Technical University of Crete

School of Electrical and Computer Engineering

Reinforcement Learning for Autonomous Unmanned Aerial Vehicles

Nikolaos Geramanis

Thesis Committee

Associate Professor Michail G. Lagoudakis (ECE)

Professor Aggelos Bletsas (ECE)

Associate Professor Panagiotis Partsinevelos (MRE)

Chania, October 2020

Πολυτεχνείο Κρήτης

Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών

Ενισχυτική Μάθηση για Αυτόνομα Μη-Επανδρωμένα Ιπτάμενα Οχήματα

Νικόλαος Γεραμάνης

Εξεταστική Επιτροπή

Αναπληρωτής Καθηγητής Μιχαήλ Γ. Λαγουδάκης (ΗΜΜΥ)

Καθηγητής Άγγελος Μπλέτσας (ΗΜΜΥ)

Αναπληρωτής Καθηγητής Παναγιώτης Παρτσινέβελος (ΜΗΧΟΠ)

Χανιά, Οκτώβριος 2020

Abstract

Reinforcement learning is an area of machine learning concerned with how autonomous agents learn to behave in unknown environments through trial-and-error. The goal of a reinforcement learning agent is to learn a sequential decision policy that maximizes the notion of cumulative reward through continuous interaction with the unknown environment. A challenging problem in robotics is the autonomous navigation of an Unmanned Aerial Vehicle (UAV) in worlds with no available map. This ability is critical in many applications, such as search and rescue operations or the mapping of geographical areas. In this thesis, we present a map-less approach for the autonomous, safe navigation of a UAV in unknown environments using reinforcement learning. Specifically, we implemented two popular algorithms, SARSA(λ) and Least-Squares Policy Iteration (LSPI), and combined them with tile coding, a parametric, linear approximation architecture for value function in order to deal with the 5- or 3-dimensional continuous state space defined by the measurements of the UAV distance sensors. The final policy of each algorithm, learned over only 500 episodes, was tested in unknown environments more complex than the one used for training in order to evaluate the behavior of each policy. Results show that SARSA(λ) was able to learn a near-optimal policy that performed adequately even in unknown situations, leading the UAV along paths free-of-collisions with obstacles. LSPI's policy required less learning time and its performance was promising, but not as effective, as it occasionally leads to collisions in unknown situations. The whole project was implemented using the Robot Operating System (ROS) framework and the Gazebo robot simulation environment.

Περίληψη

Η ενισχυτική μάθηση είναι ένας τομέας της μηχανικής μάθησης που ασχολείται με το πως οι αυτόνομοι πράκτορες μαθαίνουν να συμπεριφέρονται σε άγνωστα περιβάλλοντα μέσω μιας διαδικασίας δοκιμής και σφάλματος. Ο στόχος ενός πράκτορα ενισχυτικής μάθησης είναι να μάθει μια πολιτική ακολουθιακής λήψης αποφάσεων, η οποία μεγιστοποιεί την έννοια της αθροιστικής αμοιβής, μέσα από συνεχή αλληλεπίδραση με το άγνωστο περιβάλλον. Ένα απαιτητικό πρόβλημα στη ρομποτική είναι η αυτόνομη πλοήγηση ενός μη-επανδρωμένου ιπτάμενου οχήματος (Unmanned Aerial Vehicle – UAV) σε κόσμους χωρίς διαθέσιμο χάρτη. Αυτή η ικανότητα είναι κρίσιμη σε διάφορες εφαρμογές, όπως αποστολές έρευνας και διάσωσης και χαρτογράφηση γεωγραφικών περιοχών. Η παρούσα διπλωματική εργασία παρουσιάζει μια προσέγγιση για την αυτόνομη, ασφαλή πλοήγηση ενός UAV χωρίς χρήση χάρτη, σε άγνωστα περιβάλλοντα χρησιμοποιώντας ενισχυτική μάθηση. Πιο συγκεκριμένα, υλοποιήσαμε δύο γνωστούς αλγορίθμους, τον SARSA(λ) και τον Least-Squares Policy Iteration (LSPI), και τους συνδυάσαμε με την τεχνική τιλε ςοδινγ, μια παραμετρική, γραμμική αρχιτεκτονική προσέγγισης της συνάρτησης τιμής με σκοπό να αντιμετωπίσουμε τον 5- ή 3-διάστατο συνεχή χώρο καταστάσεων που ορίζεται από τις μετρήσεις των αισθητήρων απόστασης του UAV. Η τελική πολιτική κάθε αλγορίθμου, μετά από μάθηση σε 500 επεισόδια, δοκιμάστηκε και σε άγνωστα περιβάλλοντα πιο πολύπλοκα από αυτό της εκπαίδευσης με σκοπό να αξιολογηθεί η συμπεριφορά κάθε πολιτικής. Τα αποτελέσματα δείχνουν πως ο SARSA(λ) ήταν ικανός να μάθει μια σχεδόν-βέλτιστη συμπεριφορά, η οποία απέδωσε ικανοποιητικά ακόμη και στις άγνωστες συνθήκες, οδηγώντας το UAV σε διαδρομές χωρίς συγκρούσεις με εμπόδια. Η πολιτική του LSPI απαίτησε λιγότερο χρόνο μάθησης και η απόδοσή της έδειξε καλές προοπτικές, δεν ήταν όμως τόσο αποτελεσματική, καθώς σε κάποιες περιπτώσεις οδήγησε σε συγκρούσεις στις άγνωστες συνθήκες. Η εργασία στο σύνολό της έχει υλοποιηθεί χρησιμοποιώντας το Robot Operating System (ROS) και το περιβάλλον ρομποτικής προσομοίωσης Gazebo.

Acknowledgements

Firstly, I would like to express my gratitude and appreciation to all my professors at the Technical University of Crete and especially to Michail G. Lagoudakis and Panagiotis Partsinevelos whose guidance and support have been invaluable throughout this work.

I would also like to thank my colleagues Dimitris and Angelos whose assistance was more than enough.

I cannot forget to thank Irho, Gerasimos, Giorgos, Maria, Iosif, Roza, Giannis, Marietta, Chrisa, and Aria for all the memories we shared these beautiful years.

Last, but not least, I am grateful to my family for their continuous love, help, and support.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Learning is a fundamental ability of all intelligent beings. The bird that acquires the song of its species is a typical example of learning in the natural environment. Psychologists have conducted thousands of experiments in laboratories to study how animals can learn. Some of the ideas that they proposed inspired researchers to apply them to artificial intelligence and with the help of optimal control theory this trend led to the evolution of reinforcement learning. When thinking of reinforcement learning, the first thing that comes to mind is the interaction with the environment. The learner is not told how to behave, but instead must discover which actions lead to better results by trying them. Furthermore, long-term planning is another crucial idea of reinforcement learning. In many cases, actions may not only affect the immediate situation, but also all the following ones. These two characteristics are the core of reinforcement learning.

In recent years, researchers have achieved many impressive accomplishments by using reinforcement learning. Mnih et al. 2013 have managed to attain superhuman performance in many Atari games using only visual information. Furthermore, Silver et al. 2017 developed AlphaGo, a system that was able to defeat the world champion in the game of Go. All of that and many more show that reinforcement learning has the potential to solve many challenging problems.

Autonomous navigation has always been an interesting and challenging problem in robotics. For any mobile robot, the ability to autonomously navigate in its environment is crucial to complete any kind of task. Being able to avoid obstacles and/or find a safe route towards the target location in an unknown environment is critical in many applications, such as search and rescue operations, mobile robots in industrial environments, and mapping of geographical areas.

1.1 Thesis Contribution

This thesis describes an approach for the autonomous navigation of a UAV in unknown environments using reinforcement learning. While the most common tech-

niques to solve this problem use Simultaneous Localization and Mapping (SLAM) algorithms that consist of self-localization, map-building, and path planning, an alternative mapless method based on reinforcement learning can also be effective especially in very large environments.

In reinforcement learning, the agent learns how to behave through a trial-and-error procedure. It interacts with the environment and by receiving rewards (or penalties) it learns which actions are good and which are not in each state. In our problem, the actions are velocity commands that control the UAV and states are ranging measurements from a LiDAR sensor. Both actions and states contain some form of stochasticity that represent real-world circumstances. Our goal is for the UAV to navigate the environment successfully without crashing into walls.

We implemented and compared two different algorithms: SARSA(λ) (Rummery and Niranjan 1994), which is an online, model-free, on-policy algorithm that deals with the limitations of temporal difference and Monte Carlo learning and Least-Squares Policy Iteration (LSPI) (Lagoudakis and Parr 2003) which is a batch (offline), model-free, off-policy algorithm. Due to the large state space of the problem, we combined these algorithms with tile coding, a linear representation that was used to approximate the value function. Furthermore, we used decaying ϵ-greedy policy to deal with the exploration and exploitation trade-off.

The entire project was implemented in the Robot Operating System (ROS) framework and was tested in the Gazebo robot simulator on three different in complexity tracks. The training was conducted only on the first and simplest track of the three. Then, we tested how each policy performs on the other two unknown and more challenging tracks.

Results show that SARSA(λ) can find an efficient policy with good generalization that performs well in unknown and more complex environments in most situations. LSPI also produced a promising policy, but not as efficient as the previous one.

1.2 Thesis Outline

In Chapter 2, we present all the background information needed for this thesis. We give an overview of reinforcement learning. Specifically, we state the basic reinforcement learning problem and give all the required definitions. Moreover, we explain the use of the ROS framework and the Gazebo simulator. In addition to that, we describe the model of the unmanned aerial vehicle that was used. In Chapter 3, we specify the navigation problem and its importance in robotics and we present some related work. Chapter 4 contains a detailed description of our approach. We explicitly present the exact scenario of the experiments and analyze the model of the environment, the representation methodology, and the proposed

algorithms. The results from our experiments are shown in Chapter 5, in which we also compare the two algorithms. Finally, Chapter 6 sums up the main parts of this thesis and describes potential future work.

Chapter 2

Background

2.1 Reinforcement Learning

Reinforcement learning (RL) (Kaelbling, Littman, and Moore 1996, Sutton and Barto 2018) is an area of machine learning concerned with how agents learn to behave in an unknown environment through trial-and-error. Reinforcement learning differs from both supervised and unsupervised learning in the respect that the agent is not told what actions to take, but instead is "guided" indirectly via a reward signal. The goal of the agent is to find an optimal strategy (behavior) that maximizes the total future reward.

Reinforcement learning is also described as sequential decision-making under uncertainty, which is characterized by a stochastic environment and a long-term goal. This means that the consequences of an action may not be always the same and, may not be visible until later in the future and, for this reason, careful planning is required.

Figure 2.2 shows a very basic reinforcement learning model and how the agent interacts with the environment. At each time step t the agent receives a representation of the environment called state S_t and selects an action A_t. At the next time-step $t+1$ depending on the results of its action the agent receives a feedback called reward R_{t+1} and a new state S_{t+1}. This cycle that continues indefinitely or until the environment reaches a terminal state can be represented as a trajectory that is illustrated in Figure 2.1.

$$S_t \xrightarrow[R_{t+1}]{A_t} S_{t+1} \xrightarrow[R_{t+2}]{A_{t+1}} S_{t+2} \xrightarrow[R_{t+3}]{A_{t+2}} \cdots$$

Figure 2.1: A trajectory. The trajectory from a starting state all the way to a terminal state is called an episode

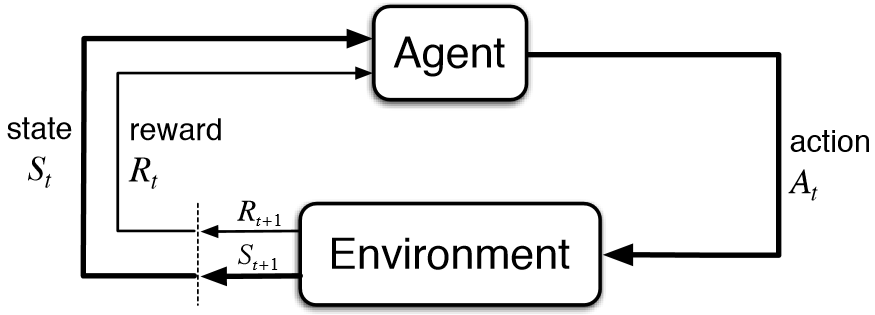


Figure 2.2: A basic reinforcement learning model (figure from Sutton and Barto 2018)

One of the challenges of reinforcement learning is the *exploration and exploitation trade-off*. To maximize the total amount of reward, the agent must choose actions that knows from experience, are effective (exploit). However, to find such actions the agent must first gain experience by trying actions that has never tried before (explore). This dilemma becomes even more complicated in a stochastic environment. Therefore, a balance between exploration and exploitation is required to find an optimal behavior.

2.1.1 Reward and Return

To determine the goal of reinforcement learning we use a *reward signal*. After each action that the agent selects, it receives a feedback that indicates how "good" or "bad" this action was. Although the reward function can be stochastic, for simplicity reasons we will assume it is deterministic.

$$R : S \times S \times A \to \mathbb{R}$$

The goal of the agent is to maximize the expected cumulative reward. This idea is the core of reinforcement learning and is based on the *reward hypothesis*, which states that all goals can be described in this way. Designing a good reward function is a challenging task that is often solved by trial-and-error.

To define the total long-term reward of a trajectory after a time-step t, we use a metric called the *return* G_t. The most commonly used model for this metric is the infinite horizon discounted reward model. The *discount factor* γ is a value between $(0, 1]$ and is used not only to bound the return, but also to model the uncertainty about the future.

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.1}$$

2.1.2 Markov Decision Process

A *finite Markov Decision Process* (MDP, Puterman 1994, Bertsekas 2005) is a mathematical model that fully describes the environment in a reinforcement learning problem. For example, given a state and an action, we can use the model to predict the reward and the next state. More precisely an MDP is a tuple that consists of the following elements:

$$MDP = (S, A, P, R, \gamma, D)$$

- The *state space* S is a finite set of all possible states of the environment.
- The *action space* A is a finite set of all possible actions that the agent can choose. Each state can have its own action space, but for consistency reasons, we will assume that all actions are available in all states.
- The *transition model* P is a matrix that defines transition probabilities from all states s to all successor states s' given the action taken a.

$$P^a_{ss'} = Pr(S_{t+1} = s' | S_t = s, A_t = a)$$

- The *reward model* R defines the reward that the agent receives at each time step.
- The *discount factor* γ is used to define the return.
- The starting state in an MDP is not always the same. For this reason, some MDPs contain the *initial state distribution* D, which is a probability distribution for all the possible starting states.

An important property of MDPs, is the *Markov property* which denotes that the next state is independent of the history given the current state. In other words, the current state holds all the information we need. This is expressed mathematically in Equation 2.2:

$$Pr(S_{t+1} | S_1, S_2, ..., S_t) = Pr(S_{t+1} | S_t) \tag{2.2}$$

There are many extensions to the MDPs, like the Partially Observable Markov Decision Processes or the Continuous Time Markov Decision Processes.

2.1.3 Policy and Value Function

A *stationary policy* π is the agent's strategy. It determines how the agent behaves in different situations. It's a function that maps all possible states to some action.

$$\pi : S \to A$$

In general the policy can also be stochastic in which case $\pi(a|s)$ denotes the probability of choosing action a in state s.

$$\pi : S \to PD(A)$$

The goal of reinforcement learning is to find an optimal policy π^* that maximizes the expected cumulative reward and not the immediate reward. This means that sometimes it has to sacrifice immediate reward for better long-term rewards. For this purpose, we use a *state-value function* V_π, which for state s predicts the total future reward that the agent is expected to accumulate, if we follow the policy π.

$$V_\pi : S \to \mathbb{R}$$

As mentioned before, we are interested in the *expected* return. Therefore, we can define the state-value function for a given policy in the following way:

$$V_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \tag{2.3}$$

Similarly, we can define the *action-value function* Q_π that predicts the expected total future reward, if in state s we select action a and then follow policy π.

$$Q : S \times A \to \mathbb{R}$$

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \tag{2.4}$$

The problem of finding a policy that maximizes the expected return can be divided into two sub-problems. The first one is called *prediction* which is how to calculate the value function of a given policy. The second one is called *control* which is how to find an optimal policy. It is clear that to find an optimal policy we must first be able to evaluate it. As a result, the problem of control involves the problem of prediction.

The value function can be expressed recursively in terms of the next time-step state-value function.

$$\begin{aligned}
V_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)\Big(R(s,a,s') + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\Big) \\
&= \sum_a \pi(a|s) \sum_{s'} P(s'|s,a)\Big(R(s,a,s') + \gamma V_\pi(s')\Big)
\end{aligned} \tag{2.5}$$

Equation 2.5 is called the *Bellman equation* for V_π. To better understand this equation we can use a back-up diagram that is shown in Figure 2.3. White circles

represent states and black circles actions. Starting from state s, the agent can choose from a set of different actions, based on its policy π. By selecting these actions, the agent can transition to a set of different states, based on the transition model P of the MDP. It also receives a reward, based on the starting state s, the action it chose a, and the next state s'.

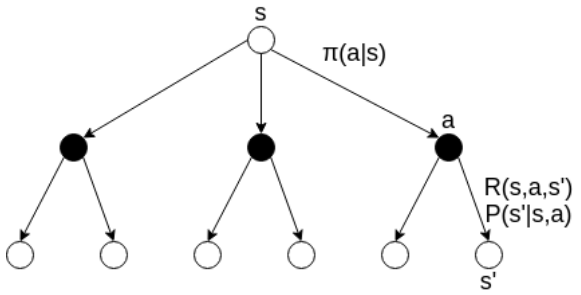


Figure 2.3: Back-up diagram for V_π

The Bellman equation for Q_π can be derived in a similar way.

$$\begin{aligned}
Q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...|S_t = s, A_t = a] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s, A_t = a] \\
&= \sum_{s'} P(s'|s,a)\Big(R(s,a,s') + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\Big) \\
&= \sum_{s'} P(s'|s,a)\Big(R(s,a,s') + \gamma \sum_{a'} \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s', A_{t+1} = a']\Big) \\
&= \sum_{s'} P(s'|s,a)\Big(R(s,a,s') + \gamma \sum_{a'} \pi(a'|s') Q_\pi(s',a')\Big) \qquad (2.6)
\end{aligned}$$

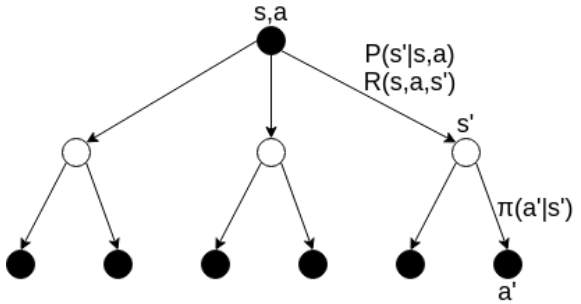


Figure 2.4: Back-up diagram for Q_π

In the above equations, we assumed a stochastic policy and a reward function dependent on the current state, the action, and the next state. The Bellman equation can be written in a similar form for different definitions of the reward function and the policy.

The Bellman equations for V_π and Q_π are linear systems and can be expressed in matrix form. Therefore, to find the actual V or Q values of a policy we can solve the systems either directly or iteratively.

2.1.4 Optimality

To solve an MDP, we have to find an optimal policy π^*. A policy π is better or equal to a policy π', if and only if it has a greater or equal value function for every state.

$$\pi \geq \pi' \Leftrightarrow V_\pi(s) \geq V_{\pi'}(s),\ \forall s \in S \tag{2.7}$$

In general, there may be more than one optimal policy, but they all share the same optimal value function, V^* or Q^*.

$$V^*(s) = \max_\pi V_\pi(s),\ \forall s \in S \tag{2.8}$$

$$Q^*(s,a) = \max_\pi Q_\pi(s,a),\ \forall s \in S,\ \forall a \in A \tag{2.9}$$

Moreover, we can write the Bellman equations for V^* and Q^* that are called *Bellman optimality equations*. Intuitively, we can think that the optimal value function of a state must be equal to the expected return of the best action from that state.

$$V^*(s) = \max_a \sum_{s'} P(s'|s,a)\Big(R(s,a,s') + \gamma V_\pi(s')\Big) \tag{2.10}$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)\Big(R(s,a,s') + \gamma \max_{a'} Q_\pi(s',a')\Big) \tag{2.11}$$

Unlike the Bellman equations, Bellman optimality equations are non-linear systems. For this reason, a direct solution is not possible and our only option is to use iterative algorithms.

The last step is to derive an optimal policy from the value function. At each state there will be at least one action that maximises the value function. Therefore, acting greedily with respect to the value function produces an optimal policy which is always deterministic.

$$\pi^*(s) = \operatorname*{argmax}_a \sum_{s'} P(s'|s,a)\Big(R(s,a,s') + \gamma V^*(s')\Big) \tag{2.12}$$

$$\pi^*(s) = \operatorname*{argmax}_a Q^*(s,a) \tag{2.13}$$

Note that using the action-value function (Equation 2.13) is simpler due to the fact that it contains information about all possible actions for each state. Moreover, it does not require the knowledge of the transition matrix P and the reward function, R which are not always known. For this reason, in control we use the action-value function instead of the state-value function.

2.1.5 Taxonomy of Reinforcement Learning Algorithms

Reinforcement learning algorithms can be classified in some major categories based on their approach, as shown in Figure 2.5.

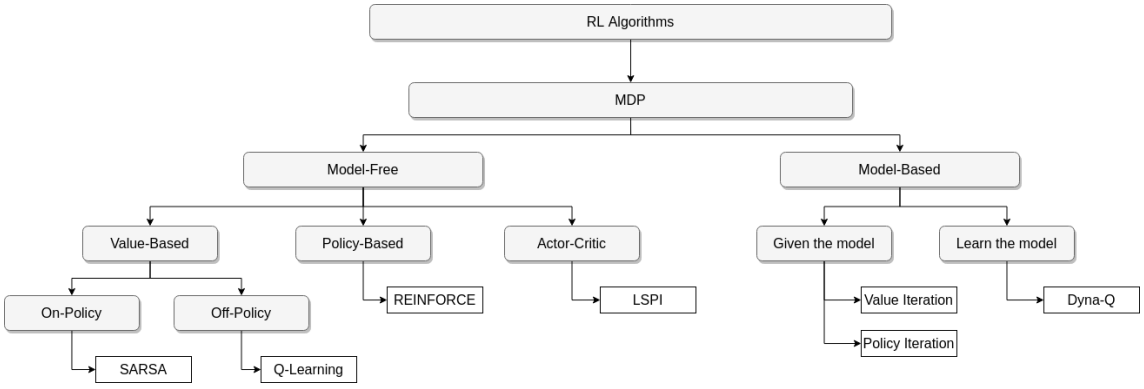


Figure 2.5: Taxonomy of reinforcement learning algorithms

To solve an MDP, we can either make use of the model or use a more direct approach. *Model-based* algorithms utilize the dynamics of the environment (transition function, reward function) to find an optimal policy. If these parameters are known, we can use dynamic programming (e.g. Policy iteration, Howard 1960). However, most of the time the dynamics of the environment are unknown. In this case, we have to learn the model and then solve it by using *planning* (e.g. Dyna-Q, Sutton 1991). On the other hand, *model-free* algorithms do not learn any kind of model, but instead are trying to learn an optimal policy and/or value function directly.

Model-free algorithms can be divided based on the function they are trying to learn. *Policy-based* methods (e.g. REINFORCE, Williams 1992) explicitly learn a policy, while *value-based* methods learn a value function. *Actor-critic* methods (e.g. LSPI, Lagoudakis and Parr 2003) are the combination of the two and learn both a policy (actor) and a value function (critic).

Last, but not least, reinforcement learning algorithms can either be *on-policy* or *off-policy*. On-policy algorithms attempt to improve or evaluate a policy that is used by the agent to behave and generate experience (e.g. SARSA, Rummery and Niranjan 1994). On the contrary, in off-policy algorithms the agent follows a different policy from the one we are trying to evaluate or improve (e.g. Q-Learning, Watkins 1989).

2.1.6 Value Function Approximation

In simple cases, the state space consists of a few states and thus the value function can be represented explicitly in a tabular form. However, most of the time and especially in robotics (Kober, Bagnell, and Peters 2013) the state space is fairly large. This is called *curse of dimensionality* and refers to the exponential increase

of the number of states over the number of dimensions. Moreover, the state space can also be continuous. This can not only cause problems in memory, but in time of convergence. The time needed to visit every state enough times to evaluate it is forbidding. Therefore, the agent must be able to *generalize* its past experiences to similar states. To solve this we use a mathematical technique called *value function approximation* (Busoniu et al. 2010) which tries to integrate function approximation techniques for the function of interest within reinforcement learning. In value-based algorithms, the function of interest is the value function V or Q.

The most commonly used value function approximation is the *parametric* approximation. This approximation uses a finite set of parameters (weights) $\mathbf{w} \in \mathbb{R}^k$ that can be tweaked appropriately in order that the approximate function fits the target function.

$$\hat{V}_\pi(s; \mathbf{w}) \approx V_\pi(s)$$

$$\hat{Q}_\pi(s, a; \mathbf{w}) \approx Q_\pi(s, a)$$

When using a function approximation, we must always have in mind that an optimal solution is not guaranteed. We sacrifice some representation accuracy to lower the storage requirements, since only the parameters $\mathbf{w}$ need to be stored and calculated. There is a large variety of parametric approximation techniques that can be divided into linear (e.g. polynomials, radial basis functions, tile coding) and non-linear architectures (e.g. neural networks).

2.1.7 Dynamic Programming: Policy & Value Iteration

Dynamic Programming refers to a family of iterative algorithms that can calculate the optimal value function by solving the Bellman (optimality) equation. However, they require full knowledge of the MDP (transition function, reward function).

The first algorithm we are going to describe is called policy iteration (Howard 1960). It consists of two main steps. The first one is the *policy evaluation* (prediction) step that calculates the state value function V_π of a given policy π. This is achieved by solving the Bellman Equation 2.5 for each state $s \in S$. Because the Bellman equation is a recursive linear equation, this step amounts to solving a linear system directly of iteratively. The next step is the *policy improvement* step. In this step, we improve the policy by acting greedily, as stated in Equation 2.12. Therefore, by repeatedly altering between policy evaluation and policy improvement, we can slowly approach optimality. This idea is illustrated in Figure 2.6 and appears in many reinforcement learning algorithms.

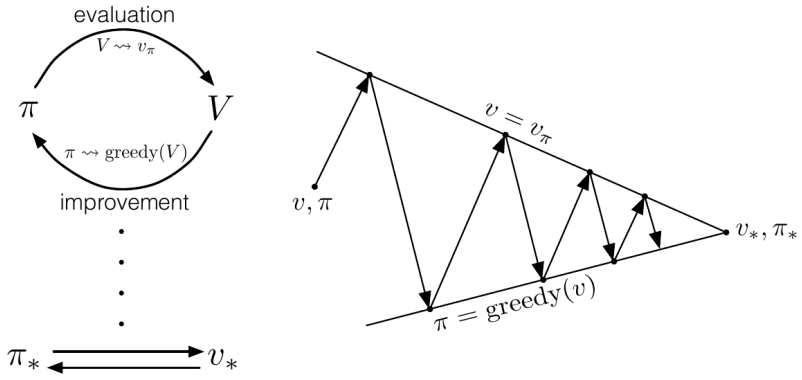


Figure 2.6: Policy iteration approach (figure from Sutton and Barto 2018)

However, policy iteration can be very computationally expensive, although in practice it converges in a small number of iterations. Another popular algorithm that is called Value Iteration (Bellman 1957) does not explicitly store the policy, but instead calculates the optimal state-value function V^* for each state $s \in S$ as stated in the Bellman Optimality Equation 2.10. This procedure needs to be done iteratively, until we have convergence of the recurrence to a certain degree. It should be noted that the number of recursive updates required for convergence may be extremely large.

2.1.8 Monte Carlo Learning

Monte Carlo (MC) learning refers to a class of model-free reinforcement learning methods that solve an MDP by averaging sample returns.

We begin by addressing the problem of prediction. As shown from Equation 2.3, an intuitive way of calculating the state-value function is to generate episodes following the policy π and for every time-step t calculate the return G_t that corresponds to a specific state s. Then, the value function for each state is the average of the corresponding returns.

In order to find an optimal policy all we have to do is improve the current policy by acting greedily as stated, in Equation 2.13. For this reason, the use of the action-value function is a one-way road. This approach is illustrated in Figure 2.7 and resembles the policy iteration method that was presented in Section 2.1.7. By repeatedly altering between these two steps we can slowly approach optimality. MC learning can also be combined with value function approximation to handle cases in which the state space is large and/or continuous (Appendix A).

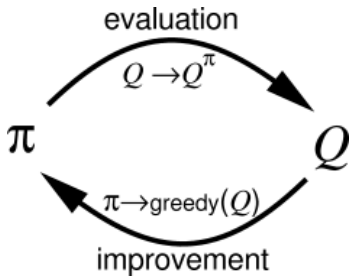


Figure 2.7: MC control (figure from Sutton and Barto 2018)

However, MC learning has some significant disadvantages. Firstly, the returns can only be calculated only once the episode has finished. This means that all episodes must terminate no matter what (*episodic tasks*) and that learning occurs after the end of the episode and not in the process. Lastly, in many applications, where each episode may contain many actions that can result in many different states and rewards, the returns are "noisy". Therefore, the MC estimator has a high variance and slow convergence.

2.1.9 Temporal Difference Learning

Temporal difference (TD) learning is another popular class of model-free reinforcement learning algorithms. TD learning uses the following general update rule to calculate the value function of a policy.

$$NewEstimate \leftarrow OldEstimate + StepSize \underbrace{\big(Target - OldEstimate\big)}_{\text{error}} \tag{2.14}$$

The expression $\big(Target - OldEstimate\big)$ is the error of the estimate and is reduced by taking a small step towards the target. Moreover, it is often desirable to decay the step size as the time passes, in order to prevent overshooting of the target. For example, a learning rate that follows a logistic curve (sigmoid) is very commonly used:

$$\alpha = \frac{\alpha_0}{1 + e^{k(ep - ep_0)}}$$

where ep is the number of the current episode, α_0 is the initial learning rate, k is the steepness of the curve and ep_0 is the midpoint of the curve.

In MC learning the target is G_t, while in TD learning the target is $R_{t+1} + \gamma V(S_{t+1})$. Therefore, in order to estimate the value function of a policy π, we need to generate (s, r, s') samples following this policy and then use the TD update rule:

$$V(s) \leftarrow V(s) + a\big(r + \gamma V(s') - V(s)\big),\ a \in (0, 1] \tag{2.15}$$

This method is called TD(0) or one-step TD, because the target is expressed as one-step roll-out of the return. The error $\big(r + \gamma V(s') - V(s)\big)$ is known as Temporal Difference (TD).

In order to perform control, we need to use the action-value function. In this thesis, we used a well-known on-policy TD control algorithm that is called SARSA (Rummery and Niranjan 1994). In each time step, it generates a (s, a, r, s', a') sample by acting greedily and then uses the rule 2.16 to update the value function.

$$Q(s,a) \leftarrow Q(s,a) + a\big(r + \gamma Q(s',a') - Q(s,a)\big) \tag{2.16}$$

The update rules 2.16 and 2.15 are viable, only when the value function can be expressed in tabular form. If we are using value function approximation, we rely on stochastic gradient descent (SGD) methods to update the weights appropriately (Appendix A).

$$\mathbf{w} \leftarrow \mathbf{w} + a\big(r + \gamma \hat{Q}(s',a';\mathbf{w}) - \hat{Q}(s,a;\mathbf{w})\big)\nabla \hat{Q}(s,a;\mathbf{w}) \tag{2.17}$$

Unlike MC learning, TD learning can be applied to both episodic and *continuing tasks*. Moreover, it does not suffer from high variance, since it does not use the full return. However, it uses another estimation to update the value function. This is called bootstrapping and as a result TD estimators are biased.

2.2 Robot Operating System

Robot Operating System (ROS) is a framework (collection of software libraries) that is used in robot software development. ROS provides tools for package management, low-level control of various devices, communication between devices and processes, debugging, and visualization. All those tools aim to hide abstraction, while keeping the required flexibility, and simplify the development of a complex robotic application. Another advantage of ROS is its integration ability with the Gazebo robot simulator. Testing a robotic application in real hardware can be both costly and dangerous. Gazebo is a powerful robotic simulator that offers the ability to test algorithms in various robots, scenarios, and environments.

ROS architecture can be represented as a graph. The main components of ROS are described below:

- A *Node* is a process that represents actuators, sensors, or parts of the robot's logic (e.g. decision making). Having nodes that control different parts of the robot is crucial in complex robotic applications. Furthermore, nodes have the ability to communicate with each other.

- A *Message* is a data type that is used in the communication between nodes. It contains fields with the necessary data to describe information like a sensor measurement or the position of the robot. It is possible to construct complex messages, like the one in Figure 2.8, that contain other messages.

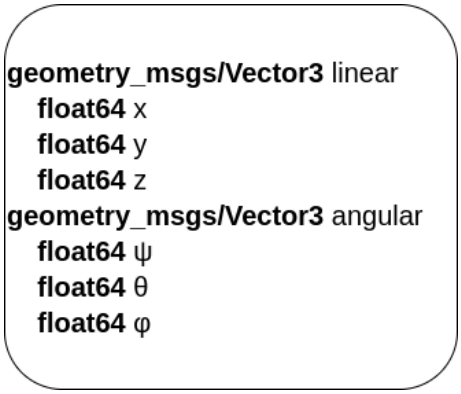


Figure 2.8: A geometry_msgs/Twist message

- One way that the nodes can communicate is by using a publish-subscribe messaging pattern. ROS messages are published to *topics* by the nodes. For a node to receive a message, it has to subscribe to the corresponding topic. There is no limitation to the number of topics that a node can subscribe to or publish to.
- *Services* is another way that nodes can communicate with each other. In contrast to topics, a service is a two-way communication, where one node (client) sends a request to another node (service) and receives a response.

Figure 2.9 shows the architecture of a simple ROS application represented by a graph. The circles represent nodes and the arrows topics, while their direction indicates the subscriber and the publisher. In this scenario, we are trying to control a virtual turtle. The node /teleop_turtle receives commands from the keyboard and publishes messages of type geometry_msgs/Twist to the topic /turtle1/command_velocity. The node /turtlesim that has subscribed to this topic, receives these messages, and moves the turtle accordingly.

Figure 2.9: An example of the architecture of a ROS application

2.3 Unmanned Aerial Vehicle

Hector quadrotor (Meyer et al. 2012) is a framework that contains packages related to modeling, control, and simulation of quadrotor UAV systems and is integrated with ROS and Gazebo simulator. Some characteristics of the simulated UAV are described below.

The pose of the UAV in three-dimensional space is described by its position and orientation. The position is a Euclidean vector that represents a point in space (world frame), while the orientation is a vector of Euler angles that refer to the rotation of the body around the corresponding axis.

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \ \boldsymbol{\theta} = \begin{bmatrix} \psi \\ \theta \\ \phi \end{bmatrix}$$

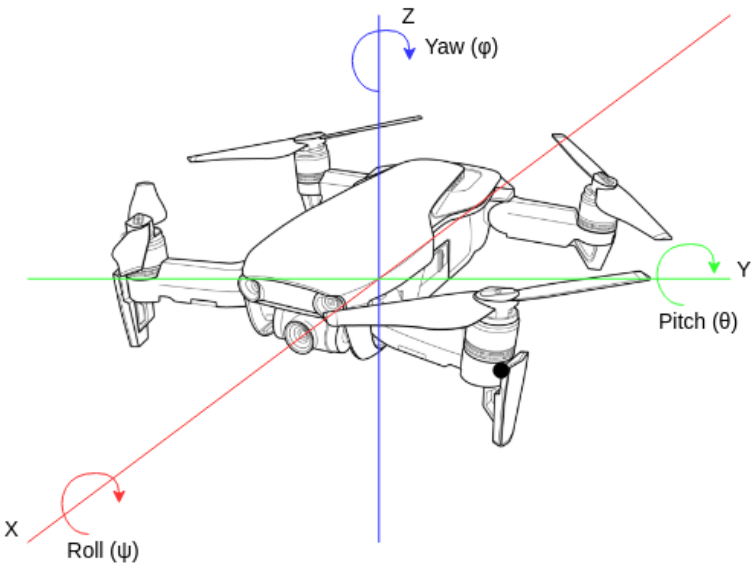


Figure 2.10: The pose of the UAV in three-dimensional space

The motion of the UAV is the rate of change of its pose. Linear velocity describes the rate of change of the position along each axis, while angular velocity the rate of change of the orientation. Velocity commands are a way of controlling the UAV. Note that unlike the pose, the velocities refer to the coordinate frame of the UAV.

$$\mathbf{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \ \boldsymbol{\omega} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The UAV is equipped with a Light Detection and Ranging system (LiDAR). LiDARs are devices that measure distances by emitting laser light in a wide range and then capturing the reflection of that light. By measuring the time that the light takes to return to the receiver, we can calculate the distance from that object.

The LiDAR used in the simulation is the Hokuyo UTM-30LX whose scanned area and specifications are shown in Figure 2.11.

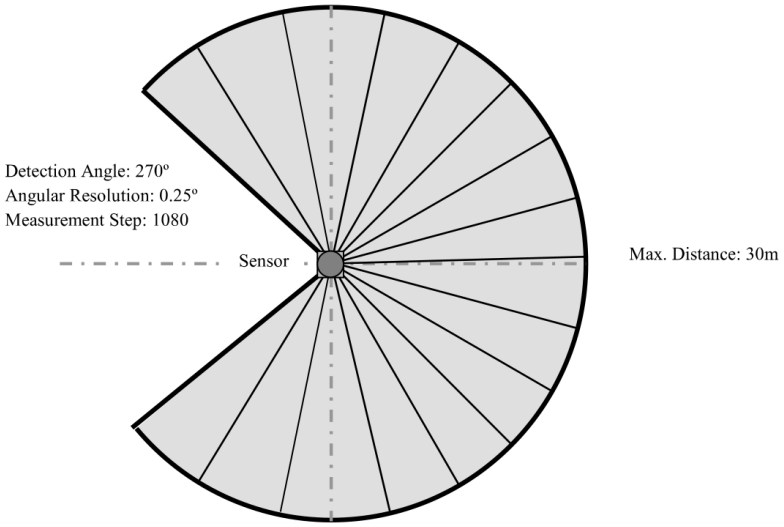


Figure 2.11: Hokuyo UTM-30LX scanning area

Additionally, to simulate real-world circumstances, all the above measurements and commands share a common first-order Gauss Markov error model. Each simulated measurement or command y is given by

$$y = \hat{y} + w$$

where $\hat{y}$ is the true value and w is white Gaussian noise. Last, but not least, it is important to state that all the above quantities are measured in the International System of Units (SI).

Chapter 3

Problem Statement

3.1 Autonomous Navigation of UAVs in Unknown Environments

When thinking of a mobile robotic application, and specifically an UAV, autonomous navigation is probably the most crucial ability that it has to possess. Moreover, in real-world applications, certain constraints make the navigation of the UAV very difficult. The exact map of the area is not always known beforehand. Furthermore, the environment can also change dynamically even during the flight, making the problem even more challenging. Search and rescue missions (e.g. a collapsed building) are an example in which such scenarios can occur. Therefore, the UAV must be able to navigate successfully in an unknown environment and have the ability to adapt to any change that might happen.

3.2 Related Work

Autonomous robotic navigation has always been an area of great research interest. When the map is known, the robot can easily navigate by using a path planning algorithm. In an unknown environment, the most standard approach is to use simultaneous localization and mapping (SLAM) algorithms (Grzonka, Grisetti, and Burgard 2009, Huang et al. 2016) that can construct a map of the environment, while keeping track of the robot's location within it. However, learning a map of an environment that stretches for kilometers (e.g. drone delivery services) and then plan a path based on it requires a lot of memory space and may be computationally expensive.

In recent years, the development of powerful reinforcement learning algorithms and their ability to solve difficult problems have urged researchers to apply them in autonomous navigation problems. Imanberdiyev et al. 2016 have developed a real-time model-based reinforcement learning algorithm, TEXPLORE, that can learn a

trajectory from a starting position to a goal by using a GPS signal. In this scenario, the UAV must also monitor its battery level and alternate its route, when necessary to recharge them. Moreover, the UAV cannot reach the target location without recharging its batteries at least once in the recharging station which location is unknown. Their approach was tested in Gazebo and proved to be more efficient than Q-Learning. However, the map is represented as a grid (discrete state domain) and does not contain any obstacles, thus making the problem less challenging.

Wang et al. 2017 also model the autonomous navigation problem as a discrete-time continuous control problem. They argue that the navigation task is a Partially Observable Markov Decision Process (POMDP) and that the Recurrent Deterministic Policy Gradient (RDPG) algorithm that is used in such problems is not efficient when used with a memory replay. As a result, they introduce a new approach called Fast-RDPG that is based on an actor-critic architecture with function approximation. They experimented in simulation environments where the UAV flies in constant altitude and velocity and uses range finder sensors and a GPS to navigate and reach the target location. Results show that the proposed algorithm is able to learn an efficient policy much faster which was tested in five virtual unknown environments with different types of complex obstacles.

Yijing et al. 2017 propose an approach called Adaptive and Random Exploration (ARE) in order to solve the UAV path planning problem in unknown scenarios. Their architecture consists of three modules. The learning module implements a 2-layer neural network and with a deep Q-Learning algorithm that uses the UAV's position, distance from target and obstacles learns a strategy of action selection. The trap-escape module helps the UAV escape dangerous situations via a random tree search algorithm. And finally, the action module receives instructions from the other two modules and let the UAV to take actions. Their method was tested in four different simulated unknown maps and was able to learn fast an accurate policy.

Pham et al. 2018 also applied The framework of reinforcement learning was also applied by Pham et al. 2018 to allow the UAV to successfully navigate in an unknown environment, which exact mathematical model is not available. They modeled the problem as an MDP and used a traditional Q-Learning algorithm in order to teach the UAV to reach a target location. The state-space consists of the position of the UAV in a discretized grid-like map and at each time step, the UAV can choose to move in one of the four cardinal directions. They also combined the learning algorithm with a PID controller in order to deal with the instability issues during movement. Their proposed approach was tested in both simulation and real implementation and showed that the UAV can successfully reach the target location using the minimum number of steps. However, in their experiments, the map was obstacle-free and consisted only of a few states.

Furthermore, Walker et al. 2019 presented a framework for UAV navigation in

indoor environments. The proposed method separates the problem into two sub-problems. The first one is the global planning problem which is modeled as a Markov Decision Process (MDP) and uses a discretized map of the environment to provide macro actions. The second problem is the local planning problem and is modeled as a Partially Observable Markov Decision Processes (POMDP) in which must perform continuous actions in order to search a local environment and avoid any obstacles. In order to solve these problems, they implemented a deep reinforcement learning algorithm. Experiments that were performed on Gazebo show that the UAV was able to transit between rooms and successfully search each room while avoiding the obstacles.

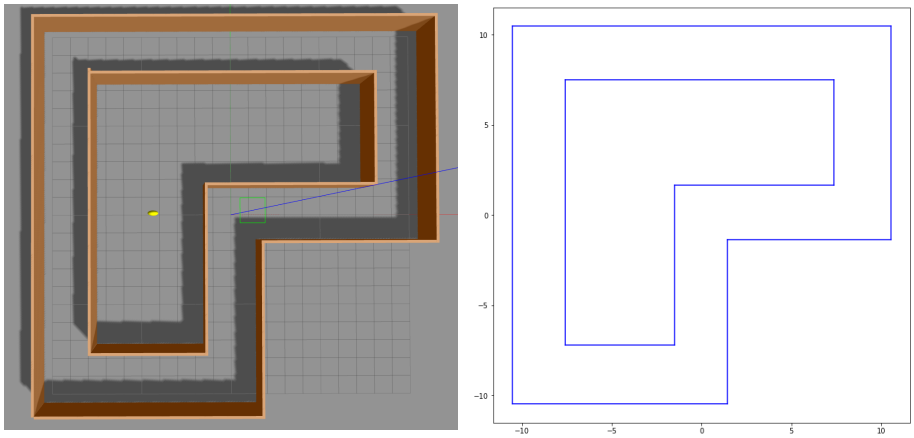
Zhang et al. 2013 addressed the path planning problem of multiple UAVs from the perspective of reinforcement learning. They argue that traditional reinforcement learning algorithms such as Q-Learning are unable to learn the geometric distance information or handle the case of multiple UAVs. For this reason, they propose a new algorithm called Geometric Reinforcement Learning (GRL) that utilize the geometric distance and risk information from detection sensors and other UAVs in order to build a general path planning.

Chapter 4

Our Approach

4.1 Model Description

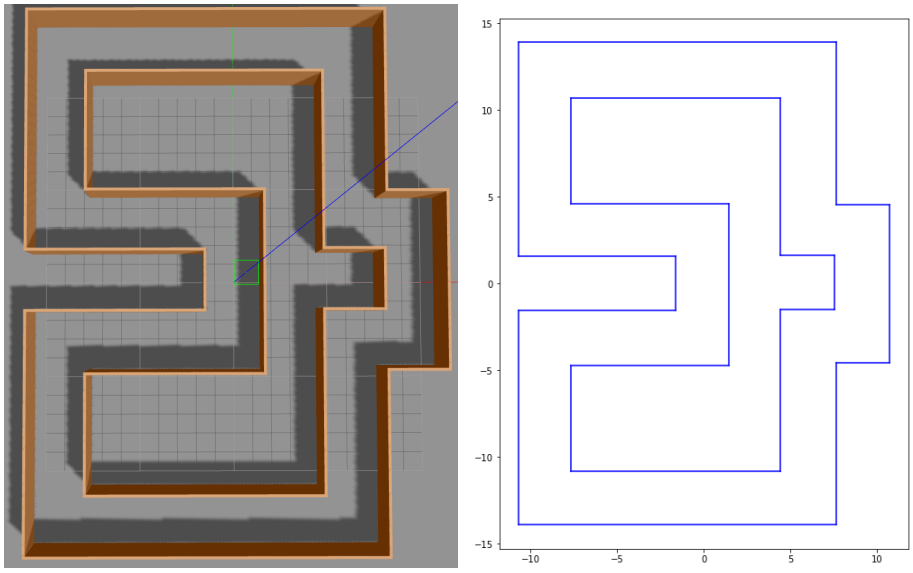
In this section, we describe the exact parameters of the MDP of the environment we implemented to test our approach. The goal is for the UAV to automatically navigate and complete the maze-like track without crashing into the wall. The tracks that were created in Gazebo are shown in Figures 4.1, 4.2 and 4.3. Note that each track has increasing complexity. The second one has more consecutive turns than the first and the third track has additionally two diagonal corridors.



(a) As shown in Gazebo

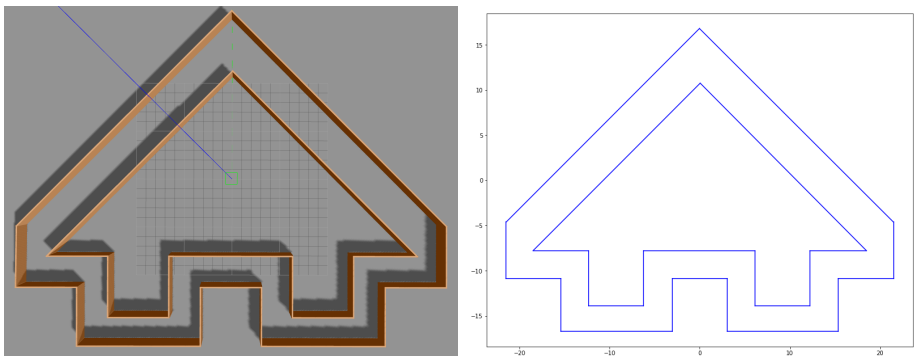
(b) As a plot

Figure 4.1: The first track



(a) As shown in Gazebo (b) As a plot

Figure 4.2: The second track



(a) As shown in Gazebo (b) As a plot

Figure 4.3: The third track

At the start of the episode, the UAV takes off and reaches its starting pose. This is achieved with the help of the implemented, controllers that use the ground truth pose of the UAV. When it has reached its starting pose, our algorithm takes control of the UAV. The flying altitude remains constant during the flight ($z = 1.5m$) which allows us to represent the problem in the 2D space. The starting pose (x position, y position, yaw angle) and thus the initial state of the UAV is chosen randomly, but with a safe distance from the walls. This randomness allows the UAV to visit more a variety of states and learn to complete the track both clockwise and anticlockwise.

While implementing our approach, we came up against an interesting problem: At each time step, the time required for the UAV to learn and select a new action is not inconsiderable. Depending on the algorithm, a high action selection time can cause various problems. During this time, the state of the environment changes and

the action selected may no longer be an optimal one. For this reason, we tried to optimize the algorithms as much as possible to have a negligible action selection time. Moreover, a low velocity for each action has been chosen to ensure a low rate of change in the environment.

As mentioned in Section 2.3, a LiDAR was used to detect the obstacles. However, due to its high dimensionality (1080 measurements), only a few measurements were used. We experimented with two different state spaces. The first one is shown in Figure 4.4. Apart from the first perspective measurement d_2, it consists of two more on each side with each one differing from the previous by $\theta = \frac{\pi}{4} rad$. These five continuous variables make up the first state space of the MDP. The second state space is like the first, but with reduced dimensionality. More precisely it contains only the three front measurements d_1, d_2, d_3. Due to the small number of measurements, in a real-life application, the LiDAR could be replaced by different, less expensive ranging sensors (e.g. ultrasonic sensors). However, in the simulation, we used the LiDAR to have more versatility.

$$S_1 = \{(d_0, d_1, d_2, d_3, d_4)\},\ 0.2 \leq d_i \leq 30 \tag{4.1}$$

$$S_2 = \{(d_1, d_2, d_3)\},\ 0.2 \leq d_i \leq 30 \tag{4.2}$$

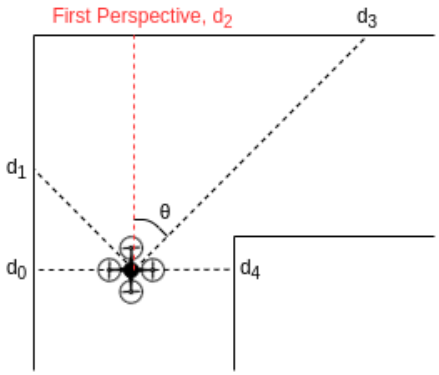


Figure 4.4: A top down (2D) perspective of the UAV showing the LiDAR measurements that were used

The action space is composed by velocity commands, as described in Section 2.3. Unlike the state space, the action space is discrete and consists only of 3 actions. Due to the fact that the UAV is flying at constant altitude (2D representation) the commands consist only of linear velocity along the x axis and yaw angular velocities. Moreover, each command lasts for 0.4 seconds.

$$A = \{forward = [0.5, 0],\ yaw\_right = [0.1, 0.5],\ yaw\_left = [0.1, -0.5]\} \tag{4.3}$$

The reward function is relatively simple. When any of the 1080 LiDAR measurements have a value lower than 0.4, we assume that the UAV has crashed into a

wall. In this case, the episode terminates and the UAV receives a penalty of -200. To encourage the UAV to move forward, every time it selects the corresponding action, it receives a reward of $+5$. Furthermore, left or right yaw rotation is penalized with a small value of -0.5. In this way, the UAV is encouraged to move forward as much as possible and only turn to avoid collisions, by performing the minimum number of rotations. Last, but not least, even though both algorithms can deal with continuing tasks, the episode terminates after 500 actions to avoid infinite-long episodes.

As stated in Section 2.3, to present a more realistic environment, we add white Gaussian noise w to both commands and measurements.

$$w \sim \mathcal{N}(0, 0.01)$$

In order to deal with the exploration and exploitation trade-off, we use a popular exploration method called *decaying ϵ-greedy*. According to this method, we behave greedily most of the time, but in random time steps (probability ϵ) we explore by selecting a random action among all available actions with equal probability. This ensures that asymptotically all actions are selected enough times in order to approach optimality. Furthermore, it is desirable to explore more in the early episodes and as time passes acting more and more greedily. For this reason, probability ϵ is diminishing with each episode. This approach is expressed mathematically with Equations 4.4 and 4.5. The practical advantage of this method is that it offers a very computationally effective, yet powerful, way of exploration.

$$\pi(s) \leftarrow \begin{cases} \text{argmax}_{a'} Q(s, a') & \textit{with probability } 1 - \epsilon \\ \textit{random action} & \textit{with probability } \epsilon \end{cases} \tag{4.4}$$

$$\epsilon = \frac{1}{episode + 1} \tag{4.5}$$

4.2 Linear Function Approximation

Due to the high dimensionality of the state space and the fact that it consists of continuous variables, we used linear function approximation (Lagoudakis 2017) to represent the value function $\hat{V}_\pi(s; \mathbf{w})$ or $\hat{Q}_\pi(s, a; \mathbf{w})$ as a linear weighted combination of k basis functions.

$$\hat{V}_\pi(s; \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(s) = \sum_{i=1}^{k} w_i \phi_i(s) \tag{4.6}$$

$$\hat{Q}_\pi(s, a; \mathbf{w}) = \mathbf{w}^T \boldsymbol{\phi}(s, a) = \sum_{i=1}^{k} w_i \phi_i(s, a) \tag{4.7}$$

The vectors

$$\boldsymbol{\phi}(s) = \begin{bmatrix} \phi_1(s) \\ \phi_2(s) \\ ... \\ \phi_k(s) \end{bmatrix}, \ \boldsymbol{\phi}(s,a) = \begin{bmatrix} \phi_1(s,a) \\ \phi_2(s,a) \\ ... \\ \phi_k(s,a) \end{bmatrix}$$

are called *feature vectors* and represent every state or state-action pair. The gradient of the value function can be easily calculated.

$$\nabla_{\mathbf{w}} \hat{V}_\pi(s;\mathbf{w}) = \boldsymbol{\phi}(s)$$

$$\nabla_{\mathbf{w}} \hat{Q}_\pi(s,a;\mathbf{w}) = \boldsymbol{\phi}(s,a)$$

Linear methods are interesting for various reasons. Even though non-linear methods (neural networks) offer a better generalization, linear methods have their own advantages. First of all, they are very easy to implement. Furthermore, in many algorithms, linear approximation offers convergence guarantees. Moreover, they can be computational efficient, due to the simple gradient of the feature vector. Last, but not least, when a linear method does not work it is fairly easy to understand why, by examining the value of the feature vector. However, the linear form does not take into account any interaction between the features and their construction requires a careful study of each problem separately.

There are many ways to construct the feature vector. Section 4.2.1 presents the method that we used. However, this method describes how to construct features $\boldsymbol{\phi}(s)$ using only state s. In order to construct features for every state-action pair $\boldsymbol{\phi}(s,a)$ we use the same basis functions $\boldsymbol{\phi}(s)$ repeated for each action, so that each action has its own parameters. The derived feature vector has the following form:

$$\boldsymbol{\phi}(s,a) = \begin{bmatrix} I(a=a_1)\phi_1(s) \\ ... \\ I(a=a_1)\phi_k(s) \\ ... \\ I(a=a_n)\phi_1(s) \\ ... \\ I(a=a_n)\phi_k(s) \end{bmatrix}$$

where I is the indicator function: $I(TRUE) = 1$, $I(FALSE) = 0$. Therefore, depending on the action a, only one block of features is active within $\boldsymbol{\phi}$ vector.

4.2.1 Tile Coding

Tile coding is a flexible and computationally efficient feature representation for multi-dimensional continuous spaces. In tile coding we discretize the state space into partitions, called tilings, and each element of the partition is called a tile. A

tile can either be active (equal to 1) or inactive (equal to 0) and for a specific tiling, a point is represented by the tile (feature) that falls into.

However, to have generalization, tile coding requires multiple overlapping tilings that offset from each other by a fraction of a tile width that is indicated by a displacement vector. A simple case of tile coding is shown in Figure 4.5. In this example, the continuous 2D state space is partitioned in four different ways resulting in four tilings. The point we want to represent (white dot) falls into exactly four tiles, one for every tiling. These four tiles represent four active features in the feature vector. Specifically, in this tile coding we have $4 \times 4 \times 4 = 64$ total features. When this state occurs, all of the features will be 0 except for the four corresponding tiles that the point falls into.

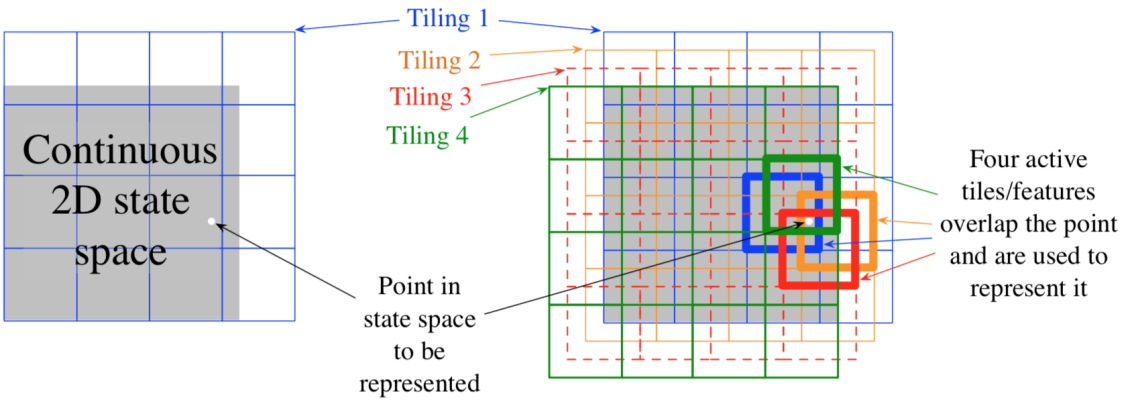


Figure 4.5: An example of tile coding (figure from Sutton and Barto 2018)

For every possible state, the number of active tiles always equals the number of tilings. This allows for an easy, intuitive way of setting the step-size parameter a. For example, $a = \frac{1}{10n}$, where n is the number of tilings, will move the parameters one-tenth of the way to target, while similar states will be moved less. Another advantage of tile coding is that due to its binary representation of the features, the value function of a state can be easily calculated. Instead of performing k multiplications and sums as indicated in Equations 4.6 and 4.7 we simply need to calculate the $n \ll k$ indices of the active features and then add up the corresponding n weights.

4.3 SARSA(λ)

λ-return algorithms are the middle ground between MC learning and TD learning that combine the advantages of both worlds. We begin by presenting the n-step methods and then extend that idea to derive the λ-return methods. A mechanism that is called eligibility traces can be combined with SARSA to obtain an efficient learning algorithm.

4.3.1 n-step Methods

n-step methods offer a unification between MC learning and TD learning. As mentioned in Sections 2.1.8 and 2.1.9, MC learning methods update the value function of a state by using the full sequence of rewards from that state. On the other hand, TD learning methods use only the next reward and then bootstrap from the value of the next state. An intermediate solution would be to use the n next rewards and then bootstrap from the value of the state n steps later. Figure 4.6 shows the backup diagram of the n-step methods. It illustrates the type of return that is used for different values of n.

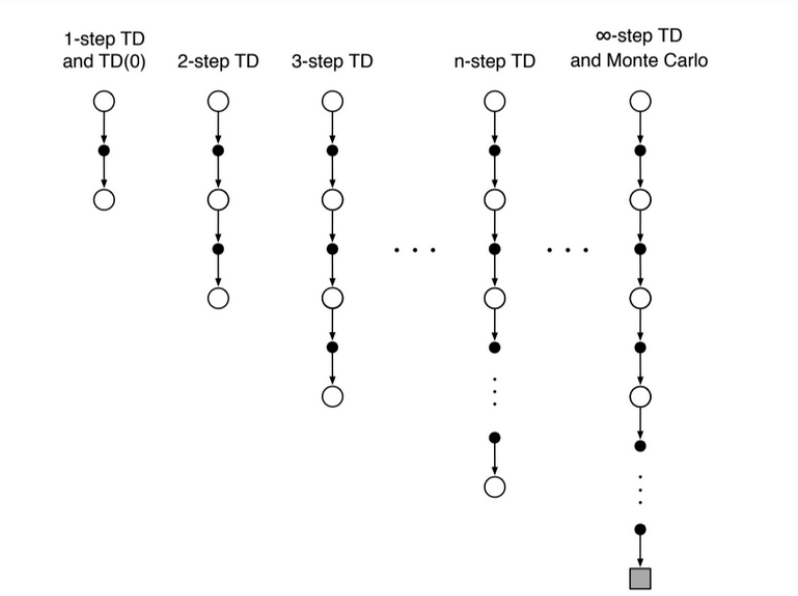


Figure 4.6: The backup diagram of n-step methods (figure from Sutton and Barto 2018)

In concrete terms, in MC learning the value function is updated in the direction of the complete return of the episode. However, in one-step TD learning the target is the one-step return:

$$G_{t:t+1} = R_{t+1} + \gamma V_t(s_{t+1})$$

where V_t here is the estimate at time t of V_π. In the same way we can define the two-step return for the two-step TD learning:

$$G_{t:t+2} = R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(s_{t+2})$$

Similarly, the target for n-step TD learning is the n-step return:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(s_{t+n}),\ t+n < T \qquad (4.8)$$

where T is the last step of the episode.

Note that n-step return contains future rewards that have not yet been received at time step t. For this reason, the value of the state at time step t can only updated at time step $t+n$. Therefore, the update rule for n-step TD prediction would be:

$$V_{t+n}(s_t) \leftarrow V_{t+n-1}(s_t) + a\big(G_{t:t+n} - V_{t+n-1}(s_t)\big),\ t < T \tag{4.9}$$

We can combine SARSA with n-step methods to derive n-step SARSA that can be used for control. We need only to switch the state-value function with the action-value function:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q_{t+n-1}(s_{t+n}, a_{t+n}),\ t+n < T \tag{4.10}$$

Then, we use rule 4.11 to update the action-value function:

$$Q_{t+n}(s_t, a_t) \leftarrow Q_{t+n-1}(s_t, a_t) + a\big(G_{t:t+n} - Q_{t+n-1}(s_t, a_t)\big),\ t < T \tag{4.11}$$

With the same way we can derive the update rules, if we are using a function approximation.

4.3.2 λ-return Methods

While n-step methods use as target the n-step return, λ-return methods extend this idea by using as target the average of n-step returns for different n's. This quantity is called λ-return and contains all the n-step returns each weighted proportionally by λ^{n-1} ($\lambda \in [0,1]$) and is normalized by a factor of $(1-\lambda)$ to ensure that the weights sum up to 1:

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{T-t} \lambda^{n-1} G_{t:t+n} \tag{4.12}$$

Equation 4.12 can be written in the following way:

$$G_t^\lambda = (1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} + \lambda^{T-t-1} G_t \tag{4.13}$$

Equation 4.13 will help us extract some interesting results. If $\lambda = 0$, the λ-return is equal to the one-step return $G_{t:t+1}$, which is equivalent to one-step TD learning. On the other hand, if $\lambda = 1$, the λ-return is equal to the full return G_t, which is equivalent to MC learning.

Therefore, TD(λ) with function approximation uses the update rule 4.14:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + a\big(G_t^\lambda - \hat{V}(s; \mathbf{w}_t)\big)\nabla \hat{V}(s; \mathbf{w}_t) \tag{4.14}$$

This approach is called *forward view*, because we update each state by looking forward in time and using future rewards and states. For this reason, the updates occur only at the end of the episode.

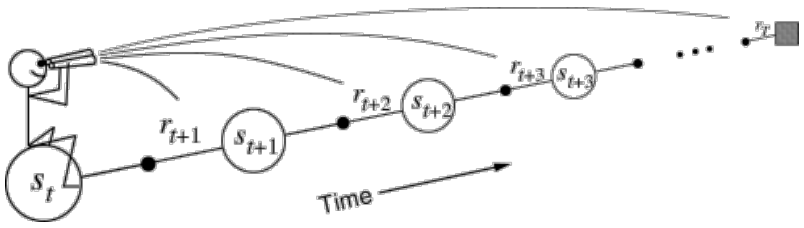


Figure 4.7: The forward view of TD(λ) (figure from Sutton and Barto 2018)

4.3.3 Eligibility Traces

As mentioned before, forward view TD(λ) updates the weights only once the episode has terminated. This limitation can be dealt with the use of eligibility traces. Even though this *backward view* algorithm that we are going to present approximates the forward view, it is very efficient.

The weight vector $\mathbf{w}$ can be viewed as a long-term memory that is accumulating experience over the lifetime of the agent. On the other hand, the *eligibility trace* is a short-term memory vector $\mathbf{z} \in \mathbb{R}^k$ that often last less than the length of an episode. It is initialized to zero at the start of the episode and then it changes based on rule 4.15:

$$\mathbf{z}_t \leftarrow \gamma\lambda\mathbf{z}_{t-1} + \nabla_{\mathbf{w}}\hat{V}_t(s_t; \mathbf{w}_t) \tag{4.15}$$

According to Equation 4.15 at each time-step the eligibility trace is incremented by the gradient of the value function of the current state and then fades away by $\gamma\lambda$. This means that the eligibility trace keeps track of which components of the weight vector have contributed, positively or negatively, only to recent learning events. The events we are interested are represented by the one-step TD error.

$$\delta_t = R_{t+1} + \gamma\hat{V}(s_{t+1}; \mathbf{w}_t) - \hat{V}(s_t; \mathbf{w}_t) \tag{4.16}$$

Then, the weight vector is updated proportionally to the TD error and the eligibility trace vector:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + a\delta_t\mathbf{z} \tag{4.17}$$

This algorithm is called backward view TD(λ), because we look back in time. At each time step, we calculate the TD error and assign it to past states depending on the contribution of that state, which is represented by the eligibility trace and fades away with time.

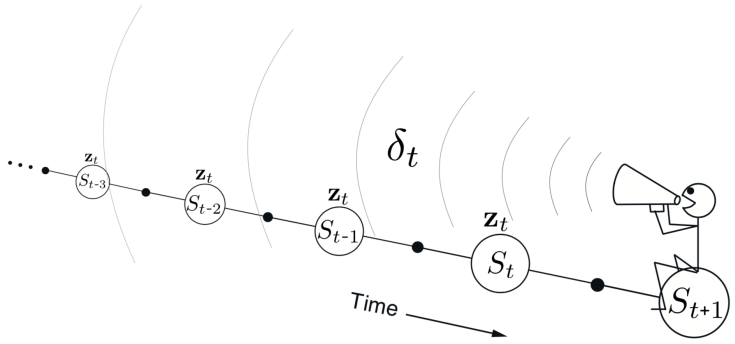


Figure 4.8: The backward view of TD(λ) (figure from Sutton and Barto 2018)

Linear TD(λ) has been proved to converge in the on-policy case, if the step-size parameter is reduced over time. To better understand the algorithm, we will examine what happens for different values of λ. If $\lambda = 0$, then at each time step the eligibility trace is equal to the gradient of the value function $\nabla\hat{V}(s_t; \mathbf{w}_t)$ and thus the update rule 4.17 becomes equivalent to the one-step TD learning or TD(0). If $\lambda = 1$, then at each time step the eligibility trace fades only by γ and it turns out to be equivalent to MC learning or TD(1). For $0 < \lambda < 1$, the larger the value of λ the more past states are affected, but each preceding state is affected less.

4.3.4 SARSA(λ)

The above ideas can be combined with SARSA in order to produce a forward view algorithm for control known as SARSA(λ) (Rummery and Niranjan 1994), which is used extensively in reinforcement learning. It has the same update rules as TD(λ), but we swap the state-value function with the action-value function:

$$\mathbf{z}_t \leftarrow \gamma\lambda\mathbf{z}_{t-1} + \nabla\hat{Q}(s_t, a_t; \mathbf{w}_t) \tag{4.18}$$

$$\delta_t = R_{t+1} + \gamma\hat{Q}(s_{t+1}, a_{t+1}; \mathbf{w}_t) - \hat{Q}(s_t, a_t; \mathbf{w}_t) \tag{4.19}$$

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + a\delta_t\mathbf{z} \tag{4.20}$$

SARSA(λ)'s backup diagram is shown in Figure 4.9. It illustrates the weighting of each type of return. The one-step return is assigned to the largest weight, $1 - \lambda$. The two-step return is assigned to the next largest weight, $(1 - \lambda)\lambda$ and so on. Note that each weight fades by λ.

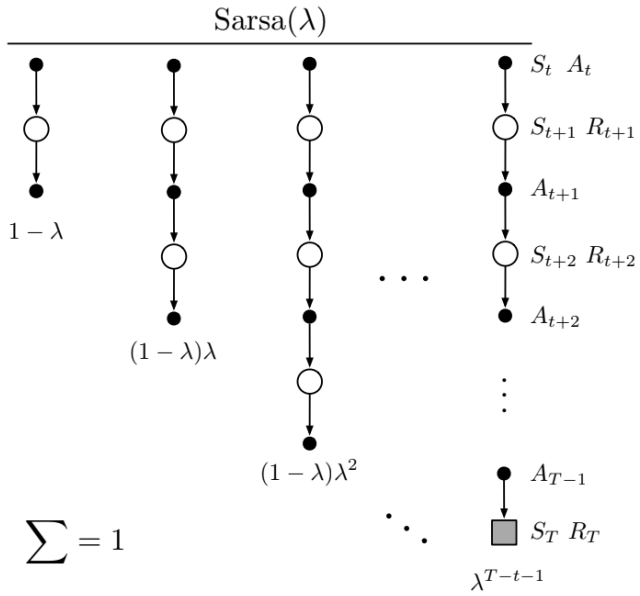


Figure 4.9: SARSA(λ)'s backup diagram (figure from Sutton and Barto 2018)

Algorithm 1 shows the pseudocode for SARSA(λ) with linear function approximation, decaying ϵ-greedy policy, and sigmoid learning rate. This implementation contains some optimizations possible when using ϵ-greedy policy. More specifically, in each-time step, it calculates the Q values for each action and stores them to be used in later computations.

Algorithm 1 SARSA(λ)

function CALCULATE_Q_VALUES($s, \mathbf{w}, \boldsymbol{\phi}$)
  for each action a **do**
    $\mathbf{Q}_s(a) \leftarrow \mathbf{w}^T \boldsymbol{\phi}(s, a)$
  end for
  return $\mathbf{Q}_s$
end function

function SARSA(λ)($\gamma, \lambda, \boldsymbol{\phi}, \alpha_0, k, ep_0$)
  Initialize $\mathbf{w}$ arbitrarily
  for each episode ep **do**
    $\alpha \leftarrow \frac{\alpha_0}{1+e^{k(ep-ep_0)}}$
    $\epsilon \leftarrow \frac{1}{ep+1}$
    Initialize s
    $\mathbf{z} \leftarrow \mathbf{0}$
    $\mathbf{Q}_s \leftarrow$ calculate_q_values($s, \mathbf{w}, \boldsymbol{\phi}$)
    $a \leftarrow \begin{cases} \text{argmax}_{a''} \mathbf{Q}_s(a'') & \textit{with probability } 1-\epsilon \\ \textit{random action} & \textit{with probability } \epsilon \end{cases}$
    while s is not terminal **do**
      Take action a, observe r, s'
      $\mathbf{Q}'_{s'} \leftarrow$ calculate_q_values($s', \mathbf{w}, \boldsymbol{\phi}$)
      $a' \leftarrow \begin{cases} \text{argmax}_{a''} \mathbf{Q}'_{s'}(a'') & \textit{with probability } 1-\epsilon \\ \textit{random action} & \textit{with probability } \epsilon \end{cases}$
      if s' is terminal **then**
        $\delta \leftarrow r - \mathbf{Q}_s(a)$
      else
        $\delta \leftarrow r + \gamma \mathbf{Q}'_{s'}(a') - \mathbf{Q}_s(a)$
      end if
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
      $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \boldsymbol{\phi}(s, a)$
      $a \leftarrow a'$
      $s \leftarrow s'$
      $\mathbf{Q}_s \leftarrow \mathbf{Q}'_{s'}$
    end while
  end for
end function

4.4 Least Squares Policy Iteration

All of the algorithms we have presented so far fall into the category of online reinforcement learning. However, learning a policy can also be done in an offline way. This is also called *batch* reinforcement learning (Lange, Gabel, and Riedmiller 2019) and its model is slightly different from that we have presented in Figure 2.2. In batch reinforcement learning, the agent does not interact with the environment, but instead is trying to find an optimal policy using a fixed set of samples $D = \{(s_t, a_t, r_{t+1}, s'_{t+1}), ...\},\ t = 1, 2, 3, ..$ that have been collected using an arbitrarily policy π_a. This is illustrated in Figure 4.10. Batch reinforcement learning algorithms come useful in applications in which gathering samples can be expensive (e.g. robotics).

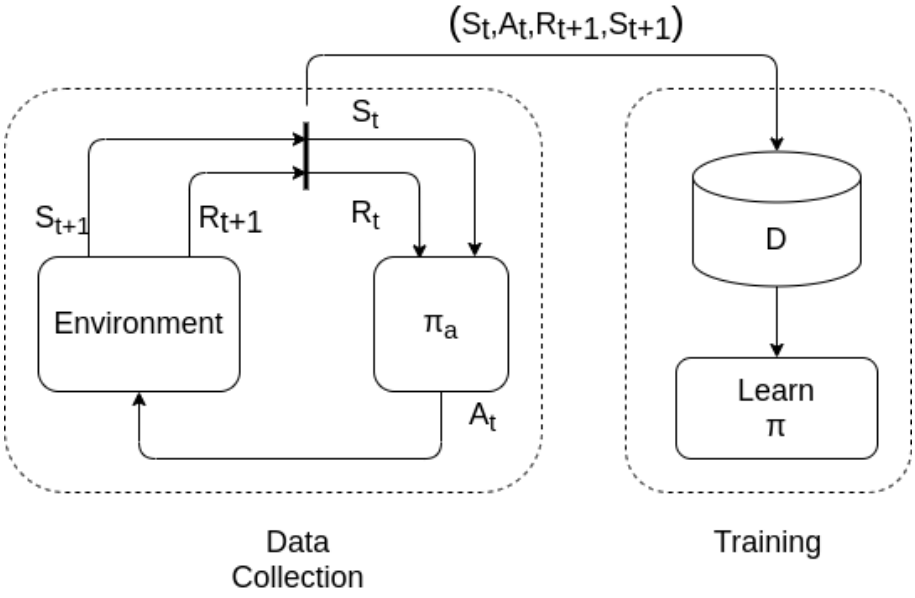


Figure 4.10: A basic offline (batch) learning model

Least-squares policy iteration (LSPI) (Lagoudakis and Parr 2003) is a well-known algorithm that solves the control problem under the framework of policy iteration that was presented in Section 2.1.7. However, LSPI does not store the policy explicitly, but instead uses only the action-value function Q. The policy can be derived by acting greedily (Equation 2.13). Moreover, LSPI uses a parametric linear value function approximation, as described in Section 4.2.

For the policy evaluation step, the least-squares temporal-difference (LSTDQ) algorithm is used. This algorithm calculates the action-value Q_π of a given policy π by using the set of samples D. Both the input policy and the derived action-value function are of course represented by a set of weights. Algorithm 2 shows the pseudocode for LSTDQ. An important part of the algorithm is that $\mathbf{A}$ will not be full rank and therefore its inverse matrix can not be calculated until enough

samples have been processed. This can be dealt with by initializing $\mathbf{A}$ to $\delta\mathbf{I}$ for a small positive δ. Another solution is to use SVD to calculate $\mathbf{A}^{-1}$.

Algorithm 2 LSTDQ

function LSTDQ($\gamma, D, \boldsymbol{\phi}, \boldsymbol{w}$)
  $\mathbf{A} \leftarrow \mathbf{0}$ // $(k \times k)$
  $\mathbf{b} \leftarrow \mathbf{0}$ // $(k \times 1)$
  for all samples $(s, a, r, s') \in D$ **do**
    if s' is terminal **then**
      $\mathbf{A} \leftarrow \mathbf{A} + \boldsymbol{\phi}(s, a)\boldsymbol{\phi}^T(s, a)$
    else
      $a' \leftarrow \text{argmax}_{a''}\, \mathbf{w}^T\boldsymbol{\phi}(s', a'')$
      $\mathbf{A} \leftarrow \mathbf{A} + \boldsymbol{\phi}(s, a)\Big(\boldsymbol{\phi}(s, a) - \gamma\boldsymbol{\phi}(s', a')\Big)^T$
    end if
    $\mathbf{b} \leftarrow \mathbf{b} + \boldsymbol{\phi}(s, a)r$
  end for
  $\mathbf{w} \leftarrow \mathbf{A}^{-1}\mathbf{b}$
  Return w
end function

After having evaluated the current policy, the policy improvement can be performed by acting greedily in respect to the action-value function Q_π. This is identical to the policy improvement step in policy iteration algorithm. However, LSPI does not need to store the policy explicitly. Therefore, the policy improvement step consists of overwriting the old weights with the ones calculated by LSTDQ. This alternation between the two steps of LSPI that is illustrated in Figure 4.11 is repeated, until the policy (weights) converge to a certain degree. Algorithm 3 shows the pseudocode for LSPI. Note that when the number of features is small, the feature vector $\boldsymbol{\phi}(s, a)$ for each sample can be calculated once and stored to be reused in each iteration of LSPI.

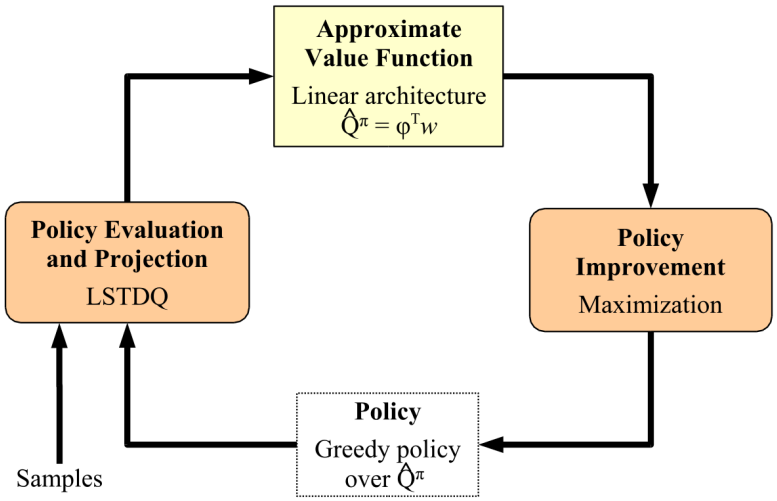


Figure 4.11: Least-squares policy iteration (figure from Lagoudakis and Parr 2003)

Algorithm 3 LSPI

function LSPI($\gamma, D, \boldsymbol{\phi}, \epsilon$)
  Initialize $\mathbf{w}'$ arbitrarily
  repeat
    $\mathbf{w} \leftarrow \mathbf{w}'$
    $\mathbf{w}' \leftarrow LSTDQ(\gamma, D, \boldsymbol{\phi}, \mathbf{w})$
  until $||\mathbf{w} - \mathbf{w}'|| < \epsilon$
end function

LSPI has proven to be quite effective compared to other reinforcement learning algorithms. First of all, LSPI is an off-policy algorithm meaning that can calculate an optimal policy from samples that have been collected by an arbitrary policy. This is not to say that we can use any policy to collect the samples. The effectiveness of the policy that LSPI will produce is closely related to the distribution of the samples. Even though LSPI is an actor-critic algorithm, it does not explicitly stores the policy (actor) which is not the case for other popular actor-critic algorithms. Therefore, LSPI eliminates the error that can occur from the approximation method of the policy. LSPI is also sample efficient, as it reuses the same set in every iteration, which is not true for other batch algorithms that require to collect new samples in later stages. Last, but not least, many reinforcement learning algorithms require careful study concerning the learning rate and its scheduling and the exploration policy and its scheduling. This parameters are not present in LSPI and thus there is less need for fine-tuning.

4.5 Implementation

In this chapter, we will discuss the implementation of the application without going into details.

As mentioned, the whole project was implemented in ROS using Python. Figure 4.12 shows the ROS graph of our application. The node /train_uav contains the training algorithm that we implemented. At the start of each episode, the UAV must reach its random initial pose. To achieve this we publish a PoseStamped message in the /command/pose topic. Then, the controller that the Hector quadrotor package provides (not shown in the figure) moves the UAV into its starting pose. Afterwards, /train_uav controls the UAV by publishing Twist messages in the /cmd_vel topic depending on the state. The state of the UAV (LiDAR measurements) is received by the /scan topic as LaserScan messages.

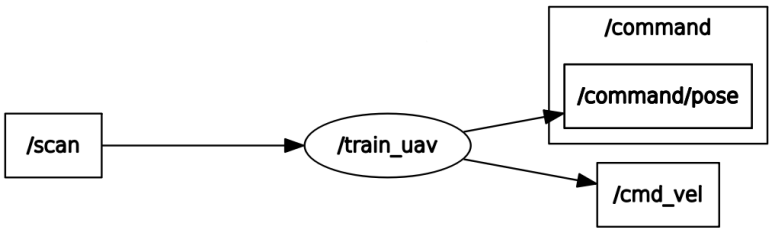


Figure 4.12: The ROS graph of our implementation

The code follows an object-oriented structure and contains two main classes. The environment class acts as a black box for the agent and implements all the necessary logic and dynamics of the environment (state space, action space, reward function, initial state, state transition, etc.). Note that the API of that class is organized in a similar way to the environments of OpenAI Gym - a popular reinforcement learning tool (Brockman et al. 2016). The corresponding classes that Gym provides were also used to define the state space and action space of the environment. The second class is the learning algorithm and contains the implementation of the algorithm that interacts with the environment.

Chapter 5

Experimental Results

In our experiments, we used SARSA(λ) and LSPI with tile coding. We also experimented with radial basis functions as a linear representation, but the results were not satisfying. Furthermore, to test how good the UAV behaves in environments which has never seen before, training occurred only in the first track, and then we measured the efficiency of the policy in the second and third track.

Since an episode can last for many actions, for both algorithms we chose a relatively high discount factor, $\gamma = 0.99$ to highlight the importance of future rewards. For the SARSA(λ) algorithm we chose $\lambda = 0.5$ which indicates that we look at the not too distant future to calculate the value function. For the scheduling of the learning rate the following parameters were used: $k = 0.2,\ ep_0 = 350,\ \alpha_0 = \frac{0.1}{\#tilings}$. As mentioned before the maximum number of actions in our experiments is 500. In the online algorithm SARSA(λ) training was occurred for 500 episodes that result in approximately 24 hours, a characteristic that prevented us from performing many experiments.

5.1 SARSA(λ) in the 5D state space

In the first experiment, we used the 5-dimensional state space S_1 and SARSA(λ).

For the tile coding approximation, we discretize each dimension in 14 tiles and an extra one tile to achieve overlap between the tilings. We used 7 tilings that differ from each other by a tenth of tile width $\frac{state\_space.high - state\_space.low}{\#tiles} = \frac{29.8}{10} = 2.98$. This results in $5,315,625$ features. However, these features must be repeated for each action (3 actions in total). Therefore the total number of features is $15,946,875$. This a very large number of features for a linear approximation. Nonetheless, as we stated in Section 4.2.1 only 10 of them are active each time and thus calculations are relatively fast. This is not to say that action selection and learning time is negligible. In fact, during training approximately $0.2s$ will pass without the UAV receiving a velocity command, which can obstruct the learning procedure. Of course, this time is much lower, if we do not perform learning and simply act greedily.

Figure 5.5 shows the accumulated reward during the training. For demonstration purposes, the curve has been smoothed by averaging the rewards every 5 episodes. Note that these rewards correspond to the learning phase. This means that they were gathered, while the UAV was exploring and thus some bad results could be due to random actions. Figure 5.2 shows the number of steps in each episode. Of course, these two curves have a similar shape. Both curves start to take-off around the episode 100. However, they maintain a high variance. This variance is reduced around episode 300 in which the learning rate starts to decrease and thus smaller steps are being taken towards the target in each update of the value function. This results in a more stable policy as illustrated by the curve.

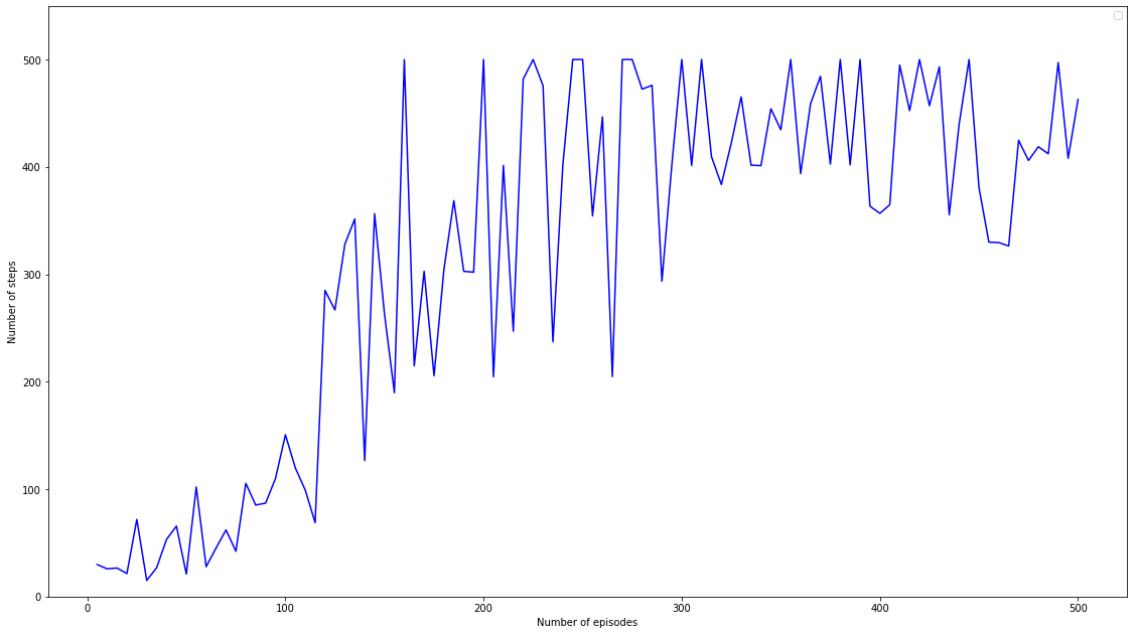


Figure 5.1: Number of steps during training for SARSA(λ) in the 5D state space

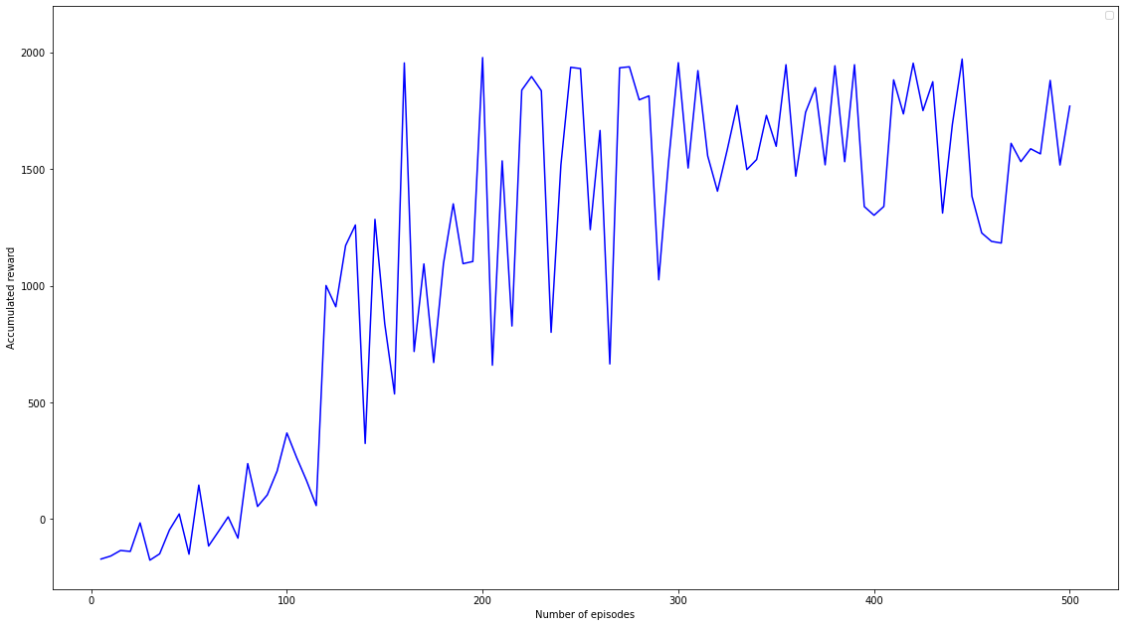
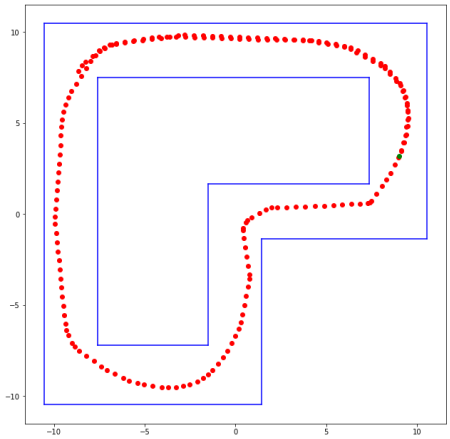


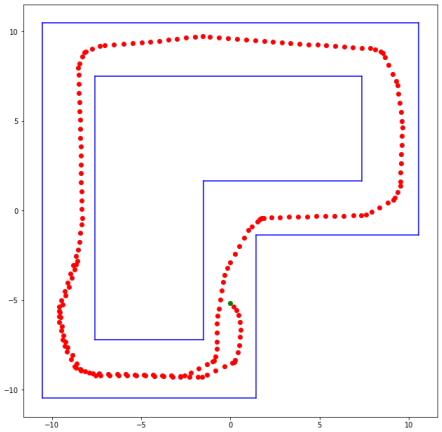
Figure 5.2: Accumulated reward during training for SARSA(λ) in the 5D state space

Figure 5.3 shows examples of trajectories in all three tracks that were produced using the final policy of $SARSA(\lambda)$. Note that in these figures only the position (x, y) is illustrated and not the rotation (yaw). All the trajectories start from a random position that is indicated by the green dot.

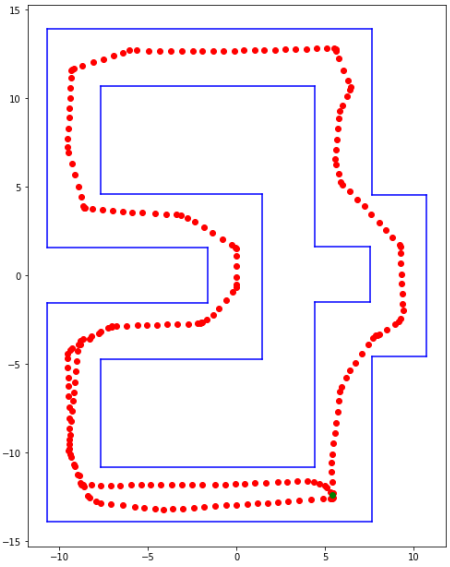
The UAV is able to complete the first track successfully both clockwise and anti-clockwise, as indicated in the trajectories shown in Figures 5.3a and 5.3b. The factor that decides if the UAV will complete the track clockwise or anti-clockwise seems to be its starting orientation and will move towards the direction that it is initially facing. Testing on the second track was quite successful too. The UAV can complete the second track clockwise as shown in the trajectory plotted in Figure 5.3c. It is even capable of dealing with the π-shaped turns on the right side of the track which are somewhat challenging. However, it was not able to complete the track anti-clockwise. This is illustrated in the trajectory of Figure 5.3d in which it fails to complete the bottom left turn. Testing on the third and more challenging track was surprisingly successful. The trajectory in Figure 5.3e shows that the UAV can complete the track anticlockwise. It is even capable of dealing with the left angular turn on the right and on top sides of the track. This is quite impressive, since the UAV has not seen this kind of turn during training. However, when moving clock-wise the UAV failed to complete the right angular turn, as illustrated in the trajectory of Figure 5.3f.



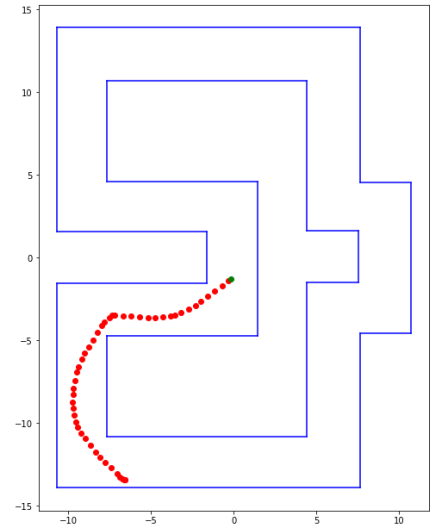
(a) $reward = 2093,\ steps = 500$



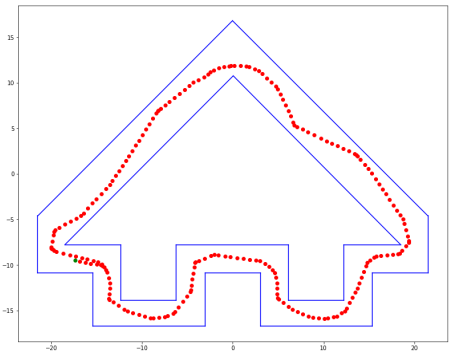
(b) $reward = 1999.5,\ steps = 500$



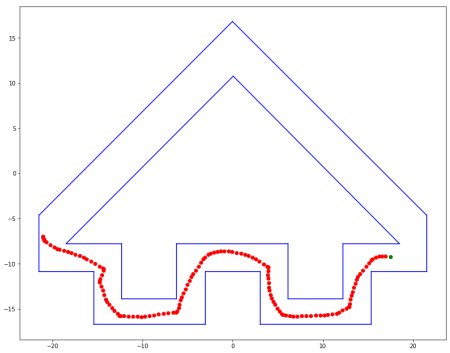
(c) $reward = 2752,\ steps = 700$



(d) $reward = 250,\ steps = 113$



(e) $reward = 2680.5,\ steps = 700$



(f) $reward = 1049.5,\ steps = 351$

Figure 5.3: Examples of trajectories from the final policy derived by SARSA(λ) in the 5D state space

5.2 SARSA(λ) in the 3D state space

In the second experiment, SARSA(λ) was used to solve the MDP that consist of the 3-dimensional state space S_2.

For the tile coding approximation, we used $10+1$ tiles and 12 tilings that differ from each other by a tile width of 2.48. This results in $47,916$ total number of features.

The accumulated reward during the training and the number of steps are shown in Figures 5.5 and 5.4 respectively. Again we smoothed both curves by averaging them every 5 episodes. The first thing to notice is that even though the average number of steps reaches the maximum value of 500 quite often, the average accumulated reward is around 1400, which is lower compared to the first experiment. This is due to a peculiarity that the produced policy contains. In this policy, the UAV can successfully complete only right turns. This is not to say that it crashes in left turns. As indicated by the trajectory examples in Figure 5.6, whenever it faces a left turn, it performs a full right rotation and then starts moving in the opposite direction. However, depending on the track, the UAV can be trapped between two left turns. In this case, it will perform many rotations and for this reason, the accumulated reward is lower. Moreover, we can observe that the average accumulated reward and the number of steps start to take off quite early.

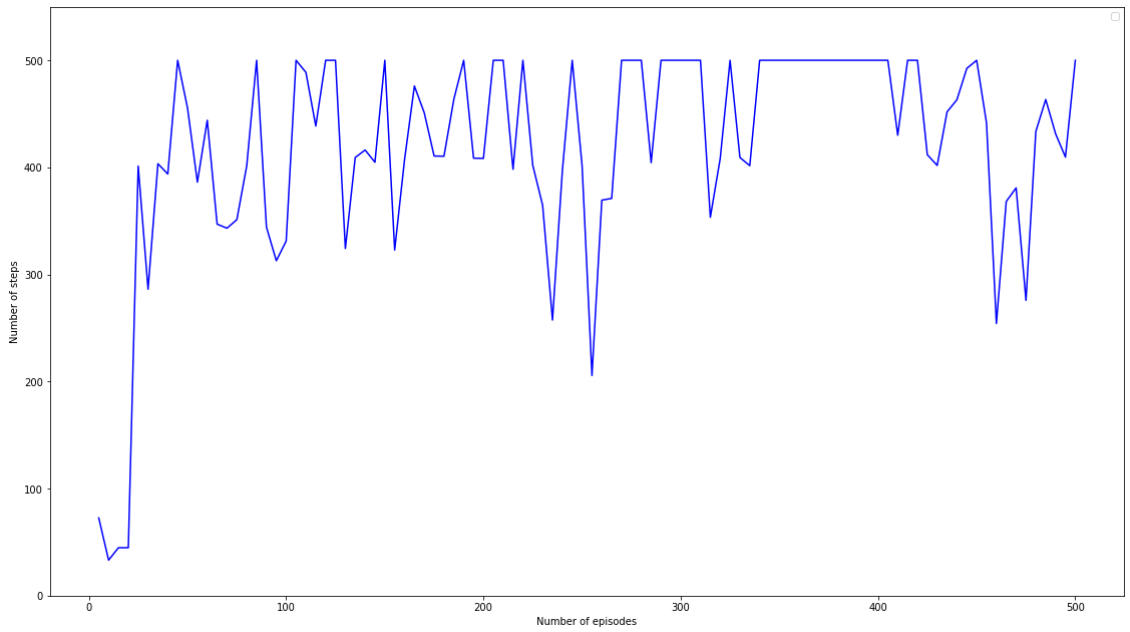


Figure 5.4: Number of steps during training for SARSA(λ) in the 3D state space

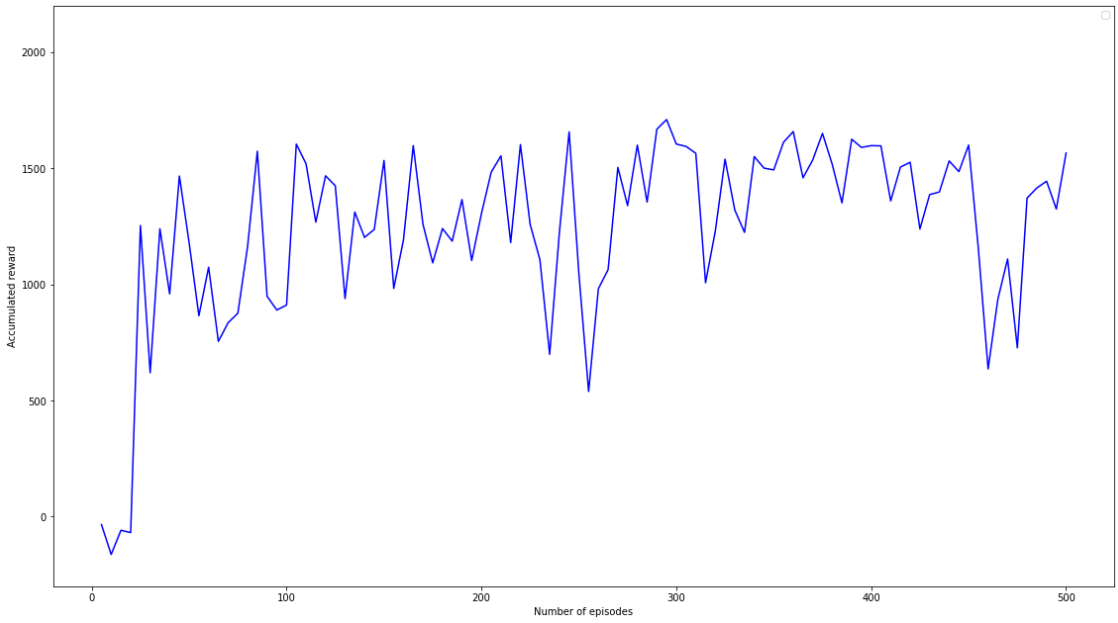
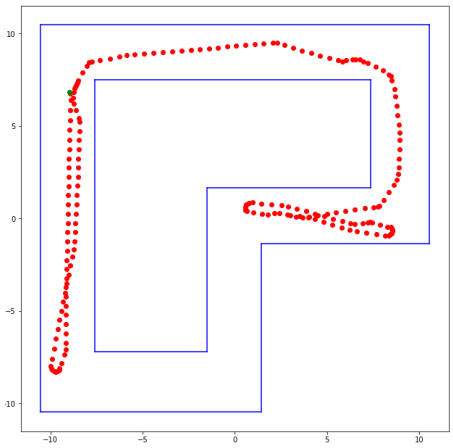
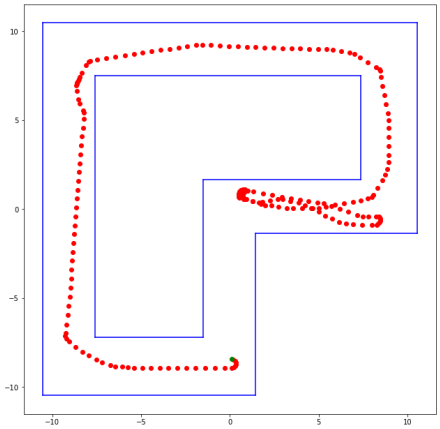


Figure 5.5: Accumulated reward during training for SARSA(λ) in the 3D state space

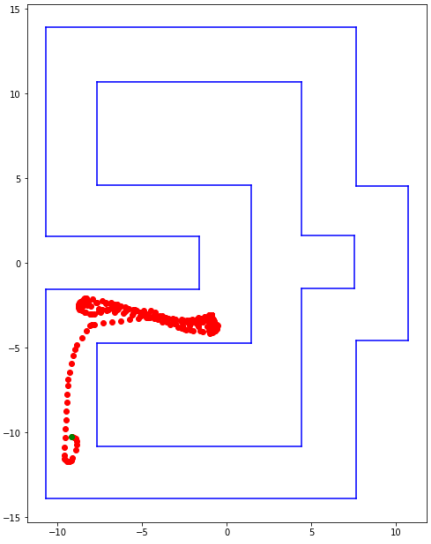
The trajectories that are shown in Figure 5.6 prove the peculiarity of the policy discussed above. In the trajectories of Figures 5.6a, 5.6b, 5.6c and 5.6f the UAV is trapped between two left turns. However, in Figure 5.6f it manages to escape and perform a left turn. In the other trajectories (Figures 5.6d, 5.6e) it crashed when it faced a left turn. However, the right turns were performed quite successfully.



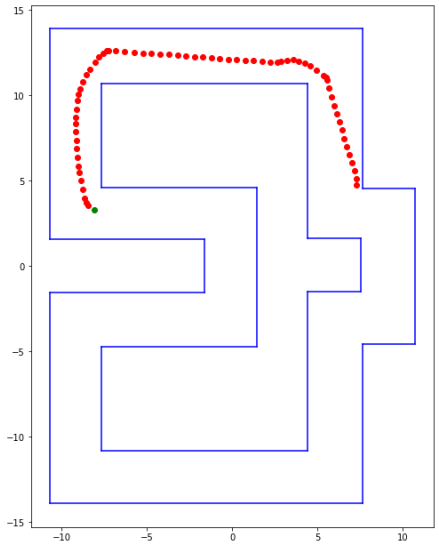
(a) $reward = 1741.0,\ steps = 500$



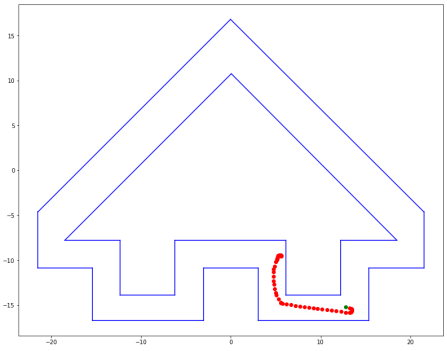
(b) $reward = 1431.5,\ steps = 500$



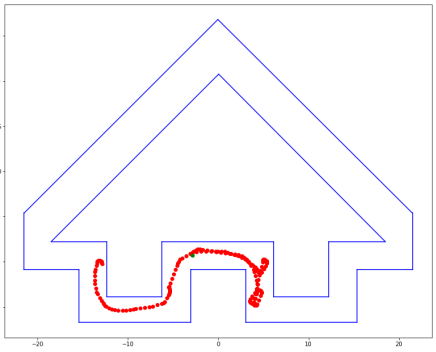
(c) $reward = 1383.5,\ steps = 500$



(d) $reward = 490.5,\ steps = 171$



(e) $reward = 88.5,\ steps = 117$



(f) $reward = 695.0,\ steps = 444$

Figure 5.6: Examples of trajectories from the final policy derived by SARSA(λ) in the 3D state space

5.3 LSPI in the 3D state space

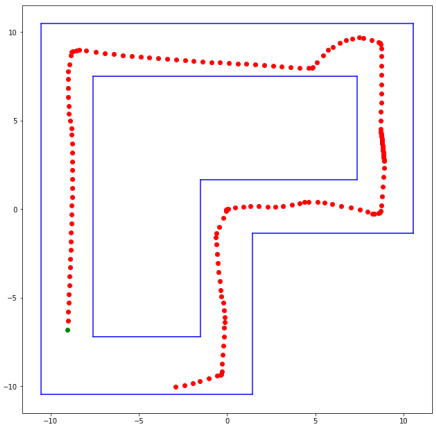
In the third experiment, we used the 3-dimensional state space S_2 in the MDP and solved it with LSPI.

In LSPI, since the dimensions of the matrix, $\mathbf{A}$, and thus the complexity of the calculations are depending on the number of features, we had to use a low discretization resolution. Therefore, we used $5 + 1$ tiles and 3 tilings that differ from each other by a tile width of 9.73. This results in 1944 total number of features. In order to train LSPI, we experimented with many different sample sets. In this experiment, we gathered the samples by running SARSA(λ) with the same parameters for 300 episodes resulting in 74367 samples. However, during training, $\mathbf{A}$ was not always full rank. This was addressed by initializing it to $0.1\mathbf{I}$. Table 5.1 shows the difference between the weights in each iteration. Note that it took only 8 iterations (approximately one hour) in order to reach convergence.

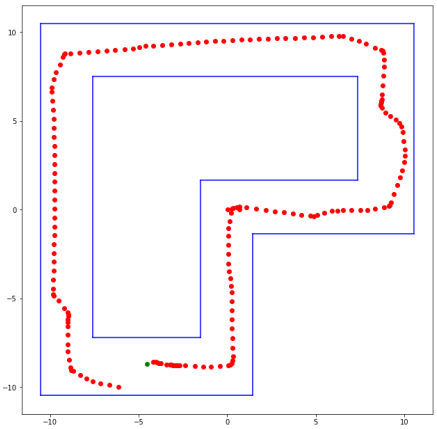
Number of iteration	$\|\|w - w'\|\|$
1	295.642431562
2	741.483850835
3	378.808579275
4	61.8432844184
5	13.8270604234
6	0.503878575229
7	0.0

Table 5.1: The difference of weights in each iteration of LSPI

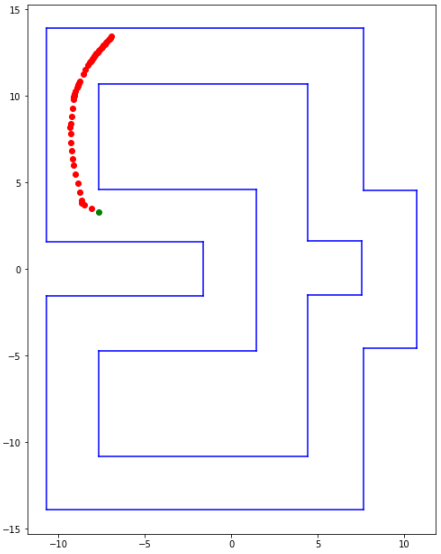
Examples of trajectories that are shown in Figure 5.7 that prove that the LSPI method has a lot of potential. It manages to almost complete the first track, both clockwise and anti-clockwise as illustrated in the trajectories shown in Figures 5.7a and 5.7b. In the second track (trajectories in Figures 5.7c and 5.7d), the results were not so positive. Note however that the policy was able to complete the left turn at the bottom of the track in which the first method failed. The results on the third track were also decent. The policy was able to complete successfully the bottom part of the track (trajectory in Figure 5.7e), but when it faces the angular turns it fails (trajectories in Figures 5.7e and 5.7f).



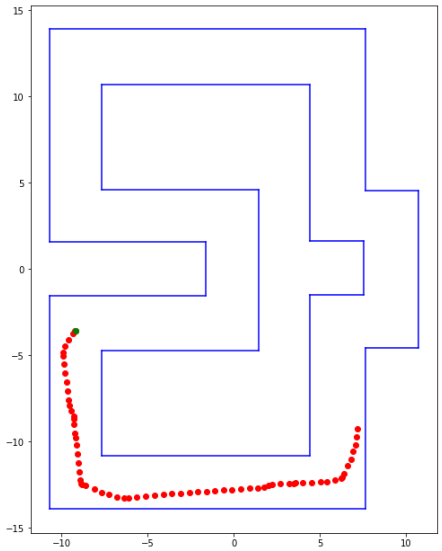
(a) $reward = 1268.5,\ steps = 430$



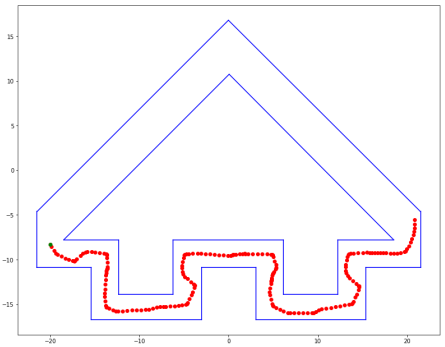
(b) $reward = 1363.5,\ steps = 471$



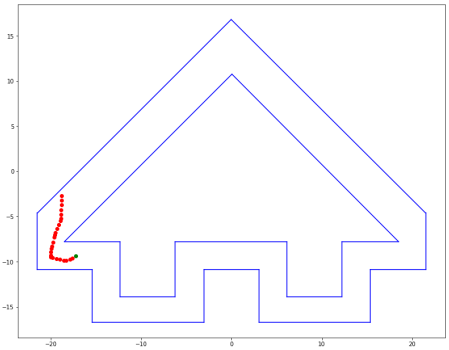
(c) $reward = 419.5,\ steps = 170$



(d) $reward = -41.5,\ steps = 135$



(e) $reward = 1455.0,\ steps = 500$



(f) $reward = 14.0,\ steps = 68$

Figure 5.7: Examples of trajectories from the final policy derived by LSPI in the 3D state space

Chapter 6

Conclusion

This thesis describes a mapless solution for the problem of autonomous navigation of a UAV in unknown environments that uses reinforcement learning. We designed an MDP to model the problem and added stochasticity in the environment to represent real-world circumstances. To learn a good decision policy, we implemented two different algorithms, SARSA(λ) and LSPI. Furthermore, tile coding, a linear value function approximation method, was used due to the large state space of the environment for the representation of the value function. Additionally, decaying ϵ-greedy policy was used to deal with the exploration and exploitation trade-off. The entire project was implemented in the ROS framework and was tested in the Gazebo simulator in three different scenarios. Training took place in a simple environment. SARSA(λ)'s produced policy was able to perform well in unknown and more complex environments in most situations. LSPI was also able to learn a decent policy, however, its efficiency was not as good as that of SARSA(λ).

6.1 Future Work

The method that was described was tested in a simulated environment. Even though we used Gazebo, which is a powerful simulator, that can imitate real-world circumstances pretty accurately and added noise to both actions and sensor measurements, the real world is much different. Further testing in real-world scenarios is required, where the obstacles may vary in size and shape and the dynamics of the UAV are different. However, due to the nature of the application, initial training in a simulated environment is necessary to construct a very basic policy that can control the UAV at a decent level. Afterward, this policy can be perfected with further training in a real UAV.

Moreover, additional testing in different scenarios is required. In our experiments, the UAV was flying at a constant altitude. This requires that all obstacles can be avoided without changing the altitude of the UAV, and thus the problem can be represented in 2D space. This assumption is not very realistic in real-world

applications. Besides, the advantage of using an aerial vehicle over a ground one is that it can easily avoid more obstacles and navigate difficult terrains. Note, however, that in such implementation the complexity of the problem will increase. A 3D LiDAR is required to take measurements in different heights and thus an increase of the state space is unavoidable. Furthermore, the action space will also increase with the addition of vertical movement. For this reason, linear methods may no longer be effective and non-linear techniques, such as neural networks, may be required. Additionally, adding a target location that the UAV has to reach is a very viable scenario that can be tested.

The model we described is not the only one that can be used. Many extensions can be implemented to make the model even more suitable for the application. First, the action space consists of three actions with standard velocities. A more realistic approach would be to have a continuous action space that consists of the two velocities. This will allow for the construction of a policy that can potentially complete the task faster. However, the algorithms that were presented do not deal with a continuous action space and a different approach is necessary. Moreover, due to the real-time characteristic of the application, a real-time Markov decision process (RTMDP) could prove more appropriate.

References

Bellman, R. (1957). "A Markovian Decision Process". In: *Journal of Mathematics and Mechanics*.

Bertsekas, D. P. (2005). *Dynamic Programming and Optimal Control, Volume 1*. Massachusetts: Athena Scientific.

Brockman, G. et al. (2016). "OpenAI Gym". In: *arXiv.org*.

Busoniu, L. et al. (2010). *Reinforcement learning and dynamic programming using function approximators*. Boca Raton: CRC.

Grzonka, S., G. Grisetti, and W. Burgard (2009). "Towards a Navigation System for Autonomous Indoor Flying". In: *IEEE International Conference on Robotics and Automation*.

Howard, R. (1960). "Dynamic Programming and Markov Processes". In: *MIT Press, Cambridge, MA*.

Huang, A. S. et al. (2016). "Visual Odometry and Mapping for Autonomous Flight Using an RGB-D Camera". In: *Journal of Intelligent & Robotic Systems*.

Imanberdiyev, N. et al. (2016). "Autonomous Navigation of UAV by Using Real Time Model Based Reinforcement Learning". In: *14th International Conference on Control, Automation, Robotics and Vision (ICARCV)*.

Kaelbling, L. P., M. L. Littman, and A. W. Moore (1996). "Reinforcement Learning: A Survey". In: *Journal of Artificial Intelligence Research*.

Kober, J., A. J. Bagnell, and J. Peters (2013). "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research*.

Lagoudakis, M. G. (2017). "Value Function Approximation". In: *Sammut C., Webb G.I. (eds) Encyclopedia of Machine Learning and Data Mining. Springer, Boston, MA*.

Lagoudakis, M. G. and R. Parr (2003). "Least-Squares Policy Iteration". In: *Journal of Artificial Intelligence Research*.

Lange, S., T. Gabel, and M. Riedmiller (2019). "Batch Reinforcement Learning". In: *IEEE Aerospace Conference*.

Meyer, J. et al. (2012). "Comprehensive Simulation of Quadrotor UAVs using ROS and Gazebo". In: *3rd Int. Conf. on Simulation, Modeling and Programming for Autonomous Robots (SIMPAR)*.

Mnih, V. et al. (2013). "Playing Atari with Deep Reinforcement Learning". In: *arXiv.org*.

Pham, H. X. et al. (2018). "Autonomous UAV Navigation Using Reinforcement Learning". In: *arXiv.org*.

Puterman, M. L. (1994). *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. New York: Wiley.

Rummery, G. A. and M. Niranjan (1994). "On-line Q-learning using connectionist systems". In: *Technical Report, Engineering Department, Cambridge University*.

Silver, D. et al. (2017). "Mastering the game of Go without human knowledge". In: *nature.com*.

Sutton, R. S. (1991). "Dyna, an integrated architecture for learning, planning, and reacting". In: *SIGART Bulletin*.

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

Walker, O. et al. (2019). "A Deep Reinforcement Learning Framework for UAV Navigation in Indoor Environments". In: *IEEE Aerospace Conference*.

Wang, C. et al. (2017). "Autonomous Navigation of UAV in Large-Scale Unknown Complex Environment With Deep Reinforcement Learning". In: *IEEE Global Conference on Signal and Information Processing (GlobalSIP)*.

Watkins, C. J. C. H. (1989). "Learning from Delayed Rewards". In: *(Ph.D. thesis), University of Cambridge*.

Williams, R. J. (1992). "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine Learning*.

Yijing, Z. et al. (2017). "Q learning algorithm based UAV path learning and obstacle avoidence approach". In: *36th Chinese Control Conference (CCC)*.

Zhang, B. et al. (2013). "Geometric Reinforcement Learning for Path Planning of UAVs". In: *Journal of Intelligent & Robotic Systems*.

Appendix A

Value Function Approximation and Gradient Methods

A.1 Gradient Descent

Gradient descent is a iterative optimization algorithm for finding a local minimum of a differentiable function. Let $F(\mathbf{w}) \in \mathbb{R}^n$ be a differentiable function of parameter vector $\mathbf{w} \in \mathbb{R}^n$. The gradient of $F(\mathbf{w})$ is:

$$\nabla F(\mathbf{w}) = \begin{bmatrix} \frac{\partial F(\mathbf{w})}{\partial w_1} \\ \dots \\ \frac{\partial F(\mathbf{w})}{\partial w_n} \end{bmatrix} \tag{A.1}$$

To find a local minimum of $F(\mathbf{w})$ we take small steps proportional to the negative of the gradient of the function:

$$\mathbf{w} \leftarrow \mathbf{w} - a\nabla F(\mathbf{w}) \tag{A.2}$$

where $a > 0$ is the step-size parameter or learning rate. This process is illustrated in figure A.1 in which we are trying to minimize a cost function.

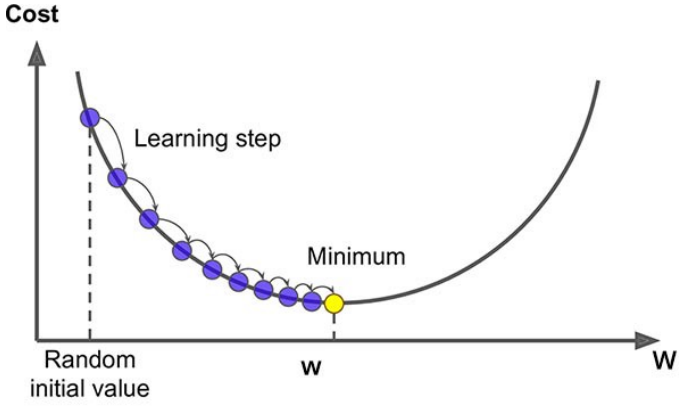


Figure A.1: An illustration of the gradient descent method

A.2 Stochastic-gradient and Semi-gradient Methods With Value Function Approximation

As mentioned in chapter 2.1.6 our goal in value function approximation is to tweak the parameters (weights) $\mathbf{w}$ appropriately in order that the approximate function fits the target function.

$$\hat{V}_\pi(s;\mathbf{w}) \approx V_\pi(s)$$

To achieve that, we first need to define the *mean squared value error* ($\overline{VE}$) between the two functions. $\overline{VE}$ is a metric that shows how much the approximate values differ from the true values. We will use the state-value function V as the function of interest but the same principles can also apply for the action-value function Q.

$$\overline{VE}(\mathbf{w}) = \mathbb{E}_\pi[(V_\pi(s) - \hat{V}(s;\mathbf{w}))^2] = \sum_s \mu(s)(V_\pi(s) - \hat{V}(s;\mathbf{w}))^2 \tag{A.3}$$

where the state distribution $\mu(s)$, $\sum_s \mu(s) = 1$ represent how much we care about the error in each state.

Then we can use the gradient descent method that was described above to find a weight vector $\mathbf{w}$ that minimizes the $\overline{VE}$:

$$\mathbf{w} \leftarrow \mathbf{w} - a\nabla\overline{VE}(\mathbf{w}) \tag{A.4}$$

$$\mathbf{w} \leftarrow \mathbf{w} - a\mathbb{E}_\pi[(V_\pi(s) - \hat{V}(s;\mathbf{w}))\nabla\hat{V}(s;\mathbf{w})] \tag{A.5}$$

We can further simplify the above update rule by using *stochastic gradient descent* (SGD), an online version of gradient descent that samples the gradient:

$$\mathbf{w} \leftarrow \mathbf{w} - a\nabla[V_\pi(s) - \hat{V}(s;\mathbf{w})]^2 \tag{A.6}$$

$$\mathbf{w} \leftarrow \mathbf{w} - a(V_\pi(s) - \hat{V}(s;\mathbf{w}))\nabla\hat{V}(s;\mathbf{w}) \tag{A.7}$$

However, the true value $V_\pi(s)$ is unknown. For this reason, we need to use an estimate of the value function in its place. If the estimate is unbiased, the method is guaranteed to converge to a local minimum. In MC learning the MC target G_t is by definition an unbiased estimate and thus we can use the following update rule:

$$\mathbf{w} \leftarrow \mathbf{w} - a(G_t - \hat{V}(s;\mathbf{w}))\nabla_{\mathbf{w}}\hat{V}(s;\mathbf{w})$$

In TD learning however, the bootstrapping targets are biased estimates as they depend on the current weight vector. For this reason, the step from equation A.6 to A.7 is no longer correct. In other words, the use of TD targets will not produce the true gradient descent method because it includes only part of the gradient and hence we call them *semi-gradient* methods. For example the update rule for TD(0) is shown below. With the same way we can define the rules that use any TD target.

$$\mathbf{w} \leftarrow \mathbf{w} - a(R + \gamma\hat{V}(s';\mathbf{w}) - \hat{V}(s;\mathbf{w}))\nabla_{\mathbf{w}}\hat{V}(s;\mathbf{w})$$