NTNU

Norwegian University of
Science and Technology

Path Following in Simulated Environments using the A3C Reinforcement Learning Method

Emil Andreas Lund

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Anastasios Lekkas, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

Using reinforcement learning as a part of a Guidance, Navigation and Control (GNC) system is a relatively unexplored field. This thesis explores the use of deep reinforcement learning in a path following control algorithm in a realistic simulated environment. A reinforcement learning method called the Asynchronous Advantage Actor-Critic (A3C) method has been implemented using artificial neural networks to model the learning, making it *deep* reinforcement learning. In addition, the software needed to use the V-REP robot simulator together with the A3C algorithm was developed. This was used to successfully learn to control a vehicle model in a path following situation in the V-REP simulator. In addition, the algorithm was used to learn to control a much simpler ship model simulated without V-REP. It was found that the learned behavior from the ship model could be transferred and used to control the vehicle model in the more complex environment, substantially speeding up the total time of learning process for the vehicle.

Sammendrag

Bruk av forsterkende læring (reinforcement learning) i fartøystyring er et nytt og lite utforsket felt. Denne oppgaven utforsker bruken av dyp forsterkende læring i stifølging. En forsterkende lærings-metode kalt A3C ble implementert. Kunstige nevrale nett ble brukt sammen med metoden, og det er det som gjør det til *dyp* forsterkende læring. I tillegg ble det utviklet programvare nødvendig for å bruke robotsimulatoren V-REP sammen med A3C algoritmen. Ved bruk av disse var det mulig å lære å kontrollere et kjøretøy i stifølging i V-REP simulatoren. I tillegg ble algoritmen brukt til å lære å kontrolere en simplere skipsmodell simulert uten V-REP. Det ble funnet ut at kunskapen man lærte ved å trene på skipet kunne overføres til kjøretøyet, og at man på denne måten kunne senke tiden det tar å trene seg opp til å styre kjøretøyet.

Preface

This master thesis was written during the spring semester of 2018 at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology.

I would like to thank my supervisor Anastasios Lekkas for all the guidance, and for letting me work on a very interesting topic. Working in parallel with his thesis on a similar topic, I want to thank Andreas Bell Martinsen who has been a good discussion partner and always available for help. He aided me with the ship model used in this thesis and came up with a way of formulating the reward function of the reinforcement learning algorithm implemented.

The following tools have been used for this project:

- The Linux distribution Ubuntu 16.04
- Computer with processor Intel(R) Core(TM) i7-4790 CPU 3.60GHz, without GPU
- Python 3.5 programming language
- V-REP simulation software [7]

The actor-critic reinforcement learning algorithm implemented in my project work preceding this master thesis was used as a basis when implementing the A3C algorithm.

Problem Description

The task is to implement a reinforcement learning method and train a vehicle to navigate in a simulated environment by path-following using this method. Explore how different design choices influences the performance and learning.

Contents

List of Figures

1 Introduction

The field of reinforcement learning have developed rapidly over the last years, but its history goes way back. The first major breakthrough was in 1963 when Donald Michie managed to create a machine that could learn to play tic-tac-toe [20]. It was a machine in the loosest of terms because it only consisted of matchboxes and beads. Each possible position in the game had its own matchbox, and the more beads it was in a box, the better this position was. This worked for a simple game as tic-tac-toe, but for more complicated games with more positions, it becomes exponentially many boxes. In 1995, Gerald Tesauro developed TD-gammon [32], a program which learns to play backgammon. An important part of TD-gammon was a breakthrough in how the learning was represented. Instead of using match boxes as Donald Michie, it successfully used a simple artificial neural network to learn how to play the game.

As computational power has grown over the last decade, complex neural networks have been proven more and more successful in fields like image and speech recognition. Larger and deeper networks are now feasible to train and is used more and more also in reinforcement learning, so-called *deep* reinforcement learning. The Google DeepMind group was able to learn to play Atari games with only the raw screen pixels as input to the algorithm, using reinforcement learning and a neural network, see [22]. They have also successfully learned to play the games chess and Go entirely from self-play with the new AlphaZero algorithm [28]. Not only does AlphaZero learn to play by itself from only knowing the rules, but have proved to learn to play better than the previously strongest chess computer. These previously mentioned examples are all learning simulated environments, but reinforcement learning has also been applied in robotics with impressive results, for example, playing games such as table tennis [16], and flying a helicopter upside down [23].

Figure 1: Image from the game in a five game match between the Go world champion Lee Se-dol and DeepMind's AlphaGo in 2016. The score ended 4-1 to AlphaGo, making it the first artificial player to consistently beat the best human player.

The usage of deep reinforcement learning in path following is a relatively unexplored field. The goal of this thesis is to explore if reinforcement learning can be used to learn this kind of motion control. Can it learn a behavior superior to the previous path following algorithms? The focus of this thesis is on real-world applications, and it was therefore chosen to build the applications to interact with a simulator called V-REP to make the model as realistic as possible. Both the simu-

lations and the interaction by having to send commands and receive sensor readings is realistic and gives some of the challenges that would have been encountered by working with a real-world model. At the same time, since it is a simulation, it is possible to achieve in the time frame of this thesis.

The theory in the following chapter about reinforcement learning is primarily found in *Reinforcement Learning: An Introduction* by Richard S. Sutton and Andrew G. Barto [29]. They provide an excellent explanation of the different parts of Reinforcement Learning. The books *Dynamic Programming and Optimal Control* by Dimitri P. Bertsekas [1] provides a more in-depth mathematical explanation of topics than Sutton & Barto and has worked as a supplement. The following chapter starts with explaining the Reinforcement Learning problem, the theory needed to implement the algorithms and the neural networks, and some explanations of terms used in this thesis. The method section describes the setup, the models used, and the software implementation choices made. The results are then presented and discussed.

2 Theory

This section starts with a high-level explanation of reinforcement learning, and where it stands compared to other machine learning algorithms. Then the more in-depth theory is explained. Following this, three reinforcement learning methods are presented: *Policy gradient, REINFORCE, and Actor-Critic.* This is done because they build on top of each other, and it was natural to include them even though it is the Actor-Critic method that is in focus. Theory on neural networks and optimization is then presented before some terms used is explained.

2.1 The Reinforcement Learning Problem

Reinforcement learning is a method in machine learning inspired by the learning witnessed in humans and animals. As an example, we can look at a person trying to shoot a basketball through the hoop. When a person is trying to learn this, he or she would repeatedly throw the ball at the hoop. For each throw, the person would do an *observation* by looking where the ball went, and get a *reward* if they scored. In the human case, this reward would be dopamine released by the brain [6] leaving the thrower feeling good about the throw, and understanding that how they threw the ball this time worked well. If they had missed, the thrower would be left without a reward, and understand that they would have to change something if to score.

The thrower, or *agent*, in this example could just as well be a robot manipulator. In contrast to the human agent, the robot has no natural way of understanding if an action is good or bad. Instead, we can introduce a *reward function* to the robot. This function would return a value, quantifying how good an action is. If it was the robot trying to score in the basketball case, the reward function could be defined as returning 10 for scoring, and -1 for missing. Defining the reward function is a crucial and difficult part of designing a reinforcement learning algorithm, as this is what will lead the change in the behavior of the agent. This is also the most impressive part of reinforcement learning. It enables an agent to autonomously explore an environment, without having its behavior explicitly programmed. This is extremely useful when the environment is complicated, or when the programmer lacks essential information about the environment.

2.2 Supervised, unsupervised and evaluative learning

Machine learning algorithms can often be classified as either supervised or unsupervised, but although reinforcement learning as a machine learning algorithm is somewhat related to both it cannot fit into any of the categories. Supervised learning occurs when the model is training on *labeled* data. For example, when trying to teach a model to recognize a cat in a picture, we would need a lot of pictures labeled as "cat", and a lot of pictures labeled "not cat". When training on these, the feedback we get would be *instructive*, as opposed to *evaluative*. Instructive feedback tells us *how* to achieve a goal, while evaluative feedback is how well we achieve a goal. Reinforcement learning is relying on evaluative feedback on each

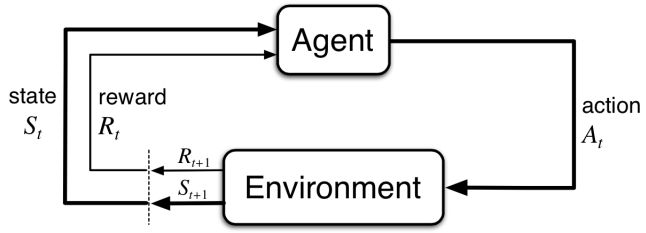


Figure 2: The agent is interacting with the environment, as defined by Markov Decision Processes. This figure is taken from [29]

action. It is important to notice that one action cannot be evaluated on its own, but only in relation to other actions.

When a reinforcement learning agent is initialized, it does not know anything about its environment. It has to explore, and label actions with the reward function by itself. In this sense, reinforcement learning is related to unsupervised learning. In unsupervised learning, there is also no labeled data. The goal for unsupervised learning is to model the underlying structure or distribution in the data in order to learn.

2.3 Markov Decision Process

To be able to describe reinforcement learning mathematically, a formal decision-making modeling called Markov Decision Processes (MDP) is used. In reinforcement learning, the agent is interacting with an environment through actions. The agent observes the state of the environment and receives a reward. This is visualized in Figure 2. As explained in [29], all of these are key elements in MDPs, as it can be fully described by the following four variables:

- S is the set of all possible states, and s_t is the state of the agent at timestep t.
- A is the set of possible actions.
- T is the state transition probability distribution of a given state and action. The probability of arriving in a state s' is given by T(s, a, s')
- R is the reward function.

To be able to model a system as an MDP, it needs to fulfill the Markov property. This is achieved if the agent at each step can observe the full state, such that no memory of earlier observations is needed to understand the current state.

2.3.1 Model-free vs. Model-based reinforcement learning

The goal of the agent in an MDP is to find a policy (explained in Section 2.5) that maximizes the expected future reward. If all of the four elements of an MDP

are available, the optimal policy can be calculated before having to execute an action in the environment. In these cases, the task is more a planning problem than a reinforcement learning problem. The challenge in most real-world tasks is that the state transition probability T is unknown. However, it is possible to find the optimal policy by learning the transition probabilities T, called model-based methods, or to avoid them, called model-free methods. Policy Gradient and Actor-Critic (Section 2.7 and 2.8) are examples of model-free methods.

2.4 Returns

The goal when choosing an action is to maximize the cumulative reward. The cumulative reward is called the *return*, and is denoted G_t. Here, t is the timestep of a certain return. The return is then defined as the following:

$$G_t := r_{t+1} + r_{t+2} + r_{t+3} + ... = \sum_{k=0}^{\infty} r_{t+k+1}$$

In the case where the timesteps goes to infinity, this does not make much sense since the return would go to infinity. It would only work if the task is limited by an upper time limit. When this is the case, the task is called *episodic*. But in both episodic and continuous cases, it is beneficial to introduce a *discount factor* γ, where $0 \leq \gamma \leq 1$ The expression for the discounted return becomes the following:

$$G_t := r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{1}$$

The intuition behind the discount factor is simple. The further into the future you see, the less certain is it that you will receive the reward. Since the processes include some uncertainty, it is beneficial to weight the rewards far in the future less than what follows immediately. The edge case is when $\gamma = 0$. This would make the agent extremely short-sighted, and only consider the reward nearest in the future, unable to take into account if it is good in the long run.

2.5 Policy

A *policy* describes a way of acting. It can be understood as the rules of choosing an action. If you are stuck in a labyrinth, a trivial policy could be "always go to the right", and you would get out in the end. In the sense of reinforcement learning, the policy can be defined as the function $\pi(s)$, that returns the probability distribution for choosing each possible action in the state s. The goal is to find the optimal policy $\pi^*(s)$. The optimal policy is the policy that tells us what action to choose to maximize the return in each state. There are several different algorithms that try to optimize the policy, and we will later discuss both value-based and policy gradient-based methods.

2.6 Value functions

The exact expected return for a state or an action can be difficult to know, especially for high-dimensional problems, however, it is possible to approximate. As earlier explained, reinforcement learning uses evaluative feedback to learn, and this means that we need a notion of how good a state or an action is. This is where the value function enters. As given in [29], the value function estimates the expected return given a state. The value function given a state s is called the *state-value function* $V^\pi(s)$, and given state-action pair it is called the action-value function $Q^\pi(s,a)$. The superscript π tells us that this is the resulting value functions when following a policy $\pi(s)$. Even for the same environment, the value function will change if you change the policy. This is because a state will have a different expected return for different behaviors. When the policy is stochastic, the expectation $\mathbb{E}$ is necessary to formulate the equation. The state-value and action-value function is defined eq. 2 and eq. 3 as in [29]. Notice that the definition of the return as defined in eq. 1 is used.

$$V^\pi(s) = \mathbb{E}[G_t|S_t = s] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{k+t+1}|S_t = s] \tag{2}$$

$$Q^\pi(s,a) = \mathbb{E}[G_t|S_t = s, A_t = a] = \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k r_{k+t+1}|S_t = s, A_t = a] \tag{3}$$

These functions can be estimated from experience. There exist several ways of updating the value function, and here the focus is on Temporal-Difference (TD). TD is a combination of the idea behind Monte Carlo methods and dynamic programming (DP). Like in DP, learning is possible even when we have not finished the environment, but without having a model of the system (model-free). This is related to Monte Carlo methods, but Monte Carlo methods need to wait until the episode is finished, and the return for each state is known, before updating the value function. Monte Carlo and DP are further explained in [29].

The TD learning comes from having the state value reflect the value of the next state after taking an action. For a tabular value function $V(S)$, the update function wolud be:

$$V(S) = V(S) + \alpha(r_t + \gamma V(s') - V(s)) \tag{4}$$

Here, α is the learning rate, r_t is the reward at timestep t, s and s' is the current and next state, and γ is a discount factor. As a special case with $\alpha = 1$, V(s) cancels out, and the state value is only dependent on the next state value. Equation 4 is the special case of Temoral Difference-error called TD(0), or one-step TD, because it only takes into account one step into the future. The TD-error δ, which is more suited for parameterized value functions and will return later, is given by

$$\delta = r_t + \gamma V(s') - V(s) \tag{5}$$

2.7 Policy gradient

The policy gradient is a method that differentiates itself from the previously mentioned algorithms. Instead of choosing a policy by using a value-function, we can parameterize a policy function directly. The difference is that the algorithm does not choose greedily the action which seemingly has the highest reward from the value function, but instead tries to learn a function π that directly map a state to a probability distribution over the actions. For each action a we can define the policy function in the following way as in [29]:

$$\pi(a|s, \boldsymbol{\theta}) = p(A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta})$$

Here s is a given state and $\boldsymbol{\theta}$ is the learned weights. If the parametrization is an artificial neural network, then $\boldsymbol{\theta}$ is the weights in the network. The p() function is the probability function, and A_t, S_t and $\boldsymbol{\theta}_t$ is the action, state and weights at a given timestep t. In this method, the policy weights $\boldsymbol{\theta}$ is learned by using the gradient of a performance measure $J(\boldsymbol{\theta})$ as given in [29]. Since one want to maximize the performance, gradient ascent can be performed on $J(\boldsymbol{\theta})$. The update step is defined as

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + \alpha\widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{6}$$

Here α is the learning rate, and $\widehat{\nabla J(\boldsymbol{\theta}_t)}$ is a stochastic estimate of $\nabla J(\boldsymbol{\theta}_t)$. The use of stochastic estimates is often used in optimization problems where the actual function cannot be computed directly, and data is only available through noisy measurements. In the case of policy gradient, the measurements are not necessary noisy, but is however built on data with uncertainty as the return is not exact.

An important part of the policy gradient algorithm is that the action choice is non-deterministic. By having the policy function output a distribution over the actions and choosing an action by sampling from this distribution, the algorithm is exploring the action space by itself. This is in contrast to the value or policy iteration where we need to apply exploration through ϵ-greedy. ϵ-greedy is a technique where instead of the calculated action, a random action is chosen with a probability ϵ and thus ensuring exploration. The non-deterministic factor also makes an important difference in environments where stochastic behavior is needed, for instance in a board game where the opponent could take advantage of deterministic behavior.

When using a direct parameterization of the policy function, the action probability distribution will change smoothly with regard to the learned parameters $\boldsymbol{\theta}$. In contrast, for value-based algorithms, the distribution can change dramatically for small changes in the action-values. For physical applications, this is an important property. If our agent is a robot arm trying to learn a task, sudden dramatic changes in the policy can result in unexpected and extreme behavior which can lead to dangerous situations. It can also lead to extreme utilization of the action space, and may then damage the actuators and the rest of the robot. For the same reason, policy search algorithms are also shown to have better convergence properties [17]. Even though value-based algorithms may converge quicker to good,

possibly globally optimal solutions, such learning processes often prove unstable under function approximation [17].

The policy gradient theorem as derived in [29] is defined as the following:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s,a) \tag{7}$$

This proportionality gives us a connection between the gradient of the cost function, and the gradient of the policy function. This is important for the derivation of the REINFORCE algorithm.

2.7.1 REINFORCE algorithm

The REINFORCE algorithm was first proposed by Williams in 1992 [35]. Following is the algorithm derived as in [29]. The policy gradient theorem given in Equation 7 is now formulated with the expected value, and leave us with a stochastic estimation of the gradient:

$$\nabla J(\theta) = \mathbb{E}[\sum_a \nabla \pi(a|s) q_\pi(s,a)]$$

Here, the gradient is given as a sum over the actions. By multiplying and dividing by the policy function, the summation can be removed, and we end up with the gradient of the logarithm of the policy function $\pi(a|s)$.

$$\nabla J(\theta) = \mathbb{E}[\sum_a \pi(a|s) q_\pi(s,a) \frac{\nabla \pi(a|s)}{\pi(a|s)}]$$
$$\nabla J(\theta) = \mathbb{E}[q_\pi(s,a) \frac{\nabla \pi(a|s)}{\pi(a|s)}]$$

The action value can be changed to the return of the state, since $q_\pi(s,a) = \mathbb{E}[G_t|s,a]$ from Equation 2. Since the derivative of the logarithm of the policy function is given as $\nabla \ln \pi = \frac{\nabla \pi}{\pi}$, the equation can be given as

$$\nabla J(\theta) = \mathbb{E}[G_t \nabla \ln \pi(a|s)] \tag{8}$$

This expression used in the update step for gradient methods explained in Section 2.11 is then given as

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + \mathbb{E}[G_t \nabla \ln \pi(a|s)] \tag{9}$$

When implementing this, the expected value is approximated by the mean of a N-sized batch of estimates:

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + \frac{1}{N} \sum_{i=0}^{N} [G_t \nabla \ln \pi(a|s)] \tag{10}$$

2.7.2 REINFORCE with baseline

The update step in Equation 9 can be generalized by adding a baseline function $b(s)$

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + (G_t - b(s))\nabla \ln \pi(a|s) \tag{11}$$

where Equation 9 is the special case when $b = 0$. It is possible to add this term because it does not change the expected value of the estimation of ∇J, the proof is given in [29]. What the baseline can change, is the variance of the update. One can use multiple different versions of baseline functions. One example is the mean of the returns:

$$b := \frac{1}{N}\sum_{i=0}^{N} G_t \tag{12}$$

As stated in Barto & Sutton, the estimated value function $V(s)$ is also a natural choice for the baseline. In this case, the update function becomes the following:

$$b(s) := V(s) \tag{13}$$

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + (G_t - V(s))\nabla \ln \pi(a|s) \tag{14}$$

2.8 Actor-Critic Method

Algorithms using parameterized policies from Section 2.5, as in the policy gradient methods, is called Actor-only methods. When using only the value function approximations from Section 2.6, it is called Critic-only methods. Combining these methods gives us Actor-Critic methods. The naming convention is as defined in [18]. The aim of these methods is to combine the strong points of both Actor-only and Critic-only methods. Here, the critic is learning an approximation of the return, and then using this to update the policy approximation of the actor. As characterized by deep reinforcement learning, the Actor and Critic are parameterized by neural networks with parameters Θ and Θ_v respectively. When discussing baseline, the possibility of using the value function as a baseline was mentioned, thus using both a parameterized policy and a value function. But as given in [29], this is not seen as an Actor-Critic Method, as it is only using the estimated value as the baseline, and not as the estimated return. The REINFORCE algorithm can be turned into an Actor-Critic method. Starting with the policy update in equation 14, the one step Temporal-Difference-error is used instead of G_t, the discounted return:

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + \delta\nabla \ln \pi(a|s) \tag{15}$$

$$\delta = R_t + \gamma V(s') - V(s) \tag{16}$$

Here, R_t is the reward, γ is the discount factor, and s and s' is the current and next state. δ is recognized as the TD-error from Equation 5, and since the TD-error is defined by the value-function V(s), it is an Actor-Critic method. The value function approximator is updated by minimizing the TD-error using gradient descent.

2.9 Asyncronous Advantage Actor-Critic Method

The Asynchronous Advantage Actor-Critic method, popularly called A3C, is a highly computational efficient way of utilizing the Actor-Critic method [21]. The efficiency comes from using separate learner agents executed in parallel. Each agent runs its own version of the environment and its own copy of the Actor-Critic neural networks. Each agent asynchronously updates a global network, using stochastic gradient descent optimization methods. Every time the agent finishes an episode, it will overwrite the local version of the network with the global version. The agents now have the ability to independently explore the environment, giving a more diverse training data. Since the processes are parallel, distributing the work on multiple CPU cores or on the GPU will decrease training time.

2.9.1 The Advantage

In Section 2.8 on the Actor-Critic method, it was explained how the combination of the Policy and state-value function was utilized. In A3C, the actor is still represented by the policy $\pi(a_t|s_t;\theta)$, but the critic is instead an estimate of the advantage function $A(s,a;\theta)$ [21]. The advantage function is defined as

$$A(s_t,a_t) = Q(s_t,a_t) - V(s_t) \tag{17}$$

This is an evaluation of the advantage in performing an action, the difference between the action value function $Q(s_t,a_t)$ and the state-value function $V(s_t)$. By inserting the definition of the action-value function from Equation 3, we can represent this by

$$A(s_t,a_t;\theta) = \sum_{i=0}^{k-1} \gamma^i r_{t+1} + \gamma^k V(s_{t+k};\theta) - V(s_t;\theta) \tag{18}$$

Here $V(s;\theta)$ is the neural network approximated state-value function with parameters θ. Using the definition for return from Equation 1, the advantage function formulation can be simplified to

$$A(s_t,a_t) = G(s_t,a_t) - V(s_t) \tag{19}$$

The objective function for the actor is then represented as the following:

$$J(\theta,\theta_v) = \ln \pi(a_t|s_t;\theta)(G_t - V(s_t;\theta_v)) \tag{20}$$

Practically, the representation of the policy π and value function V can be represented by two separate neural networks with parameters θ and θ_v. It is also possible to use one neural network for both functions, with a shared input and initial layers, but is later split to have separate output layers. For more on neural networks, see Section 2.10.

2.9.2 Entropy

To help the exploration when following a certain policy, we can make it add a higher measure of stochastic in action selection. This can be done by using the *entropy* of the policy π, as done in [21]. For this, we introduce the Shannon entropy $H(\pi)$ given as

$$H(\pi(a_t|s_t;\theta)) = -\sum_{i=1}^{M} p(a_i|s_t;\theta)\log p(a_i|s_t;\theta) \tag{21}$$

where $\pi(a_t|s_t;\theta) = [p(a_1|s_t;\theta), ..., p(a_m|s_t;\theta)]$ for a set of M actions, and $p(a|s;\theta)$ is the probability of action a given state s and parametrers θ. The Shannon entropy is maximized for a policy where the probability of a given action comes from a uniform distribution, since this is the least deterministic behavior. The entropy $H(\pi(a_t|s_t;\theta))$ can then be included in the objective function of the actor from Equation 20:

$$J(\theta,\theta_v) = \ln\pi(a_t|s_t;\theta)(G_t - V(s_t;\theta_v)) + \beta H(\pi(a_t|s_t;\theta)) \tag{22}$$

where β is the hyperparameter dictating how much to weigh the entropy. For an optimal β, the policy solution should be able to leave local minimums and explore, while being able to converge to an optimal solution.

2.10 Neural Networks

Artificial neural networks have since Hinton et al. wrote *A fast learning algorithm for deep belief nets* [11] revolutionized the world of machine learning, even though the idea of these networks has been around a long time. It is used as a function approximator, to mimic some function or model mapping an input to the desired output. It has surpassed other models in both image recognition, speech recognition, and reinforcement learning. The latter was shown as mentioned with the breakthrough of DeepMind, where an agent learns to play Atari games only from visual pixel input [22]. The idea behind artificial neural network has been around since the 1940s and was from the start a way of trying to mimic how the brain neurons function. The brain uses extremely large connected networks of neurons to process the vast information available to us. Each neuron gets input from other neurons and fires an output signal itself if the sum of input is above some threshold [12]. The artificial neural network is a simplification of what can be seen in the

brain. The basic consists of a summation of the inputs and the output is an activation function on that summation, as can be seen in Figure 5. Each signal between the neurons is multiplied by a weight of w. These weights can be tuned to fit different functions. The neurons can be stacked together in both width and layers, creating large interconnected neural networks. In Figure 6 a small artificial neural network with three inputs, one hidden layer with four neurons, and two outputs can be seen. What architecture to choose depends on the problem at hand. For instance, if the problem is complex, a larger number of neurons is needed to model the problem, than if the problem is simple. The *universal approximation theorem* states that a sufficiently large neural network actually can approximate any function mapping a finite dimensional discrete input to finite dimensional discrete output, as explained by Ian Goodfellow et al. in [10]. However, as also explained in [10], even though a neural network can represent any function, it does not mean that it is possible for the neural network to learn it.

2.10.1 Learning in a neural network

To be able to model a system using a neural network, one needs a training set of input with corresponding target outputs. Learning starts out by having a cost function that one want to minimize. This cost function is, for example, the squared error between the model output $\hat{y}$ and the target output y.

$$E = \sum(\hat{y} - y)^2 \tag{23}$$

With backpropagation as explained in [10], the weights can be updated to minimize the cost function using gradient descent. Gradient descent is explained in Section 2.11.

2.10.2 Activation functions

The important part of the activation function is that it is non-linear. This is important, as a neural network with only linear activation function, only can model linear functions, see [10]. Three common activation functions is the rectified linear (relu) function:

$$g(z) = max(0, z) \tag{24}$$

the softplus function:

$$f(z) = \ln(1 + e^z) \tag{25}$$

and the sigmoidal function:

$$\sigma(z) = \frac{1}{1 + e^{-x}} \tag{26}$$

A plot of the sigmoid and the relu functions is seen in Figure 3 and the softplus and the relu in Figure 4.

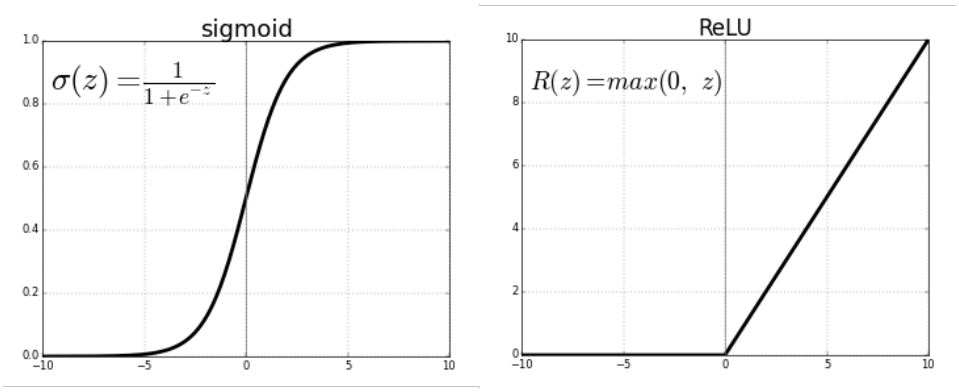


Figure 3: Plot of the sigmoid function and rectified linear (relu) function.

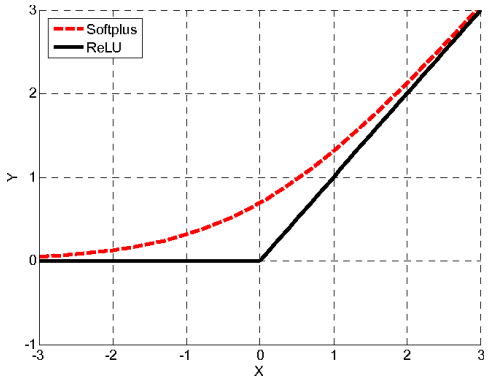


Figure 4: Plot of the softplus function and rectified linear (relu) function.

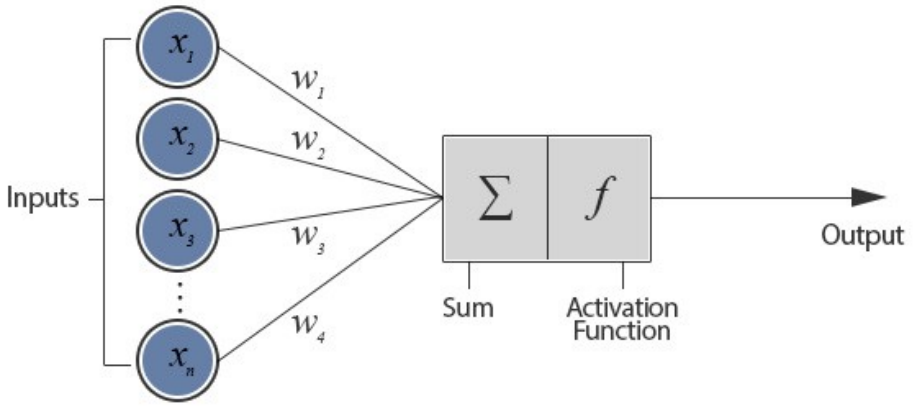


Figure 5: Model of a neuron, with inputs, summation and activation function.

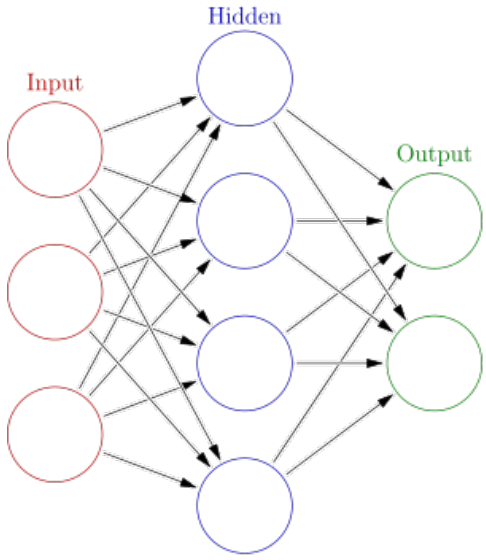


Figure 6: A small artificial neural network with three inputs, one hidden layer with four neurons, and two outputs.

2.10.3 Regularization: Dropout and L2

When training a model with a neural network, the interesting part is how the model performs on new, unseen data. The ability to perform well on these data is called *generalization* [10]. If our model is to generalize well, overfitting must be avoided. (See chapter on overfitting in [10]). This can occur when our model is trained too long on one set of data and becomes modeled especially to that set. Neural networks are complex models. If the network is capable to model something of a higher complexity than the actual model itself, it is also prone to overfitting ([10]).

Regularization techniques is applied to the network to battle overfitting. The L2-regularization (see [10]) is a weight decaying teqchnique. By adding a term with the L2 norm of the weights to the cost function in Equation 23, a lower absolute value of the weights is expected:

$$E = \sum(\hat{y} - y)^2 + \lambda|w|^2 \tag{27}$$

The λ is a hyperparameter chosen in advance to weight the importance of small weights w.

Dropout [10] is a technique meant to achieve the same as *bagging*. That is, training multiple models and use the average result in hopes for a better generalization. In dropout, each neuron in a layer of the network has a probabilistic chance of being taken out for that epoch of training. The network now only consists of the neurons not taken out, and a different subset of the network will then be trained

in each epoch. This forces the network to not rely on only some neurons, and each neuron will be forced to participate in modeling.

2.11 Gradient descent optimizer

Gradient descent is the most common way to optimize neural networks. Through backpropagation the gradient of the cost function w.r.t. the weights θ can be obtained. These gradients are then used to update the weights θ to find the minimum of the cost function. In reinforcement learning the problem is often formulated as a gradient ascend problem, where the goal is to find the maximum of some performance measure function. However, exactly the same algorithms can be used in both cases, as the maximum of the function J is the same as the minimum of $-J$.

2.11.1 Batch gradient descent

The most straightforward way of updating the weights of a neural network is the vanilla gradient descent, also called batch gradient descent:

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t - \alpha \nabla J \tag{28}$$

Here α is the learning rate, a scaling of how much to update the weights, and ∇J is the average gradient of the cost function w.r.t. the parameters θ for the whole training set. This is computationally costly, especially memory wise, as for each update all of the gradients must be calculated. On the other hand, a more consistent gradient is obtained with batch gradient descent than with stochastic gradient descent. It is also shown to have better convergence properties [10]

2.11.2 Stochastic gradient descent

Instead of calculating the gradients for the whole training set, it is with stochastic gradient descent (SGD) possible to update the network for each training sample.

$$\boldsymbol{\theta}_{t+1} := \boldsymbol{\theta}_t + \alpha \widehat{\nabla J} \tag{29}$$

The variance of the updates will fluctuate more than for batch gradient descent. This means that it is possible for SGD to jump to a different and possibly better local minimum than where it started, something that will not happen with batch gradient descent.

2.11.3 Minibatch gradient descent

Trying to merge Batch and Stochastic gradient descent, Minibatch gradient descent tries to minimize the variance as in Batch gradient descent, while utilizing the efficiency of still being SGD. Instead of training on the whole data set, a smaller batch size of n elements is chosen. Common batch sizes are between 50 and 256.

2.11.4 Moment

There are multiple challenges with SGD. It is very important to find a suitable learning rate. A small one will slow down the learning, and a too large one might hinder convergence. With SGD, the same learning rate is applied to all the weight updates. Since some weights change less often than other, it is often smart to apply a larger update step to those. Another challenge is getting stuck in a non-optimal local minimum. As discussed in Dauphin et al. [5], the largest problem might in fact not be local minima, but saddle points. They are often followed by a plateau, that makes progress hard as the gradients go towards zero.

A way to battle this is with momentum [26]. It helps to accelerate the SGD in the relevant direction by using the previous update step in addition to the current one. The term momentum comes from that it is closely related to momentum from physics. It can easily be visualized as a ball rolling down a hill. The ball starts rolling and builds up speed. If you try to apply a force on the ball to change its course, it will be hard because it still got the momentum from rolling down the hill. When you apply momentum to SGD, you apply a term looking back at the previous updates, and add a part of that gradient to the current update. Just as a ball would be able to roll over a flat surface using its momentum from the previous hill, the SGD with momentum would be able to overcome both saddle points and plateaus.

2.11.5 Adam optimizer

The Adam optimizer as proposed in [15] is utilizing both momentum and adaptive learning rates. The notation used is from [15], but changed to fit the previous notation in this paper. The update rule is given as the following:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \frac{\alpha}{\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon}\hat{\boldsymbol{m}}_t \tag{30}$$

Here $\hat{\boldsymbol{m}}_t$ and $\hat{\boldsymbol{v}}$ are both vectors and can be interpreted as the mean and the variance of the moment. As can be seen in 30, the update term is divided on the variance $\hat{\boldsymbol{v}}$. This results in that the weights that are updated less get a boost compared to the weights that are updated often. This is what is called adaptive learning rates. The moments are both exponentially decaying based on previous moments as given in 31, added with the current gradients g_t w.r.t. the network weights θ. β_1 and β_2 is the decay rates of the previous moments. Both m and v is initialized to zero, and it is proved in [15] that they are biased towards that initial value. A bias correction term is therefor added as in Eq. 32, and this gives us $\hat{m}_t$ and $\hat{v}$.

$$\begin{aligned} \boldsymbol{m}_t &= \beta_1 \boldsymbol{m}_{t-1} + (1-\beta_1)\boldsymbol{g}_t \\ \boldsymbol{v}_t &= \beta_2 \boldsymbol{v}_{t-1} + (1-\beta_2)\boldsymbol{g}_t^2 \end{aligned} \tag{31}$$

$$\begin{aligned} \hat{\boldsymbol{m}}_t &= \frac{\boldsymbol{m}_t}{1-\beta_1^t} \\ \hat{\boldsymbol{v}}_t &= \frac{\boldsymbol{v}_t}{1-\beta_2^t} \end{aligned} \tag{32}$$

2.12 Bezier Curve

Bezier curves ([13]) is a type of parametric function. It is heavily used in computer graphics and design. To a computer, straight lines are easy to represent as functions, but to mathematically represent complex curves can be unfeasible. The Bezier Curve, named after Pierre Bezier, is a way of representing curves using straight, easily represented lines, see Figure 7. For a mathematical definition, see the source [13].

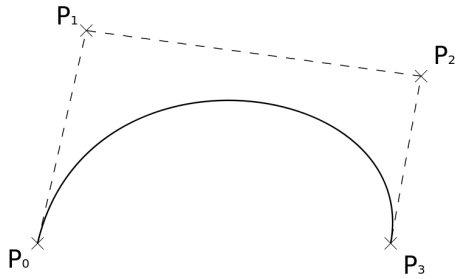


Figure 7: Example of a curve defined by three other lines

2.13 Ackerman steering

Ackerman steering [14] is a way to link the wheels of a vehicle to reduce the sideway slip. It was invented already in 1817, and used on horse-driven carriages. The geometrical solution, as shown in Figure 8, is to give the wheels different turning radius to correct for the fact that they are placed at a different length away from the turning point of the car.

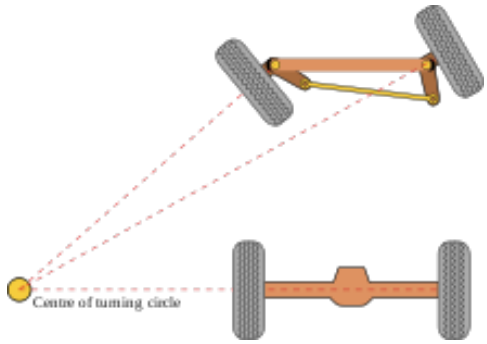


Figure 8: A figure demonstrating the different turning angles of the wheels used in Ackerman steering

3 Guidance, Navigation and Control with Reinforcement Learning

This section will explain how reinforcement learning can be implemented in the context of guidance, navigation, and control.

A motion control system can often be split into three parts denoted the *guidance*, the *navigation* and the *control* system. [9]. These are shown in Figure 9. The guidance part deals with the system that computes the reference signals used. This can be the desired position and velocity. In our path-following case, it gets data from a path planner about the desired path and calculates the reference signals. Navigation is the system that determines the position, attitude, course, and distance traveled. This can be done by using a global navigation satellite system in addition to motion sensors as accelerometers and gyros. The control module is the part responsible for determining the necessary control forces to follow the objective set by the guidance system. This is where path-following algorithm such as Line-of-Sight (LOS) or other objectives can be implemented. As we can see in Figure 9 the estimated positions and velocities from the observer are taken into the control system, so that it is able to implement feedback control laws. The output from the control is then applied to the vehicle to be controlled. The figure shows that there is a feedback signal to the guidance part. This makes the whole system a closed-loop guidance system since it makes use of the measured positions and velocities. Even though the paths are static, feedback can be used to for example avoid obstacles based on the current positions [3]. Systems without this feedback become an open-loop guidance system as it only uses the feedforward reference.

The motion control task of this thesis is path-following of an underactuated vehicle and underactuated marine craft. To understand how these systems are modeled, we need to explain some terms [9]. A *configuration space* is the n-dimensional space of which position and orientation a vehicle or craft can attain. It is also called the degree-of-freedom (DOF) of a system. The vehicle is modeled with 6 DOF $\eta_v = [x, y, z, \phi, \theta, \psi]$, which is the position in x, y and z-direction, and roll,

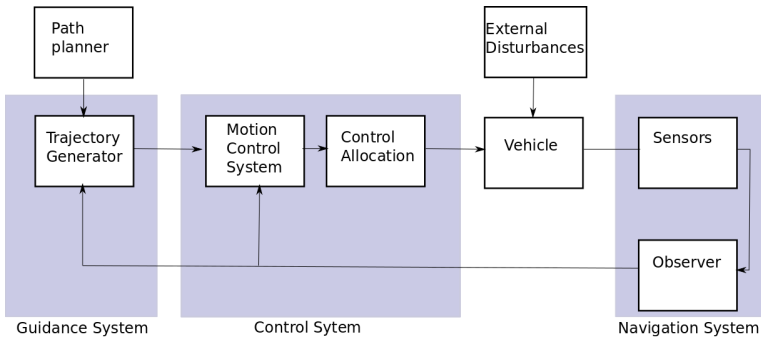


Figure 9: Illustration of a conventional of a guidance, navigation and control-system.

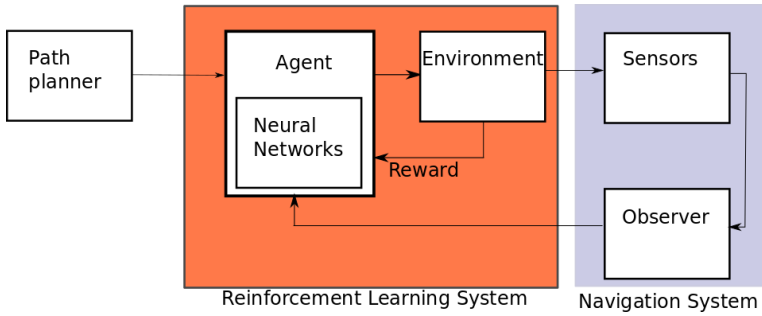


Figure 10: A motion control system using Reinforcement Learning.

pitch, and yaw respectively. We are here following the notation from [24] for marine vessels. The marine craft has 3 DOF, $\eta_m = [x, y, \psi]$, that is position in x and y-direction, and yaw angle ψ. The *workspace* of a system is the reduced m-dimensional space where m $<$ n. As we later describe, the workspace of our two systems is $m = 1$, since we are only controlling the yaw motion.

Instead of using the classical approach to path-following with a method as LOS [9], this thesis develops a reinforcement learning-based methodology to control the vehicles under consideration. In Figure 10 it is shown how the original guidance, navigation, and control-system from Figure 9 can be altered to include a reinforcement learning system. As can be seen, the reinforcement learning system has taken the place of the guidance and the control systems. The sensor data is used together with the path planner to create the states of the reinforcement learning system. These are fed to the learning representation, in this case the neural networks, to get actions to apply to the environment. An important aspect is that the reinforcement learning agent does not care about what kind of vehicle or ship model it is controlling, or how external forces are affecting the model. It could for example learn to steer a ship on a path and compensate for the current, without knowingly do so. It learns the complete model of the ship *and* the environment around it, without discriminating between the two.

4 Implementation

This section will explain the vehicle and the ship model used to test the learning algorithm. We explain the implementation details about the A3C algorithm, the paths and the simulator used. The experiments were done using the V-REP simulator setup explained in Section 4.8.

4.1 The vehicle

The vehicle used in the simulator can be seen in Figure 11. The vehicle is called Manta and comes natively with the V-REP simulator. The Manta is a relatively complex car model, having implemented both suspension and Ackerman steering. This gives us a realistic vehicle to use for our experiments. The speed is indirectly controlled by setting the torque on the motor, so by trial and error, an appropriate torque was found to minimize loss of traction, while still giving a quite high resulting speed.

- Maximum torque of the motor: 60 [Nm]
- Wheel radius: 0.09 [m]
- Width between wheels: 0.35 [m]
- Length between wheels: 0.6 [m]

For the experiments, the motor torque of the vehicle was set constant. This gives a maximum speed of 3.5 m/s. Vehicle settings:

- Maximum steering angle: 30°(turning radius: 1.2 [m])
- Constant torqe: 30 [Nm]
- Maximum speed: 3.5 [m/s]

4.2 The Mariner ship model

The ship model [4] is implemented in the Marine Systems Simulator (MSS) matlab toolbox [8]. It was translated to Python by Andreas Bell Martinsen. It is a classic ship model used extensively in the Marine Craft handbook by Fossen [9].

- Length: 160m
- Maximum rudder angle: 40°
- Cruising speed: 7.7 m/s

The equation of the mariner model is given in appendix A. The Mariner model was simulated by its equations directly in the python code using basic Euler integration.

The Manta car model

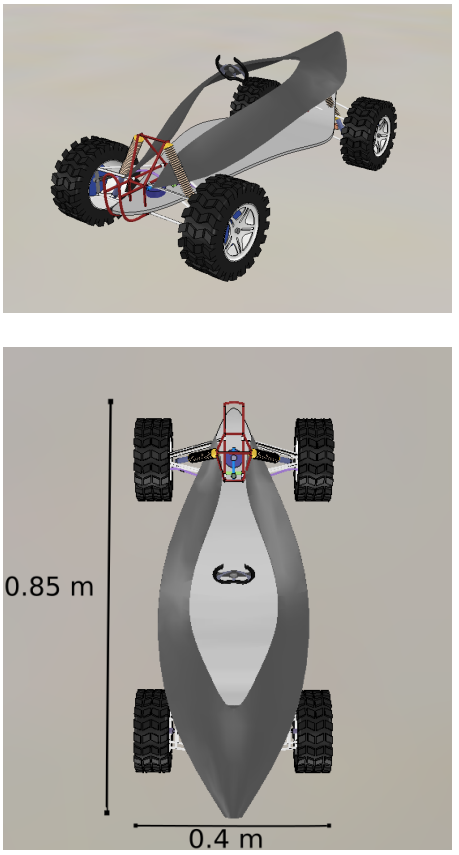


Figure 11

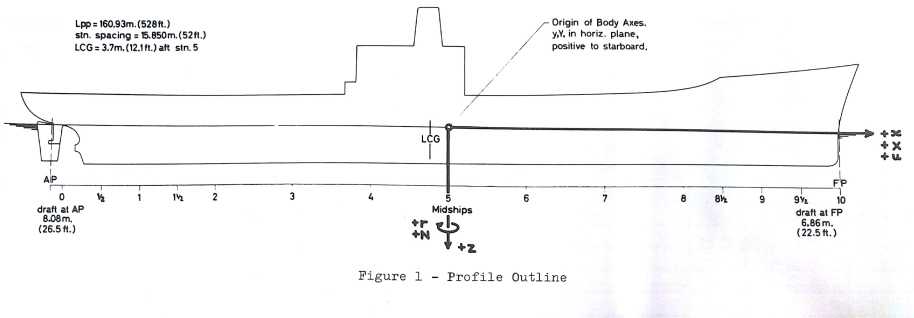


Figure 12: Illustration of the mariner model from the original paper [4]

4.3 About the models

The Manta vehicle model was the main model when starting this thesis. It is complex, and since it must be simulated in V-REP it is computationally heavy to test on. What we wanted to do, was to make a simple mathematical model to see if we were able to transfer some of the learning from the computationally light mathematical model to the Manta vehicle. As the Institute of Cybernetics in Trondheim is very involved in marine technology, we have been introduced to a wide variety of ship models during the years. We were interested if we could find some similarities between the control of the ship and control of the car model.

An interesting aspect of the models is testing whether or not the learned dynamics from the ship can be used to control the vehicle. Therefore, it was focused on making the dynamics of the two as equal as possible. The size difference between the two types of models is large, as the ship is about 200 times the car. The path to follow was therefore scaled up 200 times when training on the ship. However, the cross track-error state had to be scaled down again before sent to the neural network, for it to be indifferent to which model it was training on. The cross track-error is the only state affected by the scale. More on the states in Section 4.11.

The speed of the ship is however not 200 times the speed of the vehicle. To approximately make the dynamics equal, that is, make the actions on the ship have the same impact as actions on the vehicle, we scale the simulation time step of the ship simulation. The speed of the car is approximately 3.5 m/s, a little less than half of the speed of the ship. Since the ship should travel 200 times longer than the car after the scaling of the path, the time step is set 100 times longer than for the car. From testing, we found that the minimum timestep we could have on the car was 0.1 seconds. Less than this, and the time per simulation became too high. The V-REP recommended timestep is 0.05 second. We therefore ended up with a timestep of 0.1 seconds on the car, and 10 seconds on the ship.

4.4 The path representation

Several paths were created to train the agent, and two is presented here. The paths were created inside the V-Rep simulator environment, by utilizing a path creation tool. The paths are created by setting waypoints and then producing the smooth bezier-curve (Section 2.12) of these points. Plots of the resulting paths can be seen in Figure 14. The paths have increasing difficulty. As can be seen in Figure 13, each bezier-point includes an orientation, which is aligned with the curve tangential angle of the path. Therefore, both position and angle can be extracted directly from the path points. Defining the paths this way, compared to creating them by defining mathematical curves, makes it easier to make and tweak the paths. It is also more probable that this is how paths in a map program would be defined. However, it poses some practical problems when calculating the information needed for the car to follow. These are explained in Section 4.10.

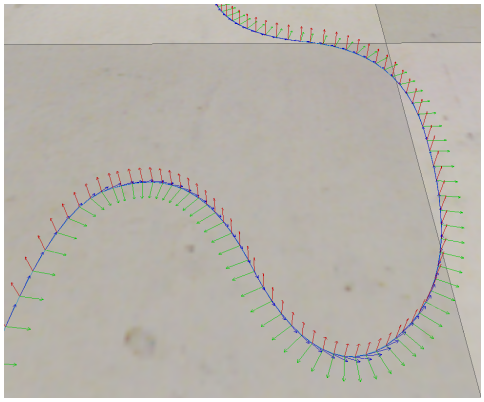
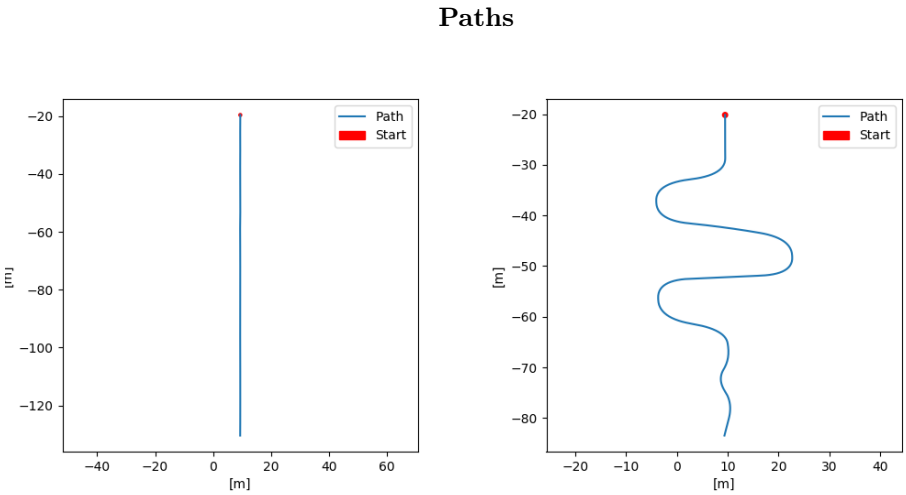


Figure 13: Image of how the points of the bezier curve looks. Each point includes orientation, which here equals the path tangential angle.



(a) (b)

Figure 14: The different paths used.

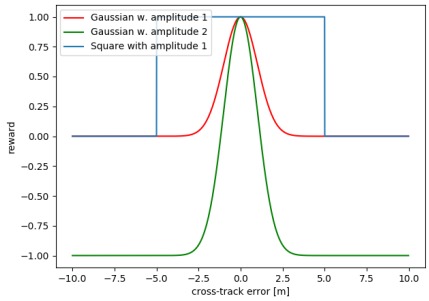


Figure 15: Reward functions with respect to cross-track error, given that the car is facing the right way down the path. There is two Gaussian distributions, and one uniform distribution.

4.5 Reward function

The reward function, defining the return as explained in Section 2.4, is a very important and difficult part of any reinforcement algorithm. This is where we have to define to the agent what we want to achieve, by setting a reward for certain behavior.

The reward functions for the path following case is made to be as simple as possible, as it is deemed as important to avoid the unforeseen behavior. By setting the motor torque constant, and only controlling the steering of the vehicle, it is possible to build a reward function based on the cross-track error of the vehicle with respect to the path. If we were to include control of the speed of the car to be learned, we would probably also have to include path progress as an element in the reward function.

The reward functions used is as shown in Figure 15, but only given that the difference in angle α between the heading of the car and the path tangential angle satisfy $-90° < \alpha < 90°$. In other words, the car must face in the correct direction, heading down the path. Since the motor torque of the car is constant, we then know that the car will be progressing down the path.

4.5.1 Penalty on turning rate

When designing an optimal solution to a control problem, we have to ensure that the solution is feasible to execute for a realistic model. A solution that might perform well in a simulated environment, might not work well in a real-world application. In an optimal control setting, a solution will often involve excessive chattering in the steering control. In a real-world application, it would result in reduced durability.

The solution in reinforcement learning is to penalize overuse of the steering, and it must be reflected in the wanted behavior defined by the reward function. The proposed solution is to add a penalty p to the reward function defined by the

rate of change in the commanded steering given as the following equation:

$$r(t) = N(e(t), \sigma^2) + p(t) \tag{33}$$
$$p = -\beta \dot{a}(t) \tag{34}$$

$r(t)$ is the reward at timestep t, N is the Gaussian distribution with variance σ^2 as visualized by Figure 15. $e(t)$ is the cross track-error at timestep t. The penalty constant β is a hyperparameter to be set, scaling the influence of the penalty on the reward. $a(t)$ is the action at timestep t. The p term is expected to encourage the agent to have a smoother usage of the steering. Choosing the term β correctly is important since a too high value will probably discourage changing the steering angle altogether, while a too low value will probably give no effect at all. Different β must be tested.

4.6 The A3C algorithm implementation

The A3C algorithm implemented in this task is built from the actor-critic algorithm from the project thesis which preceded this master thesis. The formulation of the algorithm 1 is taken from the original paper [21]. The algorithm describes one parallel learner agent. Each agent has a local step counter t, and share the global step counter T. In the implementation, the entropy regularization is also used, but as in the original paper, it is omitted in the algorithmic formulation.

Each learner synchronizes their local copy of the global network at the start of each episode. The episode is t_{max} steps. After each episode, the accumulated gradients of the weights are applied to the global network asynchronous. This is done via the Hogwild-approach [27], a method used for parallelized stochastic gradient descent. The approach is based on that multiple workers can update a shared memory, in this case, the weights of the network, without locking it. It runs the risk of having race conditions and overwriting other updates, but it is proved that it is not only fast for the program to run, but also mathematically efficient. The algorithm runs until the sum of timesteps run by all workers exceeds the number of maximum timesteps T_{max}.

The environments dealt with in this thesis has both continuous action space and state space. The continuous state space can be used just as a discrete states as inputs to the neural networks, but a continuous action space is more challenging for the reinforcement algorithm and must must be treated different. The challenge is to implement exploration of the environment when an action is chosen. If the actions is discrete, that is given a finite number of possible actions, a certain probability to be chosen can be given to each action. In the continuous case, there is infinitely many actions. One solution is to partition the action space to make it artificially discrete, but this limits the accuracy of the actions. Instead, we use a different solution. As proposed in [29], we introduce a random noise which is applied to the continuous action. A gaussian distribution was used to model the noise, as

Algorithm 1 Asynchronous advantage Actor-Critic for continuous actions with batch training

Initialize global actor and value network with weights θ and θ_v
Assume thread specific network weights θ' and θ'_v
repeat
    Reset gradients: $d\theta \leftarrow 0$, $d\theta_v \leftarrow 0$
    Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
    $t_{start} = t$
    Get state s_t
    repeat
        perform a_t according to policy $\pi(a_t|s_t;\theta')$
        Receive reward r_t and a new state s_{t+1}
        $t \leftarrow t + 1$
        $T \leftarrow T + 1$
    until terminal s_t or $t - t_{start} == t_{max}$
    $R = V(s_t, \theta')$
    for i in $\{t-1, ..., t_{start}\}$ **do**
        $R \leftarrow r_i + \gamma R$
        Accumulate gradients: $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s|i;\theta')(R - V(s_i;\theta'_v))$
        Accumulate gradients: $d\theta_v \leftarrow d\theta_v + \frac{\partial (R - V(s_i;\theta'_v))^2}{\partial \theta'_v}$
    Update θ and θ_v using $d\theta$ and $d\theta_v$
until $T > T_{max}$

proposed by [33]. A Gaussian distribution is given in Equation 35.

$$N(x|\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{35}$$

Here μ is the mean of the distribution, σ^2 is the variance and π is in this case the mathematical constant. Here the output of our policy function approximator is used as the mean of the distribution, and the Gaussian noise is added to ensure exploration. How to choose the variance of the noise is an open topic. It is possible to set it to a decaying or constant variable, as proposed by [33], or to try to learn it by parameterizing it as a part of the neural network. The latter was chosen and is explained more in Section 4.7.

4.6.1 Using Python and Green Threads

It is important to address the use of Python in the implementation of the algorithm. Python is the most popular programming language for implementing machine learning algorithms [34], much because of its simplicity and readability, and was therefore chosen for this project. It was the programming language I had the most previous experience with, and it gave the possibility to reuse software made during the project thesis. However, using Python also poses some challenges. Python has its own library implementing multithreading, *threading*, but unlike languages like C++, the threads cannot run in true parallel [25], [30]. This type of threads are called green threads and is only run on one processor core. The A3C algorithm is made specifically to be run in parallel and utilize the multiple cores of the CPU and is of course hindered by this as shown in the results section. But as we have earlier mentioned, the advantage of this algorithm is not only the parallel part but also that it is mathematically efficient. So the results will be the same as if the algorithm was run in true parallel, but it will be slower. This does however not so much affect the running of simultaneous V-REP simulations. These are separate processes and will benefit by speeding up the algorithm by running in parallel. Other options were also explored. Python has a library called *Multiprocessing* which enables the use of multiple processor cores. The difference from threading is that this library creates a completely new process with its own private memory. This makes sharing objects hard, and to us, it makes the update of the global networks complicated to implement. The solution was to implement the algorithm with the *threading* library, and settle with the longer simulation speed.

4.7 The Neural Network implementation

The neural networks in this implementation are made with the Tensorflow library [31]. Tensorflow is an open-source machine learning software specially designed for making neural networks and is originally developed by the Google Brain team. Tensorflow enables you to make neural networks by setting them up as computational graphs, separate from the programming language used to implement the algorithm. This makes it highly efficient because even though you might use a

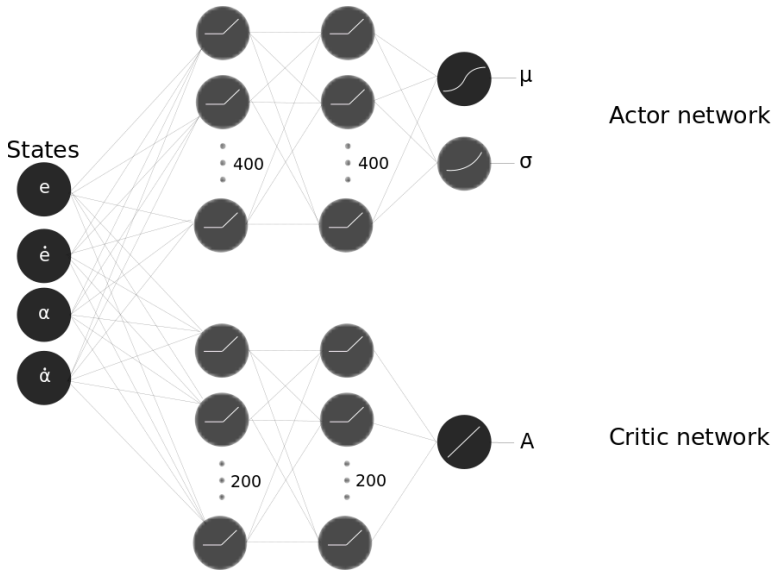


Figure 16: A visualization of the neural network used by the A3C algorithm. It is separated in an actor and a critic network, with identical input. The actor network outputs an Gaussian distribution, and the critic a value estimation. Each node visualizes which activation function is used: ReLu, tanh, softplus and linear.

slower language like Python, the neural network and the computation required is implemented in C++.

The actor and the critic are implemented as two separate neural networks, as shown in Figure 16 The input to each network is identical and is the states defined in Figure 4.11. In the original A3C paper [21], a different architecture is used. Here they implemented both the actor and critic as one network. The first layers are here shared and then split up so that each has one layer separate before the output. As the actor and critic network in our experience is likely to have to learn some of the same features from the environment, it could be efficient to share the first layers. However, it does presume and expect this to be the case without making grounds for it. Having two networks is a more general solution, and is therefore implemented.

The critic network is chosen to be a two-layer deep network, with 200 nodes in each layer using the ReLu activation function. It has one linear output, which represents the value of a given state. The actor-network is also two layers, but have 400 nodes in each layer with the ReLu activation function. The number of outputs is proportional to the number of actions the agent needs. Because we are dealing with a continuous action space, each action is represented by a Gaussian distribution, that is, a mean μ and a standard deviation σ. Since we have a one-dimensional action space, the network has two outputs. The μ-output is truncated by the tanh-function, while the σ has a softplus activation function. The Adam

Parameter	Value
Learning rate actor network	0.001
Learning rate critic network	0.001
Number of hidden layers	2
Number of neurons pr layer actor	400
Number of neurons pr layer critic	200
Discount factor γ	0.95
Entropy β	0.01

Table 1: Hyperparameters for the neural networks

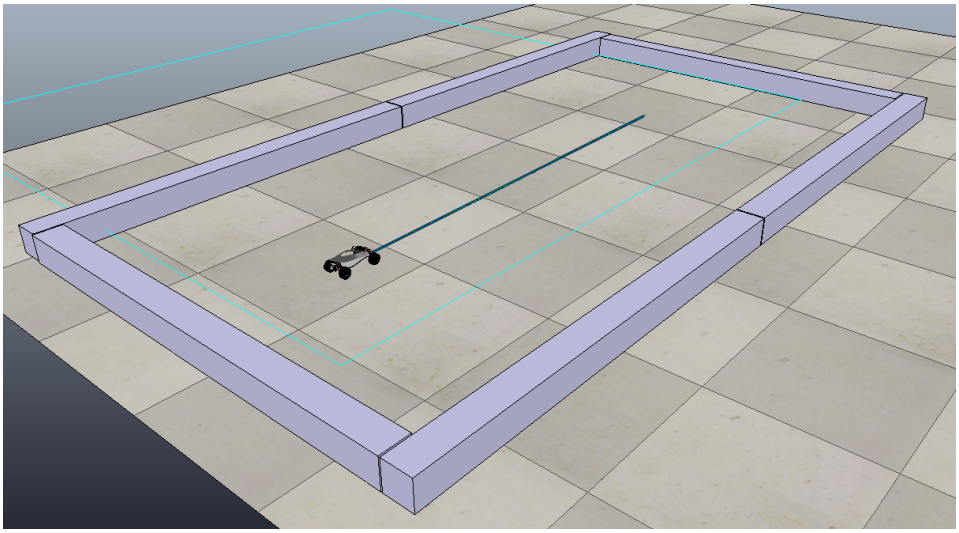


Figure 17: Overview of the V-REP simulator environment with the robot and a very short straight path.

optimization is used by the networks when learning.

The hyperparameters chosen is as given in Table 1.

4.8 System overview

This section aims at giving an overview of how different parts of the software system interacted. The two main parts are the reinforcement learning algorithm and the simulated environment. The reinforcement learning algorithm is the simplest, consisting only of the implementation of the algorithm. The environment part is split into several smaller parts, as shown in Figure 19. The most important is the simulator part, the remote API, Wrapper, and Gym environment is functioning as layers to interact with the simulator.

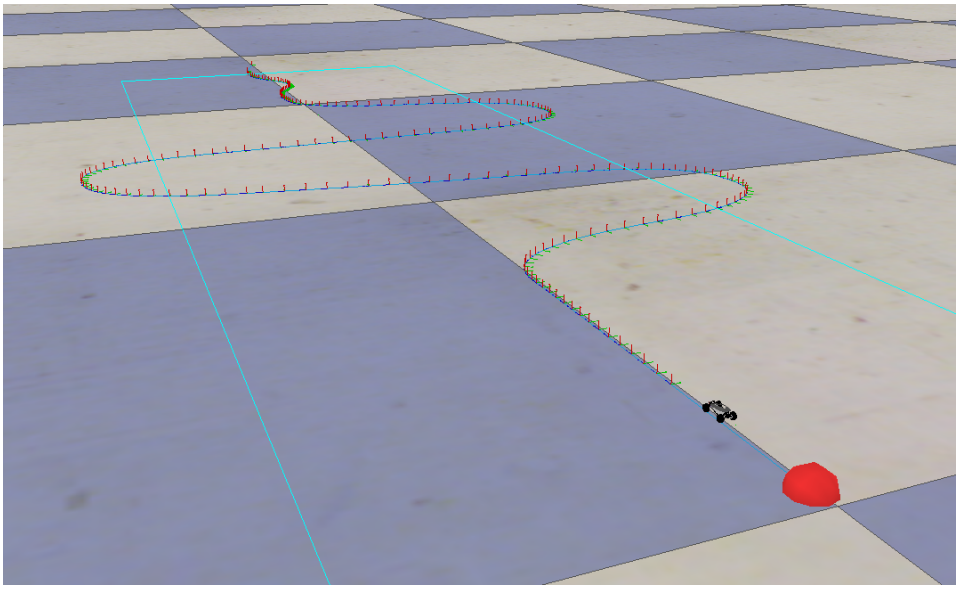


Figure 18: Overview of the simulator environment with the robot and a curved.

4.9 Simulator

4.9.1 V-REP

V-REP (Virtual Robotic Experimentation Platform) [7] is a simulation framework specifically built for robotic applications developed by Coppelia Robotics. It is a commercial product, but the educational version can be obtained for free. It is a feature-rich framework, that includes a scene and model editor, a large library of models and real-time mesh manipulation. V-REP has support for seven different programming languages, including C++, Python, and Matlab. The default language is however Lua. There is support for both local and remote control of a simulation in the V-REP environment. One can make scripts in Lua that is run inside the simulator, or one can use the remote API made to work with the other languages mentioned. You can then run your program separate, and call functions that will send messages over socket communication. Separate in this case can mean just a separate process or a separate computer. An important aspect of using this kind of communication is whether or not each call is *blocking* or *nonblocking*. A blocking call is when the remote program sends a message to the simulator, and then waits until it gets an answer before proceeding with the program. A non-blocking call would proceed before an answer is received. In our implementation, we need to know that simulating, reading sensors and setting joints happens sequential, so most communications are made blocking. E. g., it is important to know that the simulator is finished simulating one timestep before reading the states for the next.

4.9.2 The Wrapper and OpenAi Gym

The wrapper is an important part of the software. It facilitates the usage of the remote API of V-REP, for instance by wrapping the API functions in more intuitive objects, easily used in a higher level program. Each object controlled in the simulator can now be accessed by using this objects get- and set-functions, instead of calling the remote API functions directly with the object ID specified. The wrapper adds the following functionality:

- Start an instance of V-REP and set up communication on a free port
- Load premade scene
- Start/pause/stop and reset a simulation episode in V-REP
- Process return and error codes
- Print debug and logging messages
- Treat objects in V-REP (e.g. vehicles, robots) as objects in Python, simplifying reading sensors and controlling objects.

The OpenAI Gym-environments [2] is originally a collection of simulators created by the OpenAI team. However, it is possible to set up your own simulator as a Gym-environment. The environments are specifically made for interacting with a reinforcement learning agent, and therefore perfect for our task. Setting the simulator up, and being able to treat it as it as such, makes the simulator intuitive to use, and easy to reuse in future implementations of reinforcement-learning algorithms. Practically, an OpenAI Gym-environment sets a layout of function you must implement for the environment, and should work as a black box in interaction with the reinforcement learning agent.

4.10 Setup

For each episode, the vehicle is placed at the start of the path. To introduce some uncertainty, the initial position varies from 0 to 5 meters (0 to 100 for the ship) to each side of the starting point of the path, and in an angle from -45°to 45°relative to the tangent of the path. The reasoning behind this is that the vehicle is forced to deal with a randomness that will help it explore more of the environment. Even though the path might be straight, the vehicle must not only learn to keep the wheels straight to master the path, but also steer and converge to the path.

For each episode, one path is chosen to follow, either by random or not. The agent does not know which path but is only getting input from the world through the observable states. During these experiments, two different sets of states where tested.

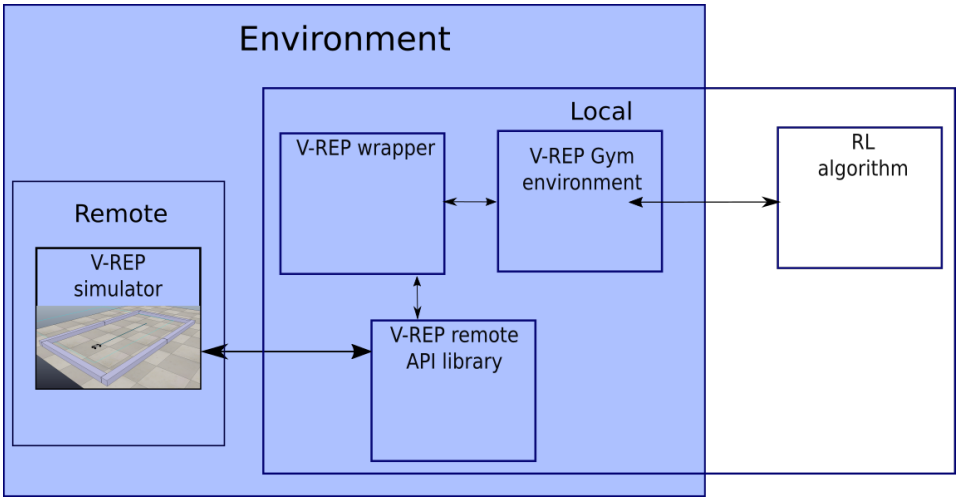


Figure 19: The connection between the software components, from the learning algorithm to the V-REP simulator. The colored area is highlighting what is hidden as the V-REP Gym environment, working as a back (blue) box.

4.11 The observable states

The observable states used was the cross-track error e, the difference in angle α between the heading of the vehicle and the line tangential to the path on the point closest to the vehicle on the path, and the derivative of these, see Figure 20. The states and their limits are shown in Figure 4.11.

States:	**Min**	**Max**
e [m]	-∞	∞
$\dot{e}[m/s]$	-∞	∞
α	-π	π
$\dot{\alpha}$	$-\infty$	∞

As explained in Section 4.4, the paths are not curves defined by mathematical functions but defined by its bezier-points. This is practical when creating and manipulating the paths, but gives us a lower precision when reading the information from the path. To minimize this precision problem, we do the following. For each timestep, the algorithm calculates the two closest bezier points on the curve with respect to the car. As explained in section 4.4, each point contains an orientation, which is aligned with the path tangential angle. This makes it easy to calculate an estimate of the difference in orientation between the car and the path. To get the cross-track error, it is possible to get an estimate just by calculating the distance from the car to the closest bezier point on the curve. This, however, would give you a low precision when the space between the points is significant, especially on the straight parts of the path. Even though the vehicle would follow the path perfectly, it would be reading a wave-like signal on the cross-track error, with the

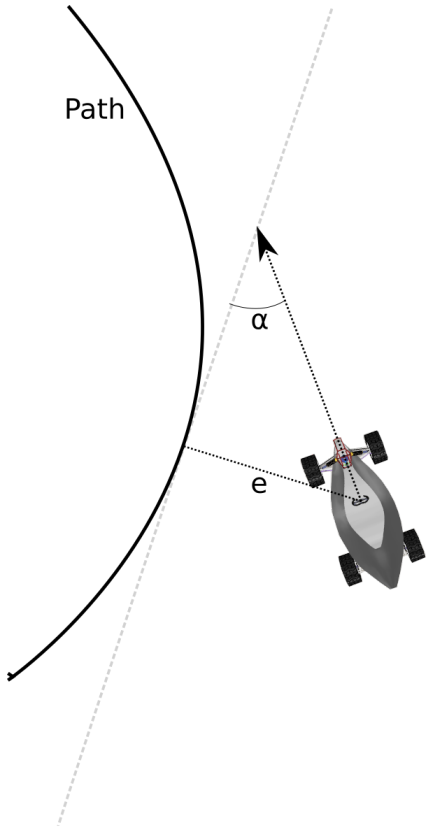


Figure 20: Figure of the cross-track error e and difference in angle α.

high points being when you are directly between two points on the path. Instead, we use the line defined by the closest point on the path and one of the adjacent points on the path, whichever is closest to the car. Using the distance to the car to this line gives us a smoother and more precise representation of the cross-track error. The formula for calculating the distance from a point $P_0 : (x_0, y_0)$ to a line $L : ax + by + c$ is defined as the following ([19], p 452):

$$distance(ax + by + c = 0, (x_0, y_0)) = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}} \tag{36}$$

Since we do not have the function for the line L, the definition can be rewritten to define the line by the two points $P1 : (x_1, y_1)$ and $P_2(x_2, y_2)$ that we already have:

$$distance(P_1, P_2, (x_0, y_0)) = \frac{|(y_2 - y_1)x_0 + (x_2 - x_1)y_0 + x_2y_1 - y_2x_1|}{\sqrt{(y_2 - y_1)^2 + (x_2 - x_1)^2}} \tag{37}$$

5 Results and Discussion

This section will present the results obtained when running the reinforcement learning simulations. It includes results from training on the Manta car vehicle and the Mariner ship model, and transfer learning from the Mariner to the car. At last follows a discussion around the time usage. All of the following results are obtained with a Gaussian reward function with amplitude = 2 (see Figure 15. The uniform reward function was not able to converge nearly as good as the Gaussian, while there was no difference in learning between different amplitudes of the Gaussian.

5.1 The Manta vehicle path following

The agent was able to learn to control the Manta vehicle to follow a predefined path. For a straight path, the learning process is shown in Figure 22. Here, the car starts at the top of the path and tries to follow it, and it is clear that it manages to learn the environment. As explained in the previous section, an initial offset of the vehicle is due to it being placed randomly around the start of the path to induce some randomness to the environment, and it can be seen that it is able to converge to the path anyways. The episode reward is plotted in Figure 21. Here the reward is scaled to have a maximum of 1 independently of how many steps the episode is. The figure shows that the reward has flattened out on about 0.8 after 400 episodes.

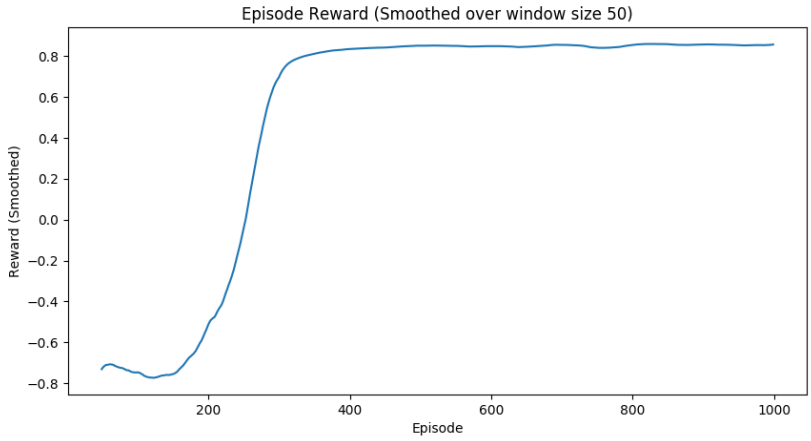
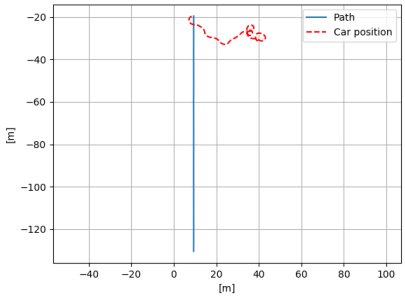
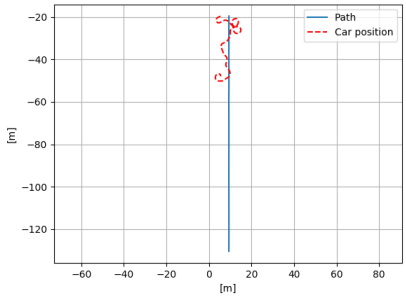


Figure 21: Training reward response when agent is trained from scratch on the displayed path

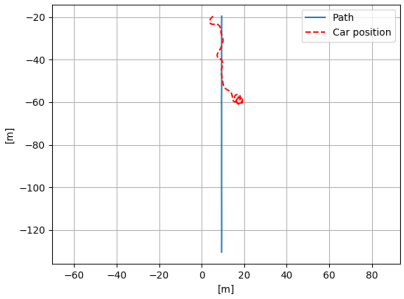
In Figure 23 the cross track-error, actual and commanded steering action, and the difference between path tangential angle and vehicle heading angle of an agent trained on the Manta vehicle are presented. The vehicle is performing well, as seen in Figure 23a, but the steering commands in Figure 23c is quite violent. Because



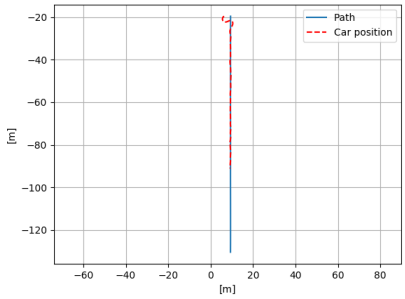
(a) After 1 episode



(b) After 120 episodes



(c) After 310 episodes

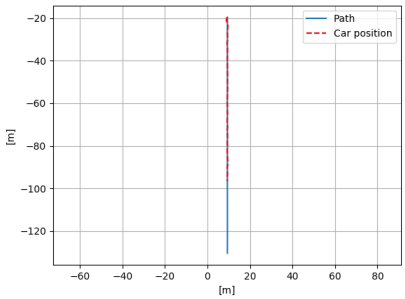


(d) After 501 episodes

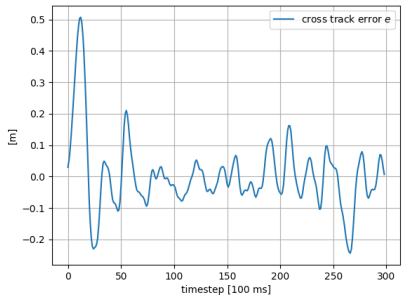
Figure 22: The progress of learning with the Manta vehicle on a straight path.

the vehicle cannot change the steering angle instantaneously, the actual steering is not as violent. The steering commands are a direct output from the actor neural network. There is, however, no penalty on the rate of change in the steering, so this result is not surprising.

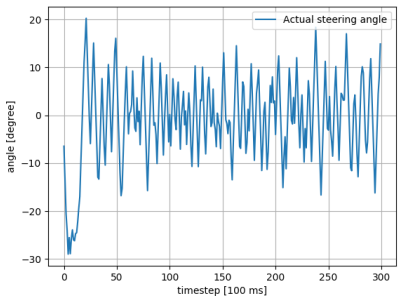
For a path with turns, the training response is comparable to the straight path, as can be seen in Figure 24. It shows that learning to follow a turning path takes almost twice as long as to follow a straight path and that it is not able to achieve as high a reward. The path and cross-track error of the curved path is plotted in Figure 25a and 25b. The cross-track error is clearly larger than for the straight path, and explains why the curved path is receiving less reward. In can be seen that the vehicle is not able to follow the path when turning sharply.



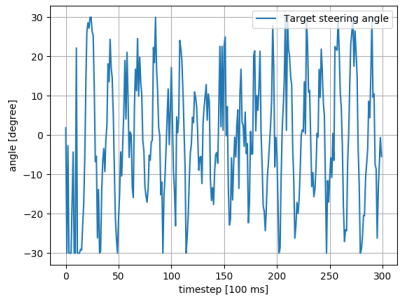
(a) Path of the vehicle



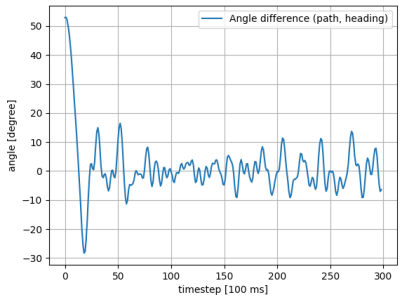
(b) Cross track-error



(c) The actual steering actions on the vehicle



(d) The steering commands to the vehicle



(e) The angle difference between vehicle heading and path tangental angle

Figure 23: The performance by the Manta vehicle of a finished training agent on a straight path.

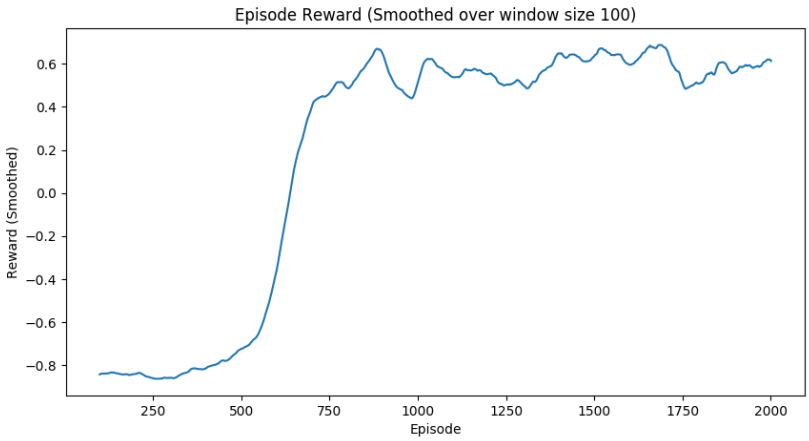
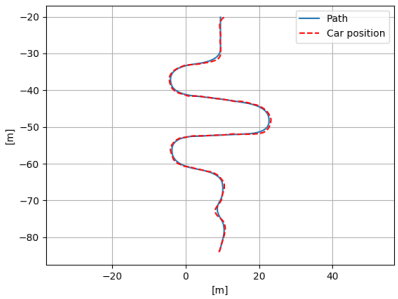
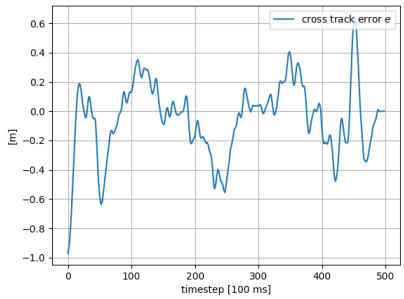


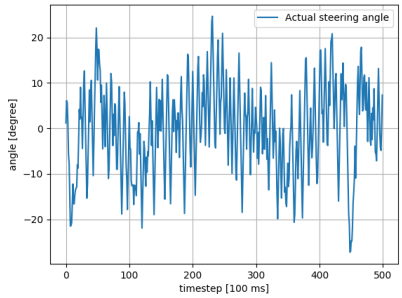
Figure 24: The reward response when training on the curvy path



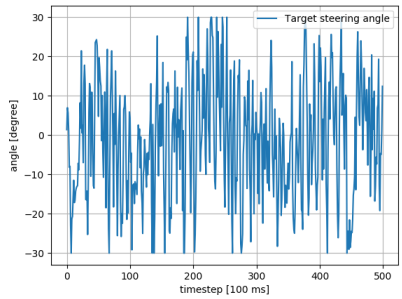
(a) Path of the vehicle



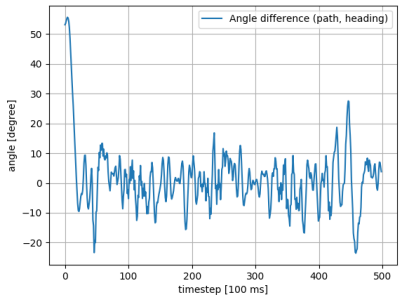
(b) Crosstrack-error



(c) The actual steering actions on the vehicle



(d) The steering commands to the vehicle



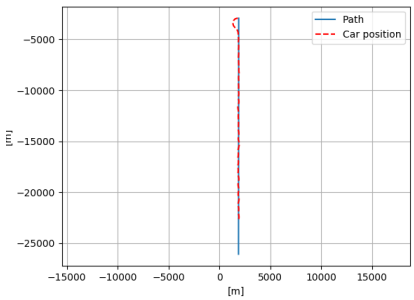
(e) The angle difference between vehicle heading and path tangential angle

Figure 25: The performance by the Manta vehicle of a finished training agent on a curved path.

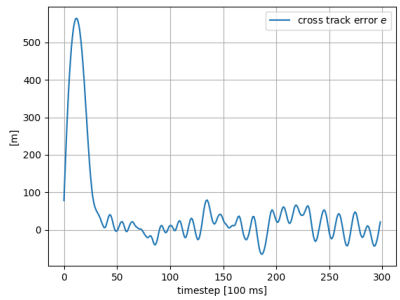
5.2 Mariner path following

Figure 26: The training reward of the Mariner model with 10 [s] time step on the curved and the straight path.

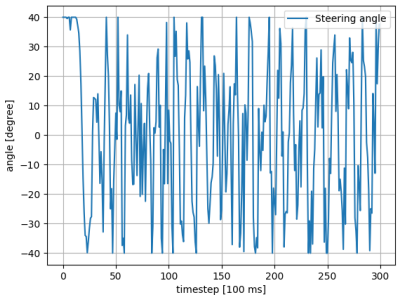
The reward from training the Mariner ship model on a curved path and straight path is shown in Figure 26. Here, the steering is updated once every 10 seconds. It is clear that the agent is able to learn the environment in both cases. As can be seen, the agent scores higher on the straight path, and this is as expected since it should be easier to follow a straight path than a curved one. This can also be seen by comparing the cross-track of the performance of the two paths in Figure 27b and 28b. Just as for the Manta vehicle, the cross-track error on the straight path shows that the ship does not converge perfectly to the path. Two different solutions to this were tested on the Mariner training. In Figure 30 the path was changed from being represented by multiple points as explained in Section 4.4, to being a single mathematically defined line. In addition, the length of the time step that is used to update the steering command was lowered to 1 s from 10 s. Even then, the ship does not converge perfectly. This might come from not being sensitive enough for the small cross-track errors. Therefore, the trained mariner model was trained again with a Gaussian function with a smaller variance than before, with the hope that it would learn to be more precise. This, however, did not improve the performance, probably because the new reward function being too sparse.



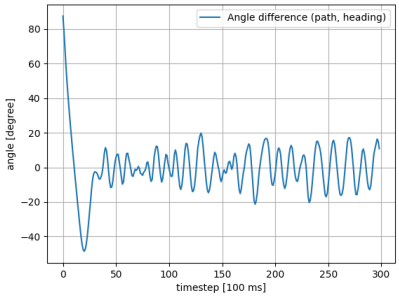
(a) Path of the vehicle



(b) Crosstrack-error

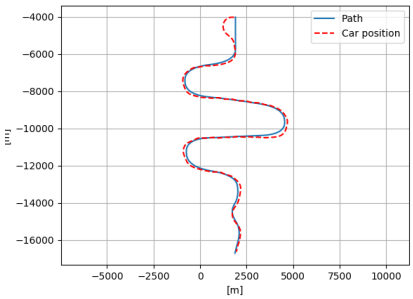


(c) The steering commands to the vehicle

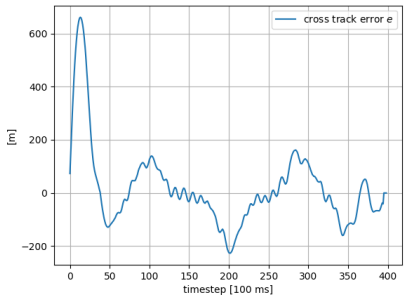


(d) The angle difference between vehicle heading and path tangential angle

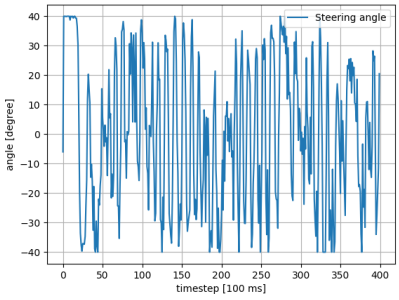
Figure 27: Performance of mariner ship model on a straight path with 10 [s] time step



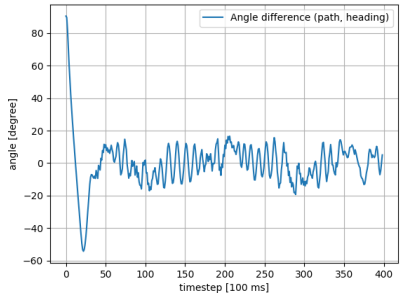
(a) Path of the vehicle



(b) Crosstrack-error



(c) The steering commands to the vehicle



(d) The angle difference between vehicle heading and path tangential angle

Figure 28: Performance of mariner ship model on curved path with 10 [s] time step

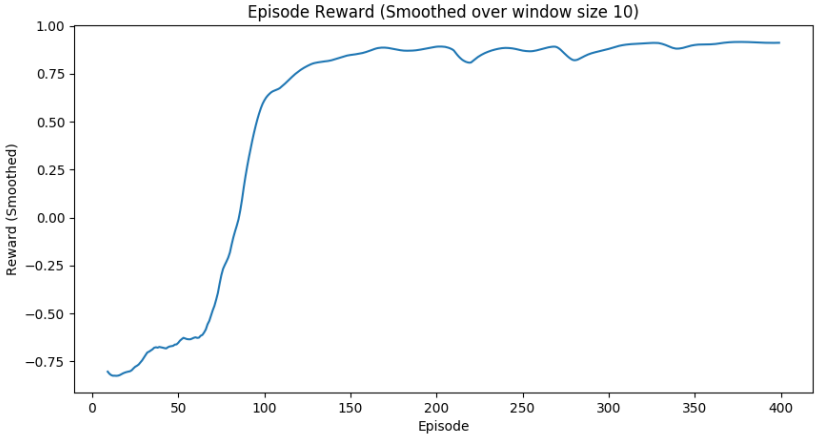
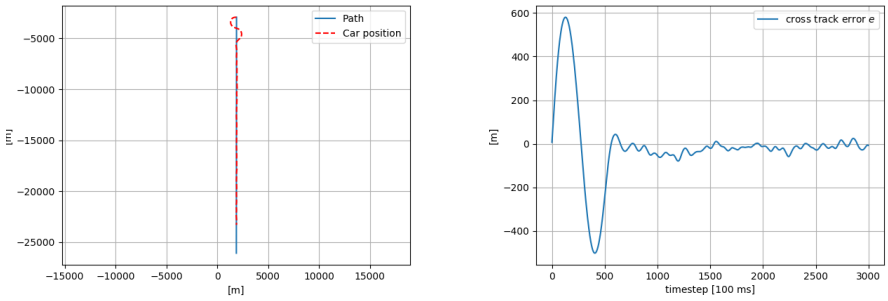


Figure 29: The training reward of the Mariner model with time step 1 [s] on a straight path.



(a) (b)

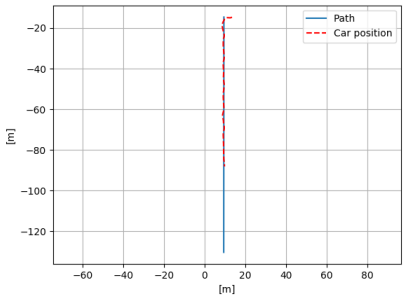
Figure 30: Performance of mariner ship model on a straight path with 1 [s] time step.

Figure 31: The training response of the Manta vehicle after previously being trained on the Mariner ship model.

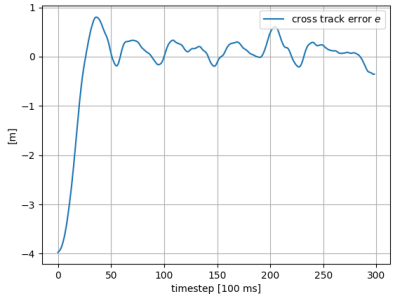
5.3 Transfer learning

In Section 4.2 it was explained that the mariner model was trained with a long time step of 10 seconds to try to have the same dynamic as the Manta car, only scaled up. By taking the neural network models trained on the mariner model which produced the results from Figure 27 and then using it on the Manta car, we get the performance in Figure 32. Looking at the cross-track error in Figure 32b there is some oscillations that also is visible in Figure 32a. If we compare this cross track error to the one in Figure 23b, it is oscillating slower. However, it has a larger error, almost half a meter even after it has had some time to stabilize. This is compared to a maximum of 20 cm as the specially trained model achieved.

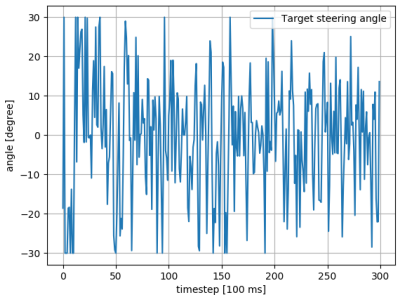
After training the agent on the Mariner model, it was again trained on the vehicle model. In Figure 31 the reward of the training on the vehicle model is presented. As we can see, the reward is increasing over the first 100 episodes. When comparing this to the reward in Figure 21, it is converging to about the same level of reward, but rather slowly. As presented in Figure 32, the basic control is already learned, so the increase in reward is minor improvements. The improvements is presented in Figure 33b. The cross-track error is clearly less than before the specific training on the vehicle.



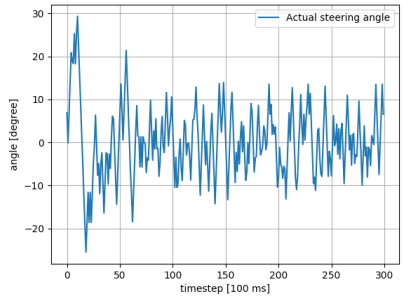
(a)



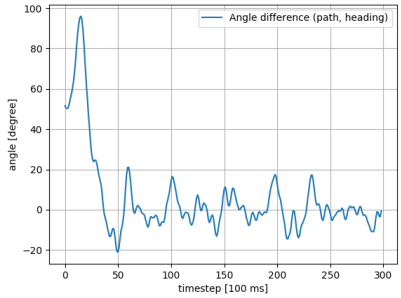
(b) Crosstrack-error



(c) The steering commands to the vehicle

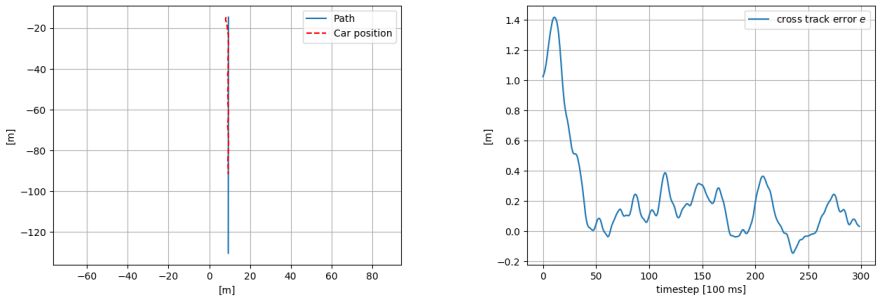


(d) The actual steering actions on the vehicle



(e) The angle difference between vehicle heading and path tangential angle

Figure 32: The performance by the Manta vehicle when the model is only trained on the Mariner ship



(a) Path of the vehicle

(b) Crosstrack-error

Figure 33: The performance by the Manta vehicle when the model is first trained on the Mariner ship, and then trained on the Manta vehicle.

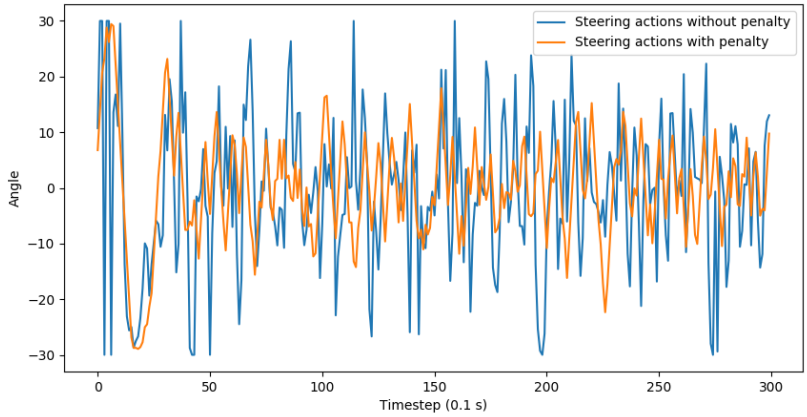
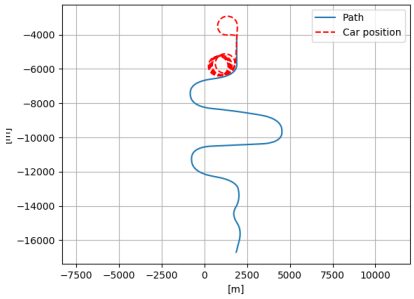


Figure 34: The steering commands on the Manta vehicle trained with and without a penalty on changes in steering commands. The penalty constant β is chosen as 0.1.

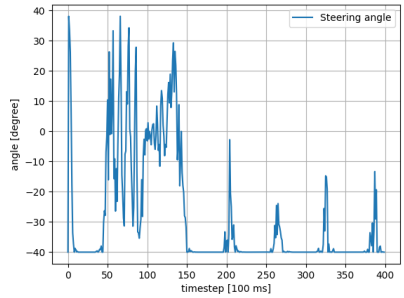
5.4 Action penalty

In Section 4.5 it was proposed to add a penalty on rapid changes in the steering action. The result of this is presented in Figure 34, where the steering commands of two cars following a straight path is plotted. The blue line is the commands of an agent trained without the penalty, and the orange is with. Here, the penalty constant β from equation 33 is chosen as 0.1. It is clear that the agent trained with the penalty is keeping a lower average angle when following the straight line. The maximum steering angle is not limited, as we can see even though it is trained with a penalty, it is still able to utilize the full range of the steering.

With a high penalty constant $\beta = 0.4$, the behavior of the system changes. In Figure 35 we can see the behavior the Mariner converges to. It has found a turn in the path where it can circulate and get a relatively high reward without changing the steering. This is a good example of how the agent can learn a behavior that was not intended, because of an untuned reward function.



(a) Path of the vehicle



(b) The actual steering actions on the vehicle

Figure 35: An optimal behavior of the system with a too high steering rate of change penalty. Here, $\beta = 0.4$

Timing diagram for a single timestep						
Model	threads	Total [s]	Algorithm[s]	Simulation[s]	Action [s]	Sensors[s]
Manta	3	0.32±0.003	0.001 ±0.0002	0.04 ± 0.001	0.08 ± 0.001	0.18 ± 0.001
	1	0.30±0.003	0.001 ±0.0002	0.04 ± 0.001	0.08 ± 0.001	0.16 ± 0.001
Mariner	8	0.066 ± 0.001				
	1	0.008± 0.0005				

Table 2: Table to be fixed. The table shows the time usage of different part of the algorithm.

5.5 Time usage

The following section presents the results from a time usage test of the algorithm. In Table 2 we separate the Manta car model and the Mariner ship model, and test for different number of threads running in parallel on the two simulations. The *Simulation* column is the time taken to simulate one timestep on the simulator, the *Sensor* column is the time taken to poll and receive a sensor reading from the simulator, while the *Action* column is the time the program uses to send the action to the simulator. The *Algorithm* column is the rest of the running time of the Python program, including neural network training, state calculations, etc.

The difference in time between running the V-REP Manta model and running the Mariner model is significant. The mariner model is 38 times faster per timestep for one thread. What is interesting, though somewhat expected, is that while multiple threads for the mariner model just scale the time linearly, the V-REP simulation is almost just as fast. This is because while the Python part of the algorithm is not running in true parallel, the V-REP simulation is. It is clear that it is the blocking communication between the python script and the V-REP simulator that is using the most time. The reason for this is unknown. It is a low amount of data to be transported, and should from previous experience be able to be done faster. When using multiple threads, it is the communication that is affected timewise. This is probably because only one thread is running at each time, and it is possible that a thread might miss its message because another thread was currently active. This assumption is strengthened by the fact that instances of the V-REP simulators start to time out when we use more than three parallel simulators. When we tried with four or more concurrent V-REP instances, some instances will time out when it fails to establish contact with its creator thread. It then gives up and stops communicating.

We should also see an increase in time in the general algorithm because of the python green threads, as we do for the mariner, but it was too small to play an important role. The result was however quite clear on that running the simulation in parallel had a big improvement on the V-REP model since it was able to actually run the simulation on multiple processor cores. This is different from the Mariner model, as the multiple threads were run on only one core, and therefore did not experience any speedup in simulation time compared to running only one thread. However, as explained in Section 4.6, the stochasticity we get when having parallel

learners is mathematically efficient.

From Section 5.1 it is shown that the agent needs 400 episodes with 300 timesteps each, to learn to follow the straight path with the Manta vehicle. This adds up to 120 000 timesteps in total. As this is run in parallel over three threads, each thread runs 40 000 timesteps each. Using the total time from Table 2, it takes 40 000*0.32 s = 12800 s, which approximates 3 hours and 33 minutes. Looking at the mariner in Section 5.2, it has learned the straight path after about 500 episodes using 8 parallel threads. Making the same calculations as with the Manta, it takes only 21 minutes to learn the environment with the Mariner model. As explained earlier, the important part is not the exact time usage, as this will vary between different hardware, but the time usage of the models compared to each other. To master the straight path, the Mariner model was 10 times faster than the V-REP Manta model. From the transfer learning we found that it took 100 episodes to train the agent to control the Manta vehicle, when previously trained on the Mariner. 100 episodes with 300 timesteps each give 30 000 timesteps, distributed over three threads. This results in 10 000 timesteps run which equals 10 000*0.32s = 3200 s $\approx$ 53 minutes. If we add this as a basis for calculating the time to train the vehicle using the model trained on the Mariner, the total time adds up to 1 hour 14 minutes. This is 2.9 times faster compared to training only on the Manta vehicle. It is clear that using a simple model to speed up training of a complex model is very effective. It is then inherently possible that this extends also to real-world applications. That is, to train a simple mathematical model with similar dynamics, and then transfer this learning to a real-world application.

6 Conclusion

An Asynchronous Advantage Actor-Critic (A3C) method to be used with the V-REP simulator was successfully implemented. In addition, the implementation was able to function with a ship environment implemented in Python. Using the A3C method, the reinforcement learning agent was able to learn to follow a path with both the Manta vehicle model and the Mariner ship model. However, the accuracy of the models following the paths showed in the results was not perfect, as we experienced some occilations in the cross-track error. The reason for this is discussed, but not exactly clear.

The agent was able to follow both a straight and a curved path. Since the vehicle model is simulated in the V-REP simulator, it is inherently slower than the Mariner, as this is run by using its equation directly in Python. We found that the mariner model could be run 10 times faster than the V-REP vehicle model for each simulated timestep. The models could be learned at about the same number of training episodes. The Mariner converged after an average of 500 episodes, versus 400 with the V-REP vehicle model. This resulted in that the agent could learn the mariner path following in only 21 minutes on the given computer, while it took 3 hours and 33 minutes on average to learn the V-REP Manta path following. This is about four times as slow as on the Mariner. The Mariner trained model can be used to transfer learning to the Manta car model. Training an agent on the Mariner

and later training it on the vehicle gave a calculated total training time of 1 hour 14 minutes, 2.9 times faster than training only on the vehicle. It is concluded that this property might extend to the real world, that is, a simple simulated model can be used to transfer learning to a real-world application.

6.1 Taking it further

There are several interesting paths that can take this work further. For example, the Manta vehicle could be given more challenging driving conditions, as the surface could be made uneven or slippery. Obstacles could be introduced, together with ways to observe and learn to avoid these. The Mariner ship model could be introduced to wind and currents that it would have to learn to compensate for. It would also be interesting to extend the input space, and find new states to take into the network. An example is to take in the rate of change of the path tangential angle, so that the vehicle could anticipate turns in the path. To make the environments more realistic, a measurement noise should be added to the sensor readings. The models would have to learn to compensate for this to be able to control a vehicle in the real world.

Appendices

A Mariner Ship Model Equations

This appendix gives the equations for the ship dynamics of the Mariner ship model as given originally in [4]. See [9] for more details on the notation. The equations are given for the motion in x and y-direction given a body fixed coordinate system. X is the force in x-direction, and Y is the force in y-direction. The moment about the z axis is given by N. The input δ is the rudder angle. Following the equation of motion, is the constants used by the Mariner ship model.

$$
\begin{aligned}
X =& X_u u + X_{uu} u^2 + X_{uuu} u^3 + X_{vv} v^2 + X_{rr} r^2 + X_{rv} rv \\
&+ X_{\delta\delta} \delta^2 + X_{u\delta\delta} u\delta^2 + X_{v\delta} v\delta + X_{uv\delta} uv\delta \\
Y =& Y_v v + Y_r r + Y_{vvv} v^3 + Y_{vvr} v^2 r + Y_{vu} vu + Y_{ru} ru + Y_\delta \delta \\
&+ Y_{\delta\delta\delta} \delta^3 + Y_{u\delta} u\delta + Y_{uu\delta} u^2\delta + Y_{v\delta\delta} v\delta^2 + Y_{vv\delta} v^2\delta + (Y0 + Y0_u u + Y0_{uu} u^2) \\
N =& N_v v + N_r r + N_{vvv} v^3 + N_{vvr} v^2 r + N_{vu} vu + N_{ru} ru + N_\delta \delta \\
&+ N_{\delta\delta\delta} \delta^3 + N_{u\delta} u\delta + N_{uu\delta} u^2\delta + N_{v\delta\delta} v\delta^2 + N_{vv\delta} v^2\delta + (N0 + N0_u u + N0_{uu} u^2)
\end{aligned}
$$

$X_{\dot{u}} = -42e^{-5}$	$Y_{\dot{v}} = -748e^{-5}$	$N_{\dot{v}} = 4.646e^{-5}$
$X_u = -184e^{-5}$	$Y_{\dot{r}} = -9.354e^{-5}$	$N_{\dot{r}} = -43.8e^{-5}$
$X_{uu} = -110e^{-5}$	$Y_v = -1160e^{-5}$	$N_v = -264e^{-5}$
$X_{uuu} = -215e^{-5}$	$Y_r = -499e^{-5}$	$N_r = -166e^{-5}$
$X_{vv} = -899e^{-5}$	$Y_{vvv} = -8078e^{-5}$	$N_{vvv} = 1636e^{-5}$
$X_{rr} = 18e^{-5}$	$Y_{vvr} = 15356e^{-5}$	$N_{vvr} = -5483e^{-5}$
$X_{\delta\delta} = -95e^{-5}$	$Y_{vu} = -1160e^{-5}$	$N_{vu} = -264e^{-5}$
$X_{u\delta\delta} = -190e^{-5}$	$Y_{ru} = -499e^{-5}$	$N_{ru} = -166e^{-5}$
$X_{rv} = 798e^{-5}$	$Y_\delta = 278e^{-5}$	$N_\delta = -139e^{-5}$
$X_{v\delta} = 93e^{-5}$	$Y_{\delta\delta\delta} = -90e^{-5}$	$N_{\delta\delta\delta} = 45e^{-5}$
$X_{uv\delta} = 93e^{-5}$	$Y_{u\delta} = 556e^{-5}$	$N_{u\delta} = -278e^{-5}$
$Y_{uu\delta} = 278e^{-5}$	$N_{uu\delta} = -139e^{-5}$	$N_{uu\delta} = -139e^{-5}$
$Y_{v\delta\delta} = -4e^{-5}$	$N_{v\delta\delta} = 13e^{-5}$	$N_{v\delta\delta} = 13e^{-5}$
$Y_{vvd} = 1190e^{-5}$	$N_{vv\delta} = -489e^{-5}$	$N_{vv\delta} = -489e^{-5}$
$Y0 = -4e^{-5}$	$N0 = 3e^{-5}$	$N0 = 3e^{-5}$
$Y0_u = -8e^{-5}$	$N0_u = 6e^{-5}$	$N0_u = 6e^{-5}$
$Y0_{uu} = -4e^{-5}$	$N0_{uu} = 3e^{-5}$	$N0_{uu} = 3e^{-5}$

References

[1] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Vol. II.* Athena Scientific, 3rd edition, 2007.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[3] Mauro Candeloro, Anastasios M. Lekkas, and Asgeir J. Sørensen. A voronoi-diagram-based dynamic path-planning system for underactuated marine vessels. *Control Engineering Practice*, 61:41 – 54, 2017.

[4] MS Chislett and J Strom-Tejsen. Planar motion mechanism tests and full-scale steering and manoeuvring predictions for a mariner class vessel. *International Shipbuilding Progress*, 12(129):201–224, 1965.

[5] Yann Dauphin, Razvan Pascanu, Çaglar Gülçehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. *CoRR*, abs/1406.2572, 2014.

[6] Espen Dietrichs. Dopamin - store medisinske leksikon. https://sml.snl.no/dopamin., 2018.

[7] M. Freese E. Rohmer, S. P. N. Singh. V-rep: a versatile and scalable robot simulation framework. In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[8] Perez Fossen. Marine systems simulator (mss). `http://www.marinecontrol.org`, 2004.

[9] Thor I Fossen. *Handbook of marine craft hydrodynamics and motion control.* John Wiley & Sons, 2011.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[11] Geoffrey E. Hinton, Simon Osindero, and Yee-Whye Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, July 2006.

[12] Jan Jansen and Joel Glover. Synapse. i store medisinske leksikon. https://sml.snl.no/synapse.

[13] Mike "Pomax" Kamermans. A primer on bezier curves, an online book. `https://pomax.github.io/bezierinfo/`, 2018. Accessed: 2018-05-05.

[14] Desmond King-Hele. Erasmus darwin's improved design for steering carriages—-and cars. *Notes and Records*, 56(1):41–62, 2002.

[15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[16] J. Kober, K. Mülling, O. Krömer, C. H. Lampert, B. Schölkopf, and J. Peters. Movement templates for learning of hitting and batting. In *2010 IEEE International Conference on Robotics and Automation*, pages 853–858, May 2010.

[17] J. Kober and J. Peters. *Reinforcement Learning in Robotics: A Survey*, volume 12, pages 579–610. Springer, Berlin, Germany, 2012.

[18] Vijay R. Konda and John N. Tsitsiklis. On actor-critic algorithms. *SIAM J. Control Optim.*, 42(4):1143–1166, April 2003.

[19] Ron Larson and Robert Hostetler. *Precalculus: A Concise Course*. Cengage Learning, 2007.

[20] Donald Michie. Experiments on the mechanization of game-learning part i. characterization of the model and its parameters. *The Computer Journal*, 6(3):232–236, 1963.

[21] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1928–1937, New York, New York, USA, 20–22 Jun 2016. PMLR.

[22] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[23] Andrew Y. Ng, Adam Coates, Mark Diel, Varun Ganapathi, Jamie Schulte, Ben Tse, Eric Berger, and Eric Liang. Autonomous inverted helicopter flight via reinforcement learning. In Marcelo H. Ang and Oussama Khatib, editors, *Experimental Robotics IX*, pages 363–372, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[24] SNAME The Society of Naval Architects and Marine Engineers. Nomenclature for treating the motion of a submerged body through a fluid. *New York: Technical and Research Bulletin*, 1950.

[25] Jan Palach. *Parallel Programming with Python*. Packt Publishing, 2014.

[26] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural Netw.*, 12(1):145–151, January 1999.

[27] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[28] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[29] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning : An Introduction*. MIT Press, 1998.

[30] QuantStart Team. Parallelising python with threading and multiprocessing. `https://www.quantstart.com/articles/Parallelising-Python-with-Threading-and-Multiprocessing`, 2018. Accessed: 2018-05-05.

[31] Tensorflow team. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[32] Gerald Tesauro. Td-gammon: A self-teaching backgammon program. In *Applications of Neural Networks*, pages 267–285. Springer, 1995.

[33] H. van Hasselt and M. A. Wiering. Reinforcement learning in continuous action spaces. In *2007 IEEE International Symposium on Approximate Dynamic Programming and Reinforcement Learning*, pages 272–279, April 2007.

[34] Christina Voskoglou. What is the best programming language for machine learning? `https://towardsdatascience.com/what-is-the-best-programming-language-for-machine-learning-a745c156d6b7`. Accessed: 2018-05-06.

[35] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Mach. Learn.*, 8(3-4):229–256, May 1992.