NTNU
Norwegian University of
Science and Technology

Introduction to Reinforcement Learning Methods for Vehicle Control

Project Thesis

Ingunn Vallestad

Trondheim, December 2018

Supervisor: Anastasios Lekkas

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

Preface

The work of this project thesis has been the first step towards what will eventually become a Master's Thesis focusing on creating an autonomous collision avoidance system for marine vehicles, using reinforcement learning. In order to achieve this goal during the next semester, it is necessary to familiarise oneself with reinforcement learning fundamentals. This report covers an introductory study of reinforcement learning methods and their application to simulations of physical systems, ranging from the simplest discrete systems to tasks involving both continuous state space and action space. An investigation of how reinforcement learning can be applied to the problem of controlling a vehicle's movement has been conducted, which will contribute to the work that is to be carried out in the Master's Thesis.

Abstract

Autonomous control of vehicles, and particularly autonomous collision avoidance, is a complex problem within control engineering. The design of control laws that result in optimal movement of a vehicle in a complicated environment is traditionally developed using intricate nonlinear methods. These usually require extensive understanding of the vehicle dynamics and kinematics. Another field that has shown promise in creating optimal control systems is reinforcement learning (RL), which is an area of work within machine learning (ML) focused on learning through exploration, or "trial and error". The combination of RL with deep learning (DL) has given rise to deep reinforcement learning (DRL), which allows for the development of complex optimal control laws in nonlinear systems without the need for knowledge about the inner workings of each system.

In this report we will discuss some fundamental RL algorithms and apply some of the most prominent ones to different control problems. Their performance in their respective tasks will be discussed, and will allow us to gain understanding of the field of reinforcement learning, and its advantages and disadvantages. An algorithm that is suitable for continuous systems will be used in the design of an autonomous vehicle control system that is to steer a vehicle from any start position to a stationary desired position. We propose a framework for such a controller and discuss some design decisions that should be made in order to achieve the objective. The controller is applied to a simulation of a simple vehicle, implemented in this project for this purpose. We show that the algorithm learns how to control the vehicle to the desired position without any a priori knowledge about the world it operates in, and we show what effects the design decisions we have made can have on the final behaviour.

The work done in this project has contributed towards the development of completely autonomous vehicle control and has demonstrated that utilising DRL for this purpose shows promise.

Contents

List of Figures

List of Tables

List of Algorithms

Nomenclature

Abbreviations

AI	Artificial Intelligence
ANN	Artificial Neural Network
APF	Artificial Potential Field
AUV	Autonomous Underwater Vehicle
$COLAV$	Collision Avoidance
$DDPG$	Deep Deterministic Policy Gradient
DL	Deep Learning
DP	Dynamic Programming
DRL	Deep Reinforcement Learning
IMO	International Maritime Organisation
LOS	Line-of-Sight
MDP	Markov Decision Process
ML	Machine Learning
MPC	Model Predictive Control
PPO	Proximal Policy Optimization
$ReLU$	Rectified Linear Unit
RL	Reinforcement Learning
ROV	Remotely Operated Vehicle
TD	Temporal Difference
UUV	Unmanned Underwater Vehicle

Subscripts and Superscripts

*	Optimality property
t	Time step

Part I

Introduction

Chapter 1

Introduction

1.1 Background and Motivation

Traditionally, navigation at sea has been performed by humans, whether that is at or below the surface. The need for intelligent autonomous systems has increased in recent years, with increased amount of underwater operations and systems that require high precision. Replacing divers with remotely operated vehicles (ROVs) or autonomous underwater vehicles (AUVs) in subsea inspection is potentially safer, more cost-effective and increases efficiency of operations. These autonomous vehicles must be able to follow a predefined path, but they must also be able to make intelligent decisions to deviate from that path in order to avoid collisions and dangerous situations. In other words, the vessels must optimise and make decisions without human interaction, to ensure safety of themselves and others. To achieve this, a common set of navigational rules must be followed to ensure that the vessels' behaviour is predictable.

On the ocean surface, the rules for ship collision avoidance given by the Convention on the International Regulations for Preventing Collisions at Sea (COLREGS) have been developed by the International Maritime Organization (IMO) [1]. For underwater operations in three dimensions, these rules can not be directly applied, and thus the authors of [2] have proposed an appropriate set of rules.

The development of control laws that comply with COLREGs while avoiding static and dynamic obstacles has been studied by many. Past works have shown that simulation-based model predictive control (MPC) can be used to optimise the planned path while avoiding obstacles [3, 4]. These works separate the collision avoidance (COLAV) system from the path following system, thus the method may be combined with several types of guidance laws. However, the optimisation is over a finite set of control laws, not a continuous space of actions. The authors of [5] use artificial potential fields (APFs) to repel the vehicle from obstacles, and the method makes vehicles able to reach moving goals. However, this only applies to course control, since constant surge speed is assumed. Other methods include, but are not limited to, dynamic window [6], set-based methods [7] and velocity obstacles [8].

The aforementioned papers focus on vehicles navigating on the ocean surface. To avoid collision in three dimensions for underwater vehicles, these methods are not directly applicable due to differing regulations and conditions, and the fact that motion in all three dimensions is possible. An example of an AUV COLAV system is the reflexive vector field with obstacle localisation in [9]. The method results in effectively avoiding obstacles using a local planner that overrides the mission planner. However, the authors

find that many obstacles may cause the local planner to take over completely and never let the vehicle return to the original mission. Also, this is a horizontal COLAV system that does not consider evasive manoeuvres in heave. An improved velocity obstacle method for unmanned underwater vehicles (UUV) is proposed in [10], where a collision risk evaluation model and motion uncertainty of obstacles increase the speed and safety of collision avoidance manoeuvres in 2D.

In [11], an APF is used to plan a path for an underwater snake robot through an environment with obstacles, then a Line-Of-Sight (LOS) guidance law is used to follow straight line segments between waypoints generated by the path planner. This requires knowledge about the obstacle locations before planning. To be able to navigate in unknown environments, a set-based method for planar underwater snake robots is presented in [12], in which a switching between one mode for path following and one mode for obstacle avoidance is enabled. This switching is based on the results of [7] and generalised to work in underwater environments and for different kinds of systems. The robot follows a straight path, and navigates past detected obstacles on the path by switching modes to follow a circle around the obstacle. The proposed switching strategy is made independent of the dynamics, and the guidance law is one that suits generic paths of both straight lines and circles. However, this method is not tested on dynamic obstacles, and not yet extended to the three-dimensional case.

In general, the above discussed methods include some drawbacks that should be considered. Some works introduce complex laws for decision making in order to comply with navigational rules [4, 5, 8], while some simplify the collision avoidance problem - by considering static obstacles only [9, 11, 12] or by only considering course control [5]. None of the presented methods have been applied to three-dimensional navigation, and it seems that solving the path following plus collision avoidance problem by a traditional mathematical approach is difficult.

A potential solution to the problem of increased complexity is to use machine learning. Many artificial intelligence (AI) algorithms, such as genetic algorithms, swarm intelligence and reinforcement learning, aim at solving an optimisation problem. There is a cost/reward function involved, and the algorithms attempt to find a behaviour that minimises cost/maximises reward. Such optimisation can be used to determine the best sequence of input signals to a system, e. g. a ship on its way to a target destination, in terms of minimal fuel consumption, minimum distance travelled, minimum time spent, or some other criteria.

The collision avoidance problem can be viewed as a multi-objective optimisation problem: find a sequence of inputs that lets you follow a path, while minimising cost associated with collision, and while complying with some well-known rules.

Reinforcement learning has been around for a long time. Turing introduced the concept of rewards and punishments in a machine's learning process in "Computing Machinery and Intelligence" from 1950 [13]. Here, he proposes that instead of trying to produce a machine that imitates an adult mind, we may try to create a child-like machine which possesses some fundamental knowledge, and then educate this machine further. Arthur Samuel developed a checkers player that learned to play using RL in the period 1952-1956 [14]. Since then this field has been extensively developed by many researchers. An example is Donald Michie and Roger A. Chambers, who described a reinforcement learning controller called BOXES in 1968 [15], which was applied to a pole balancing problem where a failure signal was given when the pole fell over or reached the edge of

its track. This is a good example of early work on reinforcement learning tasks based on incomplete knowledge. In 1996, Dimitri Bertsekas and John Tsitsiklis published the book Neuro-Dynamic Programming, which described the combination of dynamic programming and artificial neural networks and was the first textbook that fully explained this [16]. Thus, a combination of deep learning and reinforcement learning had surfaced, giving rise to deep reinforcement learning (DRL), which made it possible to apply RL methods to large problems by using function approximators. Some examples of this include DeepMind's Atari playing agent based on Deep Q-learning networks [17], and their AlphaGo Zero system that outperforms the world champion of the game Go, completely without human supervision during training [18].

In recent years, the application of deep reinforcement learning in continuous control systems has emerged, such as in [19] where several physics tasks (cartpole swing-up, dexterous manipulation, legged locomotion, car driving) are solved using an algorithm called Deep Deterministic Policy Gradient (DDPG). DDPG has also been used to achieve horizontal trajectory tracking of an AUV [20]. A path following algorithm for complex marine vessels with many nonlinearities is presented in [21]. In this work, no knowledge of the vessel dynamics is required, and it is shown that the vessel successfully learns to follow straight and curved paths, outperforming the Line-of-Sight (LOS) guidance law, one of the most widely used path following methods of today.

Combining collision avoidance systems with DRL algorithms has been explored in [22], in which it is demonstrated that the vessel can learn to navigate without prior knowledge of the world, and without defining a complex control law. The research is promising, and combining the fields of DRL and control engineering may lead to simpler designs than the traditional mathematical approaches. Using deep neural networks as function approximators means that the algorithm generalises well and thus may be applied to several types of marine vessels.

1.2 Thesis and Method

The goal of this thesis is twofold. The first objective is to build a foundation of reinforcement learning knowledge through implementation of RL algorithms on physical control tasks of varying complexity. The second objective is to use this knowledge in an investigation of how to apply RL to a manoeuvring problem for a vehicle. We propose an architecture in Figure 1.1 illustrating the components of our solution, and implement a vehicle control agent on a simplified vehicle to illustrate usefulness of the solution. It can be argued that the resulting agent can also be called a path planning agent, since its goal is to find a trajectory from a start to a goal position with accompanying control inputs to the vehicle. Thus, the terms *vehicle control system* and *path planner* will be used throughout the thesis to describe the agent.

The architecture consists of an environment and an agent, where the environment contains a model of our vehicle and how it interacts with its surroundings, and a performance measure that tells the agent about the performance level of the current state. The agent receives only the current state and performance measure from the environment and thus has no knowledge of the vehicle dynamics. The agent consists of two components - the controller, also called the policy, and a value function that provides a measure of the value of control inputs in different states. It outputs a control action that is fed back to the environment.

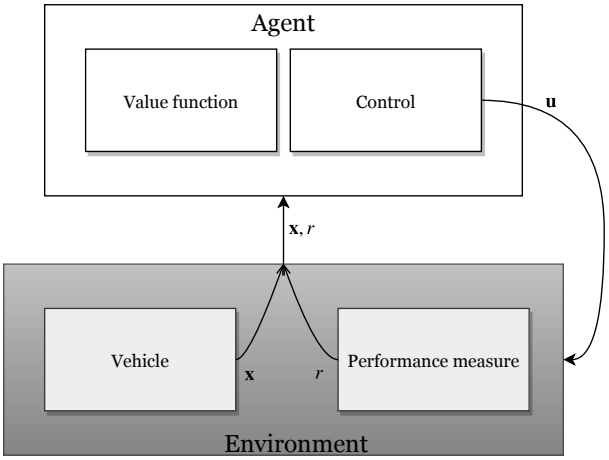


Figure 1.1: Overview of architecture of the proposed RL system

1.3 Tools

The Python programming language was used to implement the algorithms presented in this thesis. NumPy, the fundamental package for scientific computing with Python, was used in numerical computiations such as linear algebra.

To implement function approximators, Tensorflow [23] was used. Tensorflow is an open source software library for high performance numerical computation, with strong support for machine learning and deep learning. It provides building blocks to create and train neural network models, with automatic backpropagation and optimisation techniques.

Implementations of control problems were fetched from OpenAI Gym [24], which is a toolkit for reinforcement learning research. It provides a common interface to several benchmark problems such that researchers can test their algorithms. It also allows for implementation of own problems with the same interface, which has been utilised in this thesis.

1.4 Structure of Report

The report is divided into three main parts - Part I: Introduction, Part II: Theory and background material, and Part III: Results. Part I is coming to an end, and its aim has been to introduce the reader to why the work of the thesis can be of importance and how the work has been carried out.

Part II introduces concepts relevant to this thesis. First, machine learning is briefly explained, before we delve deeper into a supervised learning framework called artificial neural networks in Chapter 2. Chapter 3 presents a comprehensive description of the RL concepts and methods necessary for the contributions of the following chapters.

In Part III, the goals of the thesis are addressed and results are presented. In Chapter 4, some control tasks are described and solved using a number of different RL methods, culminating in the implementation of the algorithm that will be used in the path planning agent of Chapter 5, called DDPG. Here, a vehicle simulation on which experiments will be performed is implemented, before a series of performance measures are shaped in the hopes of achieving the desired behaviour of the vehicle. Simulation results of agents are presented in Section 5.3 and discussed in Section 5.4. Chapters 6 and 7 give a brief conclusion of the thesis and some proposed further work, respectively.

Part II

Theory and Background Material

Chapter 2

Machine Learning Background

In the following chapters, relevant concepts for this project thesis will be presented. First, an introduction to machine learning and useful theory regarding artificial neural networks will be given in Sections 2.1 and 2.2. In Chapter 3, reinforcement learning concepts will be described.

2.1 Machine Learning

Machine learning (ML) research is part of research on artificial intelligence, seeking to get computers to learn from data without relying on rule-based programming. The field of ML can be separated into three main categories, as described in [14].

Supervised learning provides an agent with labelled input-output pairs so that the agent can learn a function that maps an input to the correct output. This is a form of function approximation.

Unsupervised learning does not supply any explicit feedback, so the agent has to learn patterns found in the input on its own. A typical example is clustering, where an agent groups inputs together in clusters, that potentially provide useful information.

Reinforcement learning gives the agent an indication of whether its actions were good or bad through reinforcement signals, such as rewards or penalties. For example, a self-driving car may receive rewards when successfully stopping for pedestrians crossing the road, which indicates that the car should keep doing this in this situation. Designing the reward function in order to ensure the agent can learn the desired behaviour is an important challenge in reinforcement learning problems.

A fourth category is sometimes defined as well, useful when describing hybrid learning algorithms, called **semi-supervised learning**. It is a term for describing learning algorithms where the distinction between supervised and unsupervised is not perfectly clear.

Deep reinforcement learning (DRL) combines reinforcement learning with elements of supervised learning. As we will see in Section 3.4, the policy and value functions of a reinforcement learning agent can be represented by artificial neural networks.

2.2 Artificial Neural Networks

Artificial neural networks (ANNs) are popular in supervised learning. Their structure allows for nonlinear function approximation through a relatively small set of parameters, and we will see in Chapter 3 that this can be particularly useful in reinforcement learning for physical control tasks.

ANNs can be formed as a feed-forward network or a recurrent network. The latter form feeds its output back to its input, thus making the response of the network depend not only in the current input, but on previous inputs as well. The former case is illustrated by the example network in Figure 2.1. Feed-forward networks have connections only in one direction, and represent a function of its current input. This is the kind of ANNs this chapter will focus on.

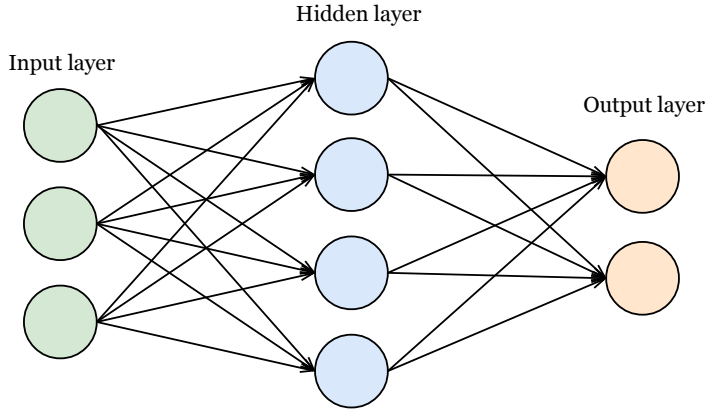


Figure 2.1: Artificial neural network

Feed-forward networks consist of *nodes*, or *units*, arranged into *layers* to form a network inspired by the neural networks found in animals. The basic structure of the nodes is illustrated in Figure 2.2, which shows that outputs of units in a layer become the inputs of the next layer's units. The output of the node is given by a function $y = f(\cdot)$, called an activation function, dependent on its inputs $\boldsymbol{x}$ and associated parameters, $\boldsymbol{w}$, and a bias b. The concept is similar to the behaviour of neurons in animals, where it can be said that a neuron "fires" when a combination of the input signals becomes larger than some threshold. The activation function is usually a hard threshold or a soft threshold (logistic function). By firing, we mean having a non-zero output which can be passed on as input to other neurons [25].

In artificial neural networks, each unit computes a weighted sum of its input, where $\boldsymbol{w}$ are the weights of the unit,

$$y = f(x_1 w_1 + x_2 w_2 + ... + x_n w_n + b) = f(\sum_{i=1}^{n} x_i w_i + b) = f(\boldsymbol{w}\boldsymbol{x} + b) \tag{2.1}$$

and thus the weight associated with inputs determines the sign and strength of the connection between each unit of the previous layer and this layer.

Typical activation functions are nonlinear, because these functions allow the network to approximate nonlinear functions. All nodes in the hidden layers usually share the same activation function, while the output layer may be different, reflecting the desired characteristics of the output. Classification tasks often use softmax in the last layer

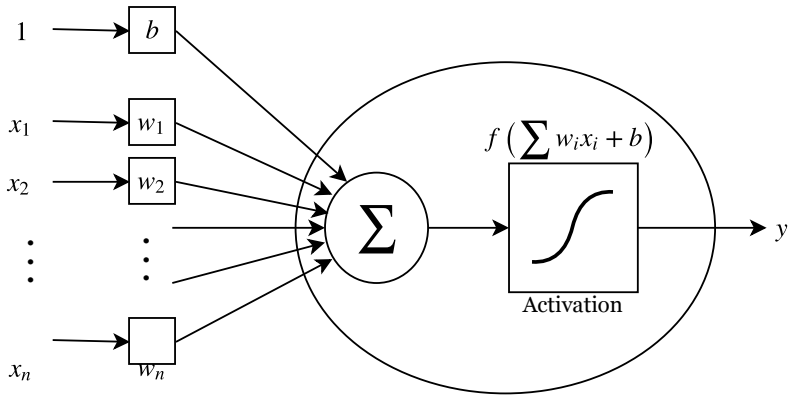


Figure 2.2: Neuron of an artificial neural network

to assign decimal probabilities to each class in a multi-class problem. Those decimal probabilities must add up to 1.0, and softmax provides this functionality. Alternatively, the hyperbolic tangent function can be used, which suppresses the output between -1 and 1. One of the most popular hidden layer activations is the Rectified Linear Unit (ReLU).

A layer can be represented by its activation function f, and a weight matrix $\boldsymbol{W}$ and a bias vector $\boldsymbol{b}$, containing the weight vectors and the biases associated with the nodes in the layer, respectively. Then the output of a layer is given by $\boldsymbol{y} = f(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$, where $\boldsymbol{x}$ is a vector of outputs of the previous layer.

The goal when training an ANN is to learn the weights of the network from data consisting of input-output pairs (x, y_{target}), such that the predicted output is close to the expected output for all inputs. An error function typically used to measure the distance to this goal is the squared error:

$$E(y_{pred} - y_{target}) = \frac{1}{2}(y_{pred} - y_{target})^2 \tag{2.2}$$

where y_{pred} is the output predicted by the network, and y_{target} is the real output which the network's output should be equal to.

To learn the weights, two phases must be carried out, called the forward propagation and backpropagation [26] phases. The forward propagation simply passes the input of a data pair to the network, receiving a predicted output and the computed error. Backpropagation then decides how to update each weight by computing the derivative of the error with respect to each weight in $\boldsymbol{W}$ and applying a gradient update

$$w_{ij} = w_{ij} - \alpha \frac{\partial E}{\partial w_{ij}} \tag{2.3}$$

where $\alpha > 0$ is referred to as the learning rate. If $\frac{\partial E}{\partial w_{ij}} < 0$, the error goes down when the weight increases, thus the weight is increased by the update. If $\frac{\partial E}{\partial w_{ij}} > 0$, the error goes up when the weight increases, thus the weight is decreased by the update. To calculate the derivatives, the backpropagation algorithm uses the chain rule to move backwards in the network, from the output to the input. Considering the weight between neurons i

and j in subsequent layers, we get

$$\begin{aligned}\frac{\partial E}{\partial w_{ij}} &= \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}\frac{\partial z_j}{\partial w_{ij}} \\ &= \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}y_i \\ &= \frac{\partial E}{\partial y_j}f_j'(z_j)y_i\end{aligned} \tag{2.4}$$

where y_i denotes the output of neuron i, $z_j = \sum_{k=1}^{n} w_{kj}y_k + b_j$ is the total input to neuron j, and $f_j(\cdot)$ is the activation function of neuron j. For neurons in the last layer, the first factor becomes $\frac{\partial E}{\partial y_{pred}} = (y_{target} - y_{pred})$, and this derivative is propagated backwards in the layers to calculate the rest of the derivatives. For neuron i in layer $L-1$, we can use the calculations from layer L:

$$\frac{\partial E}{\partial y_i} = \sum_{l \in L}\left(\frac{\partial E}{\partial y_l}\frac{\partial y_l}{\partial z_l}w_{il}\right) \tag{2.5}$$

where l denotes units in layer L. Similar arguments can be made for derivatives with respect to the biases, where $\frac{\partial z_j}{\partial b_j} = 1$.

There exist several optimisation algorithms that implement the gradient updates of backpropagation, and some commonly used in neural networks are AdaGrad [27], RMSProp [28], and Adam [29], with Adam being the most recent.

A challenge that arises when training ANNs is overfitting, in which the network fits the output only to the presented training examples, but is unable to generalise when faced with new examples. Several solutions to this has been proposed, such as *dropout* [30], *L1 or L2 weight regularization* [31] or *early stopping*. *Batch normalization* [32] is a technique that improves stability of neural network training, thus allowing higher learning rates and faster convergence.

Chapter 3

Reinforcement Learning

We introduced the term reinforcement learning, or *RL* in Section 1.1. The fundamental idea of RL is that an agent must learn by trial and error, and is rewarded for "good" actions and penalised for "bad" ones. The best outcome for this agent is to act in such a way that it maximises the total received rewards. It can thus be viewed as an optimisation problem, where the sequence of decisions is the input and the reward is the objective function, and by viewing it in this way it is evident that the idea is shared between many sciences. For example, optimisation is frequently used in both finance and optimal system control. One fundamental difference between control approaches and RL approaches, however, is that the RL problem utilises *evaluative* feedback in the form of a scalar reward, rather than instructive feedback.

This chapter introduces key concepts of RL in Sections 3.1-3.2 before moving into the realm of solution methods to the RL problem in Sections 3.3-3.4. We go through the most basic tabular methods, such as value iteration and policy iteration in 3.3.1 and Q-learning in 3.3.2, before introducing methods that use function approximators, such as policy gradient in 3.4.1 and actor-critic in 3.4.2. Section 3.4.3 introduces a recently developed method, deep deterministic policy gradient (DDPG).

3.1 The Framework

To introduce some terms and definitions that will be used in this paper, consider Figure 3.1 that describes the general RL framework.

Here, the *agent* is what we want to build when implementing an RL algorithm. Our goal is to build an algorithm that controls this agent's decision making. What the agent interacts with is called the *environment*. RL agents can observe (parts of) their environment through a state, denoted s in Figure 3.1, and can choose actions and perform them to influence the environment. The environment then responds by presenting new situations to the agent in the form of a new state. The agent has a goal, to maximize cumulative reward, and gets rewards based on how it interacts with its environment [33]. We use the subscript $t+1$ for the state to illustrate that a new state is presented to the agent after it has performed an action, and is thus given one time step after the action a_t. A reward is connected to the transition from one state s_t to the next s_{t+1} through an action a_t, so the subscript thus may be ambiguous. We choose here to associate the reward resulting from an action a_t and state s_t with time step t, and denote it r_t.

The terms *agent*, *environment* and *actions* correspond to *controller*, *plant* and *control signal* used in control theory, respectively.

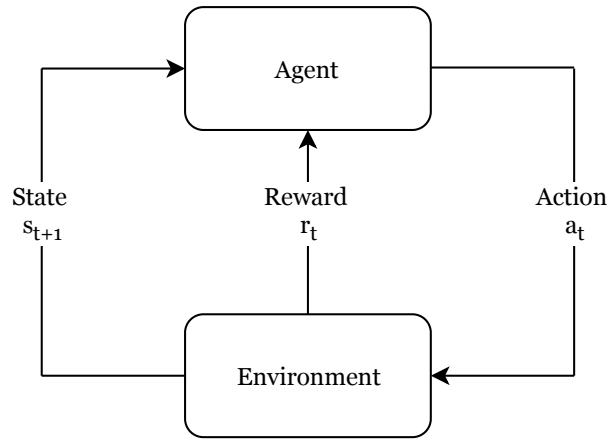


Figure 3.1: Interaction between agents and environments

An RL agent's policy determines its behaviour. This policy is a function that maps states to actions, and is denoted $\pi(a|s)$. It could be deterministic or stochastic. In the latter case, the policy is a probability distribution over actions in each state, and in the former case the policy maps each state to a single action.

Another component of RL agents is the value function, which is a measure of the value of being in a state, incorporating the expected reward received in the future by following a chosen policy. If the policy is π, the value function is denoted $v_\pi(s)$. When the value function is known, one can establish the policy by choosing the action that moves the agent to the neighbouring state with the highest value. Equation (3.1) presents the general expression of the value function, where γ $(0 \leq \gamma \leq 1)$ is a discount factor describing how much we care about future rewards.

$$v_\pi(s) \triangleq \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + ... \mid S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \Big| S_t = s \right] \tag{3.1}$$

We see that, by defining the return $G_t = \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k}\right]$, this expression can be decomposed into (i) the immediate reward of being in this state, and (ii) the discounted value of the return from the next state,

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[R_t + \gamma(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...) \mid S_t = s] \\ &= \mathbb{E}_\pi[R_t + \gamma G_{t+1} \mid S_t = s] \\ &= \mathbb{E}_\pi[R_t + \gamma v_\pi(S_{t+1}) \mid S_t = s] \end{aligned} \tag{3.2}$$

This is called a *state-value* function, because it is a function of state. Alternatively, an *action-value* function may be used in place of $v(s)$. The action-value is the value of taking action a in state s under policy π,

$$q_\pi(s, a) \triangleq \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} \Big| S_t = s, A_t = a \right] \tag{3.3}$$

which is also called the q-value.

The model in RL predicts what the environment will do next, based on the current state and the agent's action. There are typically two parts: a transition model, $\mathcal{P}$, that predicts the dynamics of the environment, and a rewards model, $\mathcal{R}$, that predicts the next immediate reward. The transition model is often stochastic, and describes how the environment reacts when a particular action is taken in a particular state, i.e. it gives a probability distribution describing what the next state will be. If $\mathcal{P}$ is deterministic, there is no uncertainty regarding what the next state will be once an action is chosen.

There exists many combinations of RL methods. The ones that learn a policy explicitly are called policy-based methods, while the ones learning a value function (and thus only implicitly learning about a policy) are called value-based methods. It is possible to learn both a policy and value function at the same time, and this is denoted as *actor-critic* methods. Model-based methods assume the transition model is known and use this for learning how to behave, while model-free methods assume no such prior knowledge.

It is important to note that value based methods must iterate through all possible actions in each state it encounters in order to determine the optimal action with the highest expected return at any step. Therefore, these methods are unsuitable for problems where the action space is continuous, since the number of actions is not finite. This is the case in most guidance and control problems, and thus this project report will focus mostly on policy based and actor-critic methods.

3.1.1 Markov Decision Processes

A state that has the *Markov property* includes all the information about past states and actions that makes a difference for the future. In other words, the next state-reward pair in the agent-environment interaction depends only on the current state and action, and is independent of any previous states and actions. Thus the probability distribution for each choice of state s and action a, given by the transition model $\mathcal{P}$,

$$P(s', r|s, a) \triangleq \mathbf{Pr}\{S_{t+1} = s', R_t = r \mid S_t = s, A_t = a\}, \tag{3.4}$$

completely characterises the environment's dynamics, and is called a Markovian transition model [14].

A sequential decision problem where the transition model is Markovian, the environment is fully observable and stochastic, and which has a reward function is called a *Markov decision process* (MDP). The tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ describes an MDP. $\mathcal{S}$ is the set of states, $\mathcal{A}(s), s \in \mathcal{S}$ is the action space, $\mathcal{P}$ is the Markovian transition model given by the transition probability $P(s', r|s, a)$ and $\mathcal{R}$ is the reward function.

The MDP is the framework we use for defining reinforcement learning problems.

3.2 The Bellman Equation

If we denote the best possible policy (or policies) by π_*, we can define optimal state-value and action-value functions as

$$v_*(s) \triangleq \max_\pi v_\pi(s) \tag{3.5}$$

$$q_*(s, a) \triangleq \max_\pi q_\pi(s, a) \tag{3.6}$$

which are obtained by following π_*. A policy π is defined to be better than or equal to another policy π' if $v_\pi(s) \geq v_{\pi'}(s)\, \forall s \in \mathcal{S}$. There may exist several optimal policies,

but they all share the same optimal state-value function and action-value function of Equation (3.5) and (3.6). q_* can be considered as the expected return of taking action a in state s and from there on following policy π_*, and thus it can be rewritten as

$$q_*(s,a) = \mathbb{E}[R_t + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \tag{3.7}$$

By understanding that the value of a state under an optimal policy must equal the expected return for the best action in that state [33], we get

$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_*(s,a) \\ &= \max_a \mathbb{E}[R_t + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} P(s', r|s, a)[r + \gamma v_*(s')] \end{aligned} \tag{3.8}$$

$$\begin{aligned} q_*(s,a) &= \mathbb{E}[R_t + \gamma \max_{a' \in \mathcal{A}(S_{t+1})} q_*(S_{t+1}, a') \mid S_t = s, A_t = a] \\ &= \sum_{s',r} P(s', r|s, a)[r + \gamma \max_{a' \in \mathcal{A}(s')} q_*(s', a')] \end{aligned} \tag{3.9}$$

where s', a' denote the next state and action after being in state s taking action a. Capital letters such as S_t, A_t, R_t are used in expectations to denote sampled states, actions and rewards.

Equations (3.8) and (3.9) are called the *Bellman optimality equations* for v_* and q_*, respectively. It is a recursive equation describing the relationship between the value of a state and the value of its successor states [33]. The value function v_* or q_* is the unique solution to its Bellman optimality equation, and solving the equation gives the solution to the reinforcement learning problem, as finding v_* or q_* will implicitly give the best possible action in each state, also known as the optimal policy π_*.

If the state space and action space is discrete and finite, and the environment model is known, we can use dynamic programming (DP) to solve the nonlinear equations. A well known DP method is called *value iteration*. It iteratively solves the Bellman equation by giving an initial guess of the value function, $V_0(s)$, and inserting it into the right-hand side of (3.8), which gives a new estimate of the value function, $V_1(s)$. This process continues for every state until the value function converges. In this thesis, capital letters in the value functions, such as V or Q, signals that these are estimates of the real value function, v_π or q_π.

A similar approach, called *policy iteration*, makes initial guesses for both the value function and the policy. It is a two-step process, where first the current policy is evaluated by computing the corresponding value function iteratively, making sure actions are chosen by following the policy. The second step is called policy improvement, and entails going trough all states, and for each state computing the Q-value of each possible action. This means that we consider the reward of each action a in state s, and the value of thereafter following the current policy. Then we choose the action with the maximum Q-value as our new policy in state s. These two steps are repeated until the policy no longer changes, which means the algorithm has converged.

Although an explicit solution to the Bellman optimality equation gives the optimal policy, the solution is generally difficult to find. The equation is actually a set of equations, one for each state, so in the case where the state space is large, solving that amount

of nonlinear equations is computationally expensive. The solution also depends on us knowing the dynamics of the environment, which is not always true. In addition, if either the state space or action space (or both) is continuous, the above mentioned DP methods cannot be used, since they depend on a finite set of states and actions in order to save the value functions as tables, with one entry per state/action. The consequence of these limitations is that approximate solutions is often the only possibility.

3.3 Tabular Solution Approaches

3.3.1 Monte Carlo Methods

A way around the assumption of knowledge about the environment, is to use experience from real or simulated situations. In Monte Carlo methods, complete episodes are sampled before updating value and policy estimates, assuming that all episodes terminate after a finite number of time steps T. Tasks that fulfil this requirement are called *episodic tasks*. The kind of learning where an episode must complete before any changes/learning can be applied, is called off-line training. The opposite case is when training updates are made after every step of an episode, called online training.

In Monte Carlo approaches, we update an estimate of the value function of Equation (3.1) or (3.3) in the direction of the complete return from the episode,

$$G_t \triangleq r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ... + \gamma^{T-1} r_{t+T-1} \tag{3.10}$$

where T is the last time step of the episode. Some such methods will be explained further in Section 3.4.1.

3.3.2 Temporal-Difference Learning

Temporal-Difference (TD) learning combines ideas from Dynamic Programming with Monte Carlo learning. TD methods learn directly from experience without the need for a model of the environment, like in Monte Carlo methods, and TD methods perform estimate updates without waiting for entire episodes to complete, like DP.

Estimating the Value Function

The simplest form of TD methods is called $TD(0)$, and it updates the estimate of the value function at every step of every episode. By looking one step into the future, it observes the received reward it got by following the policy, r, and combines this with the estimated value of the next state, $V(s')$. Then the estimate of $V(s)$ is moved towards this new observed value, called the TD target, using the update,

$$V(s) = V(s) + \alpha\underbrace{[\underbrace{r + \gamma V(s')}_{\text{TD target}} - V(s)]}_{\text{TD error, } \delta_t} \tag{3.11}$$

An equivalent update can be applied to an estimate of the action-value, $Q(s, a)$:

$$Q(s,a) = Q(s,a) + \alpha\underbrace{[\underbrace{r + \gamma Q(s', a')}_{\text{TD target}} - Q(s,a)]}_{\text{TD error, } \delta_t} \tag{3.12}$$

TD Control

When we want to use TD methods for the control problem (finding the optimal policy), Equation (3.12) must be used, since this makes it possible to estimate the value of each action in each state explicitly. The policy that is used must ensure that the state and action space is explored sufficiently, so that the action-value function can be estimated accurately in all parts of the state/action space. Often, an ϵ-greedy policy is used, which takes a greedy approach (the best action according to the estimated Q) with probability $1 - \epsilon$, and takes a random action with probability ϵ.

An on-policy TD control method is **SARSA**, whose name comes from the quintuple of events (s, a, r, s', a') that make up a transition from one state-action pair to the next. *On-policy* means that the action-value for the policy is estimated while at the same time, the policy is changed towards greediness with respect to the action-value. In Equation (3.12), this means that the next action a' is chosen by following the current policy.

Algorithm 1 SARSA

Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialise $Q(s, a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ arbitrarily. $Q(terminal, \cdot) = 0$
for each episode **do**
    Initialise s
    Choose a from s using policy derived from Q
    for each step of episode **do**
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
        $s \leftarrow s'$, $a \leftarrow a'$
    end for
end for

Q-learning is similar to SARSA, but an off-policy TD control method, meaning that the optimal action-value function q_* is directly approximated by Q, without depending on the policy being followed at any time step. The update of Equation (3.12) is modified to

$$Q(s, a) = Q(s, a) + \alpha[r + \gamma \max_a Q(s', a) - Q(s, a)] \tag{3.13}$$

and the procedure is shown in Algorithm 2.

n-step TD Methods

As we saw in Section 3.3.1, the Monte Carlo updates use the full return G_t when estimating the value function. In this section, we have shown that a one-step TD method, called $TD(0)$, observes only the immediately received reward and then estimates the remaining terms of the complete return by using the current estimated value of the successor state. It is considered a one-step update because it observes the outcome of one time step from the start state.

$$G_{t:t+1} = r_t + \gamma V_t(s_{t+1}) \tag{3.14}$$

Algorithm 2 Q-learning

Parameters: step size $\alpha \in (0, 1]$, small $\epsilon > 0$

Initialise $Q(s, a)$ for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$ arbitrarily. $Q(terminal, \cdot) = 0$
for each episode **do**
    Initialise s
    for each step of episode **do**
        Choose a from s using policy derived from Q
        Take action a, observe r, s'
        $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_a Q(s', a) - Q(s, a)]$
        $s \leftarrow s'$
    end for
end for

Here, V_t is the estimate of v_π at time t. Similarly, we could have a two-step TD update, based on the first two observed rewards, and the estimated value of the state after those two steps, at time $t + 1$. From this, we introduce n-step updates

$$G_{t:t+n} = r_t + \gamma r_{t+1} + ... + \gamma^{n-1} r_{t+n} + \gamma^n V_{t+n-1}(s_{t+n}) \tag{3.15}$$

where V_{t+n-1} is the estimated value function after n steps, at time $t + n - 1$. This means that, for $n > 1$, the n-step return involves rewards and states that have not yet been seen before reaching time $t + n$, thus the n-step return can not be computed until n steps of the episode have been carried out and the appropriate value function estimate has been computed.

By this definition, setting $n = T$ results in the Monte Carlo return, while $n = 1$ gives TD(0). According to Sutton & Barto [33, p. 157], the intermediate n-step methods will typically perform better than the extremes.

3.4 Reinforcement Learning with Function Approximation

All methods until now have been applicable to finite, discrete state spaces and action spaces, where the dimensions of the action-value matrix and state-value vector does not exceed the memory capacity of the computer solving the algorithms. This made it possible to learn a value for every single state or state-action pair, thus completely defining an optimal policy.

But what happens when the state or action space is continuous? After all, this is a more realistic scenario when developing control algorithms for physical systems. A simple solution is to discretise the spaces so that for instance Q-learning can be applied, but this gives rise to a trade-off between accuracy and computational efficiency. Information is lost when transforming continuous states and actions into discrete ones, but fine-grained discretisation could make the algorithm slow and inefficient. This section addresses how to solve problems with continuous state and action spaces.

Function approximation is way to handle such problems. Rather than using a lookup table to represent our function, such as the action-value matrix of Section 3.3.2, we can parameterise the function by a parameter vector $\boldsymbol{\theta} \in \mathbb{R}^d$. An example parameterisation

of a function $F(x)$ is a weighted linear function of a set of features, $F(x, \boldsymbol{\theta}) = \theta_1 f_1(x) + \theta_2 f_2(x) + ... + \theta_d f_d(x)$.

If we parameterise the value function or policy function by such a parameter vector, a reinforcement learning agent can learn values of the vector such that the parameterisation eventually approximates the true value function or the optimal policy. This has a few advantages, and the most obvious is that the number of parameters will be far fewer than the number of possible states and actions if the state and action spaces are large or continuous [14]. Another major advantage of the parameter vector is summarised in [14, p. 846], where a parameterised function is called a *function approximator* because it becomes an approximation of the true function:

"The compression achieved by a function approximator allows the learning agent to generalize from states it has visited to states it has not visited." ([14])

This means that in large or continuous spaces, it is no longer necessary to visit every single state or state-action pair in order to learn a good approximation of the value or policy. Assuming that the chosen parameter space makes it possible to approximate the value or policy sufficiently well, the RL agent can achieve this by adjusting the entries of $\boldsymbol{\theta}$.

3.4.1 Policy Gradient Methods

Instead of basing action selection on a learned state-value or action-value function, policy gradient methods learn a parameterised policy for selecting actions without checking the value function first. If we write the parameter vector for the policy as $\boldsymbol{\theta} \in \mathbb{R}^d$, the policy is $\pi(a|s, \boldsymbol{\theta}) = Pr\{A_t = a|S_t = s, \boldsymbol{\theta_t} = \boldsymbol{\theta}\}$. A major advantage of this kind of approach over the previously presented methods is that this allows the policy to generate actions in the complete continuous action space. A drawback is that it can get stuck in a local optimum, due to the approach being based on gradient ascent.

The only constraint on the way π is parameterised is that the gradient $\nabla_\theta \pi(a|s, \boldsymbol{\theta})$ exists and is finite for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$, $\boldsymbol{\theta} \in \mathbb{R}^d$. In a continuous action space, a deterministic policy can be a smooth function of $\boldsymbol{\theta}$, but with discrete actions, such a policy can switch between actions due to an infinitesimal change in parameters and consequently make the function non-differentiable with respect to its parameters. Therefore, in discrete action spaces it is usually required that the policy remains stochastic, i.e. that π represents a probability distribution, and its values remain in the interval $(0, 1)$ for all states, actions and parameter vectors. This also ensures exploration, since a stochastic policy introduces uncertainty in what action will be chosen from a state.

It is worth noting that a continuous action space can be represented as a probability distribution as well, and not only as a deterministic policy. For example, a normal distribution can be achieved by letting the function approximator give a mean and standard deviation representing the distribution of an action.

The Policy Gradient

We define a score function $J(\boldsymbol{\theta})$ as a measurement of the policy parameterisation's performance. In an episodic task, we choose the score function as the true value of the start state of the episode,

$$J(\boldsymbol{\theta}) \triangleq v_{\pi_\theta}(s_0) \tag{3.16}$$

and since we want to maximise the score, a way of updating the parameters is to use gradient ascent:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \nabla_\theta J(\boldsymbol{\theta}_t) \tag{3.17}$$

where α is the learning rate and $\nabla_\theta J(\boldsymbol{\theta})$ is the gradient of J with respect to $\boldsymbol{\theta}$. We do not know the true value function, thus we must approximate the gradient ascent update by obtaining samples, such that the expectation of the sampled gradient and the actual gradient are proportional to each other.

The policy gradient theorem [34] establishes that, for the episodic case,

$$\nabla J(\boldsymbol{\theta}) = \nabla v_\pi(s_0) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \tag{3.18}$$

where $\propto$ means "proportional to", and μ is the fraction of time spent in each state if following π, normalised to sum to one, called the on-policy distribution under π. From Equation (3.18) onwards, we omit the subscript θ from the gradient notation ∇ for simplicity, whenever there is no ambiguity regarding which parameter the differentiation is done with respect to.

We know that the sum over states weighed by the frequency of the states' occurrence under the policy π, as in (3.18), is the equivalent to an expectation under that same policy. By the same argument, we can replace the sum over actions of q_π by an expectation under π and then sampling the expectation, as shown below.

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\
&= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t,a) \nabla \pi(a|S_t,\boldsymbol{\theta}) \right] \\
&= \mathbb{E}_\pi \left[\sum_a \pi(a|S_t,\boldsymbol{\theta}) q_\pi(S_t,a) \frac{\nabla \pi(a|S_t,\boldsymbol{\theta})}{\pi(a|S_t,\boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[q_\pi(S_t,A_t) \frac{\nabla \pi(A_t|S_t,\boldsymbol{\theta})}{\pi(A_t|S_t,\boldsymbol{\theta})} \right] \\
&= \mathbb{E}_\pi \left[q_\pi(S_t,A_t) \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta}) \right]
\end{aligned} \tag{3.19}$$

The action-value $q_\pi(S_t, A_t)$ can be sampled from real or simulated experience, and then the gradient update of Equation (3.17) can be applied.

Monte Carlo Policy Gradient

From Section 3.3.1 we know that Monte Carlo methods sample complete episodes before making updates. Equation (3.19) can be rewritten as:

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\boldsymbol{\theta}) \\
&= \mathbb{E}_\pi \left[G_t \nabla \ln \pi(A_t|S_t,\boldsymbol{\theta}) \right] \qquad (\mathbb{E}_\pi[G_t|S_t,A_t] = q_\pi(S_t,A_t))
\end{aligned} \tag{3.20}$$

where G_t is the return from state-action pair (S_t, A_t), which includes all future rewards up until the episode's completion. Thus, using this gradient in (3.17) will be a Monte Carlo algorithm. This particular one is called REINFORCE [35] and we show the outline in Algorithm 3.

Algorithm 3 REINFORCE

Input: A differentiable policy parameterisation $\pi(a|s, \boldsymbol{\theta})$
Parameters: Step size $\alpha > 0$
Initialise policy parameter $\boldsymbol{\theta} \in \mathbb{R}^d$ (e.g to $\mathbf{0}$)
for each episode **do**
  Generate an episode $S_0, A_0, R_0, ..., S_{T-1}, A_{T-1}, R_{T-1}$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$
  for each step $t = 0, 1, ..., T-1$ of the episode **do**
    $G \leftarrow \sum_{k=t}^{T-1} \gamma^{k-t} R_k$
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta})$
  end for
end for

To explain this algorithm intuitively, let us look at the update equation in a different form, $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \gamma^t G_t \frac{\nabla \pi(A_t|S_t, \boldsymbol{\theta})}{\pi(A_t|S_t, \boldsymbol{\theta})}$. Here, $\gamma^t G_t$ is the discounted return from time step t. The discount factor is in this expression because we make an update at each time step of an episode, starting from $t = 0$, thus G_t must be discounted by t steps. The gradient vector is the gradient of the probability of taking action A_t from state S_t, and the expression in the denominator is the probability of taking that action. Combined, the update will change the parameter vector in the direction of highest increase in probability of taking this action on future visits to state S_t. Multiplying by the return makes the update largest in the directions of actions that gave higher returns, and dividing by the probability will prevent actions that occur frequently to be favoured over actions that yield better returns.

Policy Gradient with Baseline

While the stochastic policy gradient produces an unbiased estimate of the gradient of the expected total returns, these estimates can have large variance. It is possible to modify the Monte Carlo policy gradient method outlined above by including a comparison of the action value function with a baseline function, in the gradient. This can be done as long as the function $b(s)$ does not depend on a, thus the subtracted term will be zero:

$$\begin{aligned} \nabla J(\boldsymbol{\theta_\pi}) &\propto \sum_s \mu(s) \sum_a (q_\pi(s, a) - b(s)) \nabla \pi(a|s, \boldsymbol{\theta_\pi}) \\ &= \mathbb{E}_\pi \left[(G_t - b(S_t)) \nabla \ln \pi(A_t|S_t, \boldsymbol{\theta_\pi}) \right] \qquad (\mathbb{E}_\pi[G_t|S_t, A_t] = q_\pi(S_t, A_t)) \end{aligned} \tag{3.21}$$

In the above expression, $\boldsymbol{\theta_\pi}$ is the parameter vector for π.

Using a baseline function in the policy gradient estimate can reduce the variance while maintaining low bias, which helps performance. High variance results in needing more samples for the algorithm to converge, while bias can cause the algorithm to not converge at all, or converge to a poor solution.

A common choice for the baseline is an estimate of the state-value function, $V(s, \boldsymbol{\theta_v})$, parameterised by the vector $\boldsymbol{\theta_v} \in \mathbb{R}^m$. The state-value function parameters can be learned in a similar way as the policy parameters, for example by a Monte Carlo method and gradient ascent. Using the squared difference between observed return and estimated value as a score function, we get

$$J(\boldsymbol{\theta_v}) = \frac{1}{2} \left(\sum_{k=t}^{T-1} \gamma^{k-t} r_k - V(s_t, \boldsymbol{\theta_v}) \right)^2 = \frac{1}{2} \delta^2 \tag{3.22}$$

which has the gradient

$$\nabla_{\boldsymbol{\theta_v}} J(\boldsymbol{\theta_v}) = \delta \nabla_{\boldsymbol{\theta_v}} \delta = \delta \nabla_{\boldsymbol{\theta_v}} V(s, \boldsymbol{\theta_v}) \tag{3.23}$$

Then, we get the policy gradient

$$\nabla_{\boldsymbol{\theta_\pi}} J(\boldsymbol{\theta_\pi}) = (G_t - V(s_t, \boldsymbol{\theta_v})) \nabla_{\boldsymbol{\theta_\pi}} \ln \pi(a_t|s_t, \boldsymbol{\theta_\pi}) = \delta \nabla_{\boldsymbol{\theta_\pi}} \ln \pi(a_t|s_t, \boldsymbol{\theta_\pi}) \tag{3.24}$$

and the parameter update equations for the REINFORCE with baseline method become

$$\begin{aligned} \boldsymbol{\theta_v} &\leftarrow \boldsymbol{\theta_v} + \alpha_{\boldsymbol{\theta_v}} \delta \nabla V(s_t, \boldsymbol{\theta_v}) \\ \boldsymbol{\theta_\pi} &\leftarrow \boldsymbol{\theta_\pi} + \alpha_{\boldsymbol{\theta_\pi}} \gamma^t \delta \nabla \ln \pi(a_t|s_t, \boldsymbol{\theta_\pi}) \end{aligned} \tag{3.25}$$

Deterministic Policy Gradient

Rather than sampling a stochastic policy, deterministic policy gradients (DPG) use a deterministic policy $a = \mu_\theta(s)$, as the name suggests. According to [36], the deterministic policy gradient can be estimated much more efficiently than the stochastic gradient, by avoiding a problematic integral over the action space. It is shown that if we have a policy $\mu_\theta : \mathcal{S} \rightarrow \mathcal{A}$ with parameter vector $\boldsymbol{\theta}_\mu$, an action-value function $q_\mu(s, a)$ and a score function $J(\theta) = \mathbb{E}\left[R_t|S_t = s, A_t = \mu_\theta(s)\right]$, the deterministic policy gradient can be defined as

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \mu_\theta(s) \nabla_a q_\mu(s, a)|_{a=\mu_\theta(s)}\right] \tag{3.26}$$

3.4.2 Actor-Critic Methods

Methods that approximate a policy are often called actor-only methods, because a policy for action selection is learned. Methods that approximate a value function evaluates states given a policy, and are therefore called critic-only methods. The combination of policy and value approximation are therefore actor-critic methods. Here, an actor chooses and performs actions, while a critic finds the value of a state when following the policy of the actor. The critic helps the actor update its parameters to improve the policy, while also updating its own parameters. This framework is illustrated in Figure 3.2.

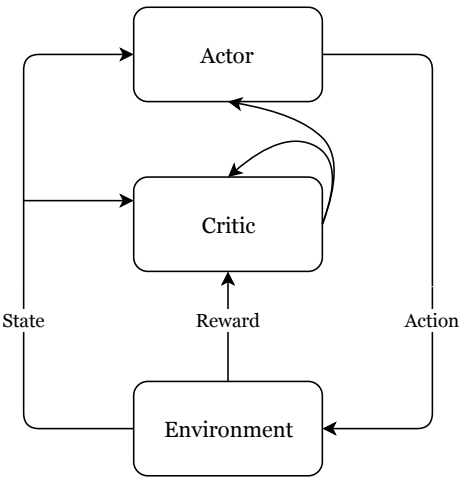


Figure 3.2: General Actor-Critic architecture

If the critic is parameterised by a vector $\boldsymbol{\theta}_Q$, and is learning an action-value function, then the output of the critic is $Q_{\theta_Q}(s, a|\boldsymbol{\theta}_Q)$. It can handle continuous states and actions, since the function approximator generalises from seen states and actions to unseen states and actions. The actor is parameterised by $\boldsymbol{\theta}_\pi$, resulting in the policy $\pi_{\theta_\pi}(a|s, \boldsymbol{\theta}_\pi)$.

The parameters $\boldsymbol{\theta}_\pi$ are adjusted by the actor according to the policy gradient as described in Section 3.4.1, but in place of the real action-value $q_\pi(s, a)$ in Equation (3.19),

the parameterised approximation $Q_{\theta_Q}(s, a|\boldsymbol{\theta}_Q)$ of the critic is used. Then, the policy gradient becomes

$$\nabla_{\theta_\pi} J(\boldsymbol{\theta}_\pi) = \mathbb{E}_\pi \left[Q_{\theta_Q}(s, a|\boldsymbol{\theta}_Q) \nabla_{\theta_\pi} \ln \pi(a|s, \boldsymbol{\theta}_\pi) \right] \tag{3.27}$$

The critic uses an appropriate method for estimating value functions, e.g. temporal-difference learning (Section 3.3.2). By this we mean that the temporal-difference error is used in the parameter update for the critic,

$$\delta_t = r_t + \gamma Q_{\theta_Q}(s_{t+1}, a_{t+1}|\boldsymbol{\theta}_Q) - Q_{\theta_Q}(s_t, a_t|\boldsymbol{\theta}_Q) \tag{3.28}$$

$$\boldsymbol{\theta}_{Q_{t+1}} = \boldsymbol{\theta}_{Q_t} + \alpha_Q \delta_t \nabla_{\theta_Q} Q_{\theta_Q}(s_t, a_t|\boldsymbol{\theta}_Q) \tag{3.29}$$

3.4.3 Deep Deterministic Policy Gradient (DDPG)

The architecture of deep deterministic policy gradient (DDPG)[19] is similar to the actor-critic architecture. In DDPG, both a policy and a value function are approximated by using neural networks as function approximators.

An action-value function is used in the critic, which enables the agent to learn an optimal policy even when it is not performing the optimal policy, because the values of all actions in all states are learned. This is especially useful here, since the learned policy in DDPG is deterministic. Thus, following the learned policy while training would lead to insufficient exploration of the environment.

The actor computes the deterministic policy parameters by utilising the deterministic policy gradient described in Section 3.4.1, such that the updates are given by

$$\begin{aligned} \boldsymbol{\theta_\mu} &\leftarrow \boldsymbol{\theta_\mu} + \alpha_\mu \nabla_{\theta_\mu} Q_{\theta_Q}(s, \mu_\theta(s)) \\ &= \boldsymbol{\theta_\mu} + \alpha_\mu \nabla_{\theta_\mu} \mu_\theta(s) \nabla_a Q_{\theta_Q}(s, a)|_{a=\mu_\theta(s)}) \end{aligned} \tag{3.30}$$

In addition to the actor-critic structure, deterministic policy and neural networks, DDPG uses experience replay [37], where experiences are saved in memory as quadruples consisting of the state, action, observed reward and the next experienced state, (s, a, r, s'), and training data is sampled uniformly from this memory. This breaks the temporal correlation between experiences, as we can sample mini batches of experiences that did not appear sequentially.

In order to stabilise training when using non-linear function approximators, separate target actor and critic networks are used for predicting the TD target in the critic's parameter update. This prevents the network parameters that are updated from also being used in the prediction that produces the update. The target network parameters are updated so that they slowly track the learned networks:

$$\begin{aligned} \boldsymbol{\theta}_{Q'} &\leftarrow \tau \boldsymbol{\theta}_Q + (1 - \tau) \boldsymbol{\theta}_{Q'} \\ \boldsymbol{\theta}_{\mu'} &\leftarrow \tau \boldsymbol{\theta}_\mu + (1 - \tau) \boldsymbol{\theta}_{\mu'} \end{aligned} \tag{3.31}$$

where τ is the rate of updates for the target networks.

As mentioned, DDPG is an off-policy algorithm, thus we also need to decide which strategy to follow when exploring the environment. The authors of [19] use an Ornstein-Uhlenbeck noise process [38] $\mathcal{N}$, which "models the velocity of a Brownian particle with friction, which results in temporally correlated values centered around 0" ([19]), and adds this to the learned policy to ensure exploration:

$$a_t = \mu_{\theta_\mu}(s_t) + \mathcal{N}_t \tag{3.32}$$

This results in Algorithm 4.

Algorithm 4 DDPG

Randomly initialise critic network $Q_{\theta_Q}(s, a)$ and actor $\mu_{\theta_\mu}(s)$ with weights θ_Q and θ_μ.
Initialise target network Q' and μ' with weights $\theta_{Q'} \leftarrow \theta_Q$, $\theta_{\mu'} \leftarrow \theta_\mu$
Initialise replay buffer R
for $episode = 1, ..., M$ **do**
    Initialise random process $\mathcal{N}$ for action exploration
    Receive initial observation state s_1
    for $t = 1, ..., T$ **do**
        Select action $a_t = \mu_{\theta_\mu}(s_t) + \mathcal{N}_t$ according to the current policy and exploration noise
        Execute action a_t and observe reward r_t and new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in R
        Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
        Set $y_i = r_i + \gamma Q'_{\theta_{Q'}}(s_{i+1}, \mu'_{\theta_{\mu'}}(s_{i+1}))$ for $i \in 1...N$
        Update critic by minimising loss: $L = \frac{1}{N}\sum_i (y_i - Q_{\theta_Q}(s_i, a_i))^2$
        Update actor policy using the sampled policy gradient:

$$\nabla_{\theta_\pi} J \approx \frac{1}{N} \sum_i \nabla_{a_i} Q_{\theta_Q}(s_i, a_i) \nabla_{\theta_\mu} \mu_{\theta_\mu}(s_i)$$

        Update the critic target network: $\theta_{Q'} \leftarrow \tau\theta_Q + (1 - \tau)\theta_{Q'}$
        Update the actor target network: $\theta_{\mu'} \leftarrow \tau\theta_\mu + (1 - \tau)\theta_{\mu'}$
    end for
end for

Part III

Results

Chapter 4

Implementation of Reinforcement Learning Methods for Simple Environments

In the following chapter, we present tasks that have been solved in this thesis, the approaches that were used to solve them and results. To become familiar with reinforcement learning methods, their advantages and limitations, three simple OpenAI control problems were solved. Section 4.1 presents a grid-world problem in which an agent must find a path from start to goal through a discrete environment. This can be viewed as an introductory problem, as the number of states and actions is low, and a novel tabular learning approach can be used to solve it. Section 4.2 introduces an environment with continuous state space, and Section 4.3 further introduces continuous actions. Consequently, these problems require more complex solution methods, and were solved using function approximation. The REINFORCE algorithm was implemented for the CartPole of Section 4.2, and for the Pendulum of Section 4.3, DDPG was implemented.

It is important for reinforcement learning agents to be able to explore their environment, in order to figure out what states and actions are good or bad. In physical systems, this presents a challenge because such exploration can put both the system that is exploring, and the environment, in danger. To minimise damage to the system, simulated environments are used here.

4.1 OpenAI Gym's FrozenLake

FrozenLake is an environment provided by OpenAI gym. The lake is a 4×4 gridworld with start state in the top left corner and the goal in the bottom right corner, with holes spread around the lake, shown in Figure 4.1. The states are numbered as

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

and an agent moving in the environment has four possible actions: $\{left = 0,\ down = 1,\ right = 2,\ up = 3\}$.

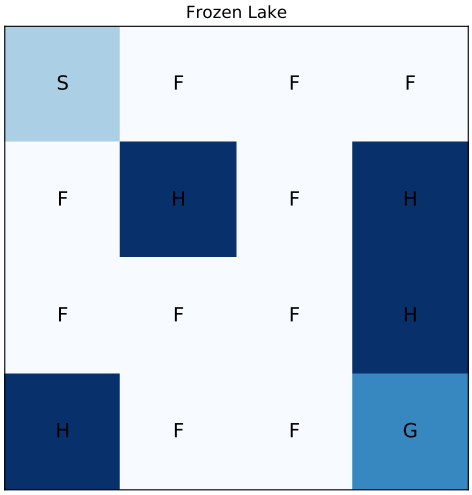


Figure 4.1: FrozenLake environment. S: Start, F: Frozen, H: Hole, G: Goal.

The world is analogous to a frozen lake, where walking on slippery ice leads to uncertain outcomes. The uncertainty is described in the transition matrix $P[s', s, a] = \mathbf{Pr}\{s'|s, a\}$, where the probability of going in the intended direction, or to the left or right of this direction, is 1/3 (thus the three probabilities sum to 1, and moving backwards is impossible). An example of trying to move down in state 1 is shown in Equation (4.1).

$$\mathbf{Pr}(s'|s = 1, a = 1) = \begin{cases} \frac{1}{3}, & \text{if } s' = 0 \\ \frac{1}{3}, & \text{if } s' = 2 \\ \frac{1}{3}, & \text{if } s' = 5 \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

A reward of +1 is given when the goal state is reached, otherwise there are no rewards. The holes and the goal are terminal states, so moving into any of these forces the agent to start over from the start state.

4.1.1 Solution Method

This environment has a limited number of states and actions, therefore a tabular solution method is suitable. Note that the transition model is available, thus an algorithm from dynamic programming may be used to solve iteratively for the value function. However, we can assume the transition probabilities are only available because we have access to the underlying implementation of the environment, thus we can let them remain unknown to the agent. Then an appropriate solution is to apply a temporal difference learning method instead. We choose Q-learning, described in Algorithm 2.

Experiment Details

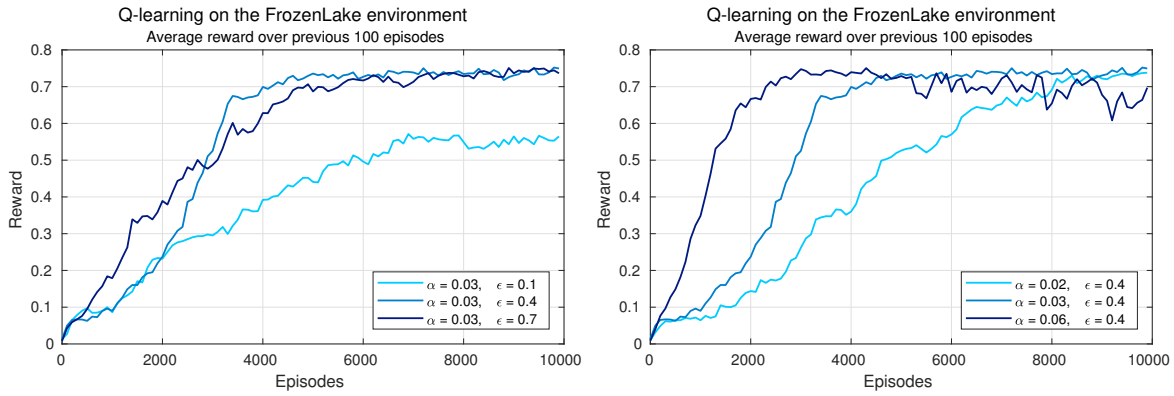
The Q-values were initialised to 0 for all state-action pairs. The algorithm then ran for 10 000 episodes, and this was repeated 10 times to obtain averaged results. For each episode of running the algorithm, 5 evaluations were carried out where exploration was

turned off, to test the currently learned policy. The test results were averaged over the 5 evaluation episodes.

The discount factor was set to $\gamma = 0.999$.

4.1.2 Simulations: Q-learning

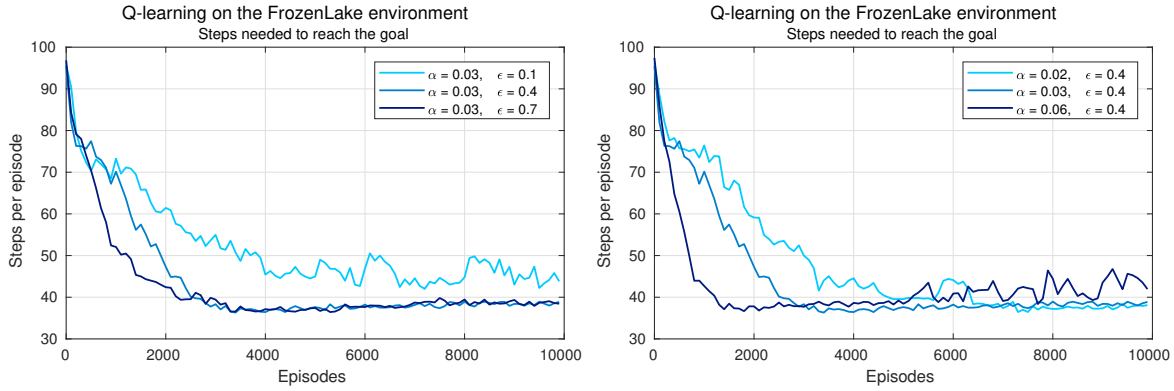
Simulation results are presented in Figure 4.2, showing the reward history, and 4.3, illustrating the number of steps the agent needs to reach the goal from the start state. A maximum test limit was set to 100 steps, and we see that in the first episodes, the agent does not reach the goal at all. The optimal policy learned by the agent and its corresponding Q-values are illustrated in Figure 4.4.



(a) Varying exploration rate

(b) Varying learning rate

Figure 4.2: Reward history for Q-learning in the FrozenLake environment



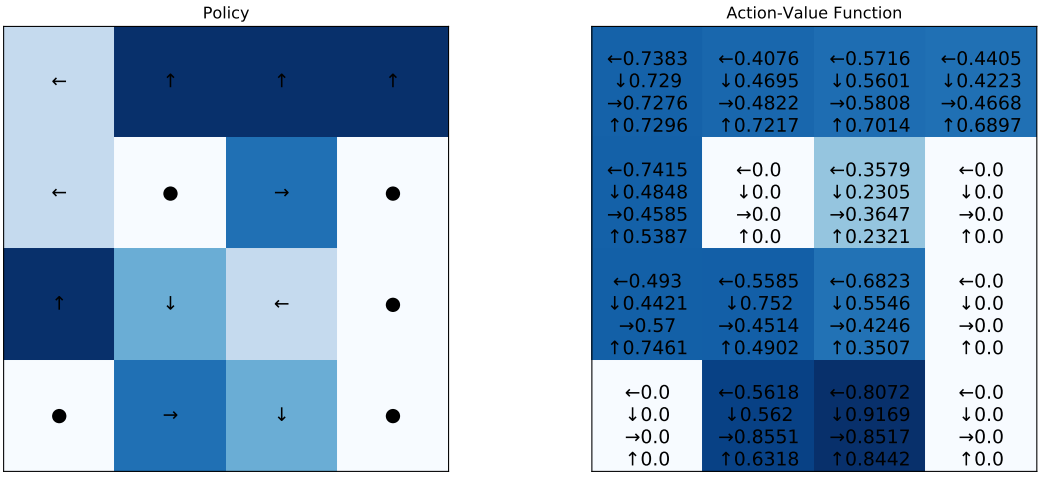
(a) Varying exploration rate

(b) Varying learning rate

Figure 4.3: Number of steps needed to reach the goal, using Q-learning in the FrozenLake environment

The convergence of Q towards q_* depends on the exploration factor ϵ, the learning rate α, and on how many time steps it is allowed to run.

Higher exploration rate can lead to less exploitation of the routes the agent has learned are good routes, and may in this case lead to falling into a hole at almost every episode and receiving zero reward, leading to states closer to the goal being less frequently visited.



(a) Optimal policy

(b) Action-values yielding optimal policy

Figure 4.4: Results in the FrozenLake environment using Q-Learning

Figure 4.2a and 4.3a shows that this may lead to the agent finding some good solutions early on, but the exploration is so high that it takes longer to reach the best policy. Here, $\epsilon = 0.7$ looks as though it will be the best exploration rate early on, but is surpassed by $\epsilon = 0.4$ Too low exploration and the agent may not learn the correct Q-value for all states, or may learn very slowly. This is reflected in Figures 4.2a and 4.3a where one can see that the lowest ϵ of 0.1 does not converge within the 10000 episodes.

Low learning rate can lead to slow convergence and thus many steps must be taken to learn the values. If too high, the algorithm may not converge or becomes unstable. Figure 4.2b and 4.3b illustrate these arguments, and we find $\alpha = 0.03$ to be an appropriate value. Here, α was constant, but it may lead to better convergence if α decreases with time.

At first glance, the policy found doesn't appear to make much sense. However, as the environment is highly stochastic, the optimal policy will be to always attempt to move in the opposite direction of nearby holes, as this is the only way to avoid any possibility of falling into them. By following the optimal policy in Figure 4.4, the only state with any possibility of falling into a hole is state 6, which is reflected by its Q-values, the lowest of all states. In this particular state, there is no obvious reason why moving right should be better or worse than moving left, as the probabilities of going up or down will be the same in both cases. Thus, the optimal Q-value should have the property $q_*(6, \text{ left}) = q_*(6, \text{ right})$. The learned Q is thus only close to optimal.

4.2 OpenAI Gym's CartPole

In the CartPole environment, a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The agent's observation consists of four values: the cart position along the track, the cart velocity, the pole angle with respect to the vertical axis, and the pole's velocity at the top of the pole. There are two available actions that push the cart to the left or to the right with a force of fixed magnitude.

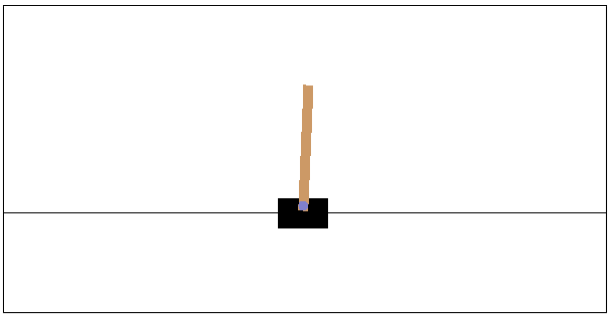


Figure 4.5: Screengrab from the CartPole environment

The pendulum starts in an upright position, with all observations assigned a uniform random value between -0.05 and 0.05. The episode terminates when the absolute value of the angle of the pole is more than 24°, or if the cart exceeds its maximum or minimum position along the track. Each episode is limited to a length of 200 steps.

The goal is to prevent the pole from falling over. A reward of +1 is given for every timestep that the pole remains upright, and the environment is considered solved when the agent gets an average reward of 195.0 over 100 consecutive trials.

Observation	Min	Max
Position	-4.8	4.8
Velocity	$-\infty$	∞
Pole angle	-24°	24°
Pole velocity	$-\infty$	∞

(a) Observation space

Action
Left
Right

(b) Action space

Table 4.1: The CartPole environment

4.2.1 Solution Method

Since the observation space is continuous and the action space is discrete, policy gradients can be applied, and a suitable solution method is REINFORCE with a neural network that approximates the policy distribution, as described in Algorithm 3. It takes states as input and gives a distribution over the two states as output. The neural network allows the agent to generalise from observed to unseen states.

Experiment Details

The network for approximating the policy had two hidden layers of dimensions 10 and 2. The input layer had four units, representing the four state values, and the output layer had two units corresponding to the two actions. The hidden layer activation function was the rectified linear unit (ReLU), and softmax was used as the output activation to ensure that the output could be considered a probability distribution over the two actions.

Adam [29] was used for learning the neural network parameters, with a learning rate of $\alpha = 0.01$. The discount factor for calculating returns was $\gamma = 0.95$. The training was carried out over 5000 episodes of maximum 200 time steps each.

4.2.2 Simulations: REINFORCE

Simulation results can be seen in Figure 4.6, where a moving average of the rewards were computed over the previous 100 episodes. Three variants are shown: with and without normalisation of returns, and a modified version of the implementation with normalisation, which will be further explained. The limit of 195.0 shows us when the environment can be considered solved.

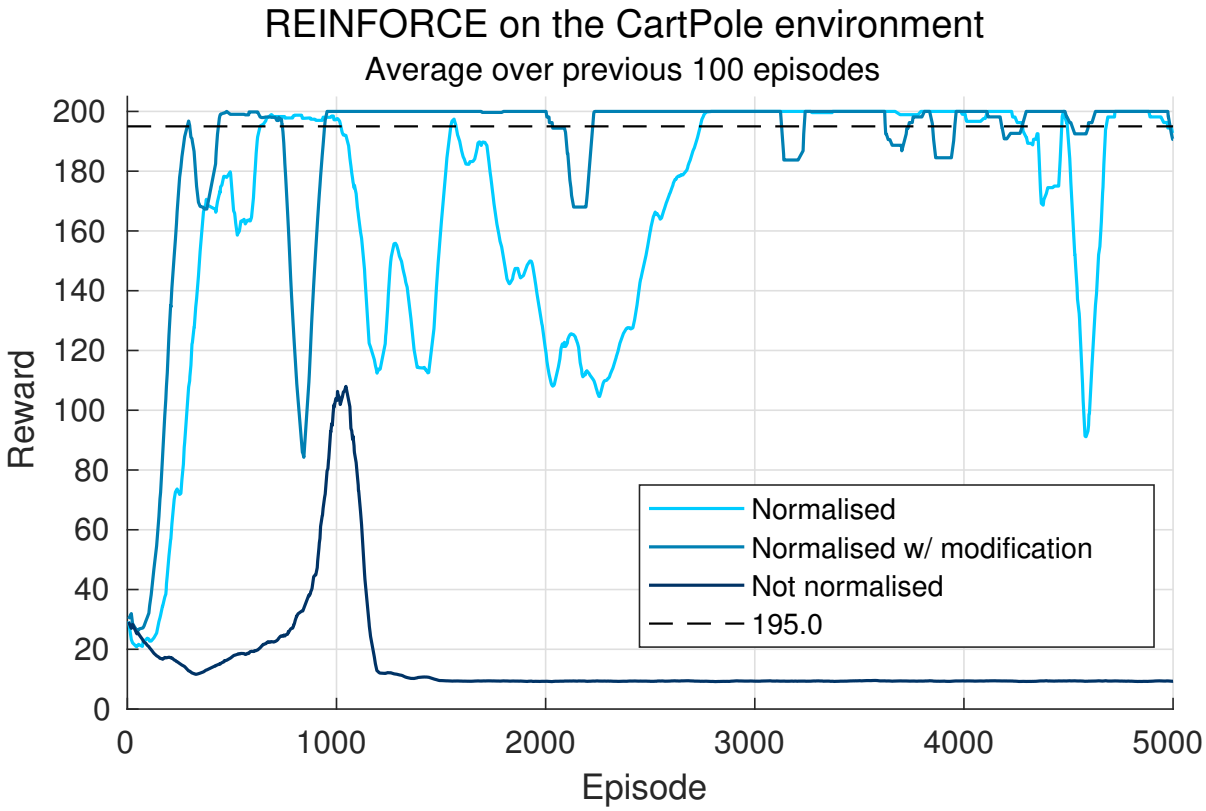


Figure 4.6: Reward history for CartPole trained using REINFORCE

The algorithm without normalisation is not even close to solving the task after 5000 episodes, while the version with normalisation of returns performs much better. In this case, it is evident that by the definition of *solved* environment, the training could have stopped as early as approximately 800 episodes, where the plotted moving average crosses the 195.0 limit. The drops in performance when training continues beyond this point, is a common occurrence in REINFORCE. This could be due to overfitting of the network to the observed data. To counteract overfitting, dropout or weight regularisation could be used. Another reason for this phenomenon could be the step limit per episode. When reaching 200 steps, the agent cannot distinguish between termination due to losing or due to the step limit being reached. This may cause the agent to attempt new combinations of actions in order to increase the score, unaware of the fact that 200 is the maximum. Once the agent realises that it cannot find better solutions, it once again learns to balance until the end of the episode.

In an attempt to mitigate this effect, a modified algorithm with normalisation of returns was implemented. In this version, whenever an episode terminated due to reaching the step limit, we did not allow the agent to update its policy parameters. We justify this by saying that in these cases the agent did not "lose", thus we do not want it to learn that

it did so. When an episode terminates due to the pole falling over, the agent learns in the same manner as before. By examining Figure 4.6 one can see that this modification seems to stabilise the training in this environment.

Another modification that could have improved performance is the introduction of a baseline function, for example by using an estimate of the value function. This would have increased complexity of the solution, since a function approximator for the value function must be implemented, however the variance of the estimated returns would be reduced, which could decrease convergence time of the REINFORCE algorithm.

4.3 OpenAI Gym's Pendulum

This is an inverted pendulum swing-up problem. An agent's available observations are the angle θ, indirectly in the form of $\sin\theta$ and $\cos\theta$, and the angular velocity $\dot{\theta}$, summarised in Table 4.2a. It can apply a torque between -2 and 2 to the pendulum in order to reach the goal of balancing the pendulum vertically at $\theta = 0$.

Observation	Min	Max
$\cos\theta$	-1.0	1.0
$\sin\theta$	-1.0	1.0
$\dot{\theta}$	-8.0	8.0

(a) Observation space

Action	Min	Max
Torque τ	-2.0	2.0

(b) Action space

Table 4.2: The Pendulum environment

The reward at each time step is

$$r = -(\theta^2 + \frac{1}{10}\dot{\theta}^2 + \frac{1}{1000}\tau^2) \tag{4.2}$$

where θ is normalized between $-\pi$ and π. The maximum possible reward is then 0, and the lowest is $-(\pi^2 + 0.1 * 8^2 + 0.001 * 2^2) = -16.2736$. Thus, achieving maximum reward means to remain at zero angle (vertical position) with minimum rotational velocity and applying minimum effort.

The starting state of each episode is decided by a random angle between $-\pi$ and π, and angular velocity between -2 and 2. The only termination criteria for this environment is the number of time steps, which is maximum 200 per episode. Unlike the CartPole environment, the Pendulum does not have a specified reward threshold at which it is considered solved.

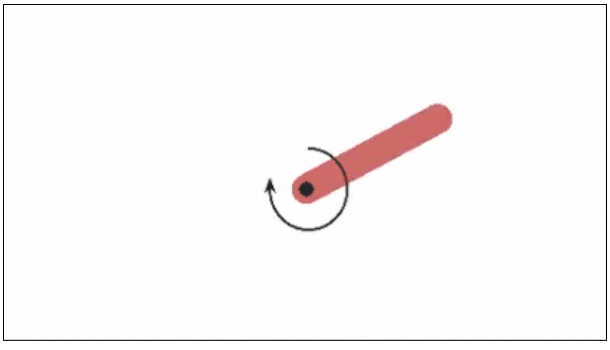


Figure 4.7: Screengrab from the Pendulum environment

4.3.1 Solution Method

Both the state space and action space of this environment is continuous. Deep deterministic policy gradients, introduced in Algorithm 4 in Section 3.4.3, are especially suitable for these kinds of environments, since the method approximates both the action-value function and a deterministic policy for action selection, using neural networks.

Experiment Details

Both the actor and critic networks had 2 hidden layers with 400 and 300 units respectively, and ReLU was used for all units in the hidden layers. The output layer of the actor used a hyperbolic tangent activation function in order to bound actions between -1 and 1. This output was then scaled to the range of the torque, which was $[-2, 2]$. The critic network's output was the Q-value, and should be able to take any value, thus no activation function was used here. Both networks had one output unit, and six input units representing the state. Actions were also used as input to the critic, but were not included until the second hidden layer.

The discount factor of the critic's TD error was $\gamma = 0.99$. For the soft target updates $\tau = 0.001$ was used. Adam [29] was used for learning the neural network parameters, with a learning rate of 10^{-4} and 10^{-3} for the actor and critic respectively. A minibatch size of 64 and a replay buffer size of 10^6 was used.

In order to explore we used an Ornstein-Uhlenbeck process [38] with $\theta = 0.15$ and $\sigma = 0.2$.

The agent trained for 600 episodes of 200 time steps each, for a total of 120000 steps.

4.3.2 Simulations: DDPG

The Pendulum environment was solved using DDPG, with function approximator designs as presented in Section 4.3.1. Results are presented in Figure 4.8, which shows the rewards from evaluation runs where exploration was turned off and only the current learned policy was followed. The opaque blue line is the real rewards, while the darker line is a smoothed version of the received rewards. From the figure we can see that the agent improves until converging to a mean reward of approximately -150, which is where it has learned to solve the task by swinging up the pendulum. The achieved reward varies because the starting conditions vary.

Testing the trained agent from a starting angle of $\theta = \pi$ and zero angular velocity, $\dot{\theta} = 0$, we get the results in Figure 4.9, where 4.9a shows the applied torque and 4.9b shows the resulting angle and angular velocity of the pendulum. For plotting purposes we do not normalise θ between $-\pi$ and π, which ensures the resulting plot is continuous and thus easier to interpret. Here, $\theta = 0$ at the top position and increases as the pendulum rotates clockwise, so that $\theta = \pi$ at rest in the bottom position, and $\theta = 2\pi$ after one complete rotation. From this, we can conclude that the pendulum swings back and forth a few times to gain momentum, before swinging up to the goal position in the counterclockwise direction. The angle and angular velocity converge to zero, while the applied torque converges to a small value.

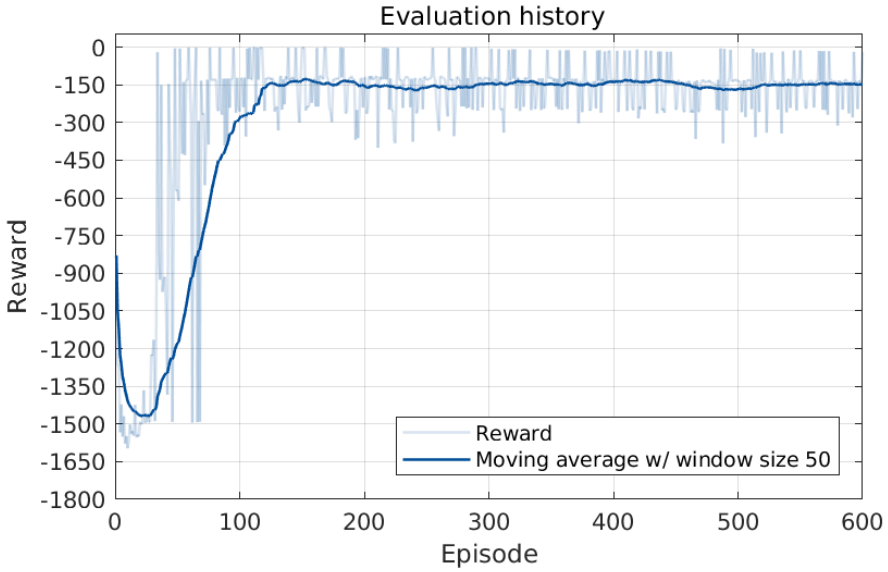
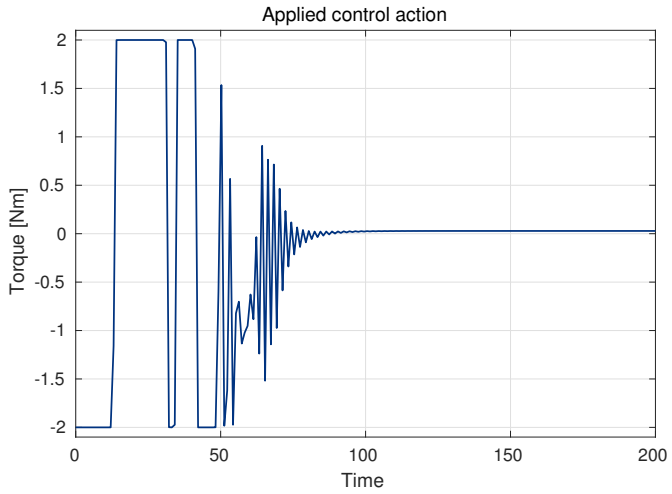
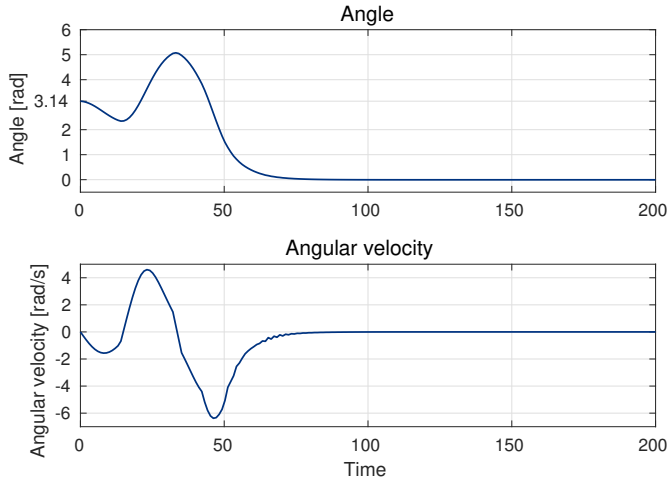


Figure 4.8: Reward history for the Pendulum trained using DDPG



(a) Control action



(b) Angle and angular velocity

Figure 4.9: Test of the Pendulum trained by DDPG

Chapter 5

Application of DDPG to a Vehicle Manoeuvring Task

This chapter shows the implementation and results of a vehicle manoeuvring task using deep reinforcement learning, which will provide insight into whether such a system can be useful. The chapter focuses on the DDPG algorithm, because this has proven to be fairly straightforward to implement, and has been tested in a simple environment in Chapter 4. The manoeuvring task has continuous state space and continuous action space, thus a deep reinforcement learning method, such as DDPG, is necessary. An alternative algorithm would have been Proximal Policy Optimisation (PPO), which is a DRL method that has shown promise in physical control tasks [39]. PPO computes an update of the policy at each time step that maximises the score function while ensuring the deviation from the previous policy is small. This makes PPO more robust in terms of tuning of step size than DDPG. It is also considered to be more sample efficient than DDPG, needing less time steps in an environment before learning tasks.

An environment consisting of a point mass whose goal is to reach a pre-defined position by applying force in a direction and with a certain magnitude is implemented in Section 5.1. The framework for implementation of RL problems provided by OpenAI Gym was used when implementing the environment. Training the DDPG agent in this environment will give an indication of whether a vehicle control system or path planner can be implemented with reinforcement learning. Several reward function designs will be considered in Section 5.2, before simulation results are presented and discussed in Section 5.3 and 5.4, respectively.

5.1 The Point Mass Environment

5.1.1 Vehicle Dynamics

A general vehicle was modelled as a point mass whose movement is decided by a force $\vec{F}_u$ with magnitude F_u and direction θ. An illustration of the point and applied force in the case where friction is ignored is presented in Figure 5.1. Friction was modelled as Coulomb friction with stiction, meaning that while the vehicle is in motion, the friction is constant and equal to the Coulomb friction, and when stationary the friction is slightly larger and equal to the static friction.

The direction θ is defined as the angle relative to the x-axis in the counterclockwise direction. Positions (x, y) and angles are defined in an inertial (stationary) coordinate

frame.

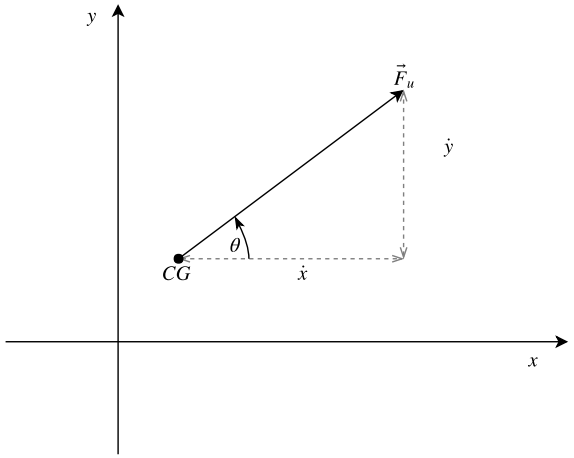


Figure 5.1: Illustration of a point mass with an applied external force

Newton's second law for the movement of a point mass in an inertial reference frame gives

$$m\vec{a} = \sum \vec{F} = \vec{F}_u - \vec{F}_f \tag{5.1}$$

$$\vec{a} = \frac{1}{m}\left(\vec{F}_u - \vec{F}_f\right) \tag{5.2}$$

where $\vec{F}_u$ is the applied force vector, $\vec{F}_f$ is the friction force, $\vec{a}$ is the point's acceleration and m is the mass. By decomposing the forces and inserting them in (5.2) we get

$$F_{ux} = F_u \cos\theta \qquad\qquad F_{uy} = F_u \sin\theta \tag{5.3}$$

$$a_x = \ddot{x} = \frac{1}{m}\left(F_u \cos\theta - F_{fx}\right) \tag{5.4}$$

$$a_y = \ddot{y} = \frac{1}{m}\left(F_u \sin\theta - F_{fy}\right) \tag{5.5}$$

Coulomb friction models the friction force as proportional to load, opposing motion and independent of contact area. Stiction, also called static friction, models how the friction force can be larger for zero velocity than for a particle in motion [40]. This means that the system sticks when the velocity is zero and the applied force is less than the stiction force, $F_u < F_s$, and the system breaks away when the applied force is equal to the stiction force, $F_u = F_s$. Here, the stiction force F_s is larger than the Coulomb force F_c. This effect can be implemented as in Equation (5.6), and an illustration of the friction model in the x-direction is shown in Figure 5.2.

$$\begin{aligned} F_{fx} &= \begin{cases} F_s, & \text{if } \dot{x} = 0 \text{ and } F_u < F_s \\ F_c \operatorname{sgn}(\dot{x}) = \mu m g \cdot \operatorname{sgn}(\dot{x}), & \text{otherwise} \end{cases} \\ F_{fy} &= \begin{cases} F_s, & \text{if } \dot{y} = 0 \text{ and } F_u < F_s \\ F_c \operatorname{sgn}(\dot{y}) = \mu m g \cdot \operatorname{sgn}(\dot{y}), & \text{otherwise} \end{cases} \end{aligned} \tag{5.6}$$

where μ is the friction coefficient. Th gravity of earth is denoted $g = 9.81 m/s^2$.

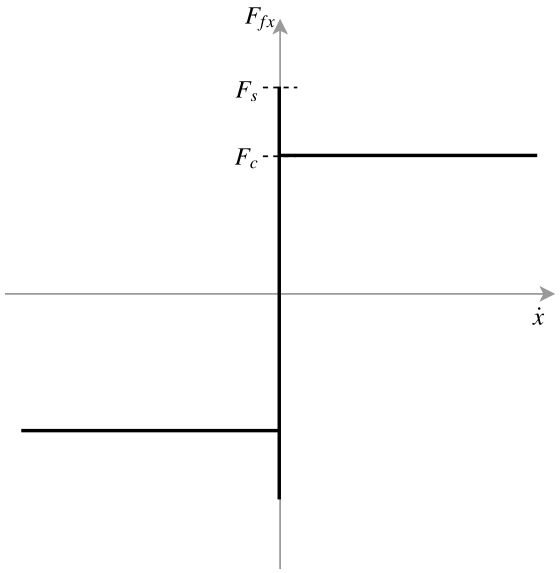


Figure 5.2: Friction model

For simulation of Equations (5.4)-(5.5), we use the simple numerical integration scheme Euler's method with step size $\Delta t = 0.05$.

In this environment, the direction θ and magnitude of force F_u are the inputs to the system, and these are applied instantly, thus there are no separate dynamics for the inputs. The following section further clarifies this.

5.1.2 Problem Formulation

The observation vector is given by

$$\boldsymbol{s} = \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \dot{\tilde{x}} \\ \dot{\tilde{y}} \\ d \\ \dot{d} \end{bmatrix} = \begin{bmatrix} x - x_g \\ y - y_g \\ \dot{x} \\ \dot{y} \\ d \\ \dot{d} \end{bmatrix} \tag{5.7}$$

where $d = \sqrt{\tilde{x}^2 + \tilde{y}^2}$ is the euclidean distance between the current position and the goal, and the action vector is

$$\boldsymbol{a} = \begin{bmatrix} \theta \\ F_u \end{bmatrix} \tag{5.8}$$

The observation and action ranges are shown in Table 5.1. The goal position is $(x_g, y_g) = (0, 0)$. The episodes begin with initial $\tilde{x}$- and $\tilde{y}$-positions, $(\tilde{x}_s, \tilde{y}_s)$, sampled from a uniform distribution between -4 and 4, and velocities at zero. The initial distance to the goal is calculated according to the position coordinates, and the derivative of the distance is zero. An episode terminates when it reaches the maximum number of time steps of 200.

By defining elements in the state vector relative to (x_g, y_g), the problem becomes independent from the placement of the origin of the chosen coordinate frame. Distance d is included in the observation because it defines the position error that is to converge to

Observation	Min	Max
$\tilde{x}$	-4.0	4.0
$\tilde{y}$	-4.0	4.0
$\dot{x}$	-1.0	1.0
$\dot{y}$	-1.0	1.0
d	-10.0	10.0
$\dot{d}$	$-\infty$	$-\infty$

(a) Observation space

Action	Min	Max
θ	$-\pi$	π
F_u	0	2.0

(b) Action space

Table 5.1: The point mass environment

zero. It is calculated using only $\tilde{x}$ and $\tilde{y}$ and thus does not introduce unique information to the system. However, augmenting the state with mappings from coordinates to other information can speed up learning, since the networks no longer have to approximate these mappings.

As mentioned in Section 5.1.1, since the angle θ and force F_u are inputs to the vehicle, there are no equations implemented that describe the change in θ and F_u for the vehicle. This means that the input vector defines the force $\vec{F}_u$, with magnitude F_u and direction θ, that is applied to the vehicle at every time step. These signals may change quickly, and so there are no limitations on the turning rate or change of acceleration that can be achieved by the vehicle.

The observation vector of Equation (5.7) is the input to the actor network of the DDPG agent, while the action vector (5.8) becomes the output of the actor network. Both the observations and actions are inputs to the critic network, where the actions are not included until the second hidden layer. The output of the critic is the estimated action-value of the observation and action. This design of the DDPG algorithm is the same as what was used on the Pendulum in Section 4.3.1.

5.2 Reward Function Design

Representative reward function designs for the presented tasks are introduced below. All rewards of this section will be tested in the environment described in Section 5.1.

The reward function is an important part of the reinforcement learning algorithm, as this is the agent's measure of how well it is performing. Therefore it is necessary to design it in a way that makes sure the agent understands what we, as the designers, consider to be the optimal behaviour.

When designing a reward signal in the point mass environment, the most important thing to consider is convergence to the goal position. A sparse signal, giving nonzero reward only exactly at the goal position is unlikely to converge, as the probability of visiting the exact goal position during exploration is small, which leads the agent to learn very little about the environment. Therefore, letting the agent receive reward for being in the vicinity of the goal might improve performance and decrease convergence time. In these environments, it is crucial to not only get to a position close to the goal, but to minimise the distance, therefore choosing a signal that increases as the distance is reduced, reaching its maximum at zero distance could be a good candidate for the reward function.

5.2.1 Reward Function 1

To achieve this, a Gaussian function is considered, which takes the shape of a bell curve as shown in Equation (5.9) and Figure 5.3, with amplitude a, mean μ and standard deviation σ.

$$g(d) = ae^{-\frac{(d-\mu)^2}{2\sigma^2}} \tag{5.9}$$

This will give the agent some knowledge about how its distance to the goal, d, relates to the performance measure. However, the control problem contains infinitely many states *and* actions, thus the transformation from states to actions may be complicated and learning may be time-consuming. In other words, this simple reward may be of use, but to speed up training of the agent we propose an additional reward when the agent reaches an area defined by a small radius around the goal, to make sure the agent tries to remain within this area. The total reward signal becomes

$$r_1 = ae^{-\frac{d^2}{2\sigma^2}} + \begin{cases} c_d, & \text{if } d < 0.05 \\ 0, & \text{otherwise} \end{cases} \tag{5.10}$$

where $a > 0$ (amplitude of the Gaussian) and $c_d > 0$ are weighting factors.

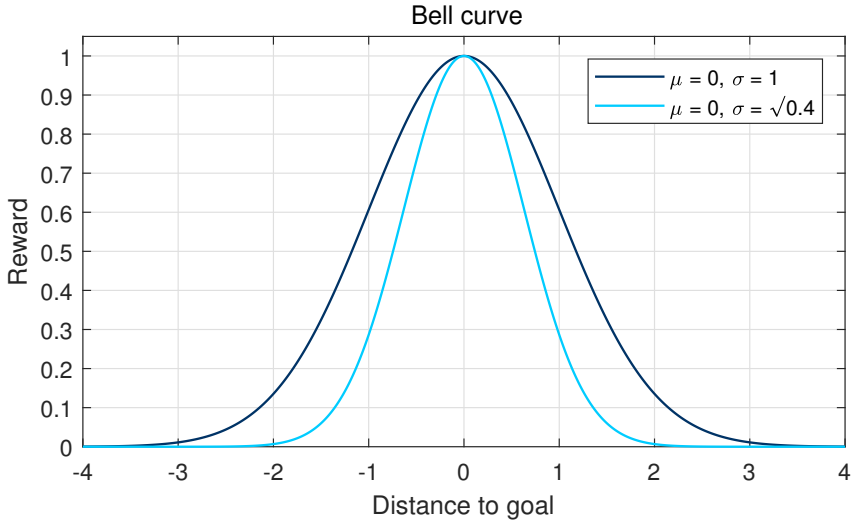


Figure 5.3: Example of reward function consisting of a Gaussian function with amplitude $a = 1$, mean $\mu = 0$ and standard deviation $\sigma = 1$ or $\sigma = \sqrt{0.4}$

5.2.2 Reward Function 2

An alternative to the Gaussian reward signal, that also holds the properties suggested in the introduction to the reward function design section, is implemented here. This reward function has a different form than r_1. Reward Function 1 will produce approximately zero reward everywhere except in some area surrounding (x_g, y_g), depending on the choice of standard deviation, which may cause the learning to be slow. Thus, the reward signal proposed in this section, called r_2, is designed to be less sparse, which can help the agent learn about good behaviour more quickly.

The design is based on receiving the largest reward at the goal, with a small area surrounding the goal giving significantly larger rewards than the rest of the environment. In this area, the signal is given by a Gaussian with an amplitude that ensures this reward

is larger than outside the radius. The Gaussian has small standard deviation in order to make it clear that the reward can increase further by reducing the distance. When outside of this area, the largest rewards are obtained when the vehicle travels straight towards the goal, and is given by a value proportional to the derivative of distance, $\dot{d}$. This is added to ensure that the agent can quickly understand what direction of travel moves it towards the goal.

It will become evident in Section 5.3 that the first reward function, r_1, described in Section 5.2.1, does not directly take into account how the control inputs should behave. In a real vehicle, it is beneficial to reduce erratic control behaviour since this in turn increases the lifespan of the actuators. By implementing a penalty on erratic control inputs, the desired outcome is that these become smoother, and ideally that they converge to zero when the vehicle is at rest. This penalty is $-c_{\dot{F}_u\dot{F}_u}\dot{F_u}^2 - c_{\dot{\theta}\dot{\theta}}\dot{\theta}^2$, where $c_{\dot{F}_u\dot{F}_u} > 0$ and $c_{\dot{\theta}\dot{\theta}} > 0$ are the weighting factors of the thrust and rudder penalties, respectively.

The reward becomes

$$r_2 = -c_{\dot{F}_u\dot{F}_u}\dot{F_u}^2 - c_{\dot{\theta}\dot{\theta}}\dot{\theta}^2 + \begin{cases} ae^{-\frac{d^2}{2\sigma^2}}, & \text{if } d < 0.05 \\ -c_{\dot{d}}\dot{d}, & \text{otherwise} \end{cases} \tag{5.11}$$

where $c_{\dot{d}} > 0$ is the weighting factor of $\dot{d}$. The $-c_{\dot{d}}\dot{d}$ term gives a reward whenever $\dot{d} < 0$, that is when distance is reduced, and a penalty when $\dot{d} > 0$.

The environment of Section 5.1 is simplified compared to vehicles in the real world in the sense that desired control inputs are applied instantly, rather than having their own dynamics. Since the vehicle of the first environment thus has the possibility of changing its direction input more quickly than what could be considered "usual", we can expect it to find solutions where the reward of changing direction outweighs the penalty of fast-changing inputs.

5.2.3 Reward Function 3

Motivated by the discussion of r_2 that can be found in Section 5.3.2, we propose a reward where the penalty on change in applied force is replaced by a penalty on the magnitude of the force. Hopefully, this motivates the agent to reduce its input to minimum. The goal is to reduce the applied force as much as possible when the goal position is reached. If F_u approaches zero, the direction is irrelevant to the movement of the vehicle. Thus, the penalty on $\dot{\theta}$ is kept as is, in the hopes of resulting in an agent that applies constant θ when stationary.

The reward is

$$r_3 = -c_{F_uF_u}F_u^2 - c_{\dot{\theta}\dot{\theta}}\dot{\theta}^2 + \begin{cases} ae^{-\frac{d^2}{2\sigma^2}}, & \text{if } d < 0.05 \\ -c_{\dot{d}}\dot{d}, & \text{otherwise} \end{cases} \tag{5.12}$$

where $c_{F_uF_u} > 0$ is the weighting factor of penalty on F_u. As in the previous section, the weighting factors of $\dot{\theta}$, $\dot{d}$ and the Gaussian signal are $c_{\dot{\theta}\dot{\theta}}$, $c_{\dot{d}}$ and a, respectively. The magnitude of the weighting factor on the force must make sure the expected reward of moving closer to the goal by applying force is greater than the penalty linked to the force.

5.3 Simulations

In this section, the reinforcement learning agent whose architecture is described in Section 4.3.1, is trained and tested using the reward signal designs with and without penalties related to control input, which were presented in Section 5.2. Each of the reward signals are implemented in the point mass environment, as implemented in Section 5.1, and results are presented and analysed below, in Section 5.3.1-5.3.3.

5.3.1 Reward Function 1

Letting $a = 1$, $c_d = 1$ and $\sigma^2 = 0.4$, and training the DDPG agent for 4000 episodes, 800000 time steps in total, we get the reward history shown in Figure 5.4. These values for the weight factors result in maximum reward of 2 at $(x, y) = (x_g, y_g)$, and the underlying bell curve will look like in Figure 5.3.

Two tests from different start positions are presented in Figure 5.5, where Figures 5.5a,c,e show what happens when the vehicle starts at position $(x, y) = (2, -2)$, and 5.5b,d,f show what happens when starting at $(x, y) = (-2, 2)$. The plotted direction θ is scaled between -1 and 1 in Figures 5.5e-f.

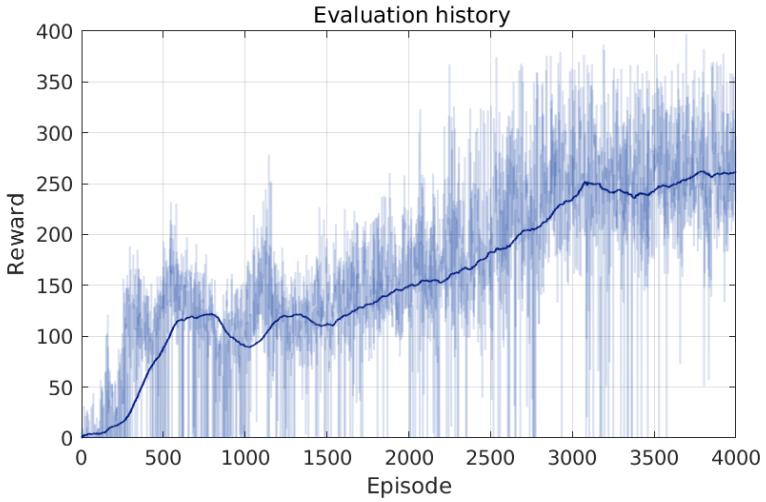
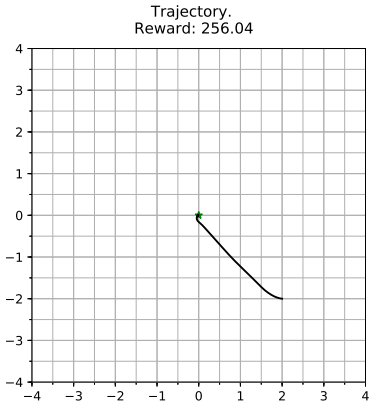
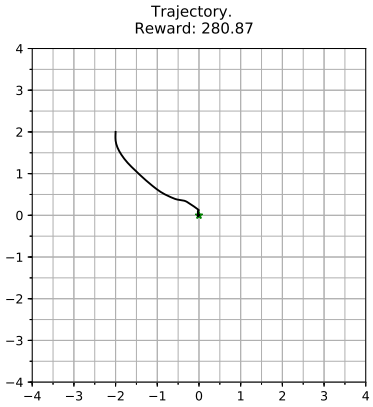


Figure 5.4: Evaluation reward history (without exploration) for point mass trained using DDPG with r_1

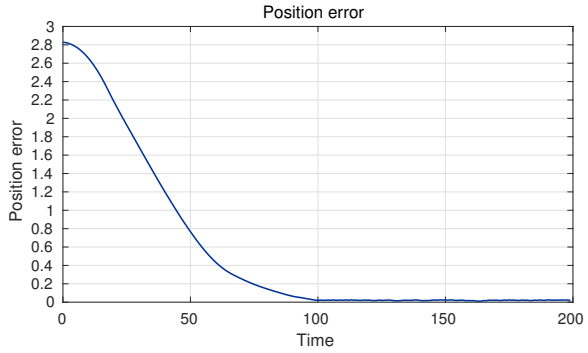
The test runs illustrate that the vehicle successfully learns how to apply actions in order to move from its start position to the goal, at $(0, 0)$. Figures 5.5c,d confirm that the position error converges to zero. However, Figures 5.5e,f show how the control actions are unpredictable. Once the vehicle has stopped at the goal, the applied force does not go to zero, and the direction is shaky. In fact, in order to stop its movement, it is not necessary for the agent to apply zero action, as there are no constraints on direction change, thus a feasible solution will be to change direction by 180° at every time step.



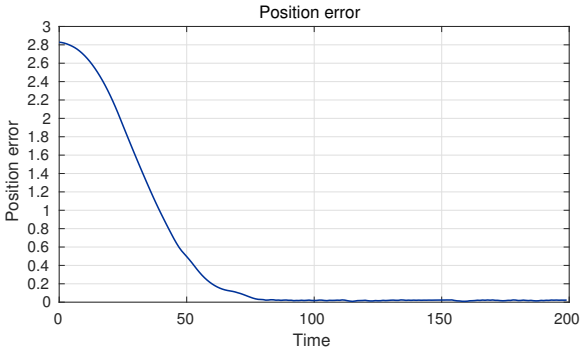
(a) Trajectory



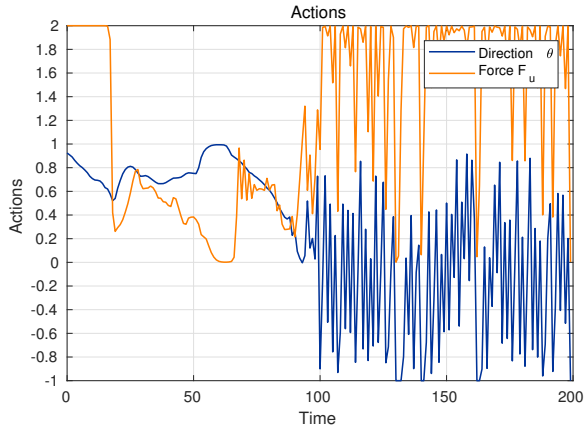
(b) Trajectory



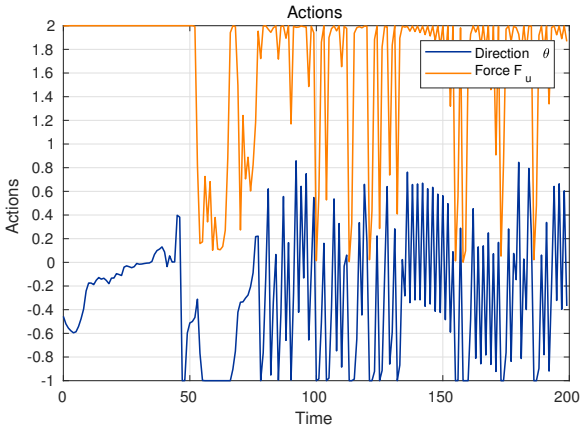
(c) Distance to goal



(d) Distance to goal



(e) Control actions



(f) Control actions

Figure 5.5: Test the point mass agent trained by DDPG with r_1, from $(x_s, y_s) = (2, -2)$ and $(x_s, y_s) = (-2, 2)$

5.3.2 Reward Function 2

We recognise through experiments that by letting the penalty on change in applied force grow large compared to the combination of the reward close to the goal and the penalty on change in direction, the agent learns to apply a constant force of $F_u = 2$. Thus only the direction of the force is utilised for speed regulation. The reason it learns this is that the chances of experiencing a situation that results in the real maximum reward, where $F_u \approx 0$, $d < 0.05$ and $\dot{\theta} \approx 0$ through exploration is low. This is especially true when, after many learning epochs, the agent has been taught that applying more force reaches the desired position faster, and reducing force will lead to a large penalty. Since maximum F_u brings the agent quickest to the goal, this approach usually gives the best reward from the agent's experience. As a result, setting $c_{\dot{F}_u\dot{F}_u} = 0$ so that $c_{\dot{F}_u\dot{F}_u}\dot{F}_u^{\,2}$ is removed from Equation (5.11) may produce more interesting results.

The remaining parameters are chosen as $a = 2$, $\sigma^2 = 0.1$, $c_{\dot{\theta}\dot{\theta}} = 0.0015$ and $c_{\dot{d}} = \frac{1}{\sqrt{2}}$. We know that, since $\theta \in [-\pi, \pi]$, then $\theta_t - \theta_{t-1} \in [-\pi, \pi]$, because all angles must be normalised to lie in this range. This value can easily be scaled to lie within $[-1, 1]$. This means that $\dot{\theta} \in [-1, 1]\frac{1}{\Delta t} = [-20, 20]$, and $\dot{d} \in [-\sqrt{2}, \sqrt{2}]$, then $c_{\dot{\theta}\dot{\theta}}\dot{\theta}^2 \in [0, 0.6]$ and $c_{\dot{d}}\dot{d} \in [-1, 1]$. So by choosing these values for the parameters, the penalty is always less than the potential reward, which motivates actions during training. The reward history of this experiment is shown in Figure 5.6. Two test runs from start positions $(x, y) = (2, 2)$ and $(x, y) = (-2, -2)$ are illustrated in Figure 5.7.

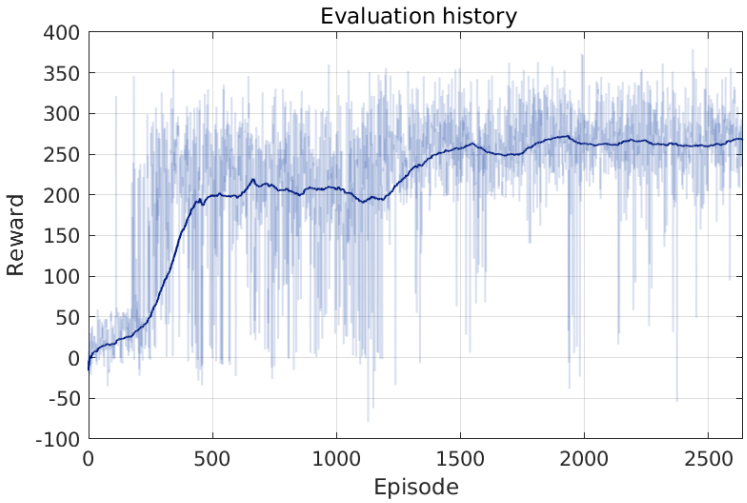
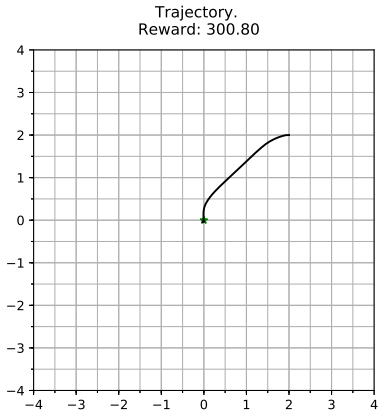


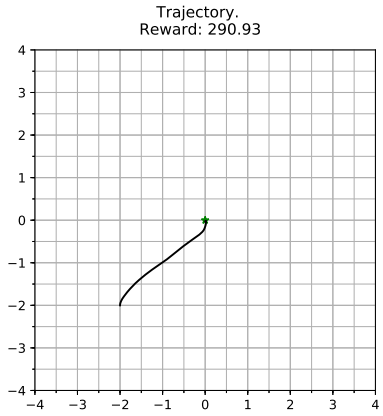
Figure 5.6: Evaluation reward history for point mass trained using DDPG with r_2

First of all, we observe that reward r_2 causes the solution to converge after only 2000 episodes, which is significantly less than what we saw in Section 5.3.1. Reward r_2 is less sparse than r_1, and therefore less exploration is needed before reaching states that give nonzero reward, and so training is faster.

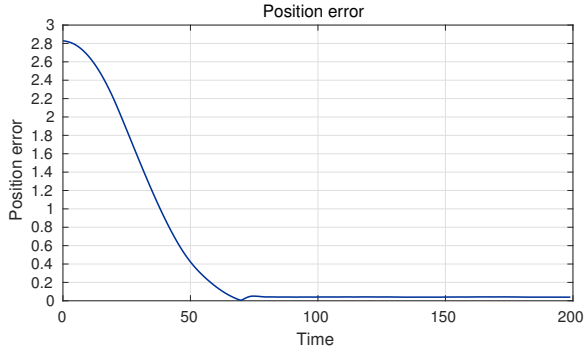
Comparing trajectories from Figure 5.7 with Figure 5.5, the new reward changes the behaviour of the vehicle, although not by extreme amounts. It can be seen that the constraint on turning rate/erratic direction causes the trajectories to appear as if they were made by a vehicle with limited turning radius. In Figure 5.5, the vehicle was turning more suddenly, thus the trajectories were less smooth. In addition, Figures 5.7e-f tell us that the penalty in change of direction has had the desired effect on the inputs. The plotted actions show a clear change in how the control inputs are being used by the



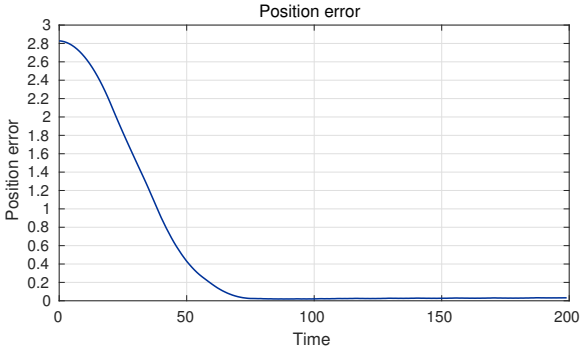
(a) Trajectory



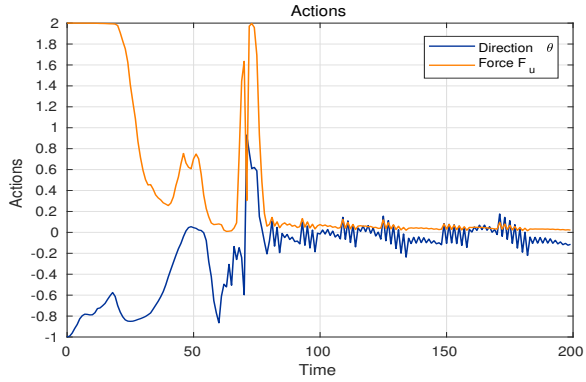
(b) Trajectory



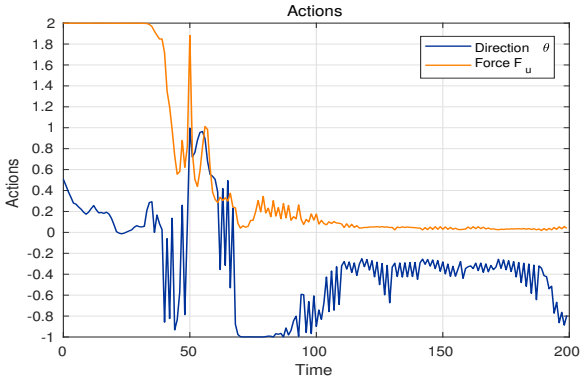
(c) Distance to goal



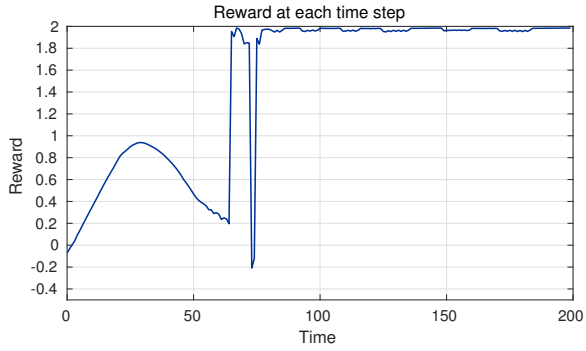
(d) Distance to goal



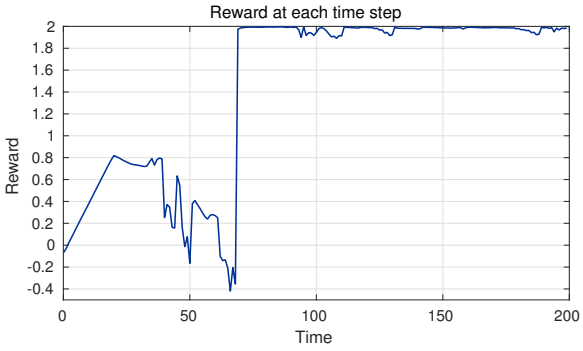
(e) Control actions



(f) Control actions



(g) Rewards



(h) Rewards

Figure 5.7: Test the point mass agent trained by DDPG with r_2, from $(x_s, y_s) = (2, 2)$ and $(x_s, y_s) = (-2, -2)$

agent, compared to r_1 in Section 5.3.1. Here, F_u approaches a steady value close to zero, while θ is less erratic than previously. The rewards corresponding to each test are shown in Figures 5.7g-h, which confirm that large changes in direction correspond to reduction in the reward signals. Even though the force is not exactly zero, due to friction the vehicle remains stationary from approximately 75 time steps. Figures 5.7c-d illustrate that this leads to a steady state error in position, as the vehicle has not learned to stop at the exact position that maximises reward.

It seems natural, then, that to keep training the neural networks will lead to a perfect solution. However, this is not the case. Some discussion regarding the cause of this can be found in Section 5.4. An example of the behaviour can be found in Appendix A.1, where it is evident that the agent has learned to reduce the position error, but it no longer understands how to reduce the derivative of the input.

5.3.3 Reward Function 3

The reward signal introduced in Section 5.2.3, given by Equation (5.12) is implemented with $c_{\dot{\theta}\dot{\theta}} = 0.0015$, $c_{F_uF_u} = 0.03$, $a = 2$, $\sigma^2 = 0.1$ and $c_{\dot{d}} = \frac{1}{\sqrt{2}}$. Then $c_{\dot{\theta}\dot{\theta}}\dot{\theta}^2 \in [0, 0.6]$, $c_{\dot{d}}\dot{d} \in [-1, 1]$ and $c_{F_uF_u}F_u^2 \in [0, 0.12]$. The penalty on F_u is designed to be the smallest, to avoid situations where remaining in the initial position by applying zero force becomes the optimal solution. The resulting reward history can be found in Figure 5.8, and two tests from starting positions $(x, y) = (2, -2)$ and $(x, y) = (-2, 2)$ are shown in Figure 5.9.

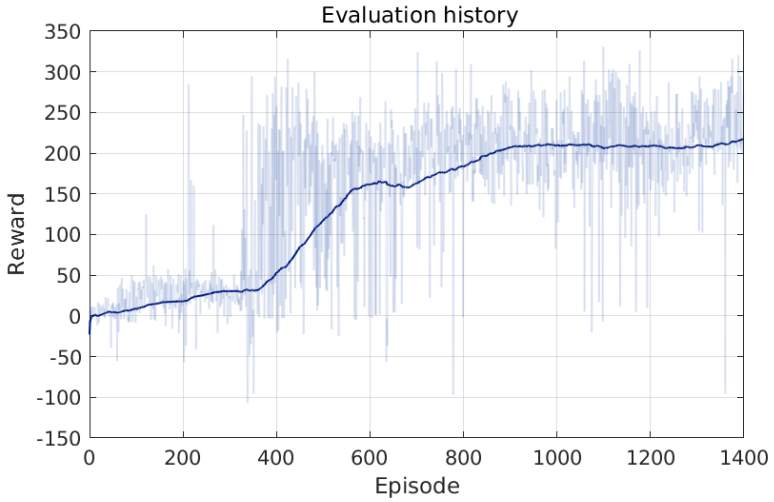
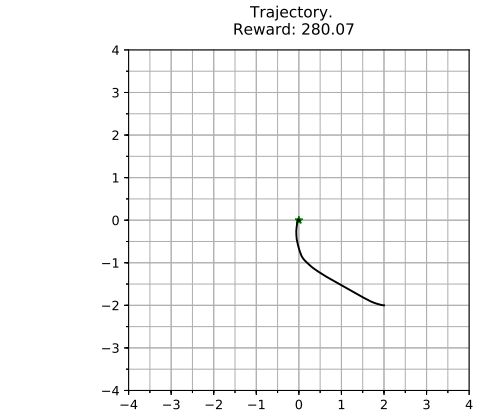


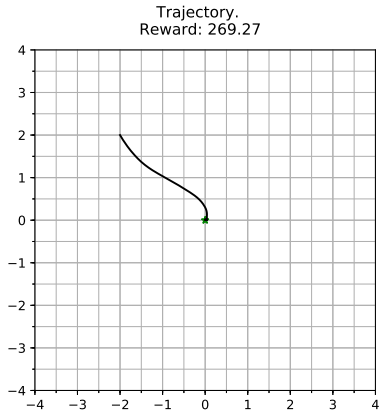
Figure 5.8: Evaluation reward history for point mass trained using DDPG with r_3

Figures 5.9e-f illustrate that the agent applies less force than previously, and that F_u becomes zero when the vehicle is stopping. The direction of the applied force is also less erratic than in the previous tests. The position error converges to zero, as shown in Figures 5.9c-d. The reward signals of Figures 5.9g-h confirm that the reward signal behaves as intended - it is reduced when F_u is large and when θ changes quickly.

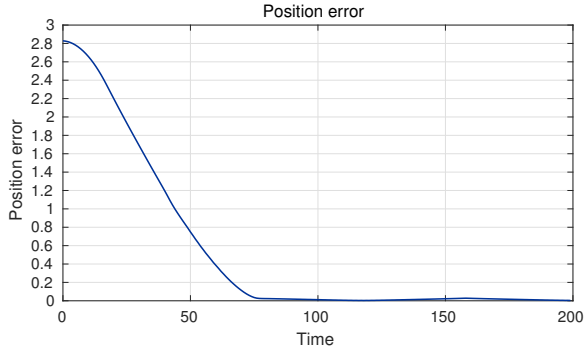
The trajectories seen in Figures 5.9a-b show that the turning of this vehicle appears even more constrained than in Section 5.3.2. More test results can be found in Appendix A.2, which show paths taken by the vehicle when the starting position is further from (x_g, y_g).



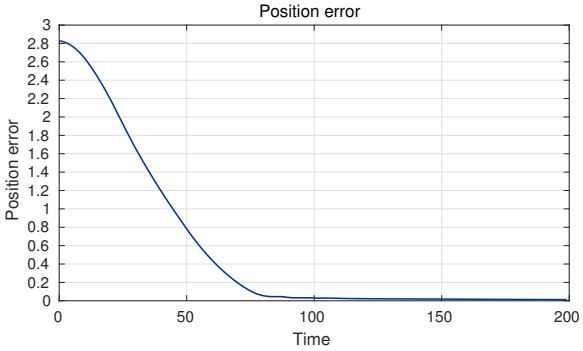
(a) Trajectory



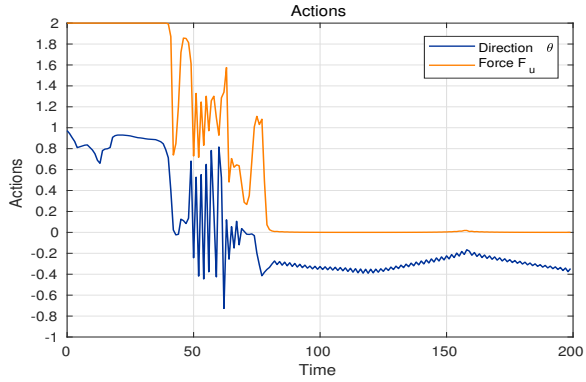
(b) Trajectory



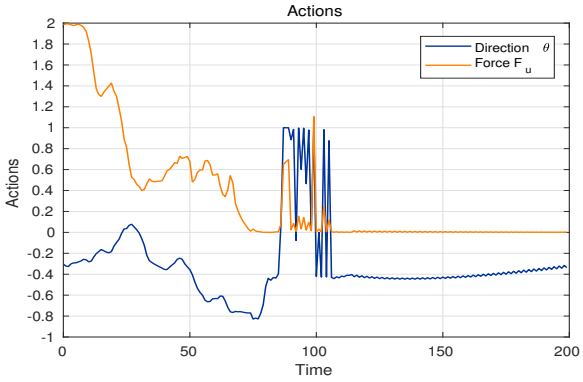
(c) Distance to goal



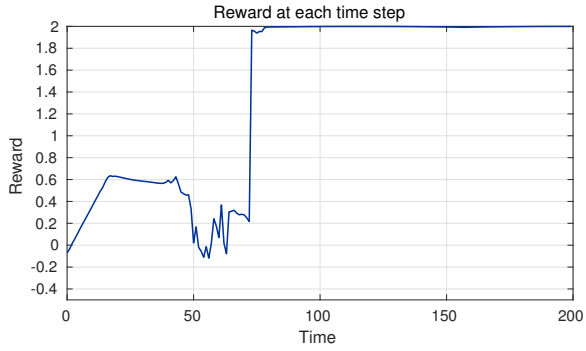
(d) Distance to goal



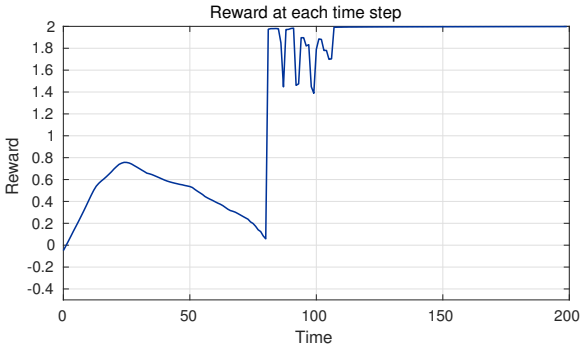
(e) Control actions



(f) Control actions



(g) Rewards



(h) Rewards

Figure 5.9: Test the point mass agent trained by DDPG with r_1, from $(x_s, y_s) = (2, -2)$ and $(x_s, y_s) = (-2, 2)$

Based on these observations, r_3 gives a relatively successful agent, which navigates from a start position to a goal by applying a force direction and magnitude. The trajectories that the agent learns to follow are not completely straight, neither with this reward nor any of the previous ones. An issue that we face when teaching an agent how to move is that there are so many possibilities for the agent's choice of action. Even with function approximators that are able to generalise and choose appropriate actions in new situations, the chances of finding *the* optimal action in a particular state can be low. This issue can be especially prominent here because of how the environment is designed. As pointed out in Section 5.1, the force is applied in the desired direction θ instantly, thus the agent may benefit from being trained on a vehicle which includes a description of the dynamics of θ. The consequence of this will be that the turning radius of the vehicle is limited, giving the agent less freedom, which may in turn produce a better solution.

5.4 Discussion

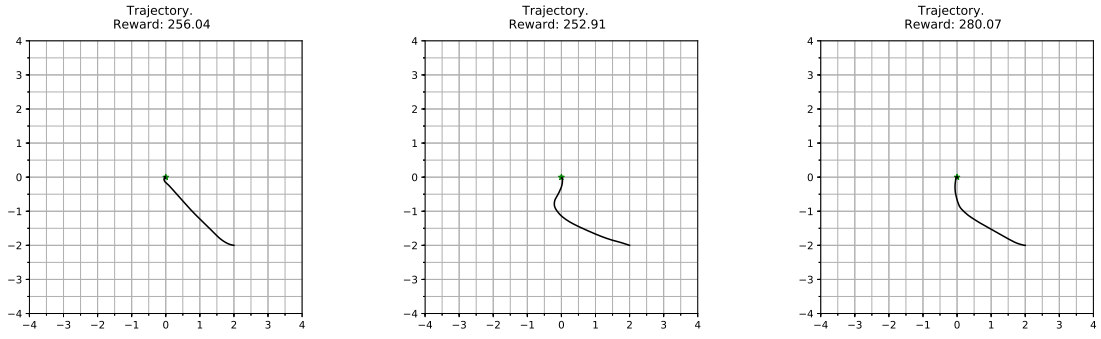
It has been pointed out during the design of reward signals r_2 and r_3 (Sections 5.2.2-5.2.3), and in the results of the previous sections, that penalising the derivative of the direction input could be beneficial. Sections 5.3.2-5.3.3 confirm that this is the case, where we see the control inputs behaving as expected, but it is also pointed out that the training of the agents can be unstable. A discussion of the reason for this behaviour follows.

The agent's observation, given by (5.7), includes the computed distance and change of distance to goal in order to reduce the complexity of the function approximator. However, the observation does not include straightforward information about the vehicle's current direction. This was initially considered to be redundant, due to the direction being a part of the action vector. Also, some knowledge is encoded in $\dot{d}$, which is maximised when the direction of travel is towards the goal. But we find that because of the discontinuity of the reward signal, $\dot{d}$ no longer affects the performance measure when the vehicle is close to the optimal position. Additionally, the velocity is typically low in this area, so all elements of the observation vector are approximately zero. It could be that the agent has trouble distinguishing between observations, and it may become difficult to learn what are the optimal actions in this area, considering that the optimal action depends on what derivative $\dot{\theta}$ it produces. Thus, an augmentation of the observation vector with the previous input θ will provide the agent with memory regarding direction, and may assist in learning the desired behaviour.

It is also important to note that the most important regularisation technique for the neural networks that has been applied here is in fact early stopping. So if the agent is allowed to train for too long, the weights of the networks may grow very large or small, undoing their ability to approximate the functions they are supposed to approximate. Thus, the addition of dropout to the networks, for instance, may result in more stable learning.

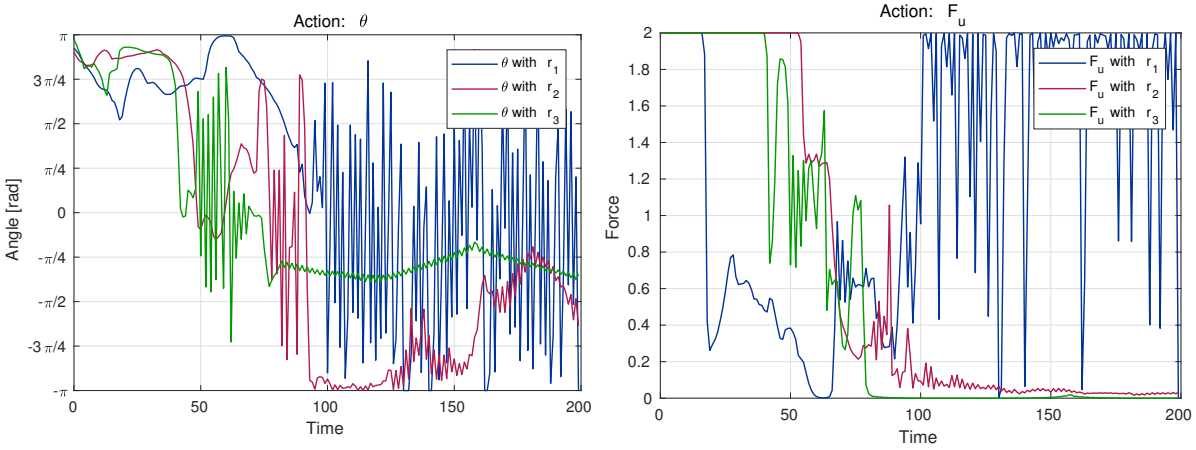
Figures 5.10 and 5.11 display a comparison of the three agents implemented with reward signals r_1, r_2 and r_3. These figures support the claims made earlier - that the main difference between the trajectories appears to be the turning radius achievable by the vehicle, and that the control inputs become less erratic when the reward function reflects a desire to reduce the change in input.

The effect on the turning radius of applying penalty to $\dot{\theta}$ is an interesting result when considering whether this system can be used in the development of a path planner

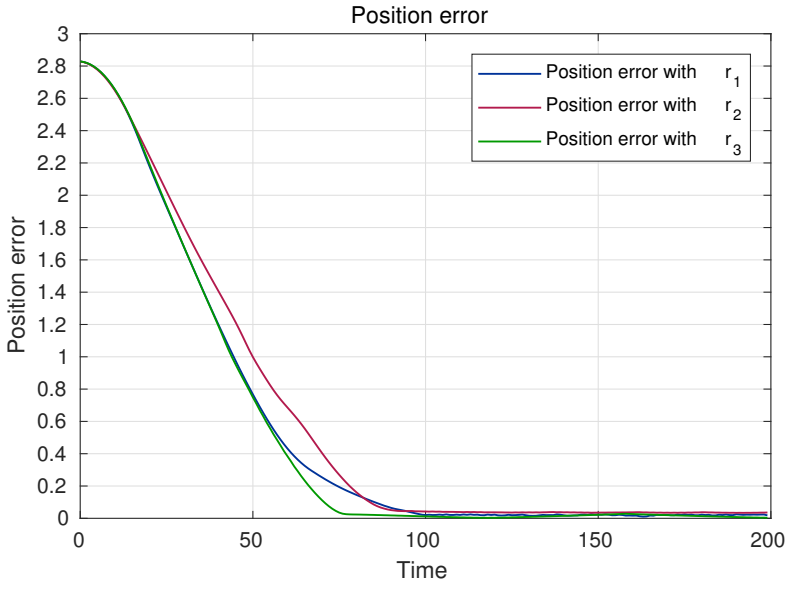


(a) Using r_1 (b) Using r_2 (c) Using r_3

Figure 5.10: Comparison of agents' sample trajectories



(a) Direction (b) Force



(c) Euclidean distance to goal

Figure 5.11: Comparison of agents' actions and position error for the sampled trajectories of Figure 5.10

for vehicles. The trajectories of Figures 5.10b and 5.10c are similar in appearance to polynomial trajectories, which is intriguing in this context because it demonstrates that the reinforcement learning agent can find feasible paths for a vehicle with constrained motion. In this case, the vehicle's motion was unconstrained, but the agent still found trajectories that could have been feasible for other vehicles. Consequently, these paths can theoretically be fed to a path following system for a constrained vehicle and they would be possible to follow.

Chapter 6

Conclusion

In this thesis we have conducted an investigation of reinforcement learning algorithms that may be suitable for control of physical systems. We have implemented some of these algorithms and applied them to control tasks fetched from OpenAI Gym, and established the resulting agents' behaviour as successful in their respective tasks.

Further, the thesis has presented a general structure for applying deep reinforcement learning to a manoeuvring or path planning problem for a vehicle. The resulting method is model-free, meaning that rather than having knowledge about how the world works before computation, the algorithm learns a representation that approximates the real world through exploration. Several reward function designs have been proposed, and results show that more than one design may lead to a solution. We have found that penalising control input change, due to less tear on actuators, is preferred. We have also found that this penalty leads a vehicle with no constraints on control inputs to behave as if such constraints were present, thus illustrating a potential for the proposed framework to be applied to other types of vehicles.

Ultimately, we have found that the results presented in this thesis are promising. They show that the path planning problem in two dimensions can be solved by using a model-free controller based on reinforcement learning.

Even though the proposed solution shows promise, some drawbacks must be considered. One such drawback has to do with stability of training. A problem may arise where the neural networks that approximate the policy and value functions do not converge to good solutions. Another issue with function approximators of this kind is that the functions they can represent are limited, therefore the true optimal policy and value function may not be possible to find by using the current network architecture. Smaller, shallower networks can generally approximate less complex functions than their deeper counterparts, but retain the advantage of being less computationally demanding. Finally, the design of performance measure and observations may reduce the complexity needed by the function approximators, and result in faster convergence.

Chapter 7

Future Work

This section will give some recommendations for future work on the vehicle control system made with reinforcement learning. Firstly, in order to test the performance of the system on the manoeuvring task more thoroughly, it should be compared to, for instance, a target tracking controller that is implemented and tuned for the vehicle. Another suggestion is to compare the trajectories found in this thesis with ones found by a polynomial trajectory planner. In addition, implementation of one or more alternative DRL methods should be considered. A possible choice is PPO, which may be more sample efficient and can give more robust training than DDPG. A more complex vehicle with curvature constraints, such as a Dubins car or a simulation of a marine vessel, may be substituted for the currently used vehicle in the agent's training, to investigate the behaviour of the proposed system in a different environment. Lastly, the incorporation of obstacles into the environment, and a redesign of the reward signal that takes the avoidance of obstacles into account, can transform the system to become an intelligent collision avoidance system. A thorough investigation of how this may be carried out should be done, which considers the design of the observation vector, the architecture of the neural networks of the function approximators, and whether both stationary and moving obstacles are to be included.

Bibliography

[1] "COLREGs — Convention on the International Regulations for Preventing Collisions at Sea," 1972. [Online]. Available: http://www.jag.navy.mil/distrib/instructions/COLREG-1972.pdf

[2] M. Candeloro, A. M. Lekkas, J. Hegde, and A. J. Sørensen, "A 3d dynamic voronoi diagram-based path-planning system for uuvs," in *OCEANS 2016 MTS/IEEE Monterey*, Sep. 2016, pp. 1–8.

[3] I. B. Hagen, D. K. M. Kufoalor, E. F. Brekke, and T. A. Johansen, "MPC-based collision avoidance strategy for existing marine vessel guidance systems," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018, pp. 7618–7623.

[4] T. A. Johansen, T. Perez, and A. Cristofaro, "Ship collision avoidance and colregs compliance using simulation-based control behavior selection with predictive hazard assessment," *IEEE Transactions on Intelligent Transportation Systems*, vol. 17, no. 12, pp. 3407–3422, Dec 2016.

[5] H. Lyu and Y. Yin, "Ship's trajectory planning for collision avoidance at sea based on modified artificial potential field," in *2017 2nd International Conference on Robotics and Automation Engineering (ICRAE)*, Dec 2017, pp. 351–357.

[6] Øivind Aleksander G. Loe, "Collision avoidance for unmanned surface vehicles," Master's thesis, Norwegian University of Science and Technology, 2008.

[7] S. Moe and K. Y. Pettersen, "Set-based line-of-sight (LOS) path following with collision avoidance for underactuated unmanned surface vessel," in *2016 24th Mediterranean Conference on Control and Automation (MED)*, June 2016.

[8] Y. Kuwata, M. T. Wolf, D. Zarzhitsky, and T. L. Huntsberger, "Safe maritime autonomous navigation with colregs, using velocity obstacles," *IEEE Journal of Oceanic Engineering*, vol. 39, no. 1, pp. 110 – 119, Jan 2014.

[9] G. Millar, "An obstacle avoidance system for autonomous underwater vehicles: A reflexive vector field approach utilizing obstacle localization," in *2014 IEEE/OES Autonomous Underwater Vehicles (AUV)*, Oct 2014, pp. 1–4.

[10] W. Zhang, S. Wei, Y. Teng, J. Zhang, X. Wang, and Z. Yan, "Dynamic obstacle avoidance for unmanned underwater vehicles based on an improved velocity obstacle method," in *Sensors*, 2017.

[11] E. Kelasidi, K. Y. Pettersen, and J. T. Gravdahl, "A waypoint guidance strategy for underwater snake robots," in *22nd Mediterranean Conference on Control and Automation*, June 2014, pp. 1512–1519.

[12] A. M. Kohl, S. Moe, E. Kelasidi, K. Y. Pettersen, and J. T. Gravdahl, "Set-based path following and obstacle avoidance for underwater snake robots," in *2017 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, Dec 2017, pp. 1206–1213.

[13] M. L. Turing, "Computing machinery and intelligence," *Mind*, vol. 49, pp. 433–460, 1950.

[14] S. J. Russel and P. Norvig, *Artificial Intelligence. A Modern Approach*, 3rd ed. Upper Saddle River, New Jersey, USA: Prentice Hall, 2010.

[15] D. Michie and R. A. Chambers, "Boxes: An experiment in adaptive control," *Machine intelligence*, vol. 2, no. 2, pp. 137–152, 1968.

[16] D. P. Bertsekas and J. N. Tsitsiklis, *Neuro-Dynamic Programming*, 1st ed. Athena Scientific, 1996.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013. [Online]. Available: http://arxiv.org/abs/1312.5602

[18] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," *Nature*, vol. 550, pp. 354–359, October 2017. [Online]. Available: https://www.nature.com/articles/nature24270

[19] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015. [Online]. Available: http://arxiv.org/abs/1509.02971

[20] R. Yu, Z. Shi, C. Huang, T. Li, and Q. Ma, "Deep reinforcement learning based optimal trajectory tracking control of autonomous underwater vehicle," in *2017 36th Chinese Control Conference (CCC)*, July 2017, pp. 4958–4965.

[21] A. B. Martinsen, "End-to-end training for path following and control of marine vehicles," Master's thesis, Norwegian University of Science and Technology, 2018.

[22] Y. Cheng and W. Zhang, "Concise deep reinforcement learning obstacle avoidance for underactuated unmanned marine vessels," *Neurocomputing*, vol. 272, pp. 63 – 73, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0925231217311943

[23] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B.

Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," 2016. [Online]. Available: http://arxiv.org/abs/1603.04467

[24] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[25] Khan Academy. (2018) The Neuron and Nervous System: Overview of neuron structure and function. [Online]. Available: https://www.khanacademy.org/science/biology/human-biology/neuron-nervous-system/a/overview-of-neuron-structure-and-function

[26] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, Oct 1986.

[27] J. C. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 07 2011.

[28] T. Tieleman and G. Hinton, "Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude," COURSERA: Neural Networks for Machine Learning, 2012.

[29] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[30] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, pp. 1929–1958, 2014. [Online]. Available: http://jmlr.org/papers/v15/srivastava14a.html

[31] A. Y. Ng, "Feature selection, L1 vs. L2 regularization, and rotational invariance," in *In ICML*, 2004.

[32] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *CoRR*, vol. abs/1502.03167, 2015. [Online]. Available: http://arxiv.org/abs/1502.03167

[33] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 2nd ed. Cambridge, Massachusetts, USA: The MIT Press, 2018.

[34] R. S. Sutton, D. Mcallester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *In Advances in Neural Information Processing Systems 12*. MIT Press, 2000, pp. 1057–1063.

[35] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, no. 3, pp. 229–256, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992696

[36] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic Policy Gradient Algorithms," in *Proceedings of the 31st International Conference on Machine Learning - Volume 32*, ser. ICML'14. JMLR.org, 2014, pp. I–387–I–395. [Online]. Available: http://dl.acm.org/citation.cfm?id=3044805.3044850

[37] L.-J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Machine Learning*, vol. 8, no. 3, pp. 293–321, May 1992. [Online]. Available: https://doi.org/10.1007/BF00992699

[38] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Phys. Rev.*, vol. 36, pp. 823–841, Sep 1930. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRev.36.823

[39] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

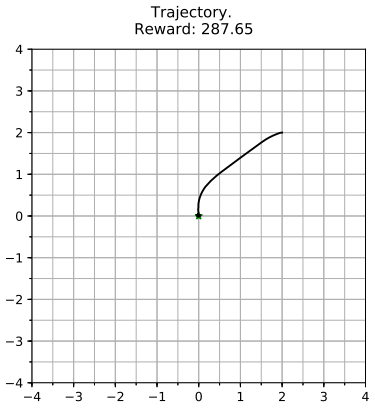
[40] T. Gravdahl and O. Egeland, *Modeling and Simulation for Automatic Control*. Marine Cybernetics AS, 2002.

Appendices

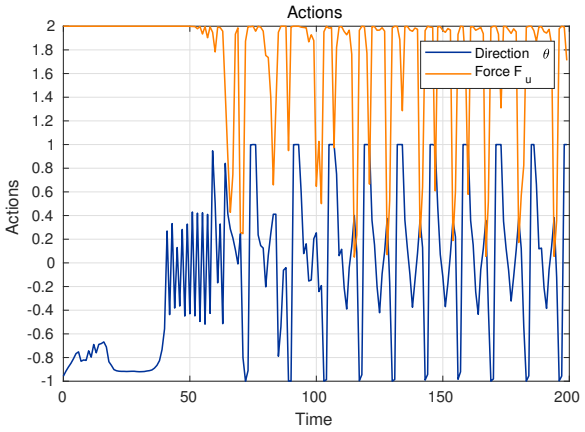
Appendix A

Additional Simulation Results of the Vehicle Manoeuvring Task

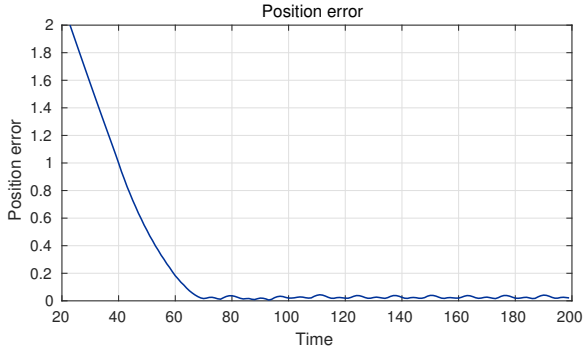
A.1 Simulation results using r_2, with additional training



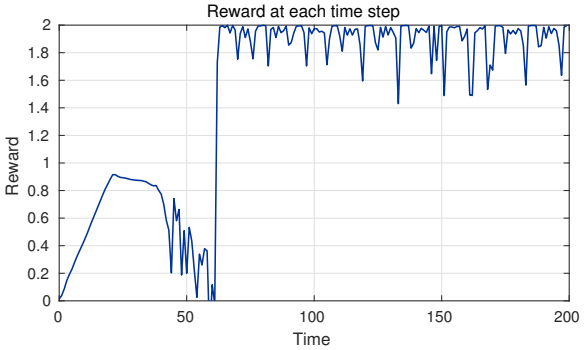
(a) Trajectory



(b) Control actions

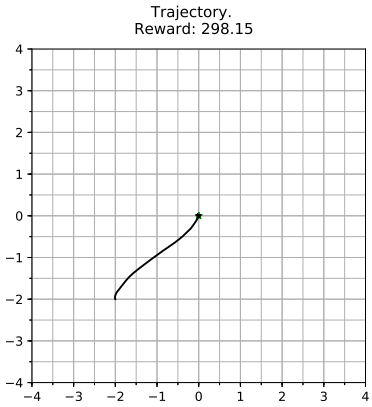


(c) Distance to goal

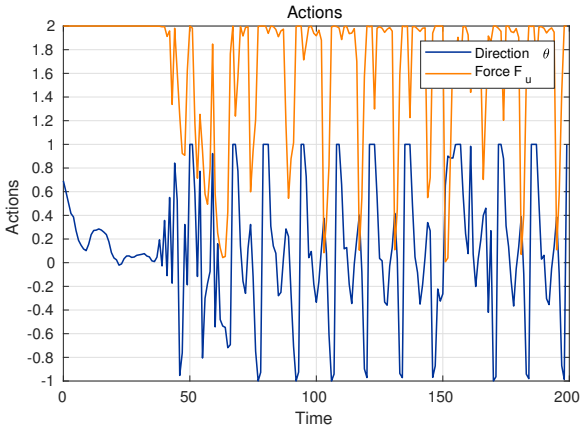


(d) Rewards

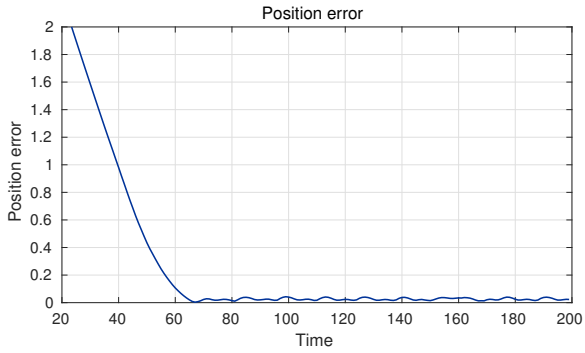
Figure A.1: Test the point mass agent trained by DDPG with r_2, from $(x_s, y_s) = (2, 2)$



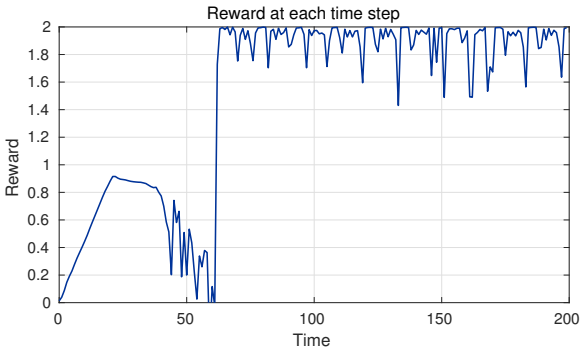
(a) Trajectory



(b) Control actions



(c) Distance to goal



(d) Rewards

Figure A.2: Test the point mass agent trained by DDPG with r_2, from $(x_s, y_s) = (-2, -2)$

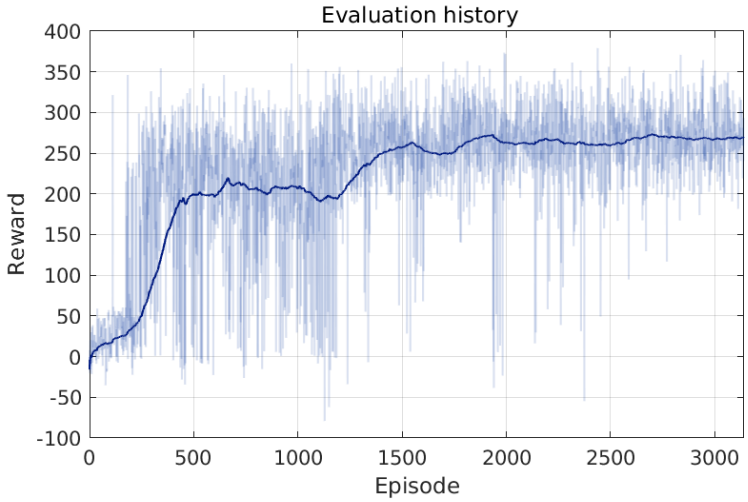
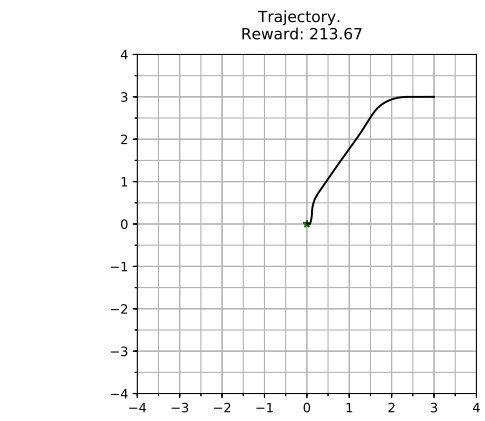
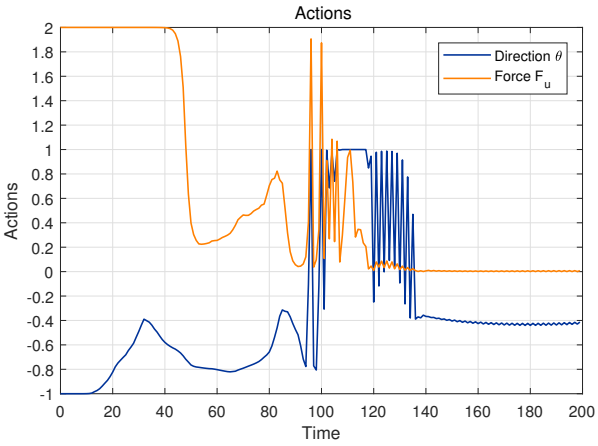


Figure A.3: Evaluation reward history for point mass trained using DDPG with r_2

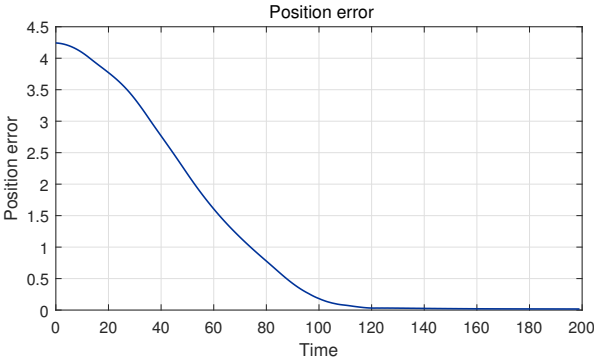
A.2 Simulation results using $\mathbf{r_3}$



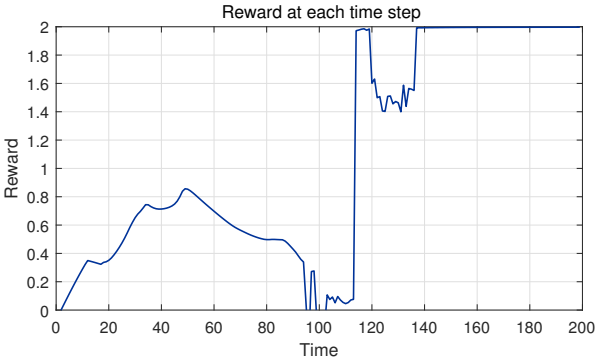
(a) Trajectory



(b) Control actions

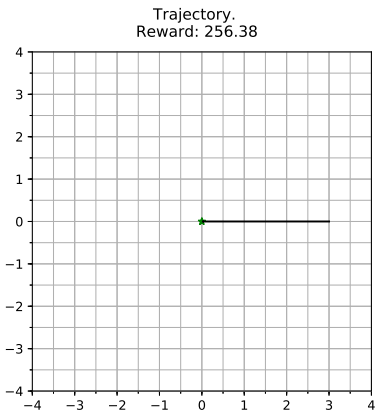


(c) Distance to goal

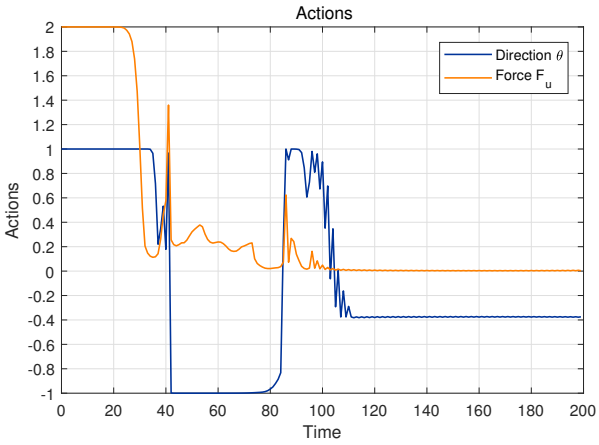


(d) Rewards

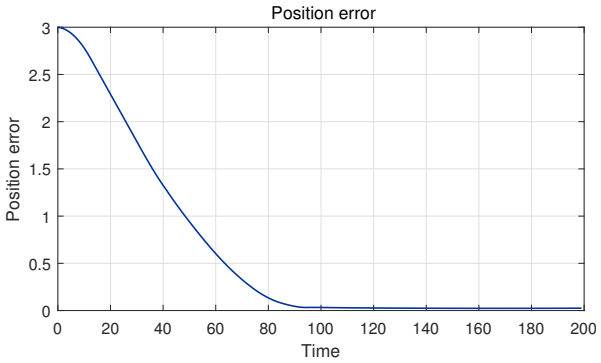
Figure A.4: Test the point mass agent trained by DDPG with r_3, from $(x_s, y_s) = (3, 3)$



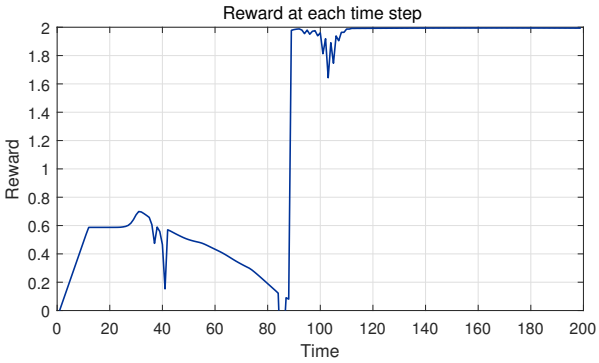
(a) Trajectory



(b) Control actions

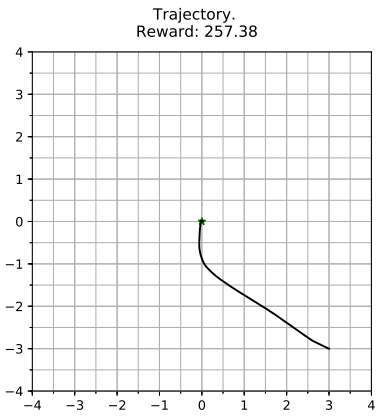


(c) Distance to goal

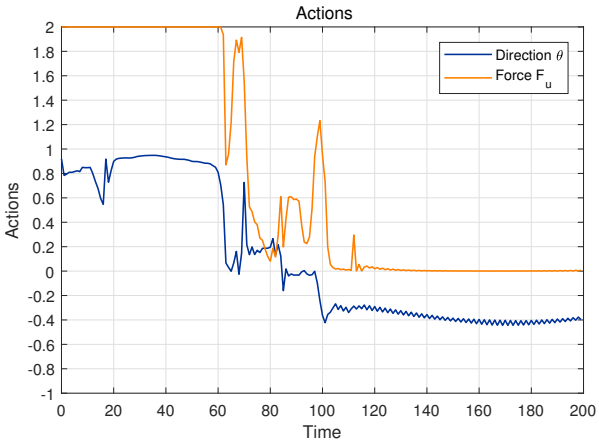


(d) Rewards

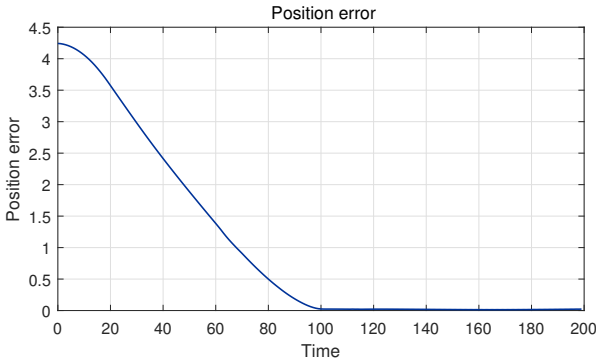
Figure A.5: Test the point mass agent trained by DDPG with r_3, from $(x_s, y_s) = (3, 0)$



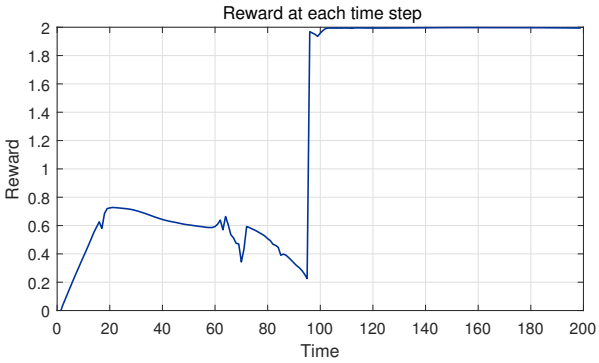
(a) Trajectory



(b) Control actions

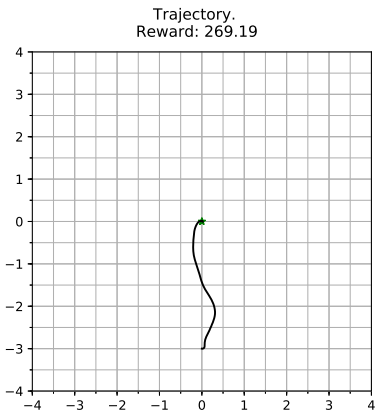


(c) Distance to goal

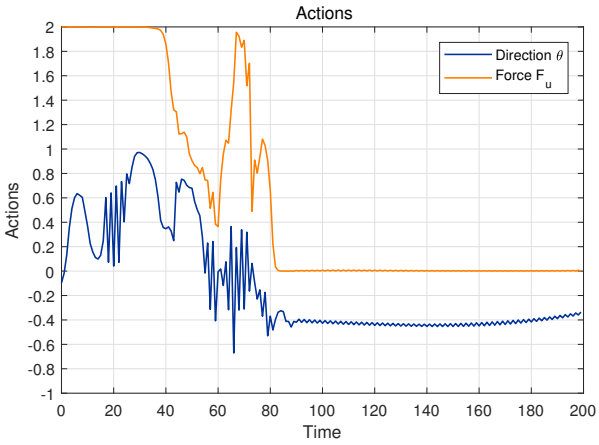


(d) Rewards

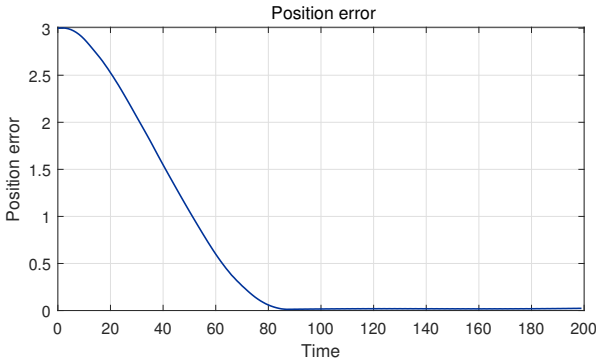
Figure A.6: Test the point mass agent trained by DDPG with r_3, from $(x_s, y_s) = (3, -3)$



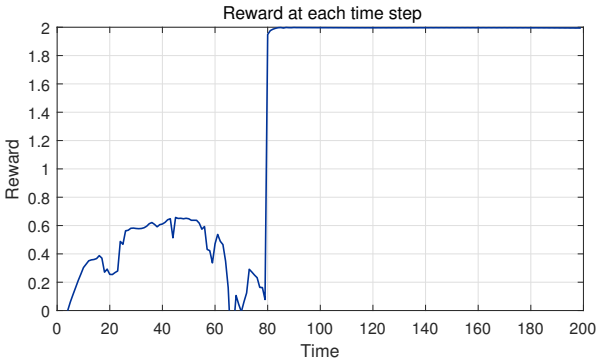
(a) Trajectory



(b) Control actions

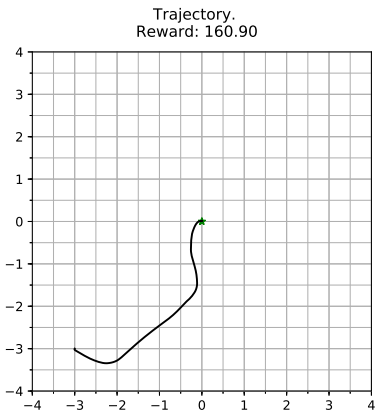


(c) Distance to goal

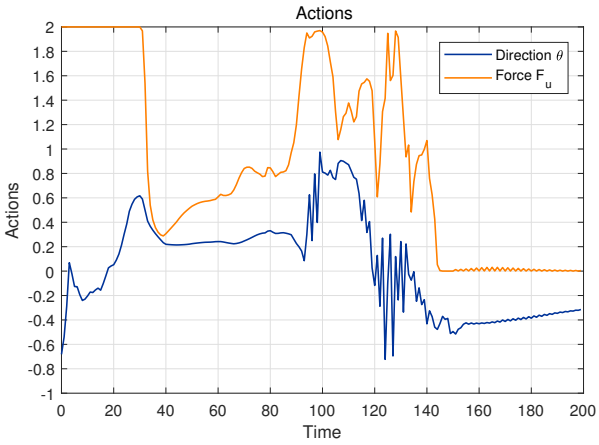


(d) Rewards

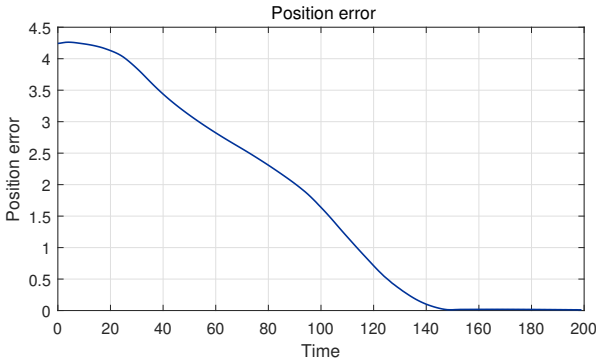
Figure A.7: Test the point mass agent trained by DDPG with r_3, from $(x_s, y_s) = (0, -3)$



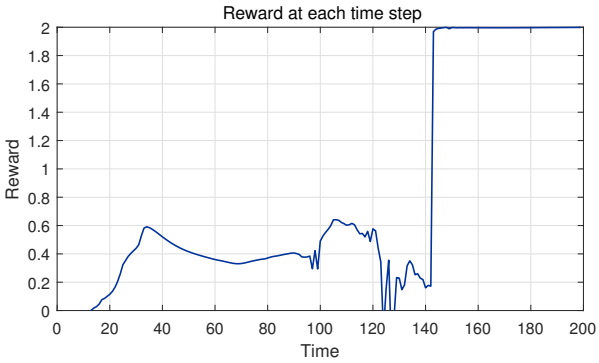
(a) Trajectory



(b) Control actions

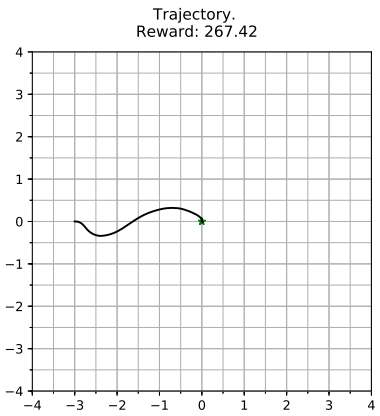


(c) Distance to goal

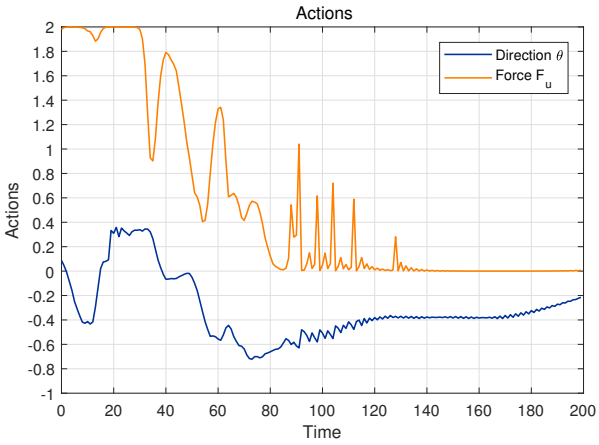


(d) Rewards

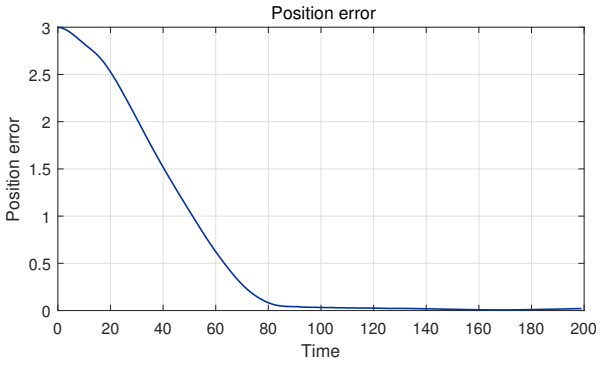
Figure A.8: Test the point mass agent trained by DDPG with r_3, from $(x_s, y_s) = (-3, -3)$



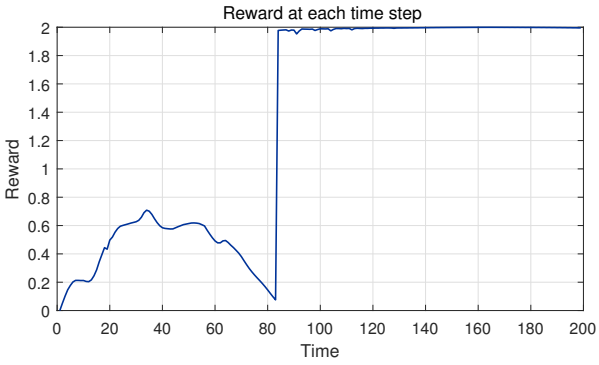
(a) Trajectory



(b) Control actions

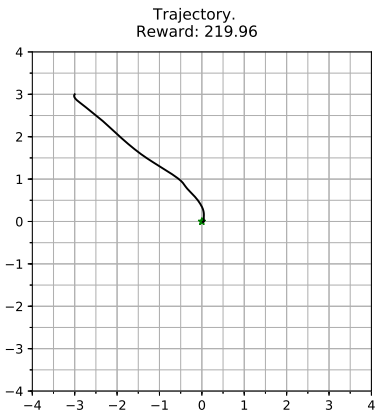


(c) Distance to goal

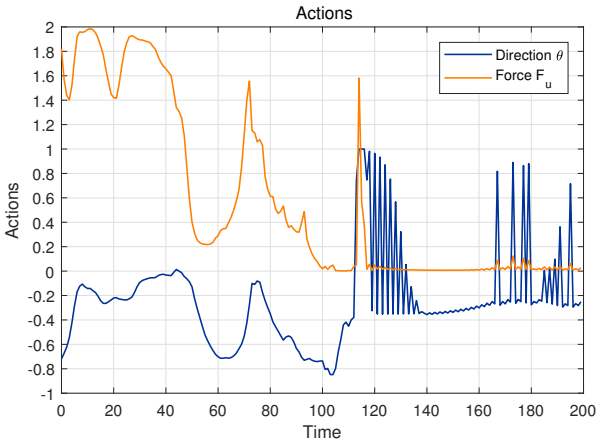


(d) Rewards

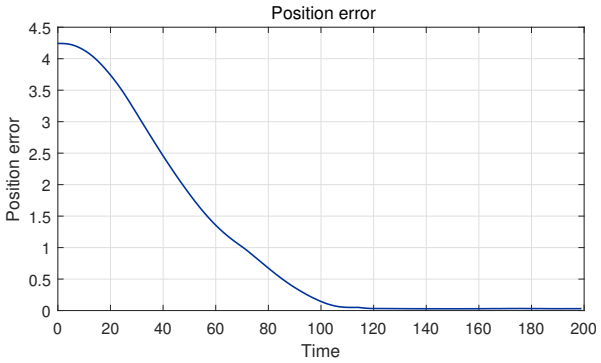
Figure A.9: Test the point mass agent trained by DDPG with r_3, from $(x_s, y_s) = (-3, 0)$



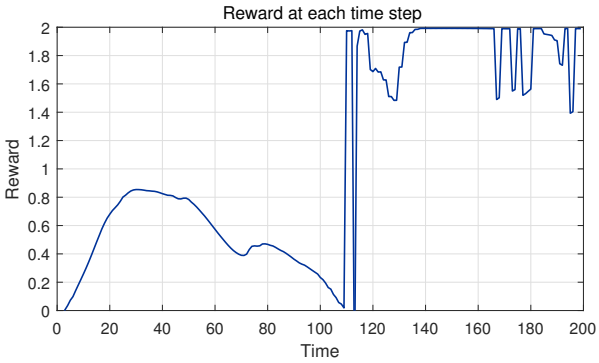
(a) Trajectory



(b) Control actions

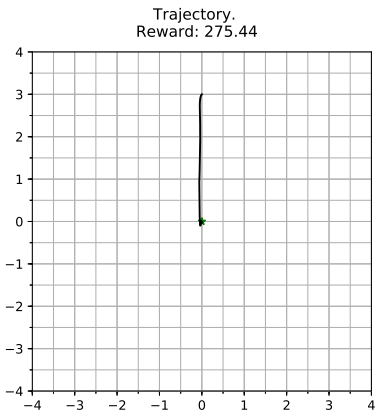


(c) Distance to goal

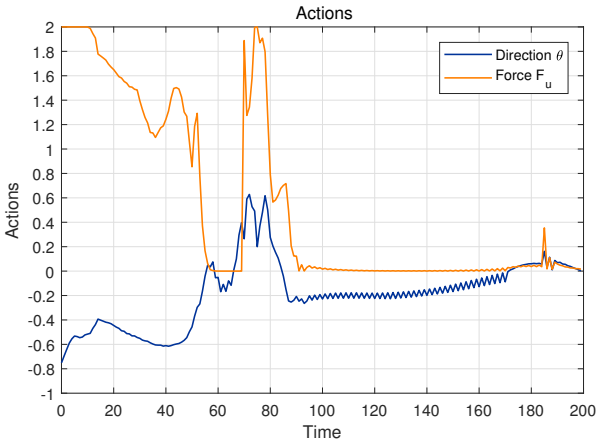


(d) Rewards

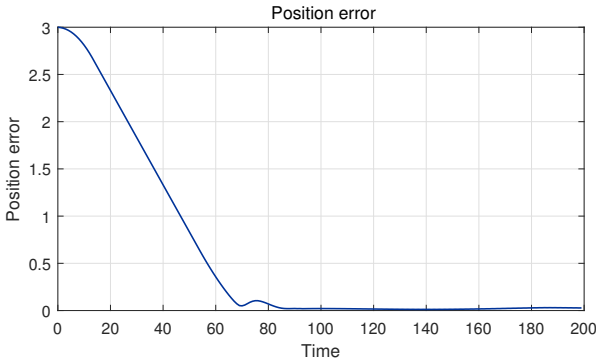
Figure A.10: Test the point mass agent trained by DDPG with r_3, from $(x_s, y_s) = (-3, 3)$



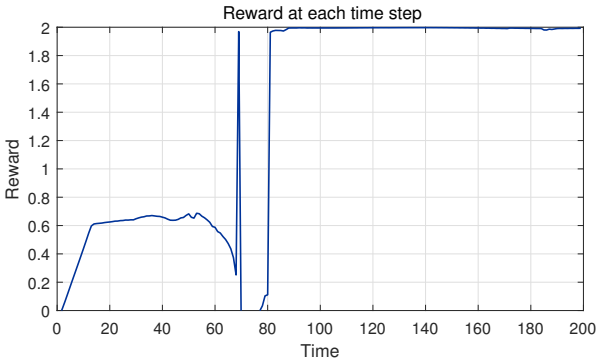
(a) Trajectory



(b) Control actions



(c) Distance to goal



(d) Rewards

Figure A.11: Test the point mass agent trained by DDPG with r_3, from $(x_s, y_s) = (0, 3)$