.com/en/latest/index.html`.

[7] Lillicrap TP, Hunt JJ, Pritzel A, Heess N, Erez T, Tassa Y, et al. Continuous control with deep reinforcement learning. CoRR. 2015;.

[8] draw.io; 2019. Acessed: 30.05.2019. Available from: `https://about.draw.io/`.

[9] Moe S, Caharija W, Pettersen KY, Schjolberg I. Path following of underactuated marine surface vessels in the presence of unknown ocean currents. In: 2014 American Control Conference. IEEE; 2014. .

[10] Fossen TI. Handbook of Marine Craft Hydrodynamics and Motion Control. John Wiley & Sons, Ltd; 2011.

[11] Rolls-Royce-com. Rolls-Royce and Finferries demonstrate world's first Fully Autonomous Ferry; 2018. Available from: `https://www.rolls-royce.com/media/press-releases/2018/03-12-2018-rr-and-finferries-demonstrate-worlds-first-fully-autonomous-ferry.aspx`.

[12] Skredderberget A. The first ever zero emission, autonomous ship; 2018. Available from: `https://www.yara.com/knowledge-grows/game-changer-for-the-environment/`.

[13] Almeida J, Silvestre C, Pascoal A. Path-following control of fully-actuated surface vessels in the presence of ocen currents. IFAC Proceedings Volumes. 2007;40(17):26–31.

[14] Lekkas AM, Fossen TI. Integral LOS Path Following for Curved Paths Based on a Monotone Cubic Hermite Spline Parametrization. IEEE Transactions on Control Systems Technology. 2014;22(6):2287–2301.

[15] Lapierre L, Soetanto D, Pascoal A. Nonlinear path following with applications to the control of autonomous underwater vehicles. In: 42nd IEEE International Conference on Decision and Control. IEEE;. .

[16] Breivik M, Fossen TI. Path following for marine surface vessels. In: Oceans '04 MTS/IEEE Techno-Ocean '04. IEEE;. .

[17] Belleter DJW, Paliotta C, Maggiore M, Pettersen KY. Path Following for Underactuated Marine Vessels. IFAC-PapersOnLine. 2016;49(18):588–593.

[18] Lapierre L, Soetanto D, Pascoal A. Nonsingular path following control of a unicycle in the presence of parametric modelling uncertainties. International Journal of Robust and Nonlinear Control. 2006;16(10):485–503.

[19] Skjetne R, Jorgensen U, Teel AR. Line-of-sight path-following along regularly parametrized curves solved as a generic maneuvering problem. In: IEEE Conference on Decision and Control and European Control Conference. IEEE; 2011. .

[20] Pedone P, Zizzari AA, Indiveri G. Path-Following for the Dynamic Model of a Marine Surface Vessel without Closed-Loop Control of the Surge Speed. IFAC Proceedings Volumes. 2010;43(20):243–248.

[21] Lekkas AM, Fossen TI. Trajectory tracking and ocean current estimation for marine underactuated vehicles. In: 2014 IEEE Conference on Control Applications (CCA). IEEE; 2014. .

[22] Sutton RS, Barto AG. Reinforcement Learning - An introduction. The MIT Press; 2018.

[23] Russell S, Norvig P. Artificial intelligence: A modern approach. 3rd ed. Pearson; 2010.

[24] Bertsekas DP. Reinforcement Learning and Optimal Control; 2019. Available from: `https://web.mit.edu/dimitrib/www/RL_MONOGRAPH1.pdf`.

[25] LeCun Y, Bengio Y, Hinton G. Deep learning. Nature. 2015;521(7553):436–444.

[26] Schmidhuber J. Deep learning in neural networks: An overview. Neural Networks. 2015;61:85–117.

[27] Goodfellow I, Bengio Y, Courville A. Deep Learning. MIT Press; 2016. `http://www.deeplearningbook.org`.

[28] François-Lavet V, Henderson P, Islam R, Bellemare MG, Pineau J. An introduction to deep reinforcement learning. Foundations and Trends® in Machine Learning. 2018;11(3-4):219–354.

[29] Mnih V, Kavukcuoglu K, Silver D, Rusu AA, Veness J, Bellemare MG, et al. Human-level control through deep reinforcement learning. Nature. 2015;518(7540):529–533.

[30] Silver D, Huang A, Maddison CJ, Guez A, Sifre L, van den Driessche G, et al. Mastering the game of Go with deep neural networks and tree search. Nature. 2016;529(7587):484–489.

[31] Gandhi D, Pinto L, Gupta A. Learning to fly by crashing. In: 2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). IEEE; 2017. .

[32] You Y, Pan X, Wang Z, Lu C. Virtual to Real Reinforcement Learning for Autonomous Driving. CoRR. 2017;abs/1704.03952. Available from: `http://arxiv.org/abs/1704.03952`.

[33] Deng Y, Bao F, Kong Y, Ren Z, Dai Q. Deep Direct Reinforcement Learning for Financial Signal Representation and Trading. IEEE Transactions on Neural Networks and Learning Systems. 2017;28(3):653–664.

[34] Martinsen AB, Lekkas AM. Straight-Path Following for Underactuated Marine Vessels using Deep Reinforcement Learning. IFAC-PapersOnLine. 2018;51(29):329–334.

[35] Tuyen LP, Layek A, Vien NA, Chung T. Deep reinforcement learning algorithms for steering an underactuated ship. In: 2017 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI). IEEE; 2017. .

[36] Martinsen AB, Lekkas AM. Curved Path Following with Deep Reinforcement Learning: Results from Three Vessel Models. In: OCEANS 2018 MTS/IEEE Charleston. IEEE; 2018. .

[37] Mitchell TM. Machine Learning. 1st ed. McGraw-Hill Education; 1999.

[38] Priddy KL, Keller PE. Artificial neural networks an introduction. SPIE; 2005.

[39] Rosenblatt F. The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain. Psychological Review. 1958;p. 65–386.

[40] Galante L, Banisch R. A Comparative Evaluation of Anomaly Detection Techniques on Multivariate Time Series Data. 2019;.

[41] Srivastava N, Hinton GE, Krizhevsky A, Sutskever I, Salakhutdinov RR. Dropout: a simple way to prevent neural networks from overfitting. Journal of Machine Learning Research. 2014;15:1929–1958.

[42] Ioffe S, Szegedy C. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In: ICML; 2015. .

[43] He K, Zhang X, Ren S, Sun J. Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2016;p. 770–778.

[44] Wiering M, van Otterlo M, editors. Reinforcement Learning. Springer Berlin Heidelberg; 2012.

[45] Arulkumaran K, Deisenroth MP, Brundage M, Bharath AA. A brief survey of deep reinforcement learning. CoRR. 2017;.

[46] Tangkaratt V, Abdolmaleki A, Sugiyama M. Guide Actor-Critic for Continuous Control. In: International Conference on Learning Representations; 2018. Available from: `https://openreview.net/forum?id=BJk59JZ0b`.

[47] Gu S, Lillicrap TP, Sutskever I, Levine S. Continuous Deep Q-Learning with Model-based Acceleration. CoRR. 2016;Available from: `http://arxiv.org/abs/1603.00748`.

[48] Amos B, Xu L, Kolter JZ. Input Convex Neural Networks. In: Precup D, Teh YW, editors. Proceedings of the 34th International Conference on Machine Learning. vol. 70 of Proceedings of Machine Learning Research. International Convention Centre, Sydney, Australia: PMLR; 2017. p. 146–155. Available from: `http://proceedings.mlr.press/v70/amos17b.html`.

[49] Watkins CJCH, Dayan P. T. Machine Learning. 1992;8(3/4):279–292.

[50] Williams RJ. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine Learning. 1992;8(3-4):229–256.

[51] Sutton RS, Mcallester DA, Singh SP, Mansour Y. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In: Solla SA, Leen TK, Müller KR, editors. Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]. The MIT Press; 1999. p. 1057–1063.

[52] Konda VR, Tsitsiklis JN. On Actor-Critic Algorithms. SIAM J Control Optim. 2003;42(4):1143–1166. Available from: `https://doi.org/10.1137/S0363012901385691`.

[53] Peters J, Schaal S. Natural Actor-Critic. Neurocomputing. 2008;71(7-9):1180–1190.

[54] Schulman J, Levine S, Moritz P, Jordan MI, Abbeel P. Trust Region Policy Optimization. CoRR. 2015;abs/1502.05477. Available from: `http://arxiv.org/abs/1502.05477`.

[55] Schulman J, Wolski F, Dhariwal P, Radford A, Klimov O. Proximal Policy Optimization Algorithms. CoRR. 2017;Available from: `http://arxiv.org/abs/1707.06347`.

[56] Konda V. Actor-critic algorithms. Cambridge, MA, USA; 2002. AAI0804543.

[57] Taylor ME, Stone P. Transfer Learning for Reinforcement Learning Domains: A Survey. J Mach Learn Res. 2009 Dec;10:1633–1685. Available from: `http://dl.acm.org/citation.cfm?id=1577069.1755839`.

[58] Taylor ME, Stone P. Behavior transfer for value-function-based reinforcement learning. In: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems - AAMAS '05. ACM Press; 2005. .

[59] Selfridge RS O ; Sutton, Barto AG. Training and tracking in robotics. Proceedings of the Ninth International Joint Conference on Artificial Intelligence . 1985;p. 670–672.

[60] Kingma DP, Ba J. Adam: A Method for Stochastic Optimization. CoRR. 2015;abs/1412.6980.

[61] Friedman E, Fontaine F. Generalizing Across Multi-Objective Reward Functions in Deep Reinforcement Learning. CoRR. 2018;abs/1809.06364.

[62] Osa T, Tangkaratt V, Sugiyama M. Hierarchical Reinforcement Learning via Advantage-Weighted Information Maximization. CoRR. 2019;abs/1901.01365.

Appendix

.1 Model of container ship

The container ship used in this project was selected from a paper about autonomous docking [1]. The vessel consisted of:

- Inertia matrix **M**, (1)
- Dampening matrix **D**, (2)
- Thruster allocation $\boldsymbol{\tau}(\boldsymbol{\alpha}, \mathbf{f})$, (3)

$$\mathbf{M} = \begin{bmatrix} 6767.7822 & 0. & 0. \\ 0. & 11346.8706 & -34032.68784 \\ 0. & -34032.68784 & 4454604.381096 \end{bmatrix} \tag{1}$$

$$\mathbf{D} = \begin{bmatrix} 7.707e+01 & 0. & 0. \\ 0. & 2.55e+02 & -2.034e+03 \\ 0. & -6.726e+02 & 3.850e+05 \end{bmatrix} \tag{2}$$

$$\boldsymbol{\tau}(\boldsymbol{\alpha}, \mathbf{f}) = \begin{bmatrix} \cos(\alpha_1) & \cos(\alpha_2) & 0 \\ \sin(\alpha_1) & \sin(\alpha_2) & 1 \\ l_{x,1}\sin(\alpha_1) - l_{y,1}\cos(\alpha_1)) & l_{x,2}\sin(\alpha_2) - l_{y,2}\cos(\alpha_2)) & l_{x,3} \end{bmatrix} \mathbf{f} \tag{3}$$

With:

- $l_{x,1} = -35$
- $l_{x,2} = -35$
- $l_{x,3} = 35$

- $l_{y,1} = 7$
- $l_{y,2} = -7$