Using Machine Learning to Learn from Demonstration: Application to the AR.Drone Quadrotor Control

Kuan-Hsiang Fu

December 15, 2015

Abstract

Developing a robot that can operate autonomously is an active area in robotics research. An autonomously operating robot can have a tremendous number of applications such as: surveillance and inspection; search and rescue; and operating in hazardous environments. Reinforcement learning, a branch of machine learning, provides an attractive framework for developing robust control algorithms since it is less demanding in terms of both knowledge and programming effort. Given a reward function, reinforcement learning employs a trial-and-error concept to make an agent learn. It is computationally intractable in practice for an agent to learn "de novo", thus it is important to provide the learning system with "a priori" knowledge. Such prior knowledge would be in the form of demonstrations performed by the teacher. However, prior knowledge does not necessarily guarantee that the agent will perform well. The performance of the agent usually depends on the reward function, since the reward function describes the formal specification of the control task. However, problems arise with complex reward function that are difficult to specify manually. In order to address these problems, apprenticeship learning via inverse reinforcement learning is used. Apprenticeship learning via inverse reinforcement learning can be used to extract a reward function from the set of demonstrations so that the agent can optimise its performance with respect to that reward function. In this research, a flight controller for the Ar.Drone quadrotor was created using a reinforcement learning algorithm and function approximators with some prior knowledge. The agent was able to perform a manoeuvre that is similar to the one demonstrated by the teacher.

Declaration

I, Kuan-Hsiang Fu, hereby declare the contents of this dissertation to be my own work, unless otherwise explicitly referenced. This document is submitted for the degree of Master of Science at the University of the Witwatersrand. This work has not been submitted to any other university, or for any other degree.

[illegible]

2015-12-15

. . . .

Signature *Date*

Acknowledgements

I would like to thank my supervisor, Professor Turgay Celik, for his insightful suggestions and guidance throughout the course of the research. I would also like to thank Pravesh Ranchod for providing vital advice and ideas to tackle this research. I am grateful to my friends, Melissa Wu, Sonali Parbhoo and Pei Chung, for proofreading this dissertation. I also acknowledge financial support from the National Research Foundation (NRF).[1]

[1]Opinions expressed and conclusions arrived at, are those of the author and are not necessarily to be attributed to the NRF.

Contents

Chapter 1

Introduction

In the field of robotics, developing robots that can operate autonomously in unpredictable environments has been a long-standing goal we would like to achieve. Robust control algorithms are needed to make the robots operate autonomously. However, the process of programming control systems for mobile robots can involve a great deal of time and requires an interdisciplinary team of experts. Policies need to be developed so that a robot can choose an action given the robot's environment and the current state it is in. Such policies are often very difficult and challenging to code by hand. An extensive knowledge about the kinematics and the aerodynamics of the robot is needed. A simple high level action would require thousands of decisions to be made at different points in time based on the robot's state. It is often difficult to put together a sequence of action primitives to achieve a specific goal. This would require knowledge from control theorists since we need to understand how the change in the speed of the motors would affect the positions of the robot. It is not possible to pre-program the robot with all the knowledge it needs about the world; there is just too much diversity in the human environment. As a result, machine learning techniques are used to develop policies that can be used by the robots [Mahadevan 1996].

Machine learning focuses on developing systems that are able to learn from data. That is, it allows computers to learn by acquiring knowledge from past experience [Anderson *et al.* 1986]. The goal is to essentially replace explicit programming by teaching. In particular, reinforcement learning algorithms would be used in this research. Reinforcement learning is a machine learning paradigm that makes use of the concept of trial-and-error to make the agent learn [Sutton and Barto 1998]. It specifies what the goal is without specifying how to achieve it and attempts to mimic the way humans learn [Lenz 2003]. The agent learns through experience instead of being told what to do. A reward is given to the agent when the agent takes an action in a given state. The goal of reinforcement learning is to maximise the cumulative reward over time. Smart and Kaelbling [2001] applied reinforcement learning on mobile robots to learn simple tasks. In particular, Q-Learning has been used and it has been shown that the method is able to learn a decent control policy within an efficient amount of time. Q-Learning is a Reinforcement learning technique that can be used to find an optimal policy (state to action mapping) [Sutton and Barto 1998]. It has been widely used in the field of robotics for control applications and for obtaining robot motions [Kormushev *et al.* 2013]. However, Q-Learning, as well as various other reinforcement learning techniques, require that the states and actions are discrete. The entire state space needs to be enumerated and stored in memory in order for the optimal policy to be computed. In the human environment, where states and actions are continuous, we are often faced with computational resource problems: it is almost impossible or infeasible to store all the state-action pairs in memory. Even if we could store everything in memory, the computation time would take too long. This computational intractability is also known as the "Curse of Dimensionality" which a large majority of reinforcement learning algorithms suffer from. Therefore, to learn the value functions for problems with continuous states, combining reinforcement learning algorithms with function approximators is an ideal approach.

In continuous state-action space domains, we can no longer represent the value functions as explicit tables, and so generalisation is required. Function approximators are usually used to generalise unseen states based on the experienced states [Sutton and Barto 1998]. In this way, the value function can be approximated and learning can be done in a reasonable amount of time and space. However, another complication that arises is that having the agent learn from scratch is not feasible: including some prior knowledge to the learning system is needed. The prior knowledge is the demonstrations performed by the teacher. This approach is known as "Learning from Demonstration" [Argall *et al.* 2009]. Learning from demonstration algorithms provide robots with the ability to come up with new policies by learning from a teacher's demonstrations. The use of learning from demonstration algorithms and function approximators allow applying reinforcement learning in continuous state-space domains become feasible with only a little loss in performance.

In reinforcement learning, a reward function is used so that the agent can learn the behavior. The reward function basically defines the formal specification of the task: it describes the objective of the task. Since the agent's primary objective is to maximise the expected sum of rewards over time, if the reward function is poorly defined, the agent may not learn the correct behaviour. In practice, it is very challenging and difficult to specify appropriate reward functions in many control problems, especially those in robotics. Trade-offs between all different features need to be considered so that the agent can learn the most desirable states and actions in the state-action space. The problem of manually specifying the reward function can be addressed in the apprenticeship learning setting [Abbeel and Ng 2004]. Apprenticeship learning via inverse reinforcement learning, one of the methods of "Learning from Demonstration", is useful in applications where reward functions are difficult to specify and expert demonstrations are easy to acquire. It is often easier to obtain expert demonstrations than to manually specify a reward function that induces the behaviour. The objective is to recover the underlying reward function from the set of expert demonstrations so that the agent can find an optimal policy with respect to this reward function.

In this research, we applied apprenticeship learning via inverse reinforcement learning to the Parrot Ar Drone quadrotor using a simulator. The results of the experiments show that the agents, when given expert demonstrations, are able to recover the unknown reward function and reproduce the manoeuvre performed by the teacher.

The remainder of the document is structured as follows: In the next chapter, the background of this research and its related work are presented. Essentially, the concept of Learning from demonstration and the existing methods are discussed. The control technology inside the Ar Drone is included in the chapter as well. The apprenticeship learning algorithm is presented later in the chapter. The chapter ends with a description of the software and libraries used in this research. The research methodology and implementation details that have been followed during the course of this research, as well as the research hypotheses, are presented in Chapter 3. Chapter 4 and 5 present the discussion on the agents that were created as well as the results of the experiments. Chapter 6 concludes the document.

Chapter 2

Background and Related Work

2.1 Introduction

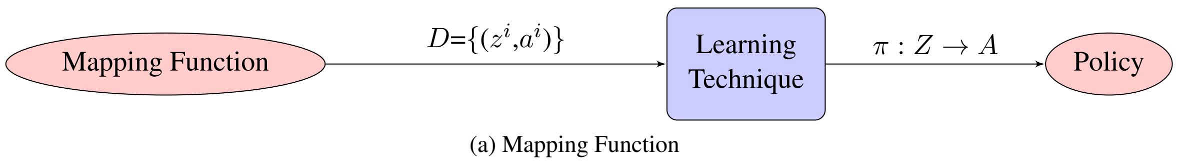
This chapter outlines the basic background relevant to the proposed research. An overview of the existing literature is presented. A summary of the key facets of reinforcement learning and function approximators is presented, particularly with respect to the algorithms which are involved in the research. The apprenticeship learning algorithm, the main focus of this research, is described and discussed in detail. The control technology inside the Ar Drone as well as the research relevant to this domain are also discussed in this chapter. We end this chapter with a discussion on the software and libraries used during the course of the research.

2.2 Learning from Demonstration

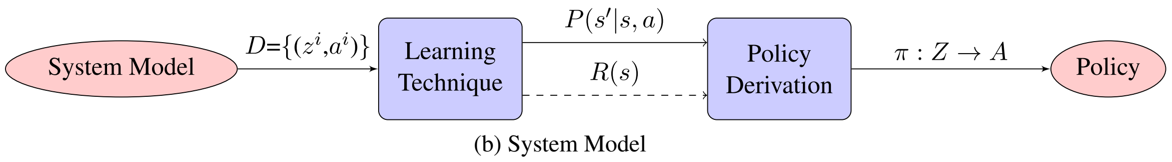
Learning techniques based on demonstration can be referred to as Learning from Demonstration, Learning by Imitation, Programming by Demonstration, Apprenticeship Learning, and Imitation Learning. Robot learning from Demonstration (LfD) is a paradigm that enables robot behaviours to be learned based on the demonstrations performed by the teacher. State-action pairs encountered during the demonstrations are recorded, which implicitly tells us the most desirable states and actions for a given task. The robot then generalises from this information to derive a policy that reproduces the demonstrated behaviour. However, the policy being derived will only perform well on the states that have been visited during the demonstration. As a result, some methods are used to improve performance by discretising continuous states to unseen states. These methods include the use of reinforcement learning and function approximators. Algorithms for Learning from Demonstration seek to expand robot capabilities to perform tasks without explicit programming. In other words, even with the absence of expert knowledge of the domain dynamics, a suitable robot controller can still be derived from the demonstrations performed by the teacher. Currently, various Learning from Demonstration algorithms have been proposed. Two recent surveys cover the scope within this field [Argall *et al.* 2009; Billing and Hellström 2010]. Argall *et al.* [2009] provides a categorisation of existing LfD approaches for policy derivation to address the robotics control problem, some of which are briefly described in this section.

2.2.1 Design Choices

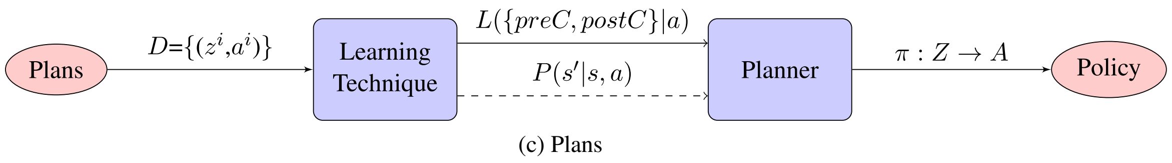
One of the common aspects of LfD is that there is a teacher providing demonstrations of the desired task, and a learner who will be supplied with a dataset of these demonstrations. The learner then derives a policy from the dataset of demonstrations that is capable of reproducing the demonstrated behavior. In developing a LfD system, many design choices need to be made. These design choices heavily influences the task formalisation and learning, since different choices result in differing representations of the learning problem that need to be solved. Some of the main design choices are described below.



(a) Mapping Function



(b) System Model



(c) Plans

Figure 2.1: Policy derivation via the generalisation approach for (a) state→action mapping function approximations, (b) dynamics system models, and (c) sequenced action plans [Argall *et al.* 2009].

Demonstration Approach

When gathering the demonstration data performed by the teacher, decisions need to be made about the demonstrator and the demonstration technique. That is, whether it should be human demonstrators, robotic teachers or hand-written control policies. This further breaks down to who controls the demonstration and who executes the demonstration. The choice of demonstrator heavily influences the type of learning algorithms that can be applied [Argall *et al.* 2009]. This is because the state and action spaces visited by the teacher might not be the ones that the learner would observe, which is known as the correspondence issue and is discussed in Section 2.2.2. As for the demonstration technique, this specifies the method by which the data is provided to the learner. Options include batch learning and on-line learning. The former is where the policy is learned once all data has been collected, and the latter refers to the policy being updated incrementally as data becomes available.

Problem Space Continuity

In developing a LfD system, choices need to be made of whether to use discrete or continuous state-action representations. If the state-action space is discrete, an appropriate policy can be derived in a reasonable amount of time by using any of the policy derivation techniques. On the other hand, if the state-action space is continuous, it is computationally intractable to derive an appropriate policy. The state-action space is so huge that it is not feasible to derive a policy that performs well in all situations. There is either not enough memory to store the state-action space or it just takes too long to derive the policy. One option in dealing with continuous domains would be to discretise the state-action space. However, if the state-action space is discretised poorly, the derived policy would not be able to perform well. The choice of discrete or continuous state-action spaces has a huge impact on how various policy derivation techniques are used to address the problem.

Policy Derivation and Performance

Several approaches exist for policy derivation. However, two key decisions need to be made on which approach to follow and whether or not the performance is able to surpass that of the teacher's demonstration. The policy derived tends to only perform well on the states that have been visited during task demonstration. We say that the policy is limited by the teacher's performance since the learner would not know what to do when it reaches states that have not been demonstrated before. If the teacher performed the task sub-optimally, the derived policy might not be able to produce the intended behaviour. As a result, policy derivation techniques seek to improve the performance by using supervised learning and reinforcement learning. The robot's task and capabilities usually determine the continuity of the action space, which in turn has an impact on the decisions being made. In practice, many control problems, especially those in robotics, deal with continuous state-action spaces. In this research, the parrot Ar. Drone is used, which comprises of continuous states and actions. Figure 2.1 shows the three approaches to derive policy from demonstration data. The three approaches are defined as *mapping function*, *system model*, and *plans* [Argall *et al.* 2009].

- Mapping function: Learning an approximation of function that maps the robot's state observations to actions from the dataset of demonstrations. This is accomplished by the use of supervised learning methods. The state and action pairs recorded in the demonstration are used as training examples. The function approximator then approximates the observed states to actions that will guide the learner to the goal state.
- System model: A state transition model of the world is created from the dataset of demonstrations, a policy is then derived from it. This is accomplished by the use of reinforcement learning methods. The transition model is learned which tells us the most desirable action to take when given a state. This eventually guides the learner to the goal state when given any state.
- Plans: Certain pre- and post-conditions must be satisfied when executing an action. Demonstration data is used to learn how actions associate with these conditions. A sequence of actions is then planned from the information which guides the learner from the initial state to the goal state.

2.2.2 Gathering Demonstration Data

Under a LfD paradigm, the teacher provides demonstrations and the learner learns from this information. However, various approaches exist for gathering the demonstration data which affects the way the demonstration dataset is built and the way the policy is derived. The platform used by the teacher for execution as well as how the demonstrations are recorded varies across approaches. For instance, one approach would use the teacher to teleoperate the learner, and the robot learner will record its own actions [Bagnell and Schneider 2001; Browning *et al.* 2004]. Another approach would use external cameras to record the demonstrations performed by the teacher [Atkeson and Schaal 1997; Bentivegna 2004]. In the latter approach, the state-action pairs recorded might not be the same as what the learner would observe. This might prevent the learner from learning the correct behaviour. In order to derive an appropriate policy, the demonstration dataset must be usable by the learner. However, we would often be faced with *Correspondence Issues* [Nehaniv and Dautenhahn 2002; Breazeal and Scassellati 2002] where a direct mapping will often not be possible due to differing sensors or motions.

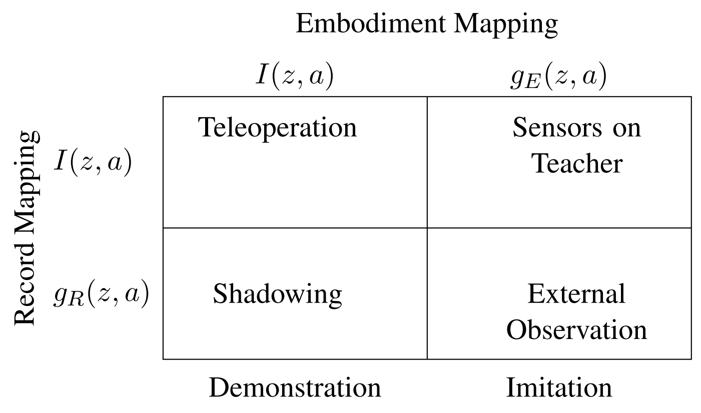


Figure 2.2: Quadrants showing the record and embodiment mappings [Argall *et al.* 2009].

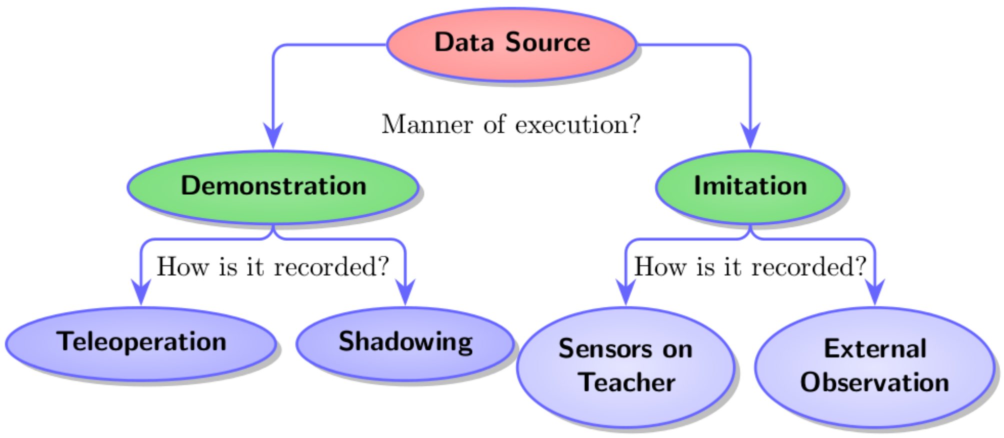


Figure 2.3: Categorisation of approaches to building the demonstration dataset [Argall *et al.* 2009].

Correspondance

The correspondance issue considers how information transfer is mapped between the teacher and the learner [Argall 2009]. We are interested in what platform has been used for execution of the task and how the task demonstration is recorded. Two mappings are considered: *Record Mapping* and *Embodiment Mapping*.

- The *Record Mapping* (Teacher Execution $\rightarrow$ Recorded Execution) describes the state-action pairs observed during teacher demonstrations. If those state-action pairs are directly recorded in the dataset, then the mapping is an identity $I(z, a)$, where z is the state observed and a is the action taken. Otherwise, some function $g_R(z, a) \neq I(z, a)$ is applied to the demonstrated data which is then recorded in the dataset.
- The *Embodiment Mapping* (Recorded Execution $\rightarrow$ Learner) describes the state-action pairs encountered by the learner. If there is a direct mapping of the state-action pairs between the dataset and the learner, then this map is the identity $I(z, a)$. Otherwise, there is some function $g_E(z, a) \neq I(z, a)$ that describes this mapping.

The intersection of the record and embodiment mappings is shown in Figure 2.2, where the contents within each quadrant represent different approaches in gathering demonstration data. Figure 2.3 shows the full categorisation of various approaches used to build the dataset of demonstrations. The four techniques are discussed below.

Demonstration

As shown in Figure 2.2, no embodiment mapping exists when teacher executions are demonstrated. This is because the demonstration is performed on the actual robot learner, the robot learner records from its own sensors, hence $g_E(z, a) \equiv I(z, a)$. However, non-direct record mapping may exist if the state-action pairs from the teacher's demonstrations are not recorded directly. Two common approaches that provide datasets of demonstrations to the robot learner include: *Teleoperation* and *Shadowing*.

- *Teleoperation*: The robot teacher operates the robot learner platform to perform a desired task. The executions are then recorded by the robot learner through its own sensors.
- *Shadowing*: The robot learner records the execution using its own sensors whilst trying to re-enact the demonstrated behaviour. Since the robot learner attempts to mimic the teacher's behaviour, this is not a direct record mapping; thus $g_R(z, a) \neq I(z, a)$.

Within the context of demonstration learning, teleoperation provides the most direct method for information transfer. This is because there is a direct record and embodiment mapping. However, operating a robot with complex motor controls via teleoperation is not manageable. As for shadowing, the record mapping is not direct since the true demonstration execution is not recorded. The mimicking execution is recorded instead. In order for the robot learner to track the teacher, an extra algorithmic component is required.

Imitation

For imitation approaches, the robot learner observes as the teacher demonstrates the desired behavior. Embodiment issues do occur since demonstration is not performed on the actual robot learner. Two common approaches for providing imitation data include: *Sensors on teacher* and *External observation*.

- *Sensors on teacher*: The teacher's execution is recorded via sensors attached to the executing platform.
- *External observation*: The teacher's execution is recorded via sensors that are located external of the executing platform.

The interested reader should refer to [Argall *et al.* 2009] for a detailed description on these approaches.

2.2.3 Deriving a Policy

Once the dataset of state-action pairs has been attained through the use of one of the above mentioned gathering methods, various techniques can be applied on the data to derive a policy. The three core approaches that have been mentioned earlier can be used to derive the policy. This includes: *Mapping function*, *System model* and *Plans*. Approaches that require few training examples with fast learning times and minimal parameter adjustments are preferable. A full categorisation of the various approaches for policy derivation from a dataset of demonstrations is shown in Figure 2.4. As shown in the figure, policy derivation approaches can be divided into three main techniques. Two of the three approaches can be further split into two techniques.

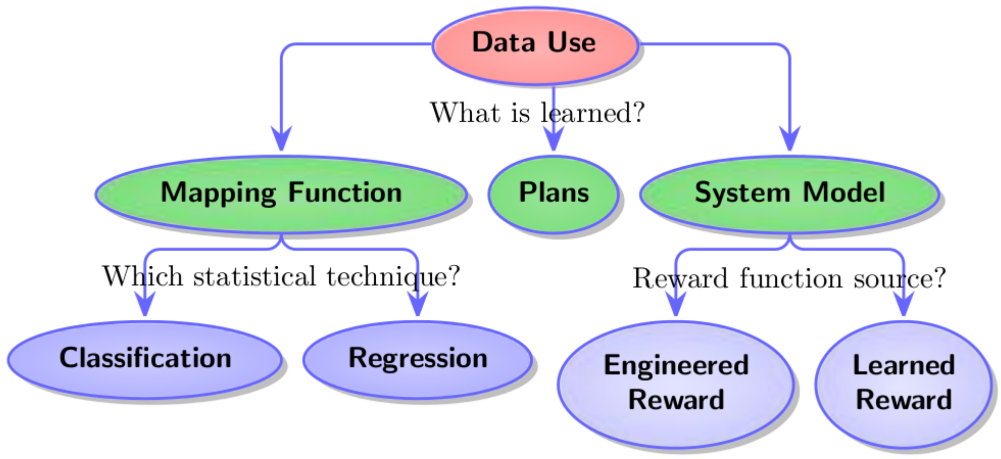


Figure 2.4: Categorization of approaches to learning a policy from a dataset of demonstrations [Argall *et al.* 2009].

Mapping Function

Mapping function approaches to policy learning approximate the state-action mapping through supervised learning methods. These methods include techniques such as *classification* and *regression*. The goal of this approach is to reproduce the underlying teacher policy, which is unknown, using any supervised learning methods. However, the state-action pairs that the learner encounters, might not be the same as the ones observed during teacher demonstration. Therefore, *classification* techniques and *regression* techniques are used to address the problem by generalizing over the set of available training examples. In this way, the learner will be able to approximate the action for states that have not been demonstrated before.

- *Classification*: Classification techniques produce discrete outputs. If the input values are continuous, similar input values are grouped together so that they can be categorised into discrete classes. Having a set of discrete classes, the classification approach can then produce a discrete output. Action controls are categorised into three levels: *low level actions*, *high-level actions* and *complex behavioral actions*. Classification algorithms can be applied at any control level. For instance, given a state of the robot, classification techniques can produce a discrete output which tells the robot to move left, right, forward or backward.
- *Regression*: Regression techniques produce continuous outputs. Input, in the form of robot states, are fed into the regressor which returns robot actions as a continuous output. Regression approaches are generally applied to low-level motions due to the fact that the continuous-valued output is often the combined result of multiple demonstration actions. The point at which the mapping approximation occurs, either at or prior to run-time, heavily influences the details of the function approximation.

The interested reader should refer to [Hastie *et al.* 2001] for a full discussion on classification and regression techniques.

Plans

An approach that represents the desired robot behaviour as a plan. The policy that is derived from this approach consists of a sequence of action primitives that lead the robot from the initial state to the goal state. The planning approach would not be employed in this research since a considerable amount of domain knowledge is needed to code the pre- and post-condition. However, the interested reader should refer to Argall *et al.* [2009].

System model

This approach adopts a state-transition model of the world, $P(s'|s, a)$, for policy learning; where s' is the state observed after taking action a in state s. A policy, $\pi : Z \rightarrow A$, can then be derived from using the model and possibly a reward function $R(s)$. Demonstration data provided to the robot as well as some autonomous exploration made by the robots generally determine the transition function. The reward function associates a reward value to a state which generally describes how desirable it is to be in that state. This approach is well defined within the field of Reinforcement learning, where the main objective of the agent is to maximise the cumulative reward over time. The reward function can either be defined by the user, or learned from the demonstrations.

- *Engineered reward functions*: Under LfD paradigm, most applications often use user defined reward functions. These rewards have a tendency to be sparse; where the reward value received for most of the states is zero. Few states such as those near obstacles will have high negative values, and states close to goal state will have high positive values. This usually causes blind exploration for the robot where it keeps exploring with no feedback. Thus the aim of demonstration-based techniques is to assist the robot in discovering the rewards. Demonstration consisting of state-action pairs performed by the teacher highlights interesting areas of the desired task. Therefore states encountered during demonstration are associated with positive rewards which eliminates long periods of exploration with no feedback [Smart and Pack Kaelbling 2002]. Reward functions and demonstrations have a huge impact on how well the learner performs and how good the derived policy is. Reward functions tell us how desirable it is to be in a particular state, therefore negative behaviors would be penalised and positive behaviors would be reinforced. Demonstrations provide recommended actions and suggest promising areas for exploration. If the demonstration performed is poor, it could actually worsen the policy learning.

- *Learned reward functions*: As mentioned previously, reward function influences the learner's performance. The reward function defines the formal specification of the task. If the reward function is poorly defined, the derived policy will not be able to reproduce the intended behaviour. However, in real world systems, defining an effective reward function can often be very difficult. One approach to resolve with this issue is to learn the reward function. *Inverse Reinforcement Learning* [Russell 1998] is a subfield within Reinforcement learning that learns the reward function. Combining the system model approach with inverse reinforcement learning we will be able to learn the reward function as well as the transition function of a given control task. This approach is known as "Apprenticeship learning via inverse reinforcement learning" [Abbeel and Ng 2004] which is the main focus of this research and will be discussed in detail later.

A more detailed description of reinforcement learning methods will be presented in the section below.

2.3 Reinforcement Learning and Markov Decision Processes

Reinforcement learning is a machine learning paradigm that has been applied widely in robotics, and has shown to be well suited and effective for robot learning [Kober and Peters 2012; Smart 2002]. In reinforcement learning, learning is accomplished within the framework of Markov Decision Processes (MDPs), which is a decision-making model and is discussed in more detail later. Reinforcement learning is an example of unsupervised learning that allows the robot to learn based on empirical experiences of the world. The robot interacts with its environment through trial-and-error to achieve a goal. The goal is achieved when an optimal behavior has been discovered. The robot selects an action a_t in state s_t at time t, resulting to a new state s_{t+1}, a feedback is provided by the environment in a form of reward which measures how good the action was. The robot's goal is to maximise cumulative reward over time so that the optimal policy $\pi^* : Z \rightarrow A$ can be learned, where Z is the set of state observations and A is the set of actions. The optimal policy maps states to actions which tells the robot which actions a to take in state s in order to maximise the cumulative reward.

2.3.1 Sequential Decision Problems

In sequential decision problems such as utilities, uncertainty, and sensing - which generalise search and planning problems; the agent's utility is dependent on a sequence of decisions [Russell *et al.* 1995]. Solutions to sequential decision problems in known, deterministic domains are very straightforward. We could apply search algorithms to generate action sequences that leads the agent to a goal state. However, due to the unreliability of the actions, such solutions would not work in stochastic domains. Since the outcome of each of the actions is stochastic, the probability of reaching state s' if action a is taken in state s can be denoted as $P(s'|s,a)$, which is known as the transition model. If the state transitions have the Markov property [Sutton and Barto 1998], then Equation 2.1 holds. That is, the probability of reaching state s' depends only on state s_t and action a_t and not on the values of the past events. Whereas if the Markov property does not hold, then the probability depends on the history of earlier states as shown in Equation 2.2. In other words, the path that leads to the current state is relevant and is included in the probability of determining the next state s'.

$$Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\} \tag{2.1}$$

$$Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, ..., r_1, s_0, a_0\} \tag{2.2}$$

Given the transition model of a stochastic environment, we also need to specify the utility function for the agent. The utility function, also known as the value function, determines the agent's utilities and thus depends on a sequence of states. For the remainder of the document we refer to the agent's utility or value as *value function*. Later in this chapter we examine how such value functions can be specified. During the interaction between the agent and the environment, for every state the agent visits, it will receive a reward. So, in the most basic case, the value function would simply be the sum of the rewards received based on the sequence of states visited. Thus, a Markov decision process (MDP) can be defined as a sequential decision problem for a fully observable, stochastic environment specified by a Markovian transition model and rewards [Russell *et al.* 1995]. A MDP is a tuple (S, A, P, γ, D, R), where S is a finite set of states; A is a set of actions; P is the set of transition probabilities; γ is the discount factor (which will be discussed later); D is the initial-state distribution, from which the initial state s_0 is drawn; and R is the reward function. A solution to a problem posed in a MDP formalism is called a policy. So, instead of providing a fixed action sequence, the policy recommends an action for any state reachable by the agent. This is useful in the sense that no matter which state the agent ends up in, the agent will always know what to do next.

The agent will be able to reach the goal state by executing the actions provided by the policy. However, there might be countless policies that are able to do the same task. Due to the stochastic nature of the environment, each of those policies might generate sequences of states that are different to each other but nonetheless accomplishes the same task. What we are interested in is the optimal policy, the policy that performs the best, where performance depends on the sequence of states generated. Since we are able to calculate the expected value of a sequence of states using a value function, the performance of a policy can be measured by the expected value of the possible sequence of states being generated. Thus, the optimal policy is a policy that generates a sequence of states yielding the highest expected value. Algorithms for calculating optimal policies are discussed later in this chapter. We now discuss different ways of evaluating state sequences as well as how performance of different policies can be compared.

Values over time

When an agent executes an action a_t at state s_t, not only will the agent reach a new state s_{t+1} but also receives a reward r_{t+1}. Thus, after executing a certain number of actions, we would also get a sequence of rewards. The agent seeks to maximise the sum of the rewards, known as the return G_t, which can be used to measure the performance of the agent. The return can be calculated using additive or discounted rewards as shown in Equation 2.3 and Equation 2.4, respectively.

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T \tag{2.3}$$

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.4}$$

In choosing what kind of rewards to use, we first need to determine whether the task at hand is episodic or continuing. In episodic tasks, there is a finite horizon for decision making, which means that after a finite number of steps, nothing else matters. This usually happens when the agent reaches the terminal state. In this case, we would have a finite number of state sequences from start state to terminal state known as an episode where the agent tries to maximise using additive rewards. In Equation 2.3, T is the final time step, and with a finite horizon, the state sequences are finite which means that the sum of rewards would also be finite. In continuing tasks, or if the terminal state does not exist, the sequence of states will be infinitely long. Calculating returns using additive rewards would therefore not be appropriate since $T = \infty$ and the sum of rewards would generally be infinite. Thus, for continuing tasks, discounted rewards are used instead. In Equation 2.4, γ is the discount factor where $0 \leq \gamma \leq 1$. The discount factor specifies how future rewards are valued by the agent and is used to keep the return finite: if γ is close to 0, future rewards are viewed as insignificant, and the agent is concerned with only maximising immediate rewards. The closer γ is to 1, the more prescient the agent and the higher it values future rewards.

Optimal policies and the values of states

Given that the value of a state sequence can be calculated using the expected return, the performance of different policies can be measured by comparing their expected values when following its policy. The value of a state s when following a policy π is defined as follows:

$$V^{\pi}(s) = E_{\pi}\{G_t | s_t = s\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s\right\} \tag{2.5}$$

which gives the expected return when starting in state s and executing policy π. The function V^{π} is known as the state-value function for policy π, which tells us the goodness of being in a particular state. The difference between the state-value function and the reward function is that the reward function provides the short term immediate reward for being in a state, whilst the state-value function provides the long term total reward from that state onward. Similarly, the action-value function for a policy π can be defined as:

$$Q^{\pi}(s,a) = E_{\pi}\{G_t | s_t = s, a_t = a\} = E_{\pi}\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a\right\} \tag{2.6}$$

which provides the value of taking action a in state s and thereafter following policy π. These value functions are used to discover the optimal policy: this is accomplished by calculating the values of each state which can then be used to select an optimal action. The optimal policy π^* thus chooses the action that maximises the expected value of the subsequent state as shown in Equation 2.7.

$$\pi^*(s) = \arg\max_{a \in A(s)} \sum_{s'} P(s'|s,a) V(s') \tag{2.7}$$

An optimal policy is a policy that performs just as well as or better than any other policy, which means that its value function is greater than or equal to every other value function for every possible state. From Equation 2.5, we see that the value of a state is the expected sum of discounted rewards from that state onward, meaning that the value of a state is proportional to that of its successor states. This recursive relationship can be expressed as follows:

$$V^{\pi}(s) = r_{t+1} + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) V(s') \tag{2.8}$$

known as the Bellman equation. Various methods exist for computing the optimal value function [Sutton and Barto 1998]. The models of the environment is learned first, which allows for the calculation of the optimal value function, and thereafter deriving the optimal policy. Problems arise when trying to learn a good model in a dynamic environment with limited data. Algorithms attempting to iteratively approximate the optimal value function are preferred and are discussed next.

2.3.2 Reinforcement Learning Algorithms

Various algorithms exist for solving MDPs by computing the optimal value functions [Szepesvári 2010], two of the most commonly used reinforcement learning algorithms are discussed below.

Q-Learning

Q-Learning [Watkins and Dayan 1992] learns the state-action value function, $Q(s,a)$. The function, $Q^{\pi}(s,a)$ is known as the state-action value function for policy π. It specifies the expected return when starting in state s, taking the action a, and thereafter following the policy π. This function therefore reflects how good it is to take action a in state s. The 4-tuples $(s_t, a_t, r_{t+1}, s_{t+1})$ are used to iteratively update the approximation to the optimal Q-function:

$$Q^*(s,a) = R(s,a) + \gamma \sum_{s'} P(s,a,s') \max_{a'} Q^*(s,a')$$

The optimal policy, $\pi^*(s)$, can then be computed:

$$\pi^*(s) = arg\max_a Q^*(s, a)$$

Algorithm 1 shows the Q-Learning algorithm. As shown in Algorithm 1, the Q-values approximation are updated iteratively, starting with random values. The value functions are stored in a tabular form, therefore states and actions are assumed to be discrete. Q-Learning has shown to converge to optimum action-values whilst all actions are repeatedly executed in all of the states and that the action-values are represented discretely [Watkins and Dayan 1992].

Algorithm 1 Q-learning: An off-policy TD control algorithm [Sutton and Barto 1998]

```
1: Initialise $Q(s, a)$ arbitrarily
2: Repeat (for each episode):
3:     Initialise $s$
4:     Repeat (for each step of episode):
5:         Choose $a$ from $s$ using policy derived from $Q$ (eg., $\epsilon$-greedy)
6:         Take action $a$, observe $r$,$s'$
7:         $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{a'} Q(s', a') - Q(s, a)]$
8:         $s \leftarrow s'$;
9:     until $s$ is terminal
```

Sarsa

Sarsa is an on-policy Temporal-difference (TD) control algorithm as shown in Algorithm 2. The main difference between the Sarsa algorithm and the Q-learning algorithm is the action-value update rule. Q-learning updates the action-value function regardless to the actual action chosen for the next state; it assumes the estimated optimal action is being chosen each time (applies the best Q-value and disregards the actual policy being followed), and is considered as an off-policy algorithm. While for Sarsa, it updates the action-value function based on the action being chosen, and so is an on-policy algorithm.

Algorithm 2 Sarsa: An on-policy TD control algorithm [Sutton and Barto 1998]

```
1: Initialise $Q(s, a)$ arbitrarily
2: Repeat (for each episode):
3:     Initialise $s$
4:     Choose $a$ from $s$ using policy derived from $Q$ (eg., $\epsilon$-greedy, taking random action $\epsilon\%$ of the
   time)
5:     Repeat (for each step of episode):
6:         Take action $a$, observe $r$,$s'$
7:         Choose $a'$ from $s'$ using policy derived from $Q$ (eg., $\epsilon$-greedy)
8:         $Q(s, a) \leftarrow Q(s, a) + \alpha([r + \gamma Q(s', a') - Q(s, a))]$
9:         $s \leftarrow s'$; $a \leftarrow a'$;
10:    until $s$ is terminal
```

TD methods follow the pattern of generalised policy iteration where the value and policy functions are updated until optimal and hence consistent with each other [Sutton and Barto 1998]. This interaction is expressed at line 7 and 8 of the Sarsa algorithm, where we keep on estimating the action-value function for the current policy whilst simultaneously adjusting the policy to be greedy with respect to the action-value function. The agent performs action selection using ϵ-greedy, which determines the exploration rate. Most of the time the agent exploits using its current knowledge, while ϵ fraction of the time it chooses a random action. For each step of an episode (alternating sequence of states and state-action pairs), the agent performs an action, observes the reward and resulting states and updates the action-value function based on those values. Since the estimate of the action-value function is updated using the difference in values between successive states, this is often called temporal-difference learning. The goal is to keep adjusting the action-value function so that it converges to the correct value.

In this research we use Sarsa instead of Q-learning since it allows the Q function to be updated on the basis of the actual action taken. This allows us to keep the agent away from states which are not preferred more often.

2.4 Eligibility Traces

The use of eligibility traces can be seen as one of the methods to speed up the learning process. When the Sarsa algorithm receives the reward r and the next state s', it only updates the action-value function for the immediate preceding state-action pair. That is, only the preceding state and action receive credit or blame. However, since the reward signal provides useful information for learning earlier estimations, those estimations should be updated as well. Eligibility traces do this by providing temporary records of events that have occurred; an example would be visiting a state or taking an action [Sutton and Barto 1998]. For each observation received, we can then update the values based on these events. The Sarsa(λ) algorithm is shown in Algorithm 3.

Algorithm 3 Sarsa(λ) [Sutton and Barto 1998]

```
1:  Initialise $Q(s,a)$ arbitrarily
2:  Repeat (for each episode):
3:      Initialise $e(s,a)$ = 0, for all $s$, $a$
4:      Initialise $s$, $a$
5:      Repeat (for each step of episode):
6:          Take action $a$, observe $r$,$s'$
7:          Choose $a'$ from $s'$ using policy derived from $Q$ (eg., $\epsilon$-greedy)
8:          $\delta \leftarrow r + \gamma Q(s',a') - Q(s,a)$
9:          $e(s,a) \leftarrow e(s,a) + 1$
10:         For all $s, a$:
11:             $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
12:             $e(s,a) \leftarrow \gamma\lambda e(s,a)$
13:         $s \leftarrow s'$; $a \leftarrow a'$;
14:     until $s$ is terminal
```

where $e(s,a)$ is the trace for state-action pair s, a; λ is the decay parameter, and $0 \leq \lambda \leq 1$. λ= 0 corresponds to updating only the preceding action-value estimate, and λ = 1 corresponds to equally updating the estimates for all the eligible state-action pairs.

2.5 Generalisation and Function Approximation

In the previous sections, we showed how TD methods can be implemented to compute the action-value function and how eligibility traces can accelerate the learning process. Sarsa and Q-learning both assume the states and actions are discrete since they need to store all the Q-values in a table explicitly. However, when we are faced with continuous states and actions, it is not possible to represent the action-value function of all the states and actions in a table. Two approaches exist that can be used to address the problem: One would be to discretise the states and actions. Another would be to generalise across the states and actions. The latter approach will be considered. If the states and actions can be usefully generalised, then a more compact representation of the value function can be achieved by using function approximation. Various function approximator methods exist [Anderson *et al.* 1986; Marsland 2011;

Hastie *et al.* 2001]. Some of the methods will be described, namely *Artificial Neural Networks*, *K-Nearest Neighbor*, *Locally Weighted Regression*, *Radial Basis Functions* and *Fourier Basis*.

Artificial Neural Networks

Artificial neural networks (ANNs) [Hastie *et al.* 2001; Hertz *et al.* 1991] can mathematically model the way biological brains work, which gives machine the capability to think like humans. ANNs compute values from input data by feeding data through the network. ANNs can therefore learn the correlated patterns between the input data and the corresponding target value. Once trained, ANNs can be used to effectively predict the outcome of unseen input data. Artificial neural networks seem to be a reasonable choice for the approximation of the value function. It has been shown that one hidden layer in the neural network is sufficient to approximate any continuous function [Funahashi 1989]. ANNs can be applied to classification and regression problems. In several Reinforcement learning systems, ANNs have been applied successfully to approximate the value function [Singh and Bertsekas 1997; Zhang and Dietterich 1995].

K-Nearest Neighbor

K-nearest neighbour (KNN) classifiers [Hastie *et al.* 2001] do not require the model to be fit. The idea behind KNN is that when the model that describes the data is not known, we look for similar data. In order to classify a query point x_0, we look at the k training points nearest to point x_0. The class of the query point will then be set to the class that is the most common within those nearest neighbours.

Locally Weighted Regression

Locally weighted regression (LWR) is a memory-based, classic approach to solve the function approximation problem [Cleveland and Devlin 1988; Atkeson *et al.* 1997; Christopher *et al.* 1997]. A local model is formed to answer each query, using a weighted regression in which close points are weighted more than distant points [Atkeson 1991]. All the data is kept in memory to calculate the prediction. This is because for every query point, LWR calculates a new model by fitting a line to the data. The training points with a closer proximity to the query point will influence the regression more heavily. Therefore, depending on the query points, different lines will be fitted to the data each time.

Radial Basis Functions

Radial basis functions (RBFs) are the natural generalisation of coarse coding to continuous-valued features [Sutton and Barto 1998]. In coarse coding, binary features are used to represent the state space. Thus, the feature can only take on a value of 0 or 1 (whether a feature is present or not). Whilst for RBFs, the feature can be a real number - which reflects varying degrees at which a feature is present. Equation 2.9 shows a Gaussian type RBF feature. The value of the RBF feature depends on the distance between the state, s, the center, c_i, and the feature's width, σ_i.

$$\phi_i(s) = exp(-\frac{\|s - c_i\|^2}{2\sigma_i^2}) \tag{2.9}$$

Fourier Basis

Fourier basis is a linear value function approximation scheme based on the Fourier series [Konidaris *et al.* 2011]. The terms of the Fourier series are used as basis functions, thus the nth order Fourier basis for d state variables is the set of basis functions defined as:

$$\phi_i(x) = cos(\pi c^i \cdot x) \tag{2.10}$$

where $c^i = [c_1, ..., c_d]$, $c_j \in [0, ..., n]$, $1 \leq j \leq d$. The value of the basis function is obtained via the vector **c** which assigns an integer coefficient (between 0 and n) to each variable in **x**; the set of basis functions is obtained by varying these coefficients. Please refer to Konidaris *et al.* [2011] for more information.

The value function is approximated as a weighted sum of a given set of basis functions $\phi_1(s), ..., \phi_n(s)$:

$$V_t(s) = \overrightarrow{\theta}_t^T \overrightarrow{\phi_s} = \sum_{i=1}^{n} \theta_t(i)\phi_i(s) \tag{2.11}$$

Since the value function is linear in the weights vector (θ), this is known as the linear value function approximation. As can be seen, the value function depends totally on $\overrightarrow{\theta}_t$. Thus changing the values of $\overrightarrow{\theta}_t$ results in a different value function approximation. The goal is to find a vector of weights that corresponds to an approximate optimal value function. This is done by using gradient descent as the update rule as shown in Equation 2.12.

$$\overrightarrow{\theta}_{t+1} = \overrightarrow{\theta_t} + \alpha[v_t - V_t(s_t)]\nabla_{\overrightarrow{\theta_t}} V_t(s_t), \tag{2.12}$$

where v_t is the target output (backup for value prediction), and V_t is the value function that can be computed by any of the supervised learning methods. Using linear function approximation methods, the gradient of the approximate value function with respect to $\overrightarrow{\theta_t}$ can be defined as:

$$\nabla_{\overrightarrow{\theta_t}} V_t(s) = \overrightarrow{\phi_s} \tag{2.13}$$

Thus the gradient descent update rule can be rewritten as follows:

$$\overrightarrow{\theta}_{t+1} = \overrightarrow{\theta_t} + \alpha[v_t - V_t(s_t)]\overrightarrow{\phi_s} \tag{2.14}$$

The Sarsa with function approximation algorithm is shown in Algorithm 4.

Algorithm 4 Sarsa(λ) with Function Approximation [Sutton and Barto 1998]

```
 1: Initialise $\overrightarrow{\theta}$ arbitrarily
 2: Repeat (for each episode):
 3:     Initialise $\overrightarrow{e}$ = $\overrightarrow{0}$
 4:     $s$, $a \leftarrow$ initial state and action of episode
 5:     $F_a \leftarrow$ set of features present in $s$, $a$
 6:     Repeat (for each step of episode):
 7:         $\overrightarrow{e} \leftarrow \gamma\lambda\overrightarrow{e}$
 8:         For all $i \in F_a$:
 9:             $e(i) \leftarrow e(i)$ + 1
10:         Take action $a$, observe $r$, and next state, $s$
11:         $\delta \leftarrow r$ - $\sum_{i \in F_a} \theta(i)$
12:         If $s$ is terminal, then $\overrightarrow{\theta} \leftarrow \overrightarrow{\theta} + \alpha\delta\overrightarrow{e}$; go to next episode
13:         With probability 1 - $\epsilon$:
14:             For all $b \in A(s)$:
15:                 $F_b \leftarrow$ set of features present in $s$, $b$
16:                 $Q_b \leftarrow \sum_{i \in F_b} \theta(i)$
17:             $a \leftarrow$ argmax$_{b \in A(s)}$ $Q_b$
18:         else
19:             $a \leftarrow$ a random action $\in A(s)$
20:             $F_a \leftarrow$ set of features present in $s$, $a$
21:             $Q_a \leftarrow \sum_{i \in F_a} \theta(i)$
22:         $\delta \leftarrow \delta + \gamma Q_a$
23:         $\theta \leftarrow \theta + \alpha\delta\overrightarrow{e}$
```

Function approximation is very useful in tasks where the state or action spaces include continuous variables. The use of function approximation provides another way of accelerating the learning process, this is because the number of parameters we use to approximate the value function is much less than the number of states. The downside, however, is that changes made to one parameter affects the predicted value of many other states. Even so, function approximation is still preferred since it allows the agent to learn more quickly and efficiently with only a little loss in performance.

2.6 Step-Size Parameter

The use of function approximation and eligibility traces is able to accelerate the learning process, however, the overall success as well as the performance depend on the learning rate or step-size parameter α.

For each step of an episode, we would receive a vector of features ϕ_t and a reward r_t. The learning algorithm then computes the prediction or TD error using

$$\delta_t = r_t + \gamma\theta_t^T\phi_{t+1} - \theta_t^T\phi_t \tag{2.15}$$

updating eligibility traces using

$$e_t = \gamma\lambda e_{t-1} + \phi_t \tag{2.16}$$

and in the end, updating the weights using

$$\theta_{t+1} = \theta_t + \alpha\delta_t e_t \tag{2.17}$$

The step-size α thus controls how far to step in the direction of the current update, in this case, how the weights should be adjusted. Since the estimated value function depends totally on the parameters of the function approximator, the step-size plays a critical role in determining the success of the learning process. Choosing a step-size that is too large could accelerate the learning process which leads to a very fast convergence, but at the same time it also has a high probability of causing the process to diverge. On the other hand, we do not want to make the step-size too small such that it could make the convergence slower. Since the performance heavily depends on the choice of step-size, a naive approach which often works well in practice would be to evaluate performance across a range of step-sizes. This, however, is computationally expensive. In order to eliminate the need to tune the learning rate, adaptive strategies (adapting the step-size based on the prediction error) are preferred. One of the strategies, Alpha-Bound, has shown to prevent function approximation divergence and out-perform related approaches with tuned parameters [Dabney and Barto 2012]. The algorithm simply uses the following equation to modify the learning rate:

$$\alpha_t = min[\alpha_{t-1}, |e_t^T(\gamma\phi_{t+1} - \phi_t)|^{-1}] \tag{2.18}$$

where $\alpha_0 = 1.0$, and the update is done before the weights are updated. Readers interested in the derivation of the bounds should refer to [Dabney and Barto 2012].

2.7 Apprenticeship Learning via Inverse Reinforcement Learning

For all control problems, the main goal is to find the control policy π^* for the given task. The control policy prescribes an action to take for each given state. However, it is very difficult to hand code the control policy in many settings. As a result, the problem is usually broken down into specifying the reward function and dynamics model and later applying the reinforcement learning algorithm to discover a good control policy.

Apprenticeship learning via inverse reinforcement learning [Abbeel and Ng 2004] can be viewed as using a system model approach to derive a policy from demonstration data. In specific, it learns the reward function then applies a reinforcement learning algorithm. The reward function defines the objective of the task, hence it is very crucial for any reinforcement learning task. In many applications, the reward function is usually manually specified by the user, which the reinforcement learning agent will use to discover a good policy. A policy which maximises the expected sum of rewards accumulated when acting according to that policy. The reward function penalises the agent when it reaches undesirable or detrimental states, and rewards the agent when it reaches desirable states. Therefore, if the reward function is poorly specified, the agent will never learn the correct behaviour. However, in many control problems, especially those in robotics, even the reward function is often difficult to manually specify. Apprenticeship learning via inverse reinforcement learning addresses the above problem.

Apprenticeship learning via inverse reinforcement learning basically comprises of two steps: an inverse reinforcement learning step and a reinforcement learning step. In inverse reinforcement learning, the objective is to recover the reward function when given the optimal policy and the dynamics model. The recovered reward function is then used in the reinforcement learning step to find the optimal policy. However, the optimal policy is not actually given in the inverse reinforcement learning step, only execution traces of the optimal policy is given. So, in the apprenticeship learning setting, we can consider the expert as attempting to maximise a reward function when demonstrating the task. The expert trajectories are execution traces of the expert policy that the algorithm will use to find a policy that induces a behaviour similar to that of the expert's. This algorithm is often used in applications where it is challenging to manually specify the reward function, since it is easier to acquire expert demonstration than manually specifying a reward function that induces the intended behaviour. The apprenticeship learning via inverse reinforcement learning algorithm does not necessarily recover the expert's reward function, but instead, produces a policy that attains performance proximate to that of the expert. The algorithm is decribed in more detail in the sub-sections that follow.

2.7.1 Markov Decision Process

We consider learning in a Markov decision process (MDP) where the reward function is not explicitly stated, but rather, expert trajectories are given which implicitly tells us the reward function that the expert is trying to maximise. A MDP is a tuple (S, A, P, γ, D, R), where S is a finite set of states; A is a set of actions; P is a set of state transition probabilities ($\{P_{sa}\}$ provides the state transition distribution upon taking action a in state s); γ is the discount factor where $0 \leq \gamma < 1$; D is the initial-state distribution, from which the start state s_0 is drawn; R is the reward function which is assumed to be bounded by 1. Since we are not explicitly given the reward function, we denote a MDP without a reward function as MDP\R, which is a tuple of the form (S, A, P, γ, D).

2.7.2 Expert Reward Function

Since only the expert trajectories are given, we can consider the expert as attempting to maximise a reward function that can be expressed as a linear combination of known features. Equation 2.19 defines the expert reward function where ϕ is a function that maps a state to a vector of values between 0 and 1 known as the feature vector; and w^* is the vector of optimal weights specifying the relative weighting between these features.

In order to ensure the rewards are bounded by 1, Equation 2.20 must hold as well. The dot product of the feature vector and the weight vector determines the reward of a given state. Given state s_k, the feature vector $\phi(s_k)$ tells us all the features that are present in the state s_k. If there are k features, we would then have $\phi : S \to [0,1]^k$, $w^* \in \mathbb{R}^k$, and $\Sigma_{i=1}^{k}|w_i^*| \leq 1$. In the highway driving simulator [Abbeel and Ng 2004], ϕ is the vector of features indicating which lane the car is currently in, as well as the closest car in the current lane. If the expert keeps hitting cars during the demonstration, the resulting weight vector w^* would induce a nasty driving style. More specifically, the feature indicating a collision would have a higher value than other features. Thus, it is the weight vector that determines the reward function. Since the expert's weight vector is unknown, the goal of the apprenticeship learning algorithm is to return a weight vector that produces a behavior similar to that of the expert's. In order to do so, the expected features of a given policy must be compared to that of the expert's.

$$R^*(s) = w^* \cdot \phi(s) \tag{2.19}$$

$$\Sigma_{i=1}^{\infty} |w_i^*| \leq 1 \tag{2.20}$$

2.7.3 Feature Expectations

In the system model approach for deriving a policy, a reinforcement learning algorithm is used. The policy returned by the reinforcement learning algorithm is a policy that maximises the accumulated discounted rewards. Using Equation 2.19, the value of a policy π can be rewritten as:

$$E_{s_0 \sim D}\left[V^{\pi}(s_0)\right] = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi\right] \tag{2.21a}$$

$$= E\left[\sum_{t=0}^{\infty} \gamma^t w \cdot \phi(s_t) | \pi\right] \tag{2.21b}$$

$$= w \cdot E\left[\sum_{t=0}^{\infty} \gamma^t \phi(s_t) | \pi\right] \tag{2.21c}$$

where the start state s_0 is drawn from D and the expectation is taken with respect to the sequence of states encountered by taking actions according to the policy π. The expected discounted accumulated feature value vector is defined as the feature expectations $\mu(\pi)$ [Abbeel and Ng 2004]. Equation 2.22 defines the feature expectations of a given policy π.

$$\mu(\pi) = E\left[\sum_{t=0}^{\infty} \gamma^t \phi(s_t) | \pi\right] \tag{2.22}$$

Using Equation 2.22, the value of a policy π can be simplified to the scalar product of the weight vector and the feature expectations of the policy as shown in Equation 2.23.

$$E_{s_0 \sim D}\left[V^{\pi}(s_0)\right] = w \cdot \mu(\pi) \tag{2.23}$$

Comparing equations 2.21a and 2.23, we see that given a policy (π), the expected sum of discounted rewards is dependent only on the feature expectations of the given policy [Abbeel and Ng 2004]. Thus, if two policies have feature expectations that are close to each other, we can expect that they have similar reward functions and behaviours.

2.7.4 Expert Feature Expectations

The expert feature expectations μ_E is calculated in the same way as any other policy. That is, the expected discounted accumulated features when acting according to the expert's policy π_E. The estimate of the expert's feature expectations μ_E is the feature expectations of the expert policy μ_{π_E}. However, the expert's policy π_E is not given and so the expert's feature expectations is estimated using the expert's trajectories. Each expert trajectory provides an expert's path through the state space: $s_0, s_1, ..., s_e$ where s_e indicates the goal or last state visited. Therefore, by observing the expert demonstrating the task several times, the empirical estimate for μ_E can be calculated as follows:

$$\hat{\mu_E} = \frac{1}{m} \sum_{i=0}^{m} \sum_{t=0}^{\infty} \gamma^t \phi(s_t^{(i)}) \tag{2.24}$$

where m is the number of trajectories. This gives the expert's average feature expectations. The aim of the apprentice learning algorithm is then to find feature expectations close to the expert's.

2.7.5 Algorithm

In the apprenticeship learning setting, we are given an MDP\R, a feature mapping ϕ and expert trajectories. The goal of the algorithm is to identify a policy with performance similar to that of the expert's, on the unknown reward function $R^*(s) = w^* \cdot \phi(s)$ [Abbeel and Ng 2004]. To accomplish this, the apprenticeship learning via inverse reinforcement learning algorithm first estimates the expert's feature expectations μ_E and then attempts to find a policy $\tilde{\pi}$ inducing feature expectations $\mu(\tilde{\pi})$ close to μ_E. The algorithm to find such a policy $\tilde{\pi}$ is presented below:

Algorithm 5 Apprenticeship Learning via Inverse Reinforcement Learning Algorithm [Abbeel and Ng 2004]

```
INPUT: An MDP\R, a feature mapping $\phi$, and the expert's feature expectations $\mu_E$.
OUTPUT: A set of policies $\{\pi^{(i)} : i = 0...n\}$.
 1: Randomly select some policy $\pi^{(0)}$, compute (or approximate via Monte Carlo) $\mu^{(0)} = \mu(\pi^{(0)})$, and
    set $i = 1$.
 2: Set $w^{(1)} = \mu_E - \mu^{(0)}$ and $\bar{\mu}^{(0)} = \mu^{(0)}$.
 3: Set $t^{(1)} = \|\mu_E - \mu^{(0)}\|_2$.
 4: if $t^{(i)} \leq \epsilon$ then
 5:     terminate.
 6: end if
 7: while $t^{(i)} > \epsilon$ do
 8:     Apply reinforcement learning algorithm to compute the optimal policy $\pi^{(i)}$ for the MDP using
    $R = (w^{(i)})^T\phi$.
 9:     Compute $\mu^{(i)} = \mu(\pi^{(i)})$ and set $i = i + 1$.
10:     Set $x = \mu^{(i-1)} - \bar{\mu}^{(i-2)}$.
11:     Set $y = \mu_E - \bar{\mu}^{(i-2)}$.
12:     Set $\bar{\mu}^{(i-1)} = \bar{\mu}^{(i-2)} + \frac{x^T y}{x^T x} x$ (This computes the orthogonal projection of $\mu_E$ onto the line through
    $\bar{\mu}^{(i-2)}$ and $\bar{\mu}^{(i-1)}$.)
13:     Set $w^{(i)} = \mu_E - \bar{\mu}^{(i-1)}$.
14:     Set $t^{(i)} = \|\mu_E - \bar{\mu}^{(i-1)}\|_2$.
15: end while
```

The algorithm takes in as input a MDP\R, a feature mapping ϕ, and the expert's feature expectations μ_E. The feature mapping ϕ maps a given state to a feature vector, which tells us all the features that are present. The expert's feature expectations is estimated from the set of expert trajectories using Equation 2.24. This is used to compare the feature expectations of other policies. The algorithm then returns a set of policies $\{\pi^{(i)} : i = 0...n\}$ where at least one of the policies induce feature expectations that are close to that of the expert's. A policy $\tilde{\pi}$ such that $\|\mu(\tilde{\pi}) - \mu_E\|_2 \leq \epsilon$, where ϵ is an arbitrarily small positive number.

At the start of the algorithm, a random policy is being generated and added to the list of policies. The feature expectations of the random policy $\pi^{(0)}$ is then computed and added to the list of feature expectations, $\mu^{(0)} = \mu(\pi^{(0)})$. The feature expectations $\mu^{(0)}$, which is the sum of the discounted features when acting according to that policy, can be computed directly using Equation 2.22 if the dynamics of the environment are known. However, if these dynamics are unknown, then the feature expectations needs to be estimated. The random policy $\pi^{(0)}$ is used to generate a set of trajectories so that the feature expectations can be estimated by averaging the discounted features of those trajectories. If m trajectories were generated by the random policy, the feature expectations can be estimated using Equation 2.24, just like how the expert's feature expectations was estimated.

In the next step of the algorithm, the weight vector is computed by taking the component-wise difference between the feature expectations of the expert μ_E and the random policy $\mu^{(0)}$. The weight vector determines the reward function and will be used in the reinforcement learning step. The expected feature expectations of the first policy $\bar{\mu}^{(0)}$ is set to the estimated feature expectations of the random policy $\mu^{(0)}$

and will be used in the next iteration of the algorithm. Followed by this step is the calculation of t, which is the 2-norm of the weight vector. This tells us the distance between the expert's and the random policy's feature expectations which in turn determines the stopping criteria of the algorithm. If this value is less than ϵ, this implies that the policy induces behaviour that is similar to that of the expert's and the algorithm terminates.

If the feature expectations induced by the random policy is not close to that of the expert's, the algorithm will then repeat the code from line 7 to line 15. Line 8 can be viewed as the reinforcement learning step where it uses the weight vector and applies a reinforcement learning algorithm to find the optimal policy. Line 9 to 14 can be viewed as the inverse reinforcement learning step in which the algorithm tries to guess or learn the reward function. This is done by repeatedly computing the orthogonal projection of μ_E onto the line through $\bar{\mu}^{(i-2)}$ and $\bar{\mu}^{(i-1)}$, which brings the feature expectations of the policy closer to the expert's after each iteration. The inverse reinforcement learning step does not necessarily recover the reward function, its aim is to find a policy that matches the expert's feature expectations. The process of repeatedly applying the reinforcement learning step and the inverse reinforcement step stops when a weight vector is found such that the expert only performs better than the returned policy by margin of ϵ, under the reward function given in Equation 2.19. It has been shown that the algorithm does terminate [Abbeel and Ng 2004].

Apprenticeship learning via inverse reinforcement learning is useful in applications where it is difficult to manually specify the reward function. The algorithm will keep guessing the reward function until it finds a policy that induces a behaviour that is close to the expert.

2.8 Parrot Ar.Drone

The Ar.Drone, as shown in Figure 2.5, is a helicopter assembled by a French company called *Parrot* in 2010 [Parrot 2012]. The drone can be remotely controlled through a user-friendly graphical interface running on Android or iOS devices. The Ar.Drone has several onboard sensors:

- 2 Cameras (vertical and horizontal)
- Ultrasound altimeter
- 3 axis accelerometer
- 2 axis gyrometer
- 1 yaw precision gyrometer

Figure 2.6 shows the coordinate system of the drone. The roll-pitch-yaw convention is very common in aerial navigation where roll is the rotation around the x axis (up and down movement of the wing tips of the aircraft); pitch is the rotation around the y axis (changes the vertical direction the head is pointing); and yaw is the rotation around the z axis (a movement of the head of the aircraft from side to side). The onboard sensors as well as the control technology embedded inside the drone give the drone the ability to hover on a spot and await for the next command; this makes flying a quadrotor easier.

Due to the onboard stabilisation, the AR.Drone is becoming very popular both as an augmented reality gaming, and as a platform for robotics research [Krajník *et al.* 2011]. The AR.Drone has been used as a research platform in [Krajník *et al.* 2011; Higuchi *et al.* 2011; Bills *et al.* 2011; Ng and Sharlin 2011].

The AR.Drone 2.0, the successor to the original, was released onto the market in 2012. The onboard sensors were made more sensitive and the camera resolution has increased. Technical specifications of the drones can be found in [Parrot 2012; Anderson 2010]. The reader should refer to [Bristeau *et al.* 2011] for a detailed information of the navigation and control technology inside the drone.

Figure 2.5: The AR-Drone quadcopter [Krajník *et al.* 2011].

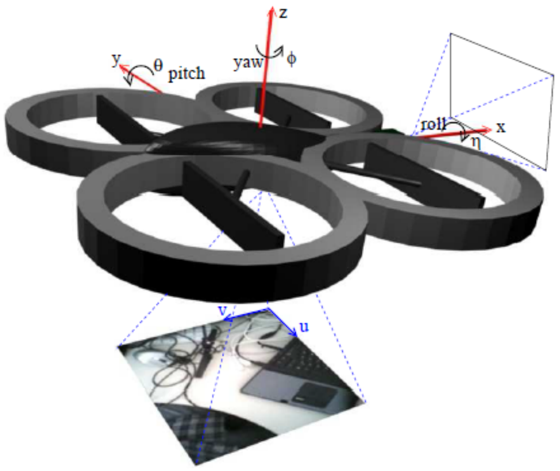


Figure 2.6: Coordinate system of the drone [Krajník *et al.* 2011].

2.9 Robot Operating System and Reinforcement Learning Library

This section discusses the software and libraries used during the course of the research. In specific, the robot framework, simulator, and the reinforcement learning library.

2.9.1 ROS

Robot Operating System (ROS) is an open-source, meta-operating system that provides a structured communications layer above the host operating systems of a heterogenous compute cluster [Quigley *et al.* 2009]. ROS is not an actual operating system or programming environment; instead, it provides services similar to that of an operating system. Such services include hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between processes, and package management. In addition to these services, ROS also provides tools and libraries for obtaining, building, writing, debugging, and running code across multiple computers [Quigley *et al.* 2009].

ROS can help to resolve some of the issues in the development of robotic software. These issues include distributed computation, software re-use, and rapid testing [O'Kane 2013].

Distributed Computation

ROS provides a mechanism that allows communication between multiple processes that may or may not reside on the same computer. This is extremely useful as it allows the robot's software to be broken down into small, standalone parts which communicate and cooperate to achieve the overall goal.

Software Reuse

ROSs standard packages provide stable, debugged implementations of many important robotics algorithms [O'Kane 2013]. Moreover, since ROS is an open-source software, the ROS website contains dozens of repositories of publicly-available ROS packages with state-of-the-art algorithms that can be shared and used in other robots without much effort. This allows robotic developers to focus more on experimenting new ideas and creating new functionalities rather than reinventing the wheels.

Rapid Testing

One of the main issues when developing robotic software is that testing can be time consuming and error-prone. Additionally, the process of working with physical robots can sometimes be slow and over-particular; whilst at other times such robots may just not be available to work with. [O'Kane 2013]. Moreover, if testing is not guaranteed to be safe, we usually would not want to take the risk of crashing the robot. Effective workarounds to this problem is available with the use of ROS. The direct low-level hardware control and high-level processing and decision making are separated by the ROS systems [O'Kane 2013]. This allows a simulator to be used to replace the low-level programs, and the high-level part of the system to test the behaviour on the simulator. In addition to this, ROS provides the recording and playing back of sensor data as well as other kinds of messages. This tool provides an effective way to process and analyse data.

2.9.2 Tum Simulator

Tum simulator is a ROS package that contains the implementation of a gazebo simulator (a tool used in ROS for robot simulation) for the Parrot Ar.Drone. This can be used to simulate both the Ar.Done 1.0 and 2.0. However, not the entire Ar.Drone sensors are 100% modeled in the simulation. The navigation data includes a lot of information of which some are not implemented in the simulation. The following are implemented:

- Message time stamp
- Message frame id
- Fly mode
- Battery percentage
- Position
- Rotation
- Velocity
- Acceleration

while the magnet, pressure, wind and tags information have been excluded.

2.9.3 RL-Glue

RL-Glue is a standard, language-independent software package that supports the development and testing of reinforcement learning algorithms [Tanner and White 2009]. RL-Glue implements a standardised reinforcement learning interface, facilitating code sharing and collaboration, which reduces the need for reinforcement learning practitioners to create their own agents and environments using incompatible software frameworks. This makes collaboration convenient and helps accelerate the pace of research in the field of reinforcement learning.

Protocol

The RL-Glue protocol is defined by four separate programs as shown in Figure 2.7: the agent, the environment, the experiment and the RL-Glue program. The agent program implements the learning algorithm: the agent is the decision maker that decides which action to take at every time step. The environment program determines the transition dynamics of the task and provides observations and rewards to the agent. The experiment program directs the experiment's execution; it states how many times to run the agent in the environment as well as evaluating the performance of the agent. The RL-Glue connects the above programs: The agent can not interact directly with the environment, the same applies to the experiment program. All contacts go through the RL-Glue interface.

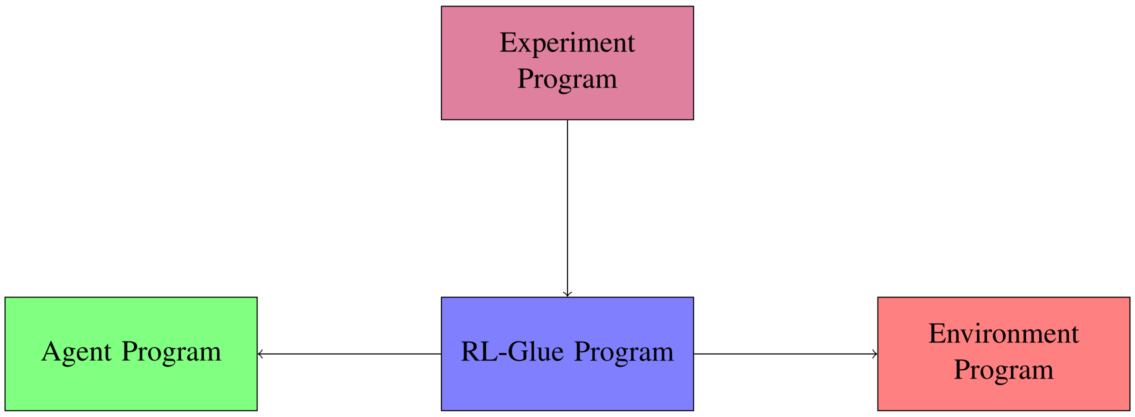


Figure 2.7: The four programs specified by the RL-Glue protocol [Tanner and White 2009].

2.9.4 Rosglue

Rosglue is a framework, developed at Brown University by the Brown Robotics Lab [Brown University], that allows robots running ROS to be environments for RL-Glue agents. It is designed to be a bridge between RL-Glue and ROS. More specifically, a simulator in ROS can be used to replace the environment program in Figure 2.7. The agent program decides the action that the robot should take, and the robot will perform that action on the environment (simulator). The simulator, which determines the dynamics of the environment, then provides observations and rewards after the action is taken. In this research, the tum simulator was used which replaces the environment program in Figure 2.7. The agent program (implementing SARSA(λ) algorithm) can then interact with the tum simulator.

2.10 Conclusion

In this chapter, the concept of Learning from Demonstration was discussed. The three core approaches used to solve the Learning from Demonstration problem include the mapping function approach, the system model approach, and the planning approach. Ways of gathering demonstration data are discussed as well, approaches include teleoperation, shadowing, sensors on teacher, and external observation. Reinforcement learning algorithms and function approximation techniques are presented. In particular, the apprenticeship learning via inverse reinforcement learning algorithm, which uses expert trajectories to learn the reward function of a given task. This is accomplished by identifying a policy that promotes feature expectations similar to that of the expert's. As mentioned earlier, this description is a gross simplification. The interested reader should refer to [Argall *et al.* 2009] for a more detailed discussion in this domain. The chapter ends by discussing the Parrot AR.Drone as well as the software and libraries used in this research. The proposed research methods are presented in the next chapter.

Chapter 3

Research Methodology

3.1 Introduction

In the previous chapter, the background relating to learning from demonstration, reinforcement learning, the Parrot Ar.Drone as well as the software and libraries used during the course of the research has been discussed. In this Chapter, the proposed research methodology for creating a flight controller for the Ar.Drone using expert's demonstrations is provided.

3.2 Aim

The aim of this research was to determine whether machine learning can be used to achieve LfD for the Ar.Drone. Reinforcement learning is a subfield of machine learning that uses the concept of trial-and-error to make the agent learn. A reward function is required by the algorithm for the agent to learn the correct behaviour while interacting with the environment. However, such a reward function is often very challenging to manually specify, especially in applications dealing with robotics control. It is often more straightforward to demonstrate the desired task than to specify the reward function, as a result, we propose using apprenticeship learning via inverse reinforcement learning to learn the reward function from expert trajectories.

Since reinforcement learning algorithms require the state space to be discrete, function approximators were used to estimate the value functions. The research aim was to see whether apprenticeship learning via inverse reinforcement learning can be applied on the Parrot Ar.Drone so that the agent, given expert's trajectories, was able to mimic the expert's behaviour. The expert's behaviour corresponds to the author's behaviour, and the expert's trajectories were created by the author.

3.3 Research Questions

The research seeks to ascertain whether reinforcement learning methods can be applied on the Ar.Drone for autonomous navigation. The scope of the research can thus be described by formulating the following research questions:

1. Can Sarsa be used with the function approximators to approximate the value functions? If so, how do the function approximators compare in performance?

2. Can radial basis functions be used to estimate the expert's feature expectations? That is, can apprenticeship learning via inverse reinforcement learning recover the underlying reward function where the reward features are constructed using radial basis functions?

3.4 Research Methodology

This research has been broken down into two stages which correspond to two different types of learning agents being implemented: the first type used a hard-coded reward function and the second type used RBF features. The desired task was to get the drone to fly from one point to another in the simulator. The resulting task has 7 continuous state variables (x, y, z position of the drone as well as the x, y, z, w orientation of the drone) and 6 continuous action variables (linear and angular velocities). To simplify the task, we decided not to include the current linear and angular velocities in the state variables and to discretise the action space into 9 finite actions.

3.4.1 Hard-Coded Reward Function

The following methods were used with a hard-coded reward function:

1. Sarsa(λ) with radial basis functions to approximate the value functions.
2. Sarsa(λ) with fourier basis functions to approximate the value functions.

Reward Function

The reward function awards high negative values for the terminal states, and smaller negative values for non-terminal states. Positive values are rewarded for the goal state.

Number of Basis Functions

We employed Sarsa(λ) with RBF bases of 2048 basis functions, 4096 basis functions, 8192 basis functions, 16384 basis functions, and Fourier bases of order 3 (16384 basis functions).

Parameter Selection

Parameters used for the experiments were: γ = 0.99, ϵ = 0.1, λ = 0.7, and α = 1.0 (Alpha-Bound). The learning algorithm was given 500 episodes and 500 steps for decision making. In order to prevent non-stationary policies, we have discretised the state space so that 500 steps were enough to reach the terminal states.

Experimental Setup

The problem with the simulator was that the drone would drift over time, so we decided to conduct the experiments using 6 actions (excluding rotation) and 8 actions (including rotation). This is to test whether the drone would be able to reach the goal state without re-orienting itself and how the number of actions available affects the performance of the drone.

Performance

The performance of the agents was evaluated in terms of the run-times of the algorithm, average reward received and the average steps taken to reach the goal state.

3.4.2 Weighted Features as Reward Function

Apprenticeship learning via inverse reinforcement learning requires expert's trajectories, a reinforcement learning algorithm and two feature mappings. One feature mapping is used to estimate the expert's feature expectation and the other one is used to estimate the value functions. The expert's trajectories are used to generate reward function for the learning agent so that a policy could be derived. From this new policy, new trajectories could be generated which leads to a new reward function. This process continues until the derived policy generates similar trajectories to that of the expert's. The following method was used to recover the expert's reward function:

1. Sarsa(λ) with 2048 radial basis functions to approximate the value functions and 100 radial basis functions to approximate the reward function.

Expert Trajectories

The author teleoperated the Ar.Drone in the simulator and the trajectories were recorded. 50 set of trajectories were used in the experiment.

Reward Function

100 radial basis function were used to represent the reward function $R(s) = w \cdot \phi(s)$, where $w \in \mathbb{R}^{100}$ and $\phi : S \rightarrow [0, 1]^{100}$. The vector w thus specifies the relative weighting between the features computed by the 100 radial basis functions.

Expert's Feature Expectations

The expert's feature expectations were estimated from the expert's trajectories using a feature mapping ϕ_i. In this case, ϕ_i was the feature mapping used in the reward function (100 radial basis functions). The expert's estimated features were compared to the estimated features of other policies until a policy is found that induces features close to that of the expert's.

Reinforcement Learning Step

Given a reward function, a reinforcement learning algorithm used the second feature mapping ϕ_j (2048 radial basis functions) to approximate the value functions. The returned policy was the optimal policy with respect to the reward function.

Parameter Selection

Parameters used were the same as previous experiments but with additional parameter ϵ_2 = 0.5, which was the terminating condition used in the apprenticeship learning via inverse reinforcement learning algorithm. Furthermore, the feature expectations were calculated using the same discount rate as the Sarsa update rule.

Experimental Setup

The experiment was conducted using 6 actions (excluding rotation). This is to test whether the drone could match the expert's feature expectation without reorienting itself.

Performance

Since the expert's reward function was unknown, we could not compare the average rewards. We thus compared the trajectories generated by the found policy to that of the expert's, and checked whether they shared any similarities.

3.5 Conclusion

In this chapter, the proposed methodology that was carried out to complete the research was discussed. This involved creating two different kinds of learning agents accomplishing similar task, one that used a hard-coded reward function and the other that used RBF features. The results of the experiments as well as further implementation details of the learning agents are presented in the next two chapters.

Chapter 4

Learning Agent I

4.1 Introduction

In the previous chapter, the scope and methods of the proposed research have been discussed. In this chapter, a learning agent that was created using reinforcement learning with hard-coded reward function is discussed. The task of the learning agent was to fly from one point to another in the simulator. The implementation of the learning agent is presented below followed by a discussion on the results of the experiments.

4.2 Implementation

In this section, we provide the implementation details of the learning agent. This includes how the MDP is constructed and the methods chosen to approximate the value function.

4.2.1 Markov Decision Process

We formulate the task of flying from one point to another as an MDP where reinforcement learning techniques could be applied to solve the problem. The environment in which the agent interacted with is shown in Figure 4.1, where the stop sign represents the goal state and the yellow lines represent the boundaries.

State Space

The state space S consists of the position and orientation of the quadrotor. The position is represented by the x, y, z coordinates of the drone and the orientation is represented as a quaternion x, y, z, w. These 7 variables all contain continuous values, thus the state space is said to be continuous. A tuple containing the 7 variables represents the state of the quadrotor at any given time step. Since the task was to fly from one point to another, it does not depend on the current velocities of the quadrotor. As a result, the linear and angular velocities were not included in the state space.

Action Space

The action space A consists of the linear and angular velocities of the quadrotor. Each of the x, y and z parameters of the linear and angular velocities can take on a real number r, where $r \in [0, 1]$. Since all the action variables are continuous, the action space in this case is also continuous. In order to simplify the learning process, the action space is discretised into 9 finite actions. The actions include the following: fly forward, fly backward, fly to the left, fly to the right, fly up, fly down, rotate counterclockwise, rotate clockwise, and stop. Table 4.1 shows the actions and their corresponding linear and angular velocities. E.g. Action number 0 will generate its corresponding linear and angular velocities that makes the quadrotor fly forward.

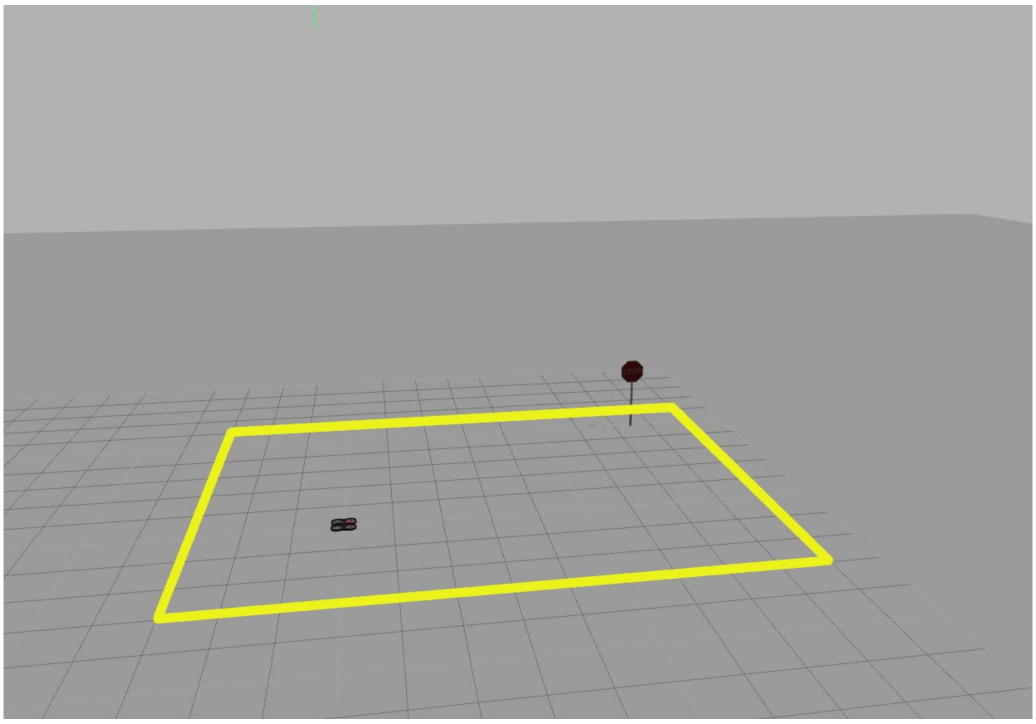


Figure 4.1: ROS tum simulator environment

Action	**Action Number**	**Linear Velocities**	**Angular Velocities**
Fly forward	0	x: 1.0, y: 0.0, z: 0.0	x: 0.0, y: 0.0, z: 0.0
Fly backward	1	x: -1.0, y: 0.0, z: 0.0	x: 0.0, y: 0.0, z: 0.0
Fly to the left	2	x: 0.0, y: 1.0, z: 0.0	x: 0.0, y: 0.0, z: 0.0
Fly to the right	3	x: 0.0, y: -1.0, z: 0.0	x: 0.0, y: 0.0, z: 0.0
Fly up	4	x: 0.0, y: 0.0, z: 1.0	x: 0.0, y: 0.0, z: 0.0
Fly down	5	x: 0.0, y: 0.0, z: -1.0	x: 0.0, y: 0.0, z: 0.0
Rotate counterclockwise	6	x: 0.0, y: 0.0, z: 0.0	x: 0.0, y: 0.0,z: 1.0
Rotate clockwise	7	x: 0.0, y: 0.0, z: 0.0	x: 0.0, y: 0.0,z: -1.0
Stop	8	x: 0.0, y: 0.0, z: 0.0	x: 0.0, y: 0.0, z: 0.0

Table 4.1: Types of actions and their respective linear and angular velocities

In order to ensure that each action executed had its intended effect, a stop action was executed for every other action taken. Thus the number of actions available to the learning agent decreased to 8. Two versions of the learning agent were created: one which used 6 actions (excluding rotation) and the other which used all the actions (including rotation).

State Transition Probabilities

$\{P_{sa}\}$ is the set of state transition probabilities (state transition distribution upon taking action a in state s) and is determined by interacting with the simulator. If the learning agent takes an action a in state s (position p and orientation o), the resulting state s' (position p' and orientation o') is provided by the simulator.

Discount Factor

$\gamma \in [0, 1)$ is the discount factor and is specified in the reinforcement learning algorithm. Since this is an episodic task, the value 0.99 was used.

Initial-State Distribution

D is the initial-state distribution, from which the start state s_0 is drawn. In this case, it was the starting position and orientation of the quadrotor in the simulator. The starting position of the quadrotor before takeoff by default is at the center of the environment as shown in Figure 4.1, where the x, y, and z coordinates are all 0. The start state s_0 is thus the state of the quadrotor after takeoff.

Reward Function

The following reward function was used:

- A reward value of 10 was given to the goal state.
- A reward value of -1 was given to states within the boundaries (except the goal state).
- A reward value of -10 was given to states outside the boundaries.

As shown in Figure 4.1, the yellow lines represent the x, and y boundaries, where $x \in [-3, 9]$ and $y \in [-3, 6]$. $z \in [0, 4]$, which is not shown in the figure represents the height boundaries. If any of the x, y, or z coordinates fall outside its respective boundaries, the state is considered out of boundary and is rewarded with a value of -10. Similarly, if all the coordinates lie within their respective boundaries, the state is rewarded with a value of -1.

The stop sign represents the goal state and is situated at position $x = 7.44$ and $y = 4.88$. Since this is a continuous domain, being at the exact same state is not possible and so states that are very close to the stop sign are considered goal states. This closeness is determined by the euclidean distance to the stop sign. Thus, if a state has an euclidean distance e to the stop sign where $e \leq 1.41$, the state is considered a terminal state and is rewarded with a value of 10.

The reward values are chosen arbitrarily, but in essence, any values would work as long as the following conditions are satisfied: awarding high negative values for the terminal states, smaller negative values for the non-terminal states, and positive values for the goal state.

4.2.2 Value Function Approximator

Two methods with basis functions of differing sizes were used along with the reinforcement learning algorithm to approximate the value function. Specifically, we employed Sarsa(λ) with RBF bases of 2048 basis functions, 4096 basis functions, 8192 basis functions, 16384 basis functions, and Fourier bases of order 3 (16384 basis functions). The basis functions were generated using the variables in the state space. The values were chosen randomly where the range is determined by the boundaries stated above.

4.3 Results and Discussions

In this section, we present the results of the experiment which uses hard-coded reward function. We begin by providing the specifications of the machine that have been used to conduct the experiment in Section 4.3.1. The performance analysis of each of the implementations is presented in Section 4.3.2, followed by the discussion of the results in Section 4.3.3.

4.3.1 System Specifications

We have used Python to implement all of the code for this research. All experimentation has been conducted on an i7 machine using ROS Hydro and Gazebo simulator. The i7 machine has the following specifications:

- Intel Core i7-3770 CPU @ 3.40GHz

- 8GB 1333 MHz DDR3
- 4 CPU Cores
- 8192 KB Cache

4.3.2 Performance Analysis of each of the Implementations

In this experiment we compare the performances of each of the implementations where the quality of the policies are assessed with respect to their run times, average reward received as well as the average number of steps taken to reach the goal state. Furthermore, for each of the implementations, we run the experiment using actions of differing amounts. This gives us an indication as to whether the number of actions available to learning agent affects its performance. Specifically, the experiment is divided into three sections. Part (i) is based on comparing the performances of each of the implementations after 500 episodes where the parameters $(\gamma = 0.99, \epsilon = 0.1, \lambda = 0.7, \alpha = 1.0)$ were used to learn the optimal policies. Each episode begins with the drone placed at the origin $(x, y, z) = (0, 0, 0)$, which it then takes off, thus the initial state of the drone is the position of the drone after takeoff (which would be different each time since the state space is not discrete); an episode ends once the drone gets close to the goal state or if it goes out of boundary. For each implementations, we execute 100 iterations to measure its performance. In part (ii), we illustrate the most frequent visited states for each optimal policies in the form of heat maps. This gives us an indication as to whether the learning agent is doing the right task. We also show a single trajectory while executing those policies, allowing us to check whether the goal state has been visited. In part (iii), we calculate the percentage of trajectories generated by the optimal policies that are successful in guiding the learning agent to the goal state. This allows us to determine whether the value functions have converged and whether 500 episodes were enough to learn the desired task. The results for (i) - (iii) is presented hereafter.

(i) Performance comparison

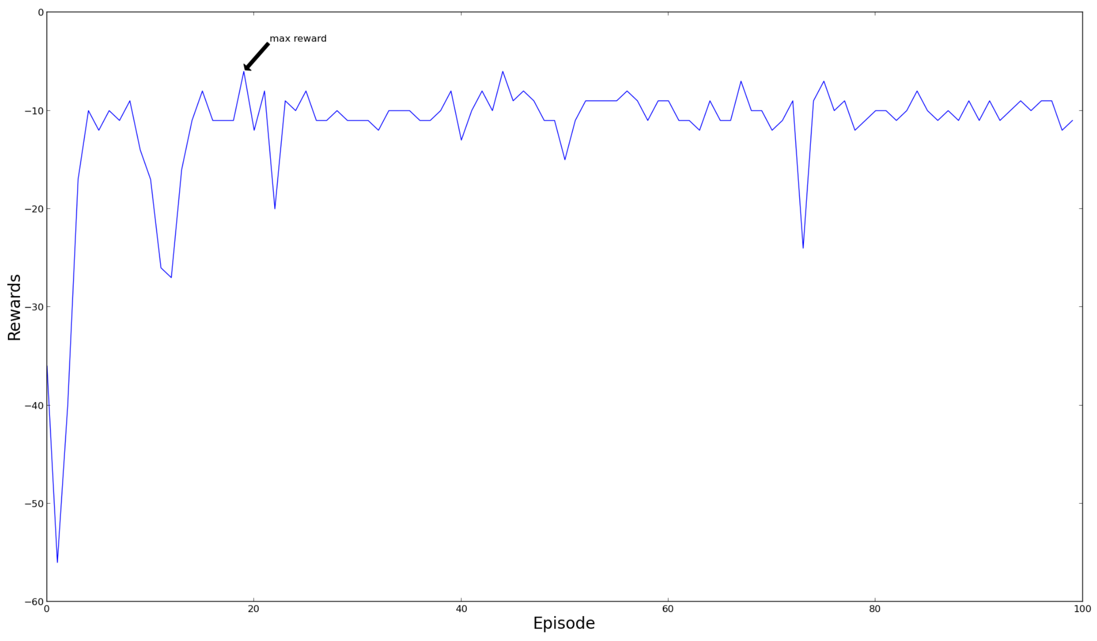
The running times of each of the implementations over 100 iterations are recorded as shown in Table 4.2. Figure 4.2 - 4.11 show the outcomes of each of the episodes in terms of the total rewards received and the number of steps the agent takes to reach the terminal state.

Type	**No. of basis functions**	**Number of actions**	**Time (seconds)**
RBF	2048	6	560.033689022
RBF	4096	6	773.334479094
RBF	8192	6	1176.70436096
RBF	16384	6	2503.86461687
Fourier	16384	6	637.762139082
RBF	2048	8	570.974934816
RBF	4096	8	751.202655077
RBF	8192	8	1225.10059404
RBF	16384	8	2614.50483012
Fourier	16384	8	583.557615995

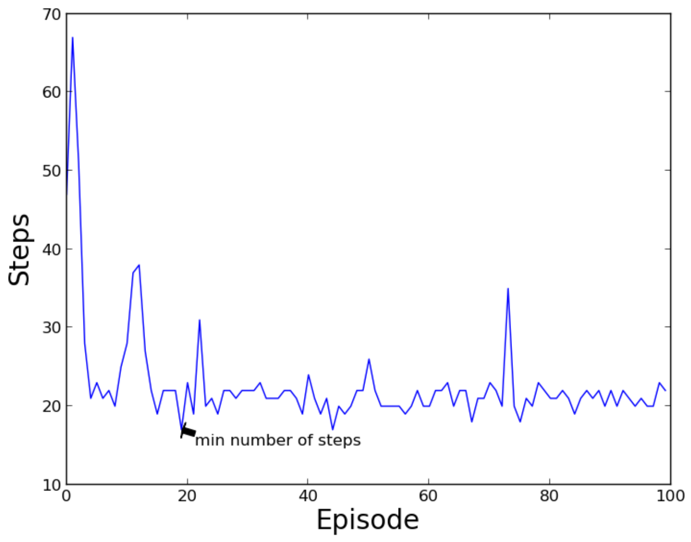
Table 4.2: Run times of each of the implementations.

The run time of each implementation increases as the number of basis functions used to approximate the value function increases. This is expected since the major computational step involves computing the features present for each visited state using the basis functions. If the number of basis functions increases, it would then take longer to compute those features. However, the additional two actions given to the agent did not have much effect on the run times of each implementation. Overall, the run times did increase, but this is due to the fact that more steps were taken by the agent during those 100 iterations.

In terms of run time, the implementation of 2048 RBFs with 6 actions has shown to outperform other implementations. Taking into account the number of basis functions, Fourier bases of order 3 has shown to outperform its RBF equivalent significantly, with a run time that is only slightly slower than 2048 RBFs.

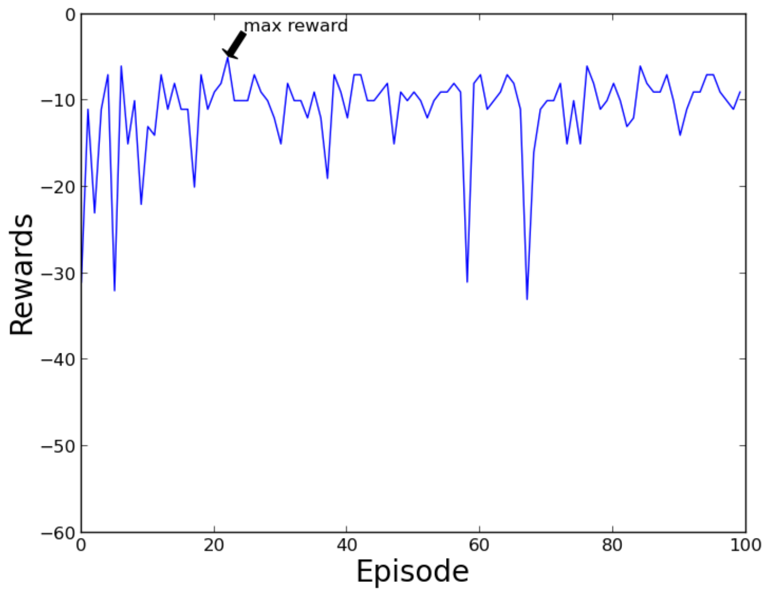


(a) Rewards

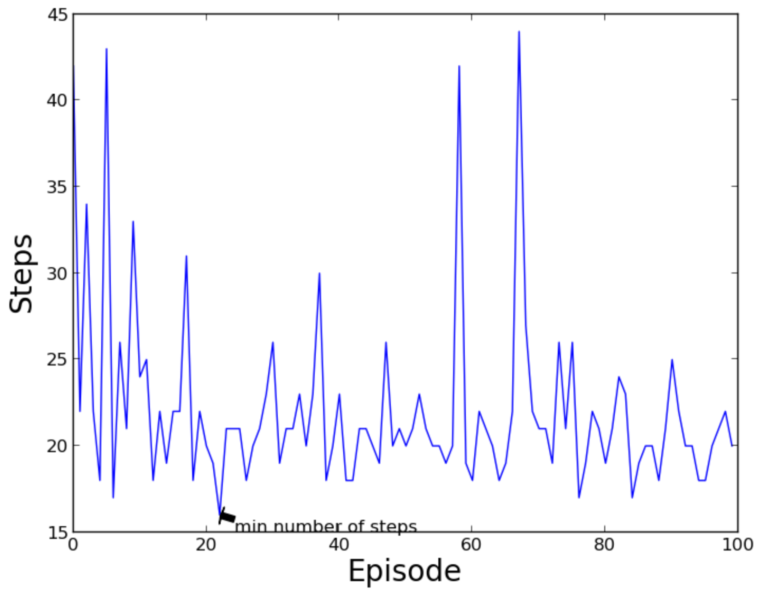


(b) Steps

Figure 4.2: 2048 RBFs with 6 actions

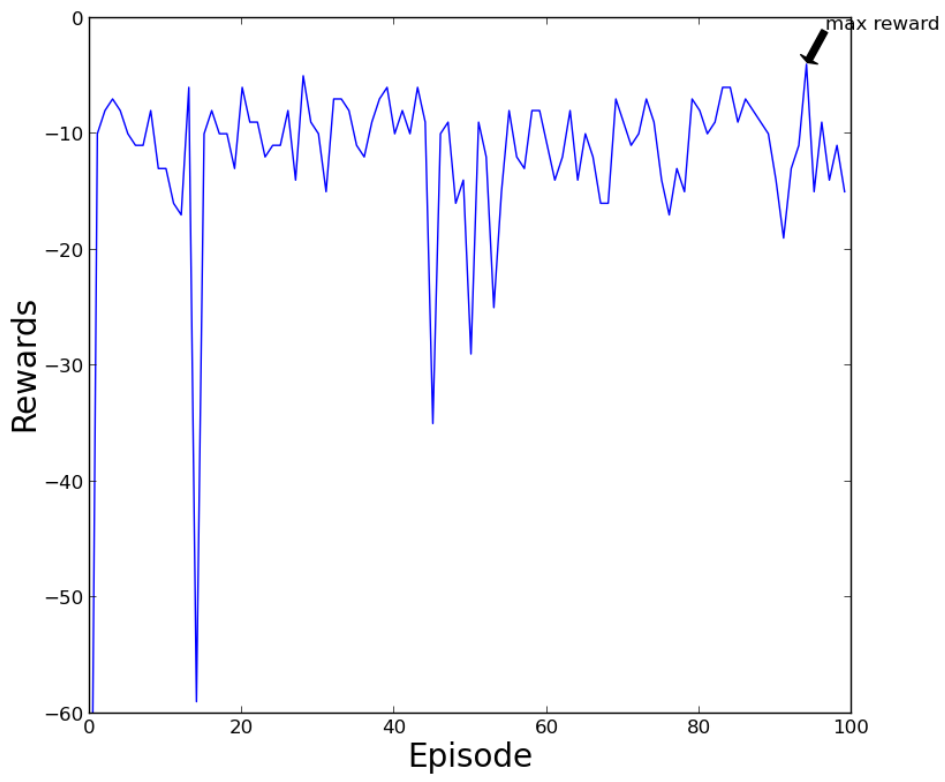


(a) Rewards

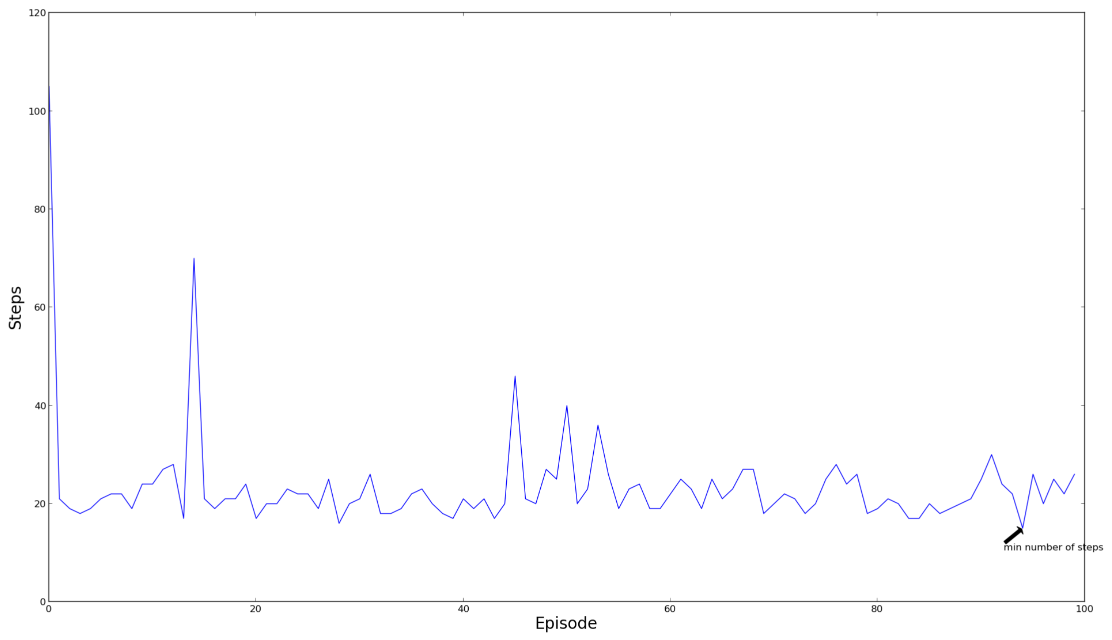


(b) Steps

Figure 4.3: 4096 RBFs with 6 actions

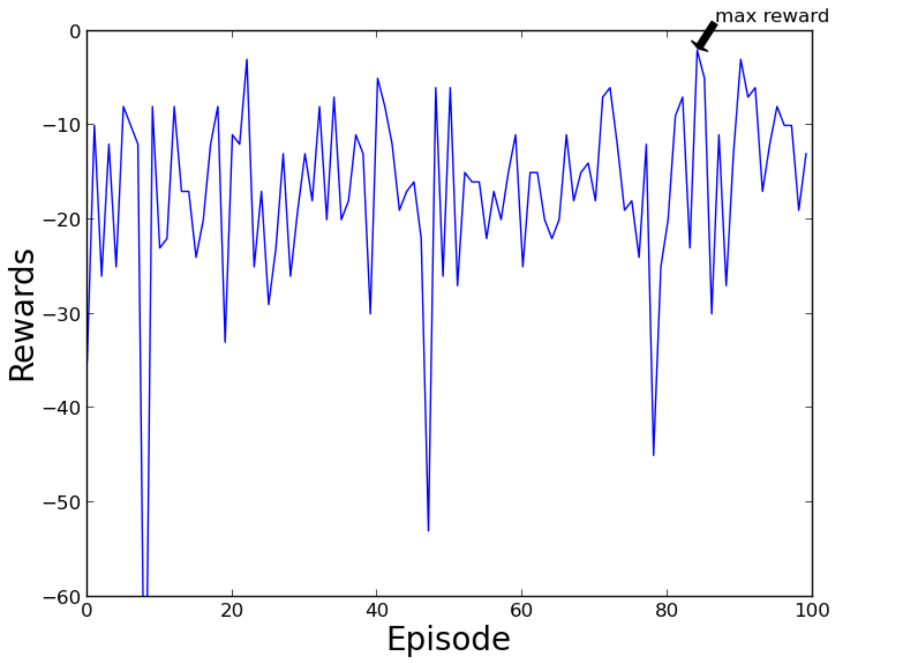


(a) Rewards

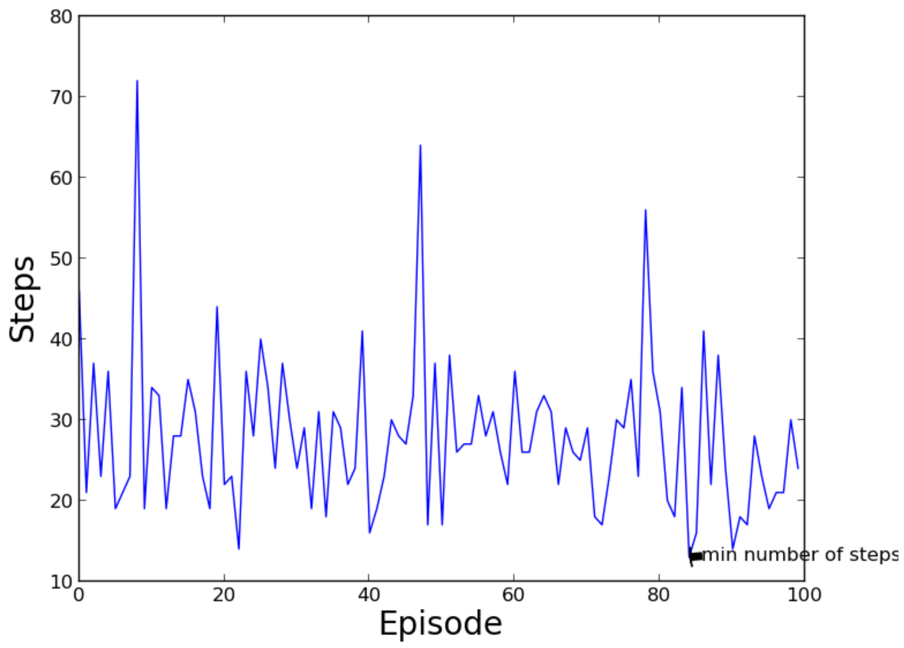


(b) Steps

Figure 4.4: 8192 RBFs with 6 actions

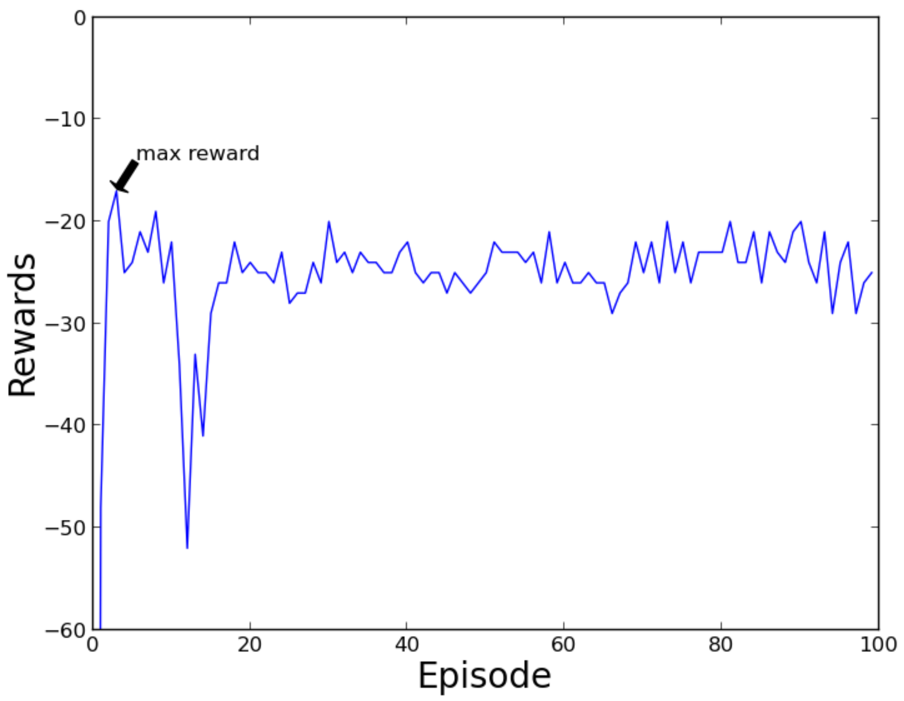


(a) Rewards

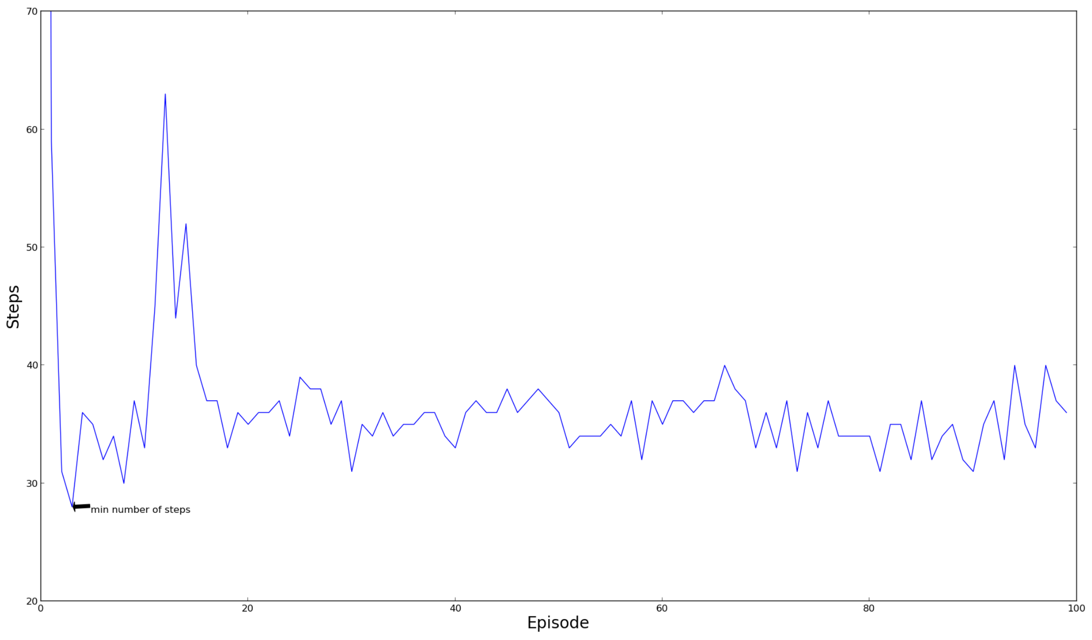


(b) Steps

Figure 4.5: 16384 RBFs with 6 actions

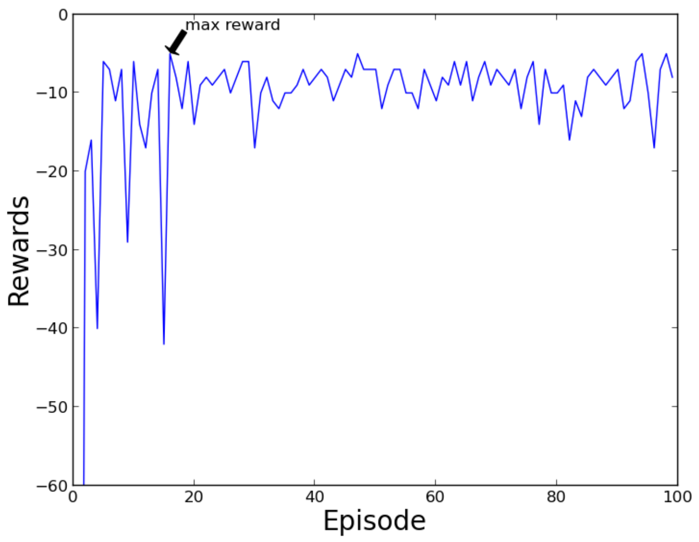


(a) Rewards

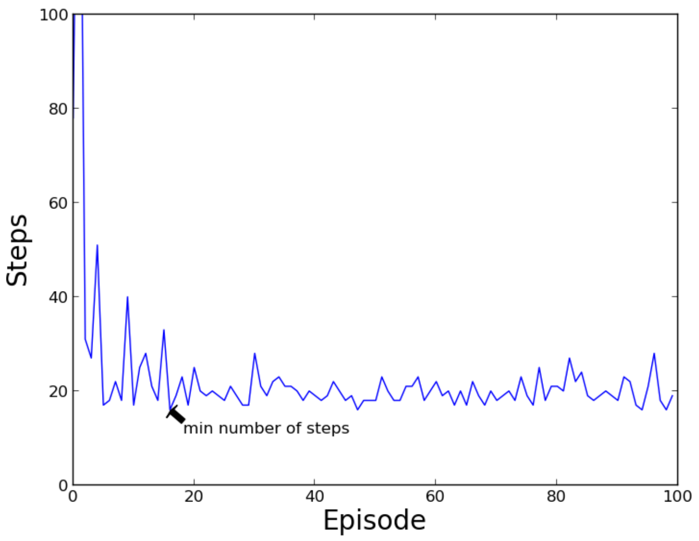


(b) Steps

Figure 4.6: Fourier bases order 3 with 6 actions

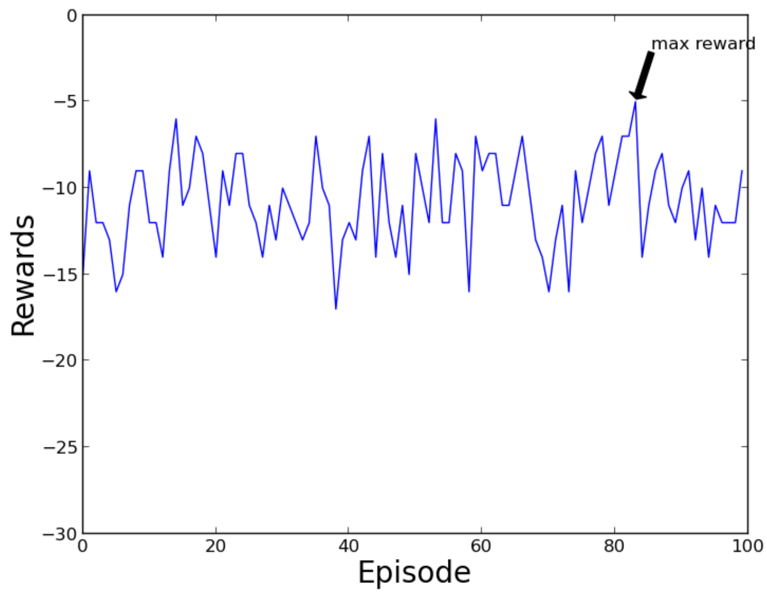


(a) Rewards

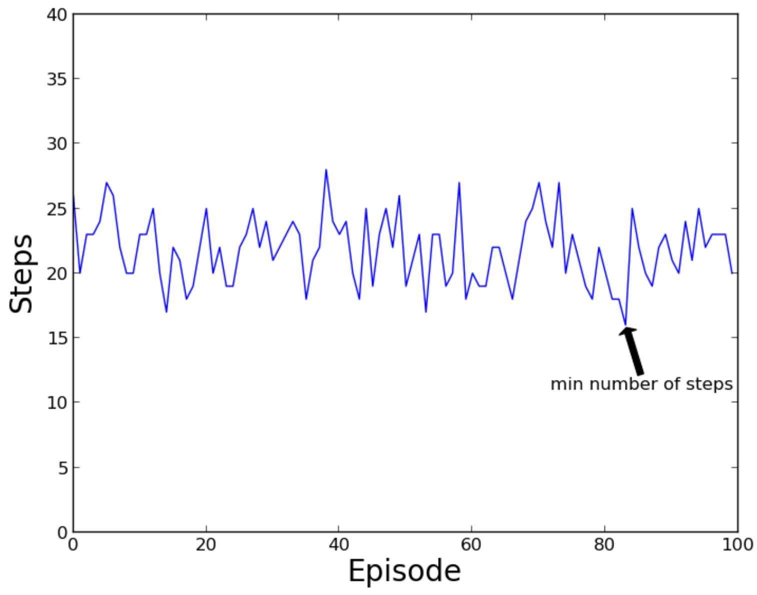


(b) Steps

Figure 4.7: 2048 RBFs with 8 actions

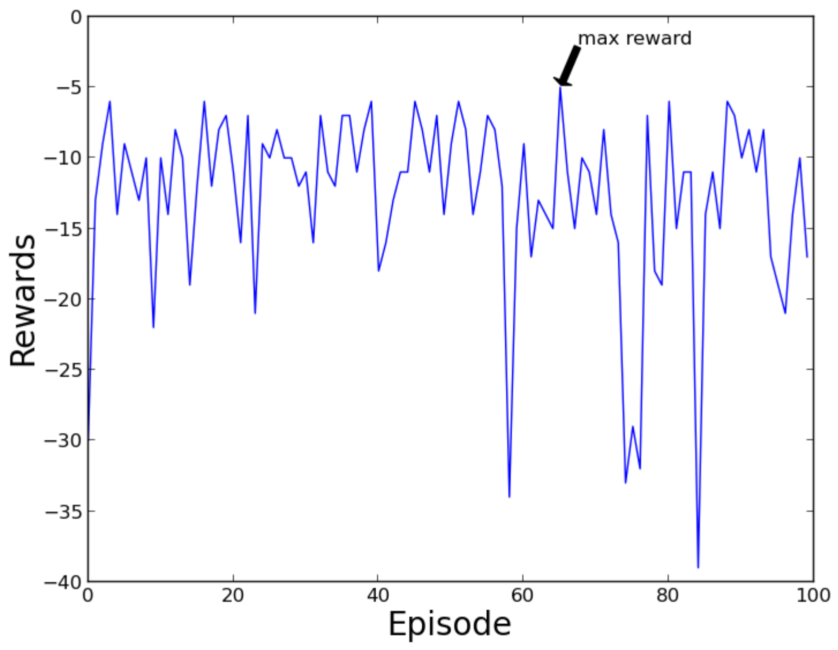


(a) Rewards

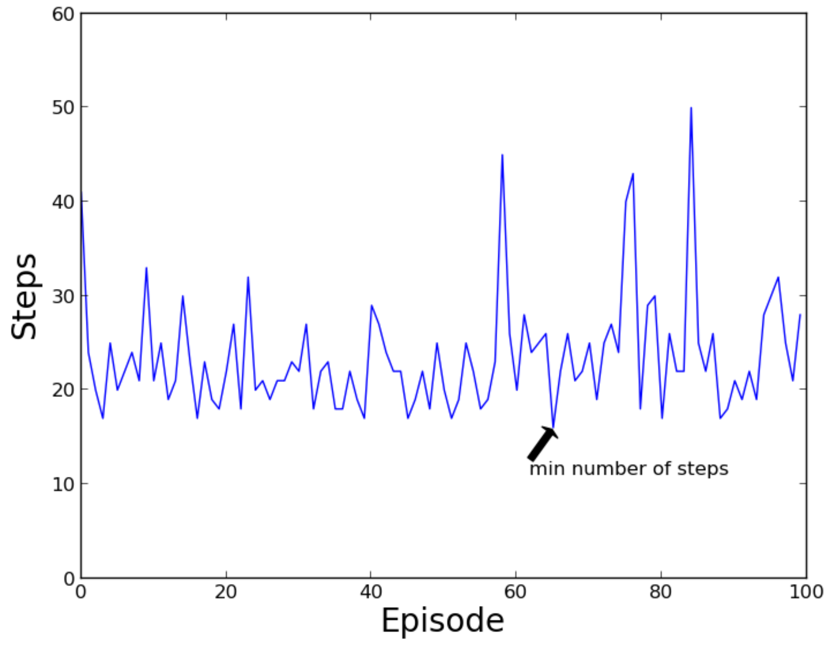


(b) Steps

Figure 4.8: 4096 RBFs with 8 actions

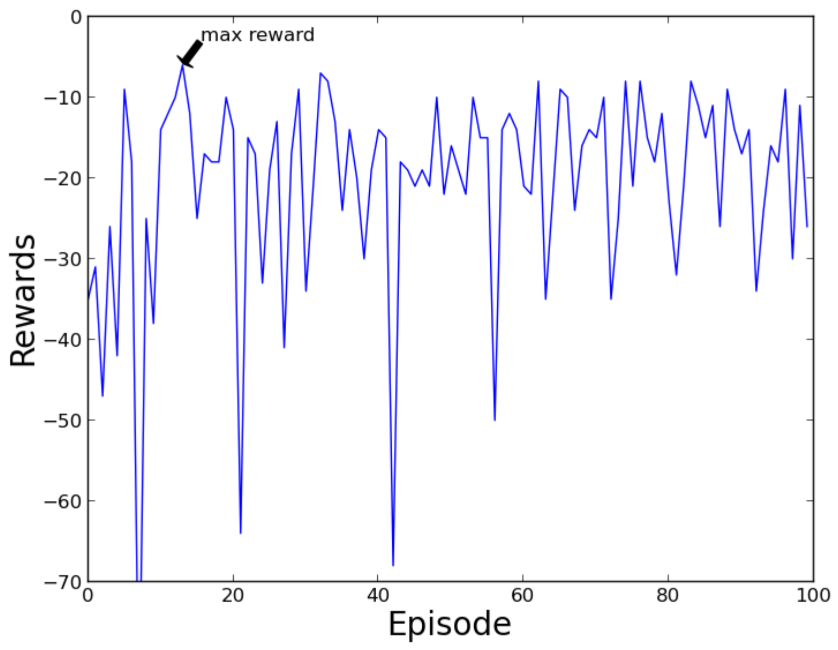


(a) Rewards

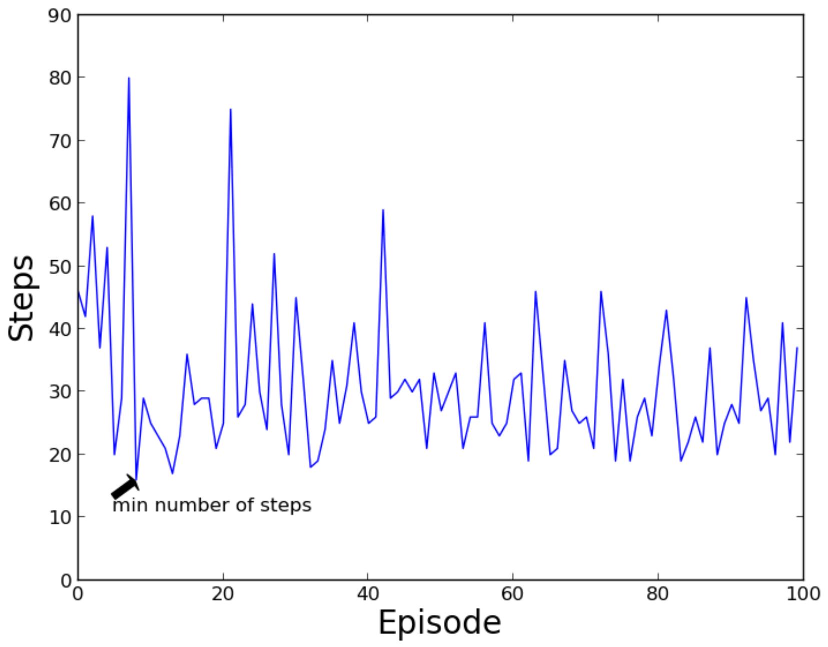


(b) Steps

Figure 4.9: 8192 RBFs with 8 actions

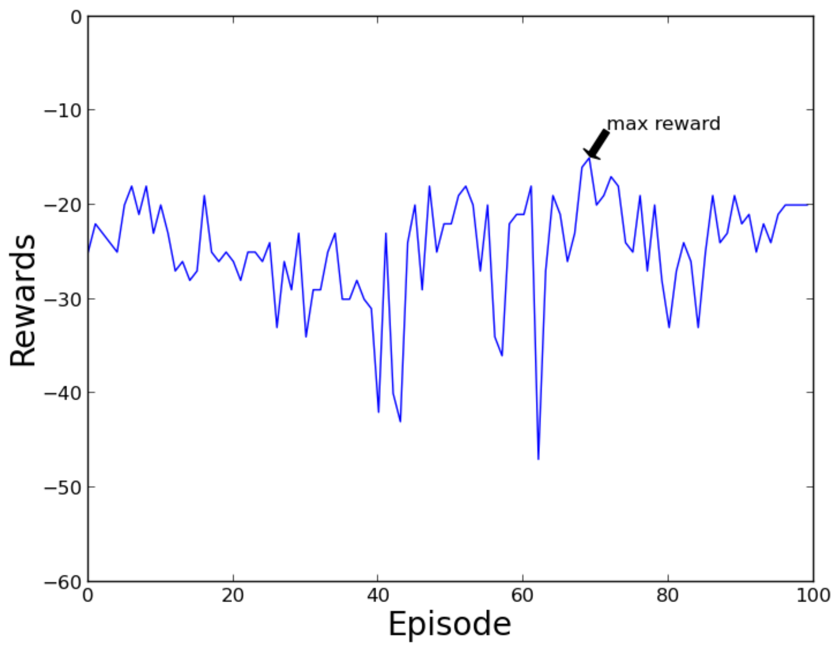


(a) Rewards

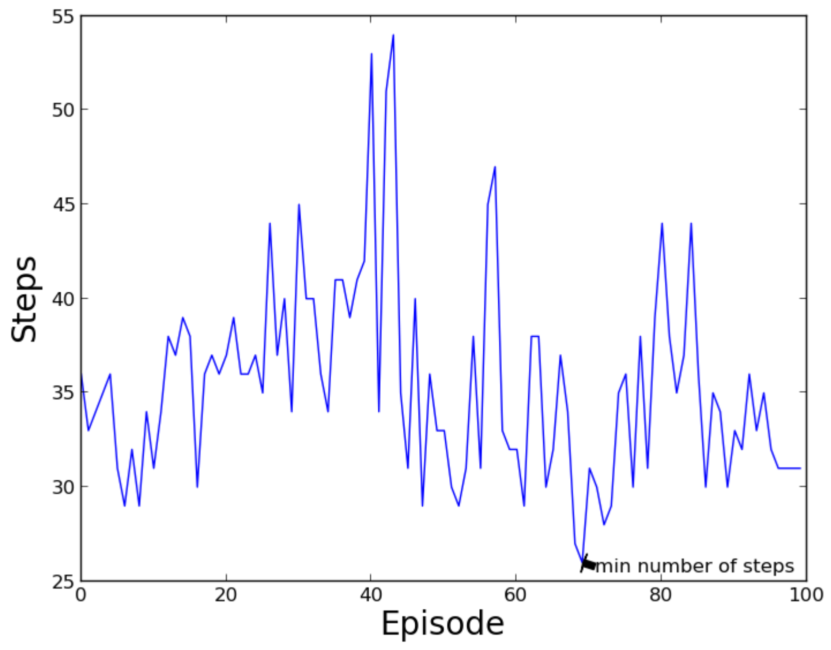


(b) Steps

Figure 4.10: 16384 RBFs with 8 actions



(a) Rewards



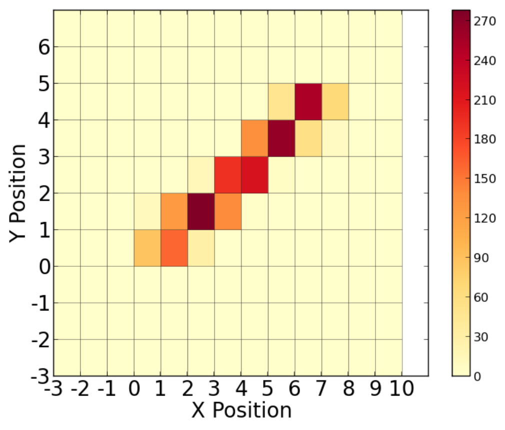
(b) Steps

Figure 4.11: Fourier bases order 3 with 8 actions

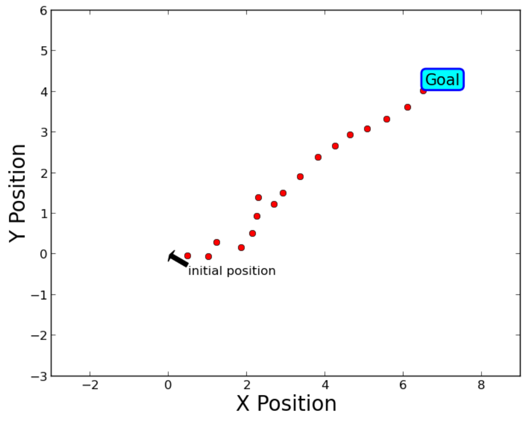
The maximum reward that the agent received and the minimum amount of steps it took to reach the terminal state are shown in Figure 4.2 - 4.11. The minimum amount of steps for each of the implementation ranges from 13-28 steps. Since rewards are all negative values and the states within the boundary are given the same reward, the episode with the minimum amount of steps also implies the maximum amount of reward. However, this is not always the case as will be shown in the next part of the experiment. The episode ends once it reaches the terminal state, and so the minimum amount of steps does not guarantee that the agent reached the goal state. Nonetheless, most of the implementations are able to achieve a minimum amount of steps of 15.

(ii) Optimal Policies discovered after 500 episodes

In this part of the experiment, we execute the optimal policies for 100 iterations and record all the visited states. Figure 4.12 - 4.21 give a representation of the optimal policy discovered in the form of heat map and a trajectory containing the minimum amount of steps from start to terminal state. The heat map shows how often the states are visited, which allows us to ascertain that the optimal policies generate successful trajectories. For all implementations, the goal state has been visited a lot of times which indicates that the optimal policies were able to guide the agent to the goal state. We observe that the path taken by those optimal policies are similar: they all go diagonally to reach the goal state. This is based on the fact the rewards given to states within the boundary are the same (negative value), and in order to maximise the rewards the shortest path needs to be taken. We see that the intensity of each state when using 16384 RBFs (Figure 4.15 and Figure 4.20) is less than that of every other implementation, moreover, the visited states are more spread out. A possible explanation for this would be that it takes very long to compute state features using 16384 RBFs and as a result the drone drifts. On the contrary, Fourier bases of order 3 computes the state features very quickly. This can be seen in Figure 4.16 and Figure 4.21 where the intensity of each state is approximately 3 times more than that of every other implementation. If we compare the trajectories generated by every optimal policy, we can see that the implementation that uses 16384 RBFs with 8 actions contains the least amount of steps to the terminal state. However, the number of steps does not necessarily guarantee the quality of the trajectory. As shown in Figure 4.20b, the trajectory contains the least amount of steps but it was unsuccessful as it did not guide the learning agent to the goal state. The number of steps only tells us that the agent has reached a terminal state, not necessarily a goal state.

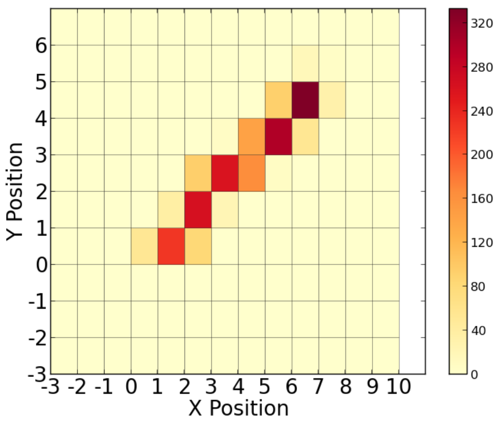


(a) Heat Map

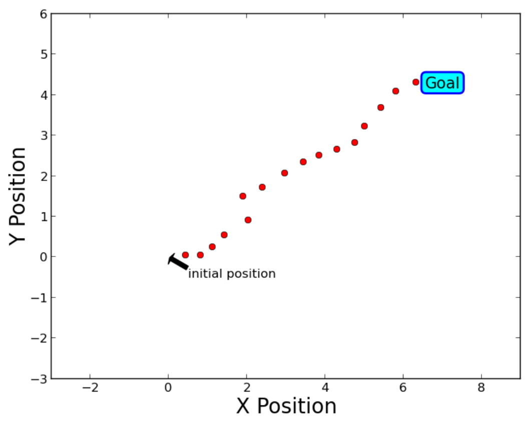


(b) Trajectory

Figure 4.12: 2048 RBFs with 6 actions

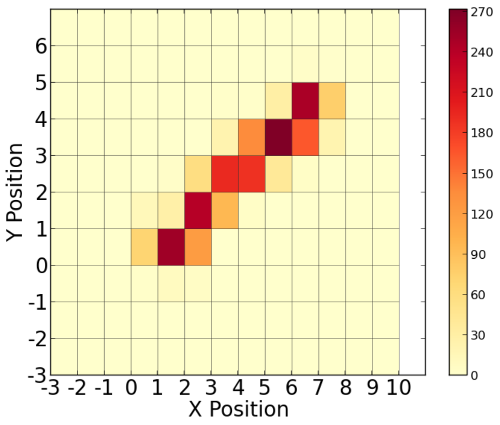


(a) Heat Map

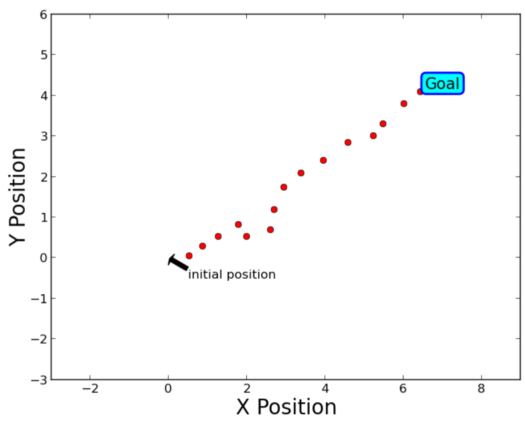


(b) Trajectory

Figure 4.13: 4096 RBFs with 6 actions

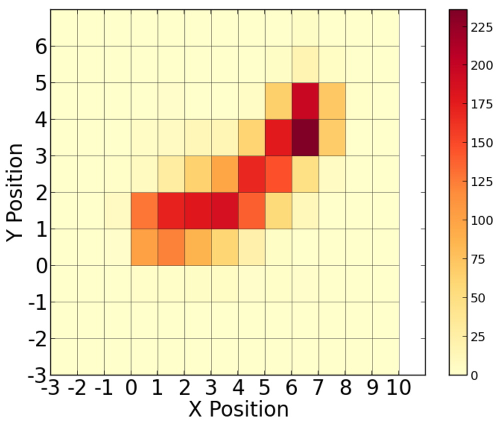


(a) Heat Map

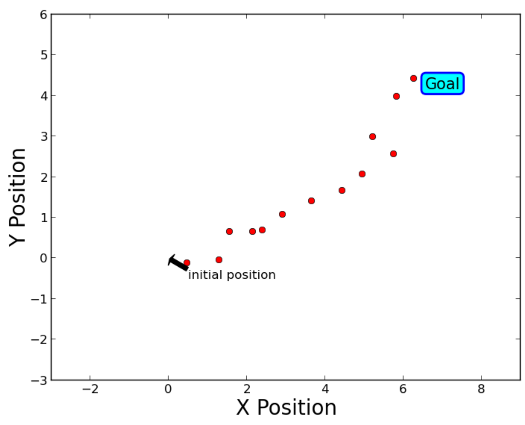


(b) Trajectory

Figure 4.14: 8192 RBFs with 6 actions

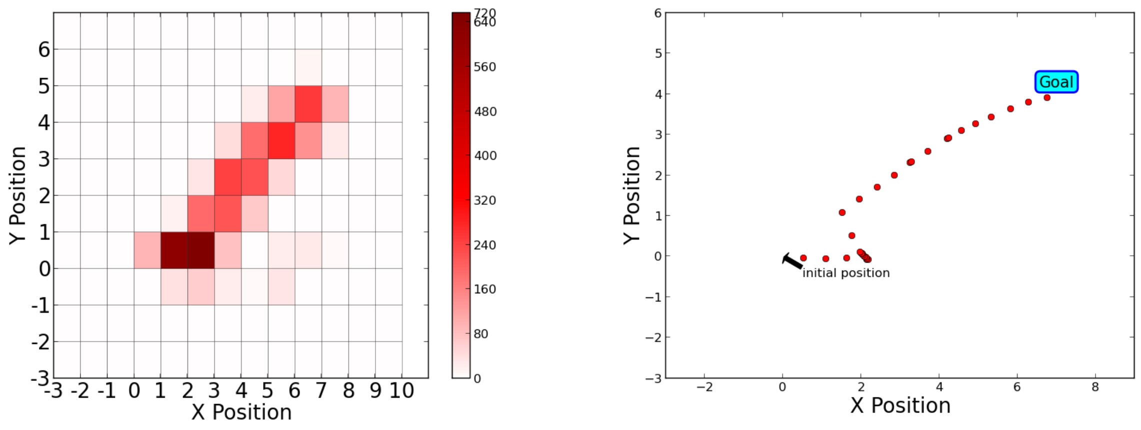


(a) Heat Map



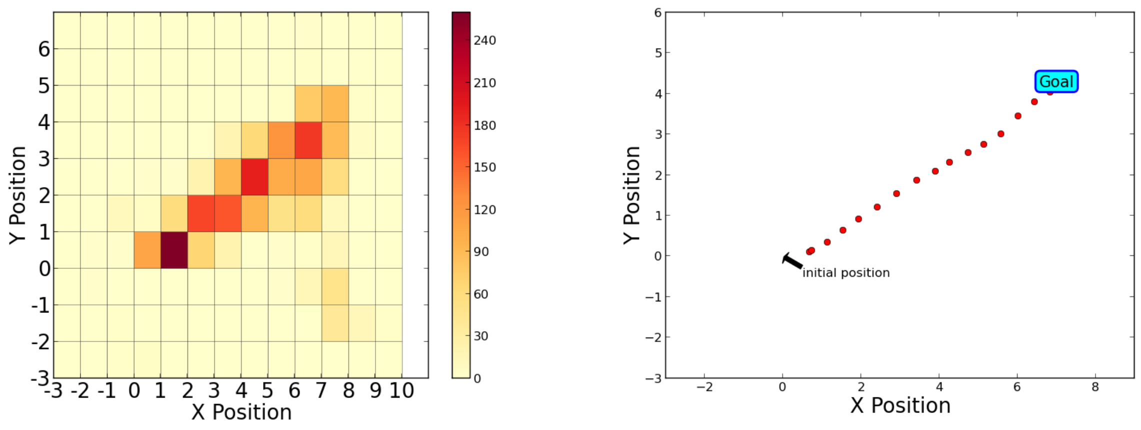
(b) Trajectory

Figure 4.15: 16384 RBFs with 6 actions



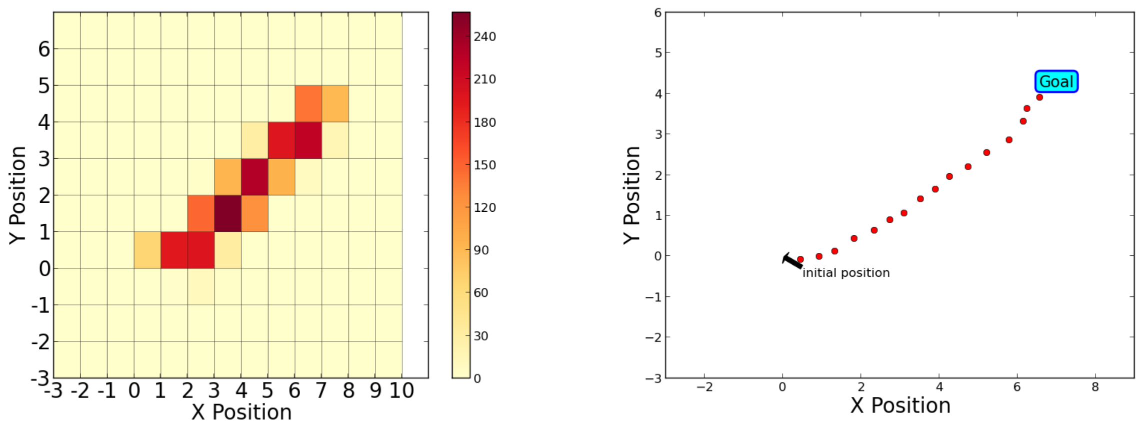
(a) Heat Map (b) Trajectory

Figure 4.16: Fourier Bases order 3 with 6 actions



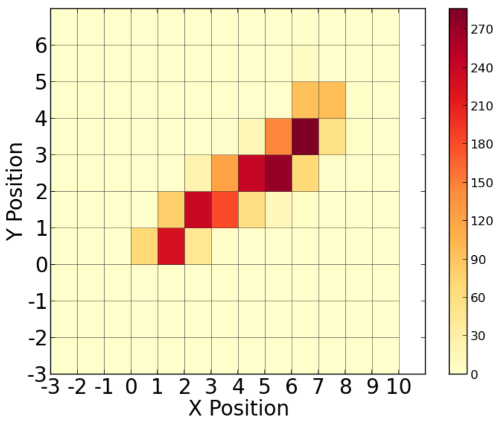
(a) Heat Map (b) Trajectory

Figure 4.17: 2048 RBFs with 8 actions

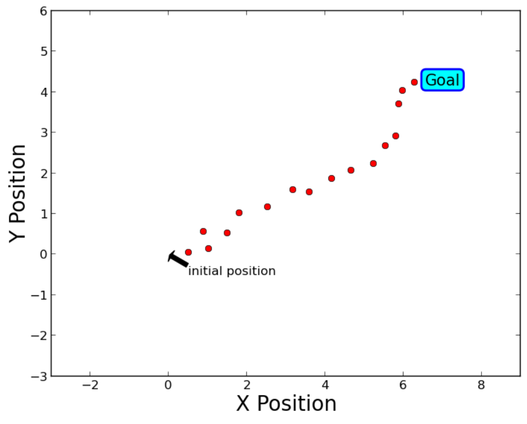


(a) Heat Map (b) Trajectory

Figure 4.18: 4096 RBFs with 8 actions

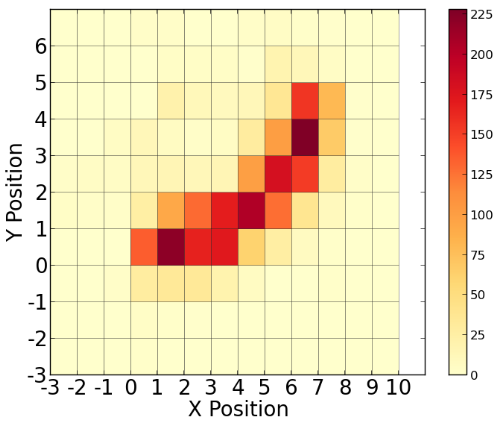


(a) Heat Map

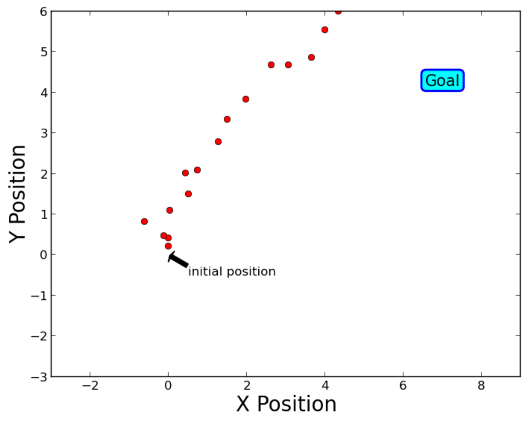


(b) Trajectory

Figure 4.19: 8192 RBFs with 8 actions

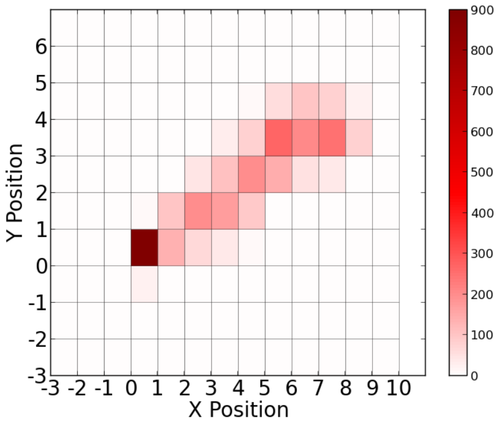


(a) Heat Map

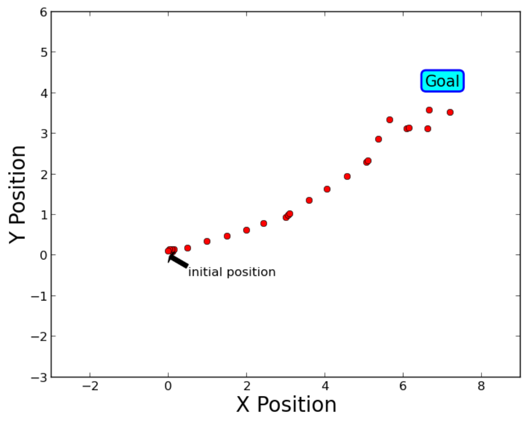


(b) Trajectory

Figure 4.20: 16384 RBFs with 8 actions



(a) Heat Map



(b) Trajectory

Figure 4.21: Fourier Bases order 3 with 8 actions

(iii) Number of successful trajectories generated for each implementation

In part (ii) of this experiment, we saw that the number of steps does not indicate that the trajectory was successful. Thus, for this part of the experiment, we record the number of successful trajectories generated for each implementation over 100 iterations. From those trajectories, we then compute the average number of steps to the goal state. This allows us to determine which implementation exhibits the best performance overall. Table 4.3 shows the number of times each optimal policy generates a successful trajectory over 100 iterations.

Type	**No. of basis functions**	**Number of actions**	**Number of successful trajectories**
RBF	2048	6	100
RBF	4096	6	100
RBF	8192	6	100
RBF	16384	6	99
Fourier	16384	6	99
RBF	2048	8	98
RBF	4096	8	100
RBF	8192	8	99
RBF	16384	8	95
Fourier	16384	8	99

Table 4.3: Number of successful trajectories generated over 100 episodes.

In Table 4.4 we record the total number of steps taken by the learning agent and compute the average number of steps required to reach the goal state.

Type	**No. of basis functions**	**Number of actions**	**Total number of steps**	**Average steps per episode**
RBF	2048	6	2283	22.83
RBF	4096	6	2208	22.08
RBF	8192	6	2333	23.33
RBF	16384	6	2717	27.44
Fourier	16384	6	3568	36.04
RBF	2048	8	2079	21.21
RBF	4096	8	2181	21.81
RBF	8192	8	2328	23.51
RBF	16384	8	2840	29.89
Fourier	16384	8	3504	35.39

Table 4.4: Average number of steps taken by each implementation to reach the goal state.

As can be seen from the tables above, all the implementations are able to produce a successful trajectory the majority of the time. The implementation of 16384 RBFs with 8 actions achieved the lowest accuracy of 95%. Even though we expect the performance to increase as the number of basis functions used increases, the results show the opposite. This low accuracy might be attributed to the fact that the value function has not fully converged. More time is required to learn the optimal value functions with the increase in the number of basis functions used. From Table 4.4 we observe that the average number of steps required by an agent using Fourier bases of order 3 tends to be greater than that of implementations using RBFs.

4.3.3 Discussion

Results from this experiment demonstrate that the use of different function approximation methods affect the performance of the learning agent. In terms of speed, the use of Fourier basis is able to compute the features of a state significantly faster than RBFs. As shown in part (i) of this experiment, implementations using Fourier basis of order 3 are able to attain a run time performance close to that of using 2048 RBFs while having 8 times more basis functions. Moreover, the total number of steps taken by agents using Fourier basis is a lot more than other implementations. However, this is actually a disadvantage since the total rewards received are also less than that of the others. Given the size of this state space and this reward function, 500 episodes were enough for the agents of each implementation to learn to fly from one point to another. Since we are only exploiting the optimal policies discovered after 500 episodes, we can not determine which implementation converges to the optimal action value functions the quickest. We can only deduce that after training has completed, all implementations are able to produce near optimal policies for this specific task. This can be seen from the heat maps and the number of successful trajectories those implementations produce. All the implementations try to get to the goal state by taking the shortest path, however, there are actually many paths that the agent could take to get there. The reward function determines which path the agent should take; a slight modification to the reward function results in a different path taken by the agent. The modification is described in the next chapter where the agent tries to accomplish the same task using apprenticeship learning via inverse reinforcement learning. The use of apprenticeship learning via inverse reinforcement learning requires an implementation to be run multiple times. Which means that an implementation that requires minimal amount of time to train and has high quality performance is preferred. Thus, we have decided to use the implementation of 2048 RBFs with 6 actions in the next chapter as it has been shown to perform very well with 100% successful trajectories produced and a decent run-time performance.

4.3.4 Discussion in Relation to the Research Question

Having completed this experimentation, the first part of the research questions set out in Chapter 3 can now be answered.

1. Sarsa can be used with function approximators to approximate the value functions. In this research, radial basis functions and Fourier basis were used and the results show that there are advantages and disadvantages associated with each method. In terms of speed, Fourier basis outperform radial basis functions. Whilst for the rewards received, implementations using radial basis functions perform better. Overall, both methods are able to learn the optimal value functions after 500 episodes. Note that the state space consists of 7 state variables, and by extending it to include various other onboard sensors, the performance might differ.

4.4 Conclusion

In this chapter, the implementation details as well as the results of the learning agent that was created using hard-coded reward function were presented. Results reveal that after 500 episodes, all implementations were able to produce an optimal policy that generates successful trajectories. In the next chapter, we examine the use of weighted features as reward function to learn the same task.

Chapter 5

Learning Agent II

5.1 Introduction

In the previous chapter, a learning agent was created using a hard-coded reward function. In this chapter, apprenticeship learning via inverse reinforcement learning was used to create the learning agent. The task of the learning agent was to reproduce the expert's behaviour given expert's trajectories. The implementation of the learning agent is presented below followed by the discussion on the results of the experiments.

5.2 Implementation

In this section, we provide the implementation details of the learning agent. This includes the following: expert trajectories, MDP, feature vector, expert's feature expectations, computing optimal policy and new feature expectations.

5.2.1 Learning Agent's Expert Trajectories

The expert trajectories used for the learning agent were created using the tum simulator. The author teleoperated the quadrotor via a joystick using the simulator, the states encountered as well as various other features were recorded. Such features include: the position, which is determined by the x, y and z coordinates of the quadrotor; and the orientation, represented as a quaternion, which is a four-element vector used for encoding any rotation in a 3D coordinate system.

5.2.2 Markov Decision Process

The MDP was constructed in the same way as in the previous chapter; the only deviation being that of the reward function. As seen from the previous chapter, the reward function used resulted in the agent taking the shortest path. However, there are actually countless ways to move from one point to another. Therefore, the task now is to fly from one point to another by taking a specific path as shown in Figure 5.1. That is to go straight then left. It is quite complicated to define a reward function for this task; we can not assign positive rewards to states near the red arrows such that the agent would just hover around the initial state, neither can we assign slightly higher rewards to those areas within the red arrows where the agent could be better off taking the shortest path. The changes made to the reward function are described below.

Feature Vector

The reward function that we tried to recover is assumed to be expressible as a linear combination of known features. Thus, a feature mapping ϕ is needed to map a given state to a vector of values between 0 and 1. In cases where the state and action spaces are discrete, the state and action spaces can be directly

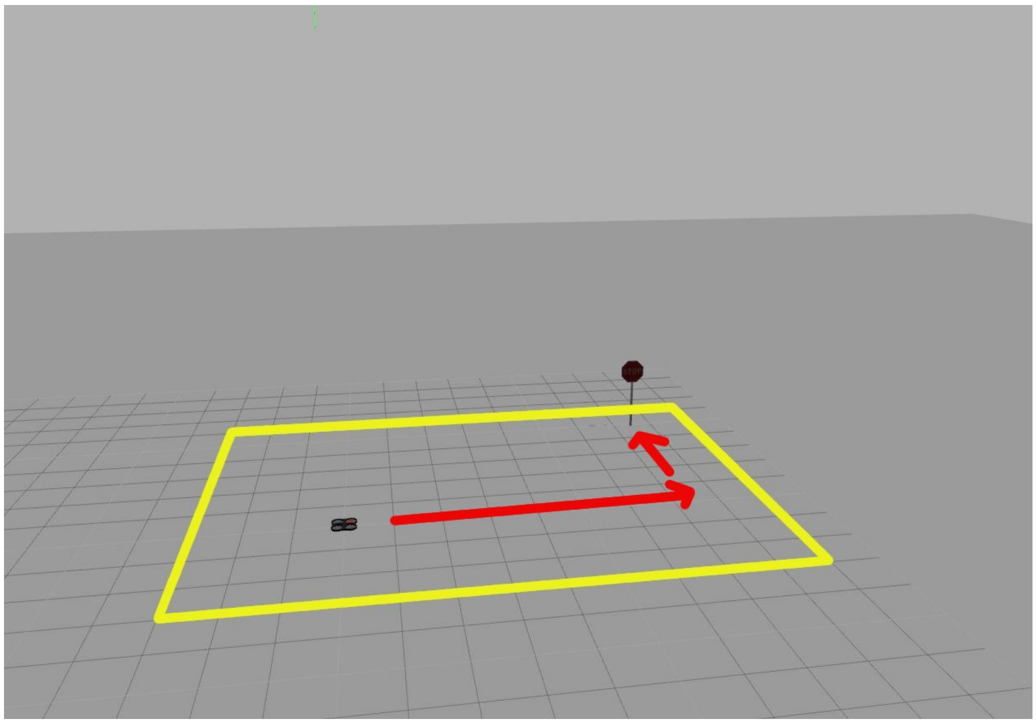


Figure 5.1: ROS tum simulator environment showing path.

used as features. Since we were dealing with continuous state space, radial basis functions were used as the feature mapping. This is because using binary features in continuous state space is not appropriate: the feature could either be on or off. Whereas using radial basis functions allows us to indicate the degree in which a feature is on or off. As a result, the agent would get rewarded if it visits states that are close to the expert's. Figure 5.2 shows an example of a feature vector where the state and action spaces are discrete. In this case, the feature vector would be of length $states * actions$, where a given state refers to the x, y, and z GPS coordinates. That is, if each of the x, y, and z GPS coordinates has 20 discrete values, then the resulting vector would be of length 20*20*20*9 = 72000. Figure 5.3 shows an example of a feature vector in continuous state and action spaces using RBFs to represent the features, where n is the number of radial basis functions used. In this research, 100 radial basis functions were generated randomly, thus the length of the feature vector was 100.

$$\begin{pmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Figure 5.2: Feature vector for discrete state and action spaces $\phi : S \times A \rightarrow [0, 1]^{S \times A}$.

$$\begin{pmatrix} 0.0 \\ \vdots \\ 0.1 \\ 0.05 \\ 0.01 \\ \vdots \\ 0.0 \end{pmatrix}$$

Figure 5.3: Feature vector for continuous state and action spaces using RBFs $\phi : S \times A \to [0, 1]^n$.

With the use of the feature vector ϕ and the expert's trajectories, the expert's feature expectations can be estimated.

5.2.3 Expert's Feature Expectations

The expert's feature expectations vector, μ_E, is the expected discounted accumulated features encountered when following the expert's policy π_E. However, π_E is unknown and as a result, μ_E must be estimated from a given set of trajectories. The estimated feature expectations of the expert, $\hat{\mu_E}$, is the expert's averaged discounted accumulated features, where the features are extracted from the expert's trajectories. $\hat{\mu_E}$ is calculated as shown in equation 2.24. The inner sum calculates the feature expectations, where the discounted features are summed together using the feature mapping ϕ, which in this case was the radial basis functions. The outer sum runs through the given number of expert trajectories which gives the total feature expectations. In this experiment, the learning agent used 50 expert trajectories, m = 50, and the goal was to see if the learning agent can learn how to fly from Point A to Point B based on the given trajectories. In order to incorporate flying patterns into the learning agent, multiple demonstrations are required. We chose m=50 since providing too few demonstrations would result in the feature vector being too sparse. The results of the experiment are presented in section 5.3.

Apprenticeship learning via inverse reinforcement learning tries to find a policy whose performance is as close as to the expert's by using expert's trajectories. That is, it tries to find a policy that induces feature expectations close to that of the expert's.

5.2.4 Reinforcement Learning Step

The Sarsa algorithm was used in the reinforcement learning step of the apprenticeship learning via inverse reinforcement learning algorithm. In this step, a reward function is given in which the reinforcement learning algorithm was applied to find the optimal policy. That is, the Sarsa algorithm tries to find the optimal policy given the weight vector w and the feature mapping ϕ which was the RBF. However, since the state space is continuous and cannot be represented as discrete tables, function approximation is needed. In this experiment, RBF was used to approximate the value functions. In specific, the implementation of 2048 RBFs with 6 actions was used. The algorithm was run for 500 iterations, and the resulting policy was used to calculate new feature expectations.

5.2.5 Calculating New Feature Expectations

The reinforcement learning step returns a policy in which new feature expectations $\mu(\pi)$ are calculated. Since the policy is non-deterministic, the feature expectations is calculated by taking an expectation as shown in Equation 2.22. This is accomplished by running the policy using the simulator. The greedy policy was chosen using the Q functions, and the states encountered were recorded. New feature expectations were then calculated using those trajectories. These feature expectations are then used to find a new weight vector and a new reward function. The process repeats until a policy has been found that induces feature expectations that are close to that of the expert's.

5.3 Results and Discussions

In this section, we provide the results of the experiments using weighted features as the reward function. In particular, we examine whether or not the use of apprenticeship learning via inverse reinforcement learning allows us to recover a reward function that induces performance close to that of the expert's.

5.3.1 Experimentation using Weighted Features

In this experiment, we run the apprenticeship learning algorithm in an attempt to recover the expert's reward function.

Expert Demonstrations

The expert, in this case the author, demonstrated how to reach the goal state via a specific path 50 times. These demonstrations are provided to the learning agent which using the apprenticeship learning algorithm then learns the underlying reward function. Figure 5.4 gives a graphical representation of the expert demonstrations in the form of heat map. A single expert trajectory (X and Y positions) is shown in Figure 5.5.

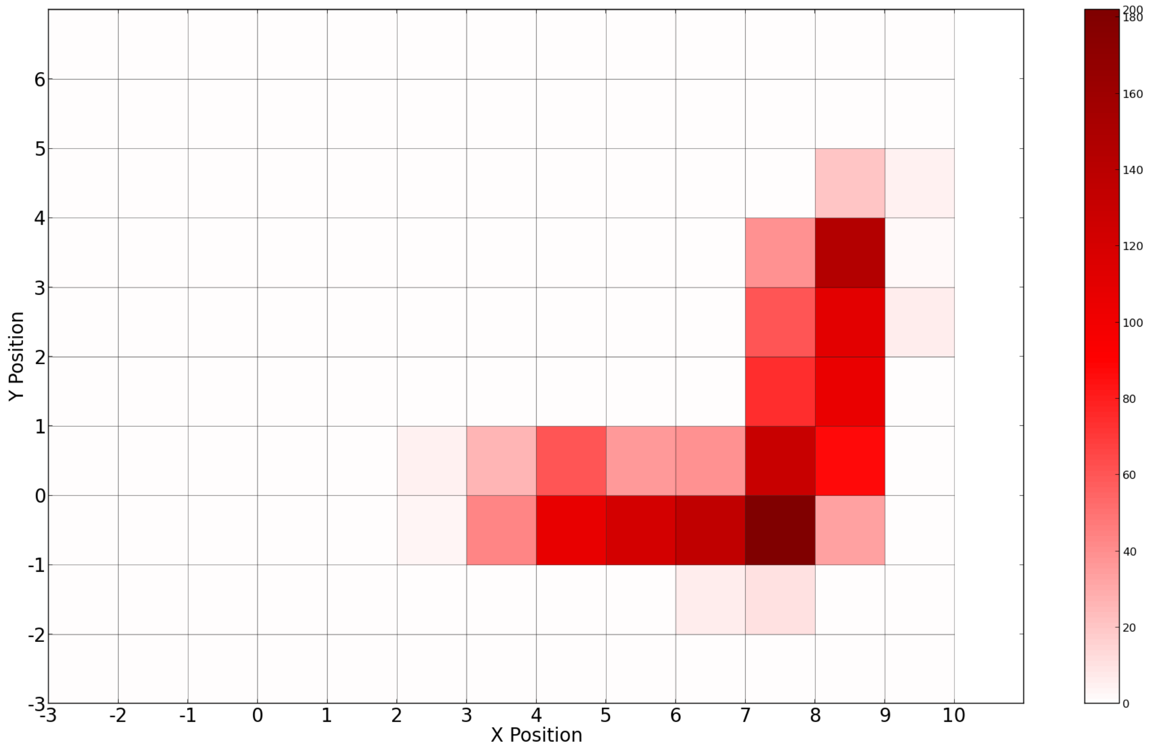


Figure 5.4: 50 expert trajectories.

As can be seen in Figure 5.4 and 5.5, the trajectories were incomplete. This is due to time delays, where some of the states were not recorded, especially those near the initial state. Nonetheless, from the figures, we could still determine the path taken by the expert. That is, to fly in the East direction until in line with the goal state then fly North.

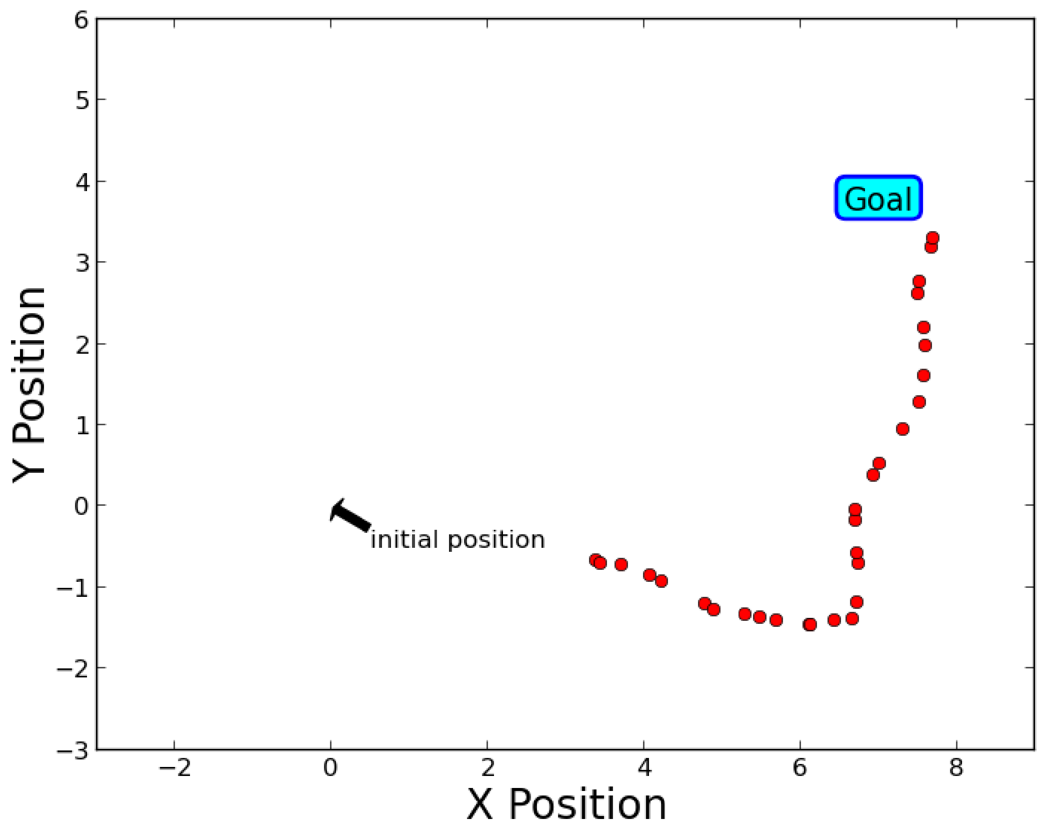


Figure 5.5: Single expert demonstration.

Feature Mapping

Since the reward function is represented as the weighted sum of features, a feature mapping is needed to extract the features from a set of trajectories. 100 radial basis functions were used and the centers in the X and Y dimensions can be seen in Figure 5.6.

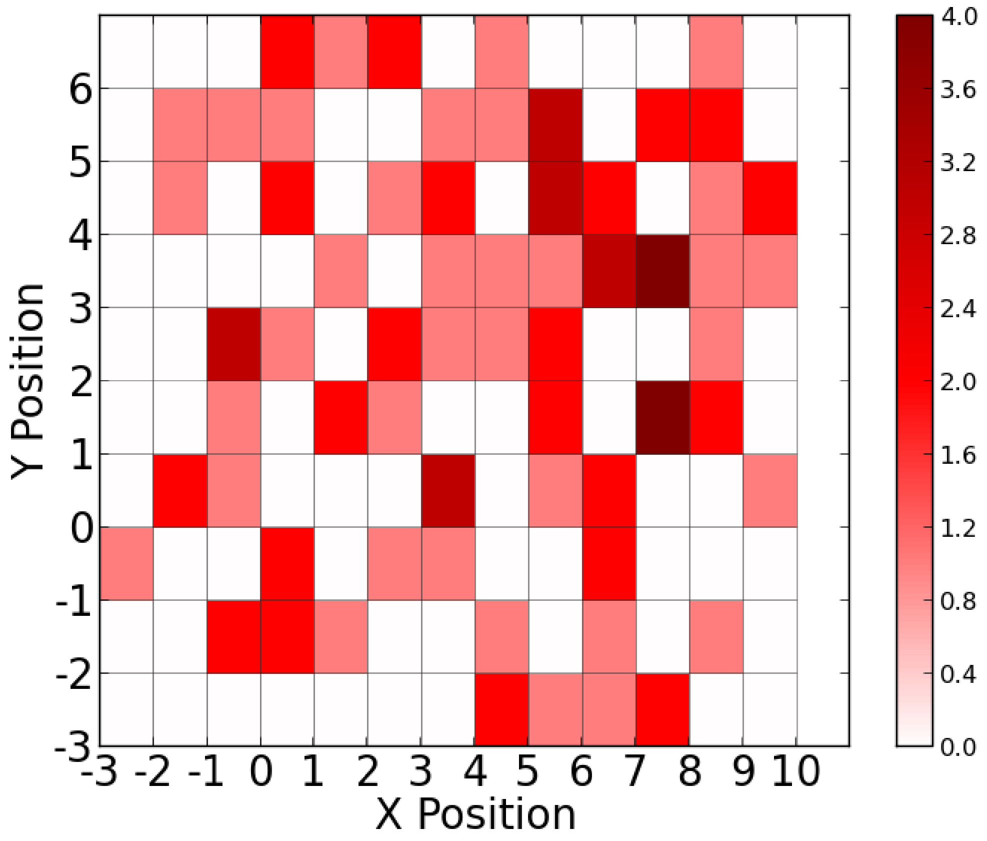


Figure 5.6: Number of radial basis functions with centers in X and Y dimensions.

Note that a RBF consists of 7 state variables, but only the centers in X and Y dimensions are shown here as we believe they are more relevant to the task we are trying to accomplish. We observe that even though 100 basis functions were used, they weren't enough to cover the whole state space in the X and Y dimensions. This could affect the performance of the learning agent as it is possible that the desired task could not be adequately represented by those basis functions. However, there are quite a few basis functions near the goal state which should be able overcome this matter.

Distance to Expert's Feature Expectations

The shorter the euclidean distance to the expert's feature expectations means that the closer we are in finding a policy whose performance is similar to the expert. The euclidean distance to the expert's feature expectations after each iteration of the apprenticeship learning algorithm is shown in Figure 5.7, where distance is calculated with respect to the expert's feature expectations and the current found policy's feature expectations.

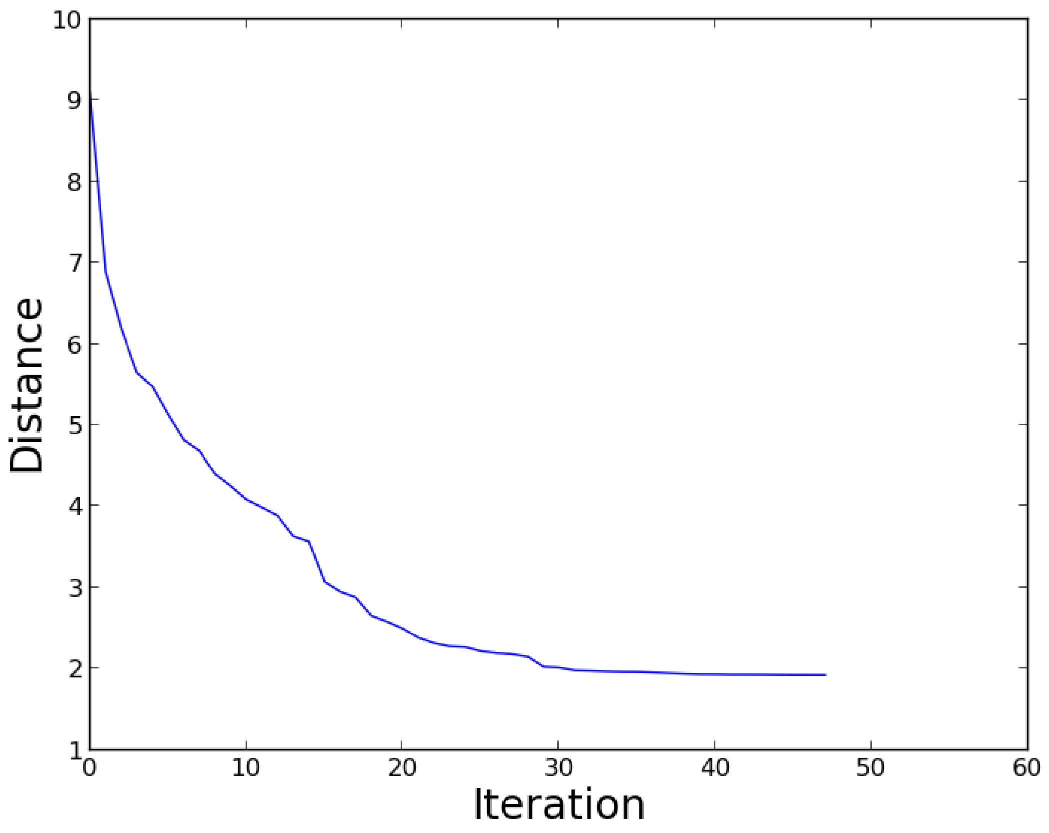


Figure 5.7: Euclidean distance to expert's feature expectations.

We see in Figure 5.7 that for each iteration of the apprenticeship learning algorithm, it brings us closer to the expert's feature expectations. We observe that the euclidean distance decreased from 9.097 to 1.925. Even though 1.925 is still a lot greater than the terminating condition for the apprenticeship learning algorithm (0.5), we decided to stop here. This is because the distance to the expert's feature expectations started to decrease at a much slower rate, but more importantly, we have found a policy that performs as well as the expert.

Discovered Policy

The task of the learning agent is to fly to the goal state via a specific route. Therefore we would expect the discovered reward function to assign higher rewards to those states along the path as well as near the goal state. Figure 5.8 shows the expert reward function discovered after 47 iterations in the form of a heat map. The points in blue correspond to states with negative rewards; the points in red represent states with positive rewards. Table 5.1 and 5.2 give the basis functions with the highest and lowest weights.

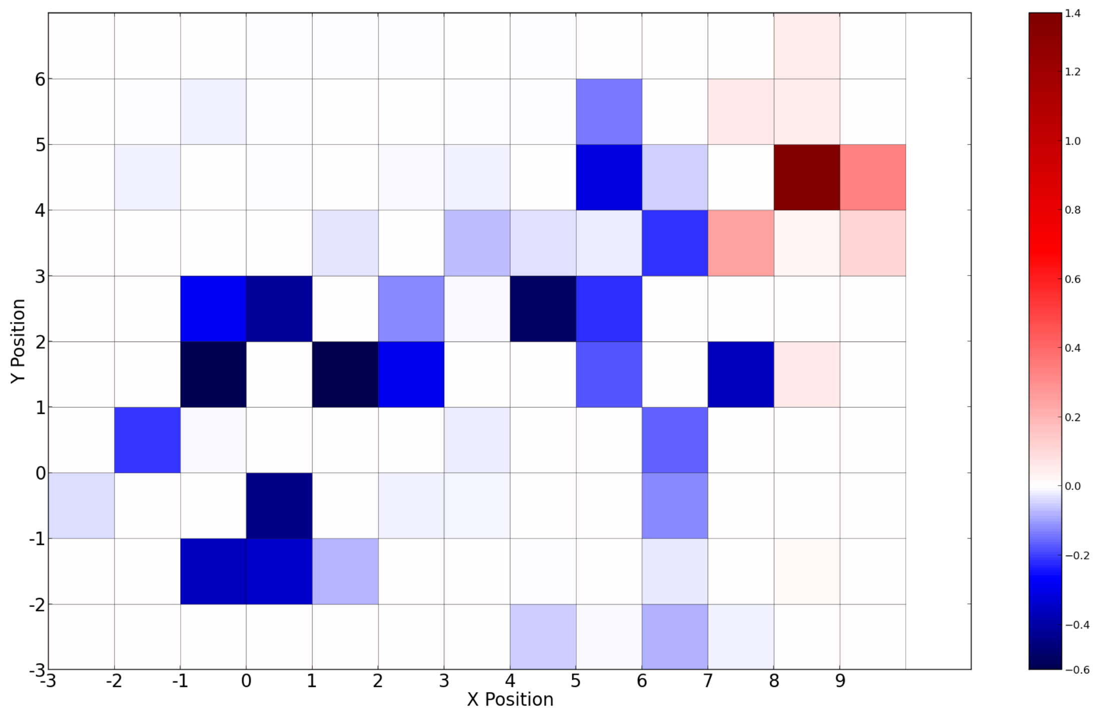


Figure 5.8: Expert reward function discovered.

RBF	X Center	Y Center	Weight
45	7.6668317	3.8086124	1.3687283
93	7.1647356	3.0345893	0.3192707
85	8.7211365	3.7076802	0.1991550
63	8.7242511	3.8807252	0.1399921
70	7.5472815	0.6660204	0.1169506

Table 5.1: Top 5 basis functions with the highest weights.

RBF	X Center	Y Center	Weight
12	-0.9821142	0.9198778	-0.5194532
47	4.2515919	1.5184586	-0.4951706
94	0.0322013	-1.3110444	-0.4196577
32	0.3095117	1.5302734	-0.4155340
64	1.4898608	1.2482849	-0.3031006

Table 5.2: Top 5 basis functions with the lowest weights.

We observe that the discovered reward function actually makes intuitive sense; it assigns slightly higher rewards to those state which are more preferable. Note that the reward function is represented as a weighted sum of features, therefore the use of the apprenticeship learning algorithm basically modifies the weights of those 100 RBFs after each iteration. We see that the basis functions above the initial state are assigned with unfavourable rewards which penalises the learning agent for getting too close to these states. This is expected since we do not want the learning agent to take that path, those negative rewards would force the learning agent to fly in the east direction instead. As can be seen from the tables, the preferable states, especially those near the goal state, are assigned with positive rewards. While the undesirable states, especially those above and below the initial state, are assigned with negative rewards. One interesting point to note is that, from Table 5.1, the 70th RBF has a positive value. This is because at this point, the learning agent should be in line with the goal, and the positive reward forces the agent to fly towards the goal state. By maximizing the rewards of this reward function, we get a resulting policy as shown in Figure 5.9. Figure 5.10 shows a single trajectory generated by the optimal policy.

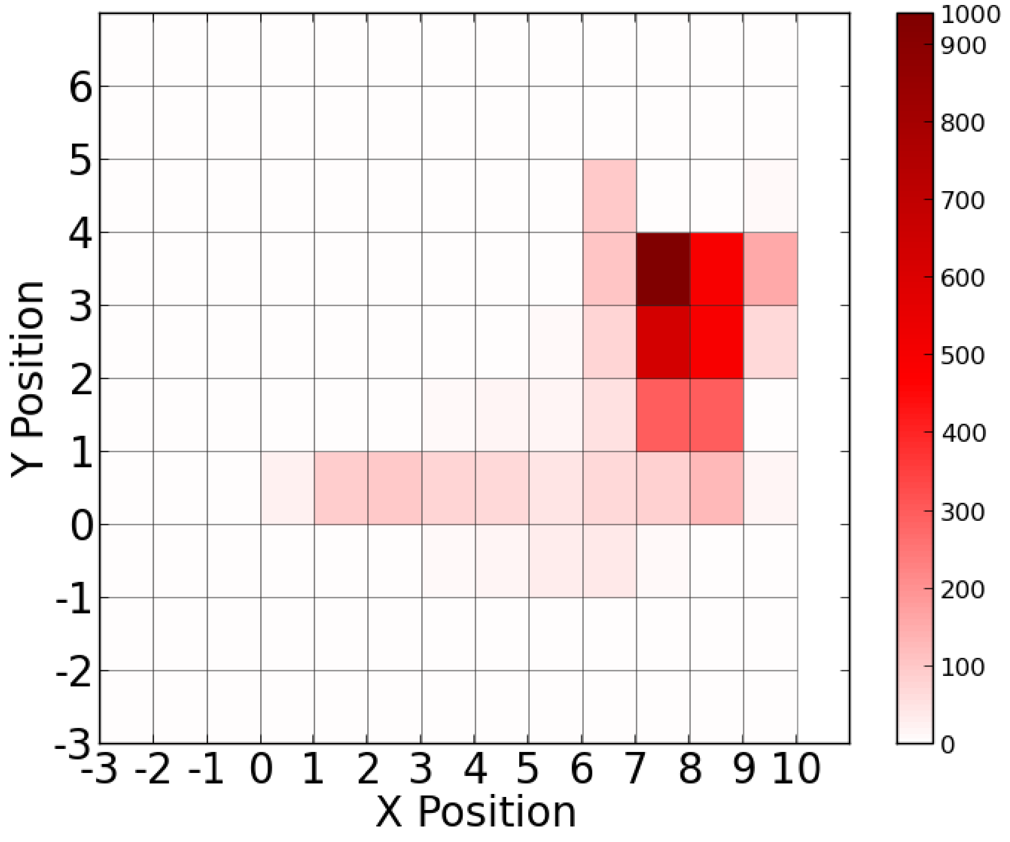


Figure 5.9: Final learned policy.

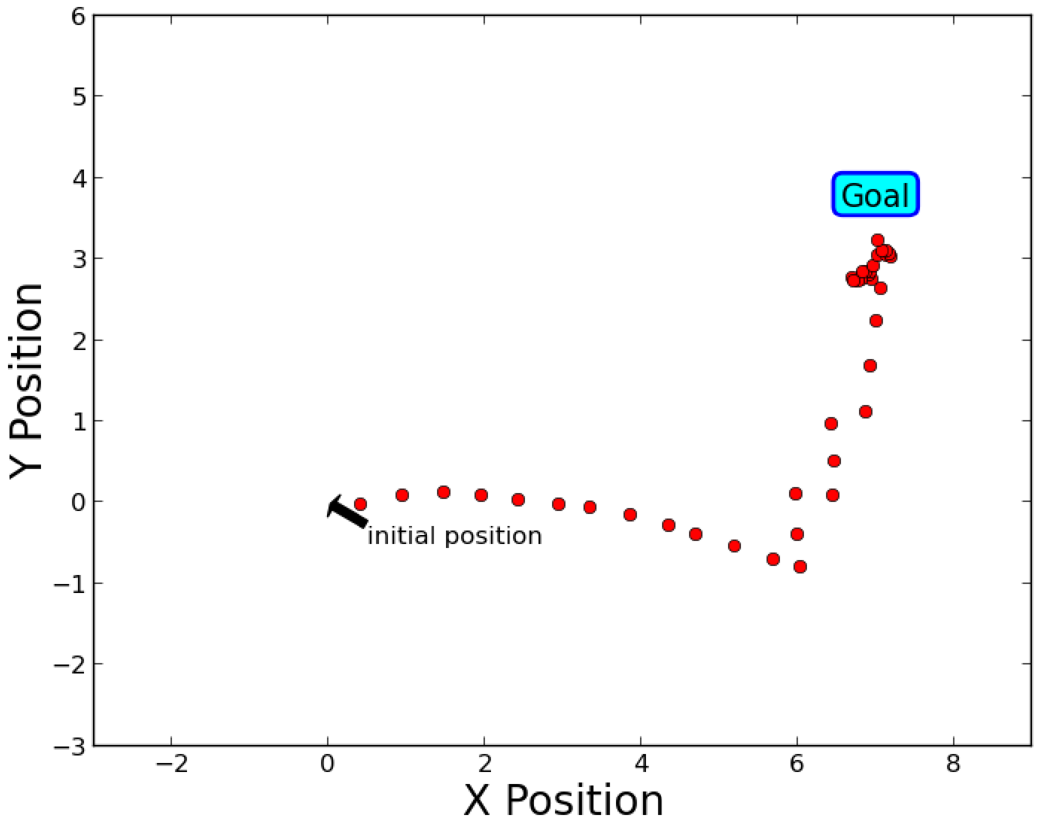


Figure 5.10: Single trajectory generated from the final learned policy.

From the figures, we observe that the learning agent learned to take the desired path. We also observe that most of the time, the learning agent decides to hover around the goal state. This is probably because those states surrounding the goal state are assigned with positive rewards. Even though there quite a few basis functions around the goal state, they have different z positions and different orientations. Nonetheless, the learning agent was able to use the recovered reward function and attain a performance similar to the expert's.

5.3.2 Discussion

Results from this experiment show that the use of apprenticeship learning was able to output a policy with a performance close to the expert's. The learning agent is capable of learning the task demonstrated by the expert when given 50 expert trajectories. We observe that for each iteration of the apprenticeship learning algorithm, the performance of the learning agent increases with respect to the expert's unknown reward function. This can be seen from the decrease in the euclidean distance to the expert's feature expectations. The final discovered policy had a euclidean distance to the expert's feature expectations of 1.925, which is not really that close. The reason we did not achieve a euclidean distance that is less than 1 might be attributed to the fact that the demonstrations provided were incomplete. This has a large implication on the performance of the learning agent since the states that are inevitable, especially those states near the initial state, were missing from the expert demonstrations. As a result, the learning agent could not match the expert's feature expectations as close as possible. Furthermore, when the author was demonstrating the task, a joystick was used to control the drone. These actions might be slightly different to the ones provided to the learning agent. We also observe that the 100 randomly generated RBFs did not fill up the state space in the X and Y dimensions sufficiently, which affects how the learning agent is rewarded or penalised in states where no RBF is present. Despite all these factors affecting the performance of the learning agent, the final discovered policy was able to generate trajectories bearing resemblance to the expert demonstrations. In fact, the discovered policy might even be better since the trajectory it generates shows the complete path from the initial state to the goal state. However, the downside to the apprenticeship learning algorithm is that it requires many iterations to be run, each having a different reward function. This is much more time consuming compared to the case where we know the reward function beforehand, and we just need to find the optimal policy with respect to that reward function.

5.3.3 Answering Research Question

Having completed this experimentation, the second part of the research questions set out in Chapter 3 can now be answered.

2. Radial basis functions can be used to estimate the expert's feature expectations. In this experiment, the reward function used by the algorithm is constructed using 100 radial basis functions. Results show that the recovered reward function was able to reward preferable states and penalise undesirable states. That is, the algorithm was able to assign higher weights to the RBFs whose centers are close to the goal states, and lower weights to the RBFs who centers are close to states that are not preferred.

5.4 Conclusion

This chapter described the implementation details as well as the results of the learning agent that was created using weighted features. Results reveal that the use of apprenticeship learning is very effective; the discovered policy was able to generate trajectories bearing resemblance to the expert's demonstrations. The next chapter concludes this dissertation by summarising the research and presenting potential future work.

Chapter 6

Conclusion

Designing control algorithms by hand for robots are often very difficult; an extensive amount of knowledge about the robotic platform domain is required. The Parrot AR.Drone is the robotic platform selected in this research. The drone can perform manoeuvres in three dimensions and collect information with its onboard sensors, which increases the difficulty of designing control algorithms by hand. Futhermore, the hand-coded algorithm must take into account all the circumstances that might arise during task execution. For every unexpected circumstance that is encountered, the robot would fail and the algorithm needs to be amended. One solution to address this problem would be to use learning from demonstration. The concept of learning from demonstration is that the teacher provides the learner with a set of demonstrations. The learner can then reproduce the desired behavior by using those demonstrations. If an unexpected circumstance occurs, the teacher just needs to perform another demonstration.

Given a set of demonstration data, the three core approaches that can be used for policy derivation include mapping function approach, system model approach, and planning approach. The mapping approach involves learning an approximation of function that maps the robot's state to actions. Classification and regression algorithms are used to learn the state-action mapping. The system model approach involves learning the state-transition model of the world dynamics. Reinforcement learning algorithm and reward function are used to produce an optimal value function which tells us the preferred action to select for any given state. The planning approach involves planning a sequence of actions by learning the rules associating every action with a set of pre- and post-conditions.

In this research, we have taken the system model approach in the absence of a reward function to derive a policy for a given task. In specific, apprenticeship learning via inverse reinforcement learning was used to learn the task demonstrated by the expert, which was to fly from one point to another. This approach eliminates the need to manually specify the reward function and is thus often used in applications where it is much easier to demonstrate the desired task. The goal of this approach is to discover a reward function from the demonstration data, where the reinforcement learning algorithm can be applied to learn a policy that attains performance close to that of the expert. The underlying reward function can be viewed as trying to achieve the expert's performance by rewarding trajectories that contain similar features to the expert trajectories being presented. Results from experimentation show that the use of apprenticeship learning via inverse reinforcement learning is very effective; the agent was able to learn the desired task based on expert trajectories presented to it. The recovered reward function was able to reward preferred states and penalize undesirable states, thus forces the agent to take a specific path to the goal state. Even though the final policy output by the algorithm did not induce feature expectations very close to the expert's, it was able to generate trajectories bearing resemblance to the expert trajectories.

The algorithm used in this research assumed that the reward function can be expressed as a linear function of known features. However, in most applications we would expect the rewards to be spike. That is, you either get rewarded for doing the right task or you do not. Thus, an interesting approach to consider might be to extend the algorithm into learning reward functions incorporating nonlinear features. Furthermore, the quadrotor used in this research does not have a GPS device attached to it. Hence, an interesting direction to consider for future work might be to replace the GPS positions by camera images. In this case, the quadrotor locates itself based on the features present in the images. In addition, it might be worth investigating the use of just onboard sensors and the time as state variables. The agent would then learn the correct action to take for any given time stamp. This would be useful in situations where the surrounding areas are similar and the images are not very helpful.

References

[Abbeel and Ng 2004] Pieter Abbeel and Andrew Y Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the twenty-first international conference on Machine learning*, page 1. ACM, 2004.

[Anderson *et al.* 1986] John Robert Anderson, Ryszard Spencer Michalski, Ryszard Stanisław Michalski, Thomas Michael Mitchell, et al. *Machine learning: An artificial intelligence approach*, volume 2. Morgan Kaufmann, 1986.

[Anderson 2010] C Anderson. *Parrot AR.Drones specs*, 2010. Retrieved 11 August 2013, from `http://diydrones.com/profiles/blogs/parrot-ardrones-specs-arm9`

[Argall *et al.* 2009] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2009.

[Argall 2009] Brenna D Argall. *Learning mobile robot motion control from demonstration and corrective feedback.* PhD thesis, University of Southern California, 2009.

[Atkeson and Schaal 1997] Christopher G Atkeson and Stefan Schaal. Robot learning from demonstration. In *ICML*, volume 97, pages 12–20, 1997.

[Atkeson *et al.* 1997] Christopher G Atkeson, Andrew W Moore, and Stefan Schaal. Locally weighted learning for control. *Artificial intelligence review*, 11(1-5):75–113, 1997.

[Atkeson 1991] Christopher G Atkeson. Using locally weighted regression for robot learning. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 958–963. IEEE, 1991.

[Bagnell and Schneider 2001] J Andrew Bagnell and Jeff G Schneider. Autonomous helicopter control using reinforcement learning policy search methods. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 2, pages 1615–1620. IEEE, 2001.

[Bentivegna 2004] Darrin Charles Bentivegna. Learning from observation using primitives. 2004.

[Billing and Hellström 2010] Erik A Billing and Thomas Hellström. A formalism for learning from demonstration. *Paladyn*, 1(1):1–13, 2010.

[Bills *et al.* 2011] Cooper Bills, Joyce Chen, and Ashutosh Saxena. Autonomous mav flight in indoor environments using single image perspective cues. In *Robotics and automation (ICRA), 2011 IEEE international conference on*, pages 5776–5783. IEEE, 2011.

[Breazeal and Scassellati 2002] Cynthia Breazeal and Brian Scassellati. Robots that imitate humans. *Trends in Cognitive Sciences*, 6(11):481–487, 2002.

[Bristeau *et al.* 2011] Pierre-Jean Bristeau, François Callou, David Vissière, Nicolas Petit, et al. The navigation and control technology inside the ar. drone micro uav. In *18th IFAC World Congress*, pages 1477–1484, 2011.

[Brown University] *Brown University Repository for ROS Packages.* `https://code.google.com/p/brown-ros-pkg/`.

[Browning *et al.* 2004] Brett Browning, Ling Xu, and Manuela Veloso. Skill acquisition and use for a dynamically-balancing soccer robot. In *AAAI*, pages 599–604, 2004.

[Christopher *et al.* 1997] G Atkeson Christopher, W Moore Andrew, and Schaal Stefan. Locally weighted learning. *Artificial Intelligence Review*, 11(1):11–73, 1997.

[Cleveland and Devlin 1988] William S Cleveland and Susan J Devlin. Locally weighted regression: an approach to regression analysis by local fitting. *Journal of the American Statistical Association*, 83(403):596–610, 1988.

[Dabney and Barto 2012] William Dabney and Andrew G Barto. Adaptive step-size for online temporal difference learning. In *AAAI*, 2012.

[Funahashi 1989] Ken-Ichi Funahashi. On the approximate realization of continuous mappings by neural networks. *Neural networks*, 2(3):183–192, 1989.

[Hastie *et al.* 2001] Trevor. Hastie, Robert. Tibshirani, and J Jerome H Friedman. *The elements of statistical learning*, volume 1. Springer New York, 2001.

[Hertz *et al.* 1991] John A Hertz, Anders S Krogh, and Richard G Palmer. *Introduction to the Theory of Neural Computacion*, volume 1. Basic Books, 1991.

[Higuchi *et al.* 2011] Keita Higuchi, Tetsuro Shimada, and Jun Rekimoto. Flying sports assistant: external visual imagery representation for sports training. In *Proceedings of the 2nd Augmented Human International Conference*, page 7. ACM, 2011.

[Kober and Peters 2012] Jens Kober and Jan Peters. Reinforcement learning in robotics: a survey. In *Reinforcement Learning*, pages 579–610. Springer, 2012.

[Konidaris *et al.* 2011] George Konidaris, Sarah Osentoski, and Philip S Thomas. Value function approximation in reinforcement learning using the fourier basis. In *AAAI*, 2011.

[Kormushev *et al.* 2013] Petar Kormushev, Sylvain Calinon, and Darwin G Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2(3):122–148, 2013.

[Krajník *et al.* 2011] Tomáš Krajník, Vojtěch Vonásek, Daniel Fišer, and Jan Faigl. Ar-drone as a platform for robotic research and education. In *Research and Education in Robotics-EUROBOT 2011*, pages 172–186. Springer, 2011.

[Lenz 2003] J. Lenz. *Reinforcement Learning and the Temporal Difference Algorithm*, 2003.

[Mahadevan 1996] Sridhar Mahadevan. Machine learning for robots: A comparison of different paradigms. In *Proceedings of the Workshop on Towards Real Autonomy, IEEE/RSJ Internaltional Conference on Intelligent Robots and Systems (IROS96)*. Citeseer, 1996.

[Marsland 2011] Stephen Marsland. *Machine learning: an algorithmic perspective*. CRC Press, 2011.

[Nehaniv and Dautenhahn 2002] Chrystopher L Nehaniv and Kerstin Dautenhahn. 2 the correspondence problem. *Imitation in animals and artifacts*, page 41, 2002.

[Ng and Sharlin 2011] Wai Shan Ng and Ehud Sharlin. Collocated interaction with flying robots. In *RO-MAN, 2011 IEEE*, pages 143–149. IEEE, 2011.

[O'Kane 2013] Jason M. O'Kane. *A Gentle Introduction to ROS*. Independently published, October 2013. Available at `http://www.cse.sc.edu/~jokane/agitr/`.

[Parrot 2012] SA Parrot. *AR.Drone*, 2012. Retrieved 11 August 2013, from `http://ardrone2.parrot.com/`

[Quigley *et al.* 2009] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[Russell *et al.* 1995] Stuart Russell, Peter Norvig, and Artificial Intelligence. A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25, 1995.

[Russell 1998] Stuart Russell. Learning agents for uncertain environments. In *Proceedings of the eleventh annual conference on Computational learning theory*, pages 101–103. ACM, 1998.

[Singh and Bertsekas 1997] Satinder Singh and Dimitri Bertsekas. Reinforcement learning for dynamic channel allocation in cellular telephone systems. *Advances in neural information processing systems*, pages 974–980, 1997.

[Smart and Kaelbling 2001] William D Smart and Leslie Pack Kaelbling. Reinforcement learning for robot control. *Mobile Robots XVI*, 2001.

[Smart and Pack Kaelbling 2002] William D Smart and L Pack Kaelbling. Effective reinforcement learning for mobile robots. In *Robotics and Automation, 2002. Proceedings. ICRA'02. IEEE International Conference on*, volume 4, pages 3404–3410. IEEE, 2002.

[Smart 2002] William Donald Smart. *Making reinforcement learning work on real robots*. PhD thesis, Brown University, 2002.

[Sutton and Barto 1998] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*, volume 1. Cambridge Univ Press, 1998.

[Szepesvári 2010] Csaba Szepesvári. Algorithms for reinforcement learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 4(1):1–103, 2010.

[Tanner and White 2009] Brian Tanner and Adam White. Rl-glue: Language-independent software for reinforcement-learning experiments. *The Journal of Machine Learning Research*, 10:2133–2136, 2009.

[Watkins and Dayan 1992] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[Zhang and Dietterich 1995] Wei Zhang and Thomas G Dietterich. A reinforcement learning approach to job-shop scheduling. In *IJCAI*, volume 95, pages 1114–1120. Citeseer, 1995.