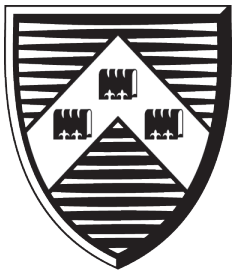
THE UNIVERSITY of York

Department of Computer Science

Submitted in part fulfilment for the degree of
MEng in Computer Science with Artificial Intelligence.

Reinforcement Learning of Helicopter Control

Harrison Spain
20th April 2016



Supervisor:
Dr Daniel Kudenko

Number of words = 15,995 , as counted by wc -w.
This includes the body of the report, but not Appendix A .

Abstract

This project explores the use of reinforcement learning for learning simulated helicopter controls. The simulator used is that of the 2014 Reinforcement Learning Competition. Helicopter control is a challenging task due to a helicopter's unstable nature. In terms of reinforcement learning the helicopter domain's environment is continuous and high dimensional in both the state and action space as well as being noisy and stochastic. Because of these complexities, this environment makes for a challenging test-bed for reinforcement learning.

Several reinforcement learning techniques are implemented and explored. Varying reinforcement learning parameters and using different techniques impact an agent's learning speed, maximum performance, learning stability, memory usage and computation time. It is demonstrated that the state space must be discretised to allow for generalisation; this is done by function approximation via tile coding. On top of this, the use of eligibility traces, reward shaping, state space representation altering and hierarchical reinforcement learning techniques are explored with each providing advantages and disadvantages in terms of the aforementioned properties. From the experiments some theory from the literature and properties observed in other domains are identified. Learning is observed with the use of general purpose reinforcement learning methods; however, performance is lower than specially tailored algorithms and hard-coded controllers.

I would like to thank my supervisor Daniel Kudenko for the support and guidance throughout this project. Emily, thank you for always being there for me, I love you.

Contents

List of Figures

List of Tables

1 Introduction

Machine learning can be, and has been, used to train a computer to perform tasks. These tasks can be pattern recognition, prediction, control and others. Most machine learning methods can be assigned to one of two broad categories: *supervised learning* and *unsupervised learning*. In supervised machine learning, training examples are presented to the learner. These examples consist of input values and desired output values. These are analysed to produce an inferred function which is then used on previously unseen input data to produce outputs. Unsupervised machine learning, on the other hand, does not present desired outputs. Instead, patterns within the input data are found [8, 9].

Not all problems fit perfectly into these two categories. For instance, it can be infeasible to produce training examples for the use of classic supervised learning methods. *Reinforcement learning* sits in a space between supervised and unsupervised learning. Rather than a 'correct' answer, a scalar *reward* signal, indicating a level of success, is presented to the learner (known as the *agent*). The main aim of reinforcement learning is for solving sequential decision problems where the agent must learn about its surroundings (the *environment*) then make a series of decisions based on its observations [10].

Many real world and complex problems can be formulated as reinforcement learning problems. Examples include: game-playing (both board [11] and computer [12] games), system scheduling [13] and, of closest relevance to this project, robot control [14, 15].

This project explores the use of reinforcement learning to learn helicopter controls. Rather than learning the controls of a physical helicopter, this project is in the context of simulated helicopters, in particular for the task of learning helicopter hovering. The helicopter simulator used is provided by the 2014 Reinforcement Learning Competition [16]. The reinforcement learning competition ran annually from 2006 to 2014; 2008 was the first year that helicopter hovering was included as a competition domain. Competitions of this nature encourage researchers to apply their theoretical approaches to (simulated) real-world tasks allowing the performance and characteristics of the approaches to be compared [17].

Helicopter control is a difficult task in general, not just for reinforcement learning. There are many complex forces acting on a helicopter. Unlike a fixed wing aircraft that can glide, a helicopter is naturally unstable and will fall from the sky if corrective actions are not immediately taken [18]. In terms of the reinforcement learning task there are several challenges. The first of which being that the twelve state variables and four action variables used by the agent are all continuous. Continuous states and actions mean there are an infinite number of states, actions and combinations of states and actions; if not addressed this will prevent any learning taking place. Secondly, the environment in which the agent exists is noisy and stochastic. The state the agent observes is noisy and therefore not the 'true' state. When the agent chooses an action wind in the environment adds a random effect to the action which is unknown to the agent. However, real-world helicopters would also have to deal with latency between the true state and the observed state and between selecting an action and it having an effect on the environment.

This project takes the form of a pseudo entry into the 2014 Reinforcement Learning Competition helicopter domain. The goal of this project is to perform an exhaustive parameter study of reinforcement learning approaches in the context of helicopter control. A series of helicopter agents are implemented and tested with the competition simulator. It was found

that without any method of dealing with the continuous states-action space the agent indeed did not learn and did not perform well. With the addition of generalisation and function approximation via the use of tile coding [3] the agent is able to learn and performs comparably to previous entries to the competition. Enhancements to reinforcement learning such as the use of eligibility traces, reward shaping and alternative state representations have advantages and disadvantages in terms of learning speed, learning stability, maximum performance, memory usage and computation time. However, no learning method or parameter choice were found to match the performance of a hard-coded controller provided with the competition software.

1.1 Report Structure

Chapter 2: Literature Review

In this chapter the background of the field of reinforcement learning is established. Starting with the fundamental concepts behind reinforcement learning then the various learning methods and algorithms. How to deal with continuous state spaces builds on the fundamentals: a necessity for this project. Finally, additional reinforcement learning techniques such as eligibility traces and reward shaping are introduced.

Chapter 3: Helicopter Control

Here the helicopter domain itself is considered. This includes how helicopters fly in reality and how the helicopter simulator used for this project works. Previous work in the use of reinforcement learning for both real-world and simulated helicopters is also presented.

Chapter 4: Helicopter Learner

The focus of this chapter is more towards the practicalities of the project. Firstly the objectives and requirements are established, along with the development approach used for the implementation of reinforcement learning for use in the experiments. The design of the implementation is then outlined. Finally, the experimental method used to test and evaluate the various agents is given.

Chapter 5: Experimentation and Results

The techniques established in chapter 2 are applied to the helicopter domain introduced in chapter 3. A series of experiments are undertaken using the method given in chapter 4; each testing the effects of either varying a reinforcement learning parameter or using a different reinforcement learning technique.

Chapter 6: Conclusions

The findings from chapter 5 are compared against other implementations of helicopter controllers. These other controllers include other work on reinforcement learning, other forms of machine learning and non-learnt controllers. The results of the experiments are summarised and future work proposed.

Appendix A: Definitions

A list of keywords and their definitions that are used throughout this report. Additionally, the symbols and their meanings are listed.

1.2 Statement of Ethics

There are no ethical concerns directly related to this project. There is no human involvement in this project and no data is collected. Therefore, informed connect and data protection are not necessary.

This project sits in the wider scope of autonomous vehicles, in particular autonomous drones[1]. Drones do have concerns with regards to privacy, safety and their use in warfare [19, 20]. Future work in this area may have to deal with these issues; however, this project is fully simulated and therefore these issues are not directly relevant.

[1] Unmanned or autonomous aerial vehicles

2 Literature Review

2.1 Agents, Environments and Rewards

Two key concepts in Reinforcement Learning are that of *Agents* and *Environments*. Agents are the entities in the world that learn and make decisions. Everything in the world that is outside an Agent is the Environment. Agents continually perceive from and interact with the environment using sensors and actuators respectively. Based on the *State* that the Agent perceives from the Environment, the Agent selects an *Action* to perform. The environment then responds to this action and then the agent perceives a new state. This process of perceiving and acting is repeated indefinitely or until some goal is reached, usually in discrete time steps [3, 21]. These relationships can be seen in figure 2.1.

The boundary between the Agent and the Environment is not necessarily at the physical edge of the Agent's body. It may in fact be closer to the Agent's "brain" [3]. For instance, an Agent may send control signals to motors but how the motors respond to these is not under the Agent's control; therefore the motors will be part of the Environment.

Another concept in Reinforcement is that of a *reward*. The reward is a number generated by the Environment. It can be positive or negative and indicates how 'good' the agent's current state is. The reward is perceived by the Agent along with the state from the Environment at each time step. The general objective of the Agent is to gather the most reward it can over time [3].

2.2 The Markov Property and Markov Decision Process

At any particular time, an agent perceives a state from the environment containing all the available information at that time. This state will include any sensor measurements (such as position) but may also be processed to include information built up over time (such as velocity from previous position measurements). However, the state may not include all information about all aspects of the environment [3].

If the agent's next state and reward received are only dependent on the previously perceived state and executed action and are also independent of all past states, then the state has the

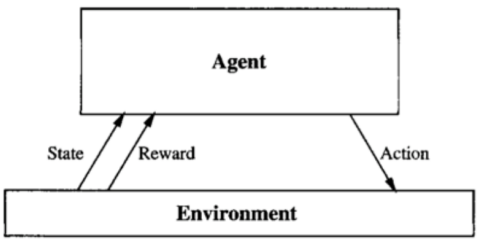


Figure 2.1: An Agent interacting with its Environment [1]

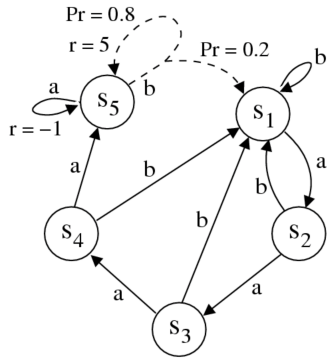


Figure 2.2: Example MDP Transition Graph [2]

Markov Property and is a *Markov State*. If all the states are Markov States, then the environment has the Markov Property. It is possible, but not necessary, that any state can encode all previous states' information leading up to the current time; so only a single state is needed for deciding the next action. Therefore, from any state, all future states can be predicted [3].

The Markov property can be formalised by the equivalence of the following two probably distributions at time t:

$$Pr\left\{s_{t+1} = s', r_{t+1} = r' | s_t, a_t, s_{t-1}, a_{t-1} \ldots r_1, s_0, a_0\right\} \equiv Pr\left\{s_{t+1} = s', r_{t+1} = r' | s_t, a_t\right\} \tag{2.1}$$

for all states s' and rewards r and all possible values of past events $s_t, a_t, r_t, r_1, s_0, a_0$.

In Reinforcement Learning it is assumed that decisions are based only on the current state; therefore, these states should have the Markov property. When a Reinforcement Learning task has the Markov property it can be modelled using a *Markov Decision Process* (MDP). Modelling a task without the Markov property with an MDP is possible and can in-fact perform well [3, 22]. It is also possible to adapt reinforcement learning algorithms to use a non-Markovian Decision Process [23].

An MDP is a tuple (S, A, T, R) where S is a set of states, A is a set of actions, T is a transition function $T : S \times A \times S \to [0, 1]$ and R is a reward function $R : S \times A \times S \to \mathbb{R}$. The transition function moves the agent from one state to another by applying an action, this function can be deterministic or stochastic. The reward function gives a numerical reward for transitioning from one state to another by applying an action [22, 24].

An example MDP transition graph can be seen in figure 2.2. It has five states and two actions. All actions are deterministic except for taking action b in state five where there is an 80% chance of returning to state five and a 20% chance of moving to state one. Taking action a in state five gives a reward of -1 and taking action b and returning to state five gives a reward of 5. All other actions from all other states have a reward of 0.

2.3 Policies and Value Functions

At each time step an agent must decide what action to take. The action to be taken at a particular state is determined by the agent's *policy*. A policy maps a state to an action that will be performed at that state. The policy can be deterministic $\pi : S \to A$ or stochastic $\pi : S \times A \to [0, 1]$ [22].

To determine which action to take (which state to transition to) it must be possible to say if one state is better than another; this is given by the state's (or state-action pair's) *value function*.

The value function for a state represents how much reward the agent can expect to get in the future from being at that state [3]. To calculate the value function, first we must have the concept of the return.

The return at a particular time is the sum of future rewards after that time:

$$R_t = r_{t+1} + r_{t+2} + \cdots + r_T \tag{2.2}$$

The return can be discounted so that future rewards are worth less than more immediate ones:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \tag{2.3}$$

where γ is the discount factor, $0 \leq \gamma \leq 1$. The return can have a finite or infinite horizon. A finite horizon means that the return is calculated upto a final time step (T in equation 2.2). This is used in applications were there is a final step such as reaching a goal state; these tasks are called *episodic*. Infinite horizon is used for *continuing* tasks that dont terminate and the return includes all future time steps (the sum in equation 2.3). The smaller the discount factor the greater the learning impact of early steps; for episodic tasks the dicount factor can be set to 1 as all episodes are guarenteed to end [3].

The value function of a state s uses the expected discounted return $E_\pi \{R_t\}$ when using a particular policy π and is given by:

$$V^\pi(s) = E_\pi \{R_t | s_t = s\} \tag{2.4}$$

The value of taking a particular action a at state s is given by:

$$Q^\pi(s, a) = E_\pi \{R_t | s_t = s, a_t = a\} \tag{2.5}$$

V^π is called the state-value function for policy π and Q^π is called the action-value function for policy π. The values of these functions can be estimated from experience, by the agent taking actions at various states and remembering the rewards it received [3].

2.3.1 The Optimal Policy

The aim of reinforcement learning is to find the policy that gives the most reward over time; known as the optimal policy. One policy π is better than or equal to another π' if for all states $V^\pi(s) \geq V^{\pi'}(s)$. There is always at least one policy that is better than or equal to all others; this is the optimal policy π^*.

The optimal state-value function is given by:

$$V^*(s) = \max_\pi V^\pi(s) \tag{2.6}$$

And the optimal action-value function is given by:

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) \tag{2.7}$$

An important thing to note for many reinforcement learning algorithms is that Q^* can be used to calculate V^*:

$$V^*(s) = \max_a Q^*(s, a) \tag{2.8}$$

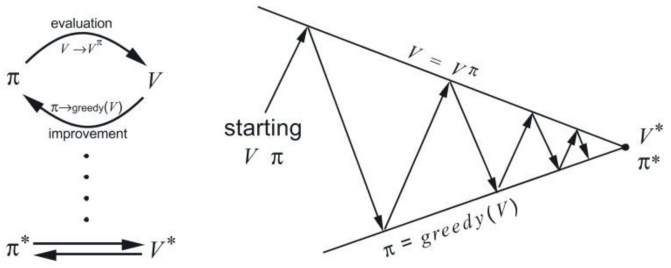


Figure 2.3: Visual representation of Generalised Policy Iteration [3]

This allows learning when there is no model of the environment. From these equations the optimal policy can be calculated:

$$\pi^*(s) = \arg\max_a Q^*(s,a) \tag{2.9}$$

[3, 22]

2.4 Learning Methods

The previous sections gave much of the required theory behind reinforcement learning techniques; this will focus on the methods used to learn the optimal policies described in the previous section. There are two reinforcement learning methodologies that all methods fall under: *model-based* and *model-free* [3, 22]. These are described in the following sections.

An important concept that underlies all the following methods is *Generalised Policy Iteration* (GPI). GPI is made up of two steps: policy *evaluation* and policy *improvement*. Policy evaluation estimates the value of a given policy. Policy improvement evaluates the value of actions at states to find possible improvements to the policy [3]. The details of these two steps vary between algorithms and methods. A visual representation of these steps can be seen in figure 2.3.

2.4.1 Model-Based Methods

Dynamic Programming (DP) algorithms compute optimal policies with a perfect MDP model of the environment. The assumption that there is a perfect model of the environment available is a limitation of DP as this is often not the case in reinforcement learning problems [3, 22]. There are two fundamental DP algorithms: *Policy Iteration* [25] and *Value Iteration* [26]. Policy Iteration follows closely the GPI algorithm described above; an iterative process: evaluation of the policy followed by improvement of the policy. This two phase process can be slow as the optimal policy is only calculated in the limit. Value Iteration combines the two phases into one, only estimating the value function for the optimal policy. Equation 2.10 shows the sequence of policies and value functions generated by policy iteration; equation 2.11 shows the sequence of value functions generated by value iteration.

$$\pi_0 \to V^{\pi_0} \to \pi_1 \to V^{\pi_1} \to ... \to \pi^* \tag{2.10}$$

$$V_0 \to V_1 \to V_2 \to ... \to V^* \tag{2.11}$$

2.4.2 Model-Free Methods

Model-free methods, on the other hand, do not rely on a model of the environment being available. These methods must therefore sample from the environment to gather information about the model, which is unknown. This *exploration* is done by applying actions at states and seeing what reward is received; from this, action-value functions can be estimated and learnt from. Many model-free methods also utilise value iteration but estimate a Q function rather than a value function.

2.4.2.1 Exploration vs Exploitation

For an agent to gain the most reward, it must take actions it already knows to be good. This *exploitation* is in contrast to the above exploration. The agent must therefore have a balance between exploration and exploitation; it must explore new areas of the state space while still exploiting areas already known to be good. There are several ways this trade off can be done; the first is the ε-greedy method. In this method, the currently estimated best action (exploitation) is taken with probability $1 - \varepsilon$; a new randomly selected action (exploration) is taken with probability ε where $0 \leq \varepsilon \leq 1$. The value of ε can either be set to a number chosen empirically or decrease over time having the result that the agent follows its learnt policy closer in later episodes. Having a smaller ε gives slower learning; however, eventually it can achieve better results [27].

Another exploration strategy is the Boltzmann method. In this method the probability of selecting an action is proportional to that action's Q-value. Therefore, good actions will be taken more often than bad actions. Here there is an additional 'temperature' parameter; a high temperature makes the behaviour more random and a low temperature makes the behaviour more greedy. The Boltzmann equation can be seen in equation 2.12 [3, 22]. It is possible to combine this with ε-greedy so that when ε-greedy selects a random action, this action is determined by the Boltzmann strategy [28].

$$P(a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_i e^{\frac{Q(s,a_i)}{T}}} \tag{2.12}$$

2.4.2.2 Temporal Credit Assignment Problem

A problem faced when attempting to learn without a model is that it can be hard to tell how good an action was when a reward is not received until later on. For example, getting 0 reward for an action where this action was crucial in getting a high reward in the future. This problem is known as the *temporal credit assignment problem* [29].

There are two main methods for dealing with this problem. Firstly, one can wait until the end of an episode then punish or reward actions taken during the episode. This method however can use a lot of memory. The alternative method is to update an actions value straight away, taking into account the immediate reward of that action and the estimated reward of future actions [1, 22].

2.4.2.3 Monte Carlo

Monte Carlo methods can be either model based or model-free reinforcement learning methods. The task must be episodic, meaning the task will always 'end' at some point no matter what actions are taken. Monte Carlo (model-free) methods proceed as follows: a whole episode is performed using a particular policy. During the episode, states are *visited* several times by the agent, with rewards being generated at each visit.

There are two methods of handling these visits: *every-visit* and *first-visit* Monte Carlo. In every-visit Monte Carlo the rewards for all visits to state-actions are averaged to give the value for that state-action in that episode. For first-visit Monte Carlo, the reward from the first visit to a state-action is the value for that state. In both methods these values are averaged over all episodes. When each state-action is visited infinitely often, both methods' value estimates converge to the true value of the policy used. Nither method is proven to be better than the other; however, every-visit MC has a bias as the many returns from the same trial share the same rewards and are thus not independent [30]. To ensure all state-action values are visited, random actions must sometimes be taken (exploration) [3, 22, 31].

2.4.2.4 Temporal Difference

Another model-free method is *temporal difference learning* [32]; this is the main method used in reinforcement learning. In this method, estimated state-action pair values are updated throughout execution, after every action when a reward is received, in contrast to Monte Carlo which waits until the end of an episode. This can make learning faster when episodes are long or in non-episodic tasks. At each of these updates, the previous value estimate and the reward are part of the update; therefore, this method uses *bootstrapping* (estimates are based on previous estimates).This method can also be used *on-line*, meaning that it can learn while interacting with the environment [3, 22].

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t) \right] \tag{2.13}$$

The basic temporal difference learning algorithm is TD(0), given by equation 2.13, estimates the value of the policy on a state. γ is the discount factor and $0 \leq \alpha \leq 1$ is the learning rate. This algorithm is on-line and incrementally updates the value of a state. As can be seen from the equation, the new value is based on old values (bootstrapping) after a reward from moving from one state to another is received. The learning rate α is used to determine how much current value determines the new value. As with model-free Monte Carlo, all states (or state-action pairs) must be visited infinitely often to be guaranteed to converge to the true values.

2.5 Learning Algorithms

2.5.1 Q-Learning

Q-Learning, first proposed by Watkins [33], is a temporal difference reinforcement learning algorithm. Q-Learning is an *off-policy* algorithm, meaning that in estimating the optimal policy π^* it actually follows a different policy π. In Q-Learning, values of state-action pairs (*Q values*) are incrementally estimated based on the reward feedback for taking the action in the state and the current Q-value. It is very similar in form to TD(0) and the update rule can be seen in equation 2.14. Rather than just the value of the next state being used to estimate the current Q-Value, the maximum Q-value when varying the action used at the next state is used. It has been shown that when every state is visited infinitely often Q-Leaning will converge to the optimal policy [3, 22, 34]. The algorithm is given by algorithm 1.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.14}$$

Algorithm 1 Q-Learning Algorithm [22]

Require: discount factor γ, learning parameter α
  initialise Q arbitrarily
  for each episode **do**
    initialise s to the starting state
    repeat
      choose an action $a \in A(s)$ based on Q and an exploration strategy
      perform action a
      observe new state s' and recieved reward r
      $Q(s,a) := Q(s,a) + \alpha \left(r + \gamma \cdot max_{a' \in A(s')} Q(s',a') - Q(s,a) \right)$
      $s := s'$
    until s' is a goal state
  end for

2.5.2 SARSA

SARSA (State-Action-Reward-State-Action) [35] is very similar to Q-Learning with the key difference being that it is *on-policy*, which means that the policy it is learning and the policy that it is following are the same. This difference is shown in the update rule by replacing the max operator with the estimated value for the next state-action given the current policy. The update rule for SARSA is given by equation 2.15 where a_{t+1} is the action that is executed by the policy when in the next state s_{t+1}. Like other TD algorithms, this will converge to an optimal policy and value if all state-actions are visited infinitely often [3, 22, 35]. In the limit the optimal policy will be the greedy policy [36]. An advantage of SARSA over Q-Learning is that when the environment is *non-stationary* (i.e. constantly changing over time) SARSA performs better as it is not possible to reach an optimal policy in such an environment [22]. The SARSA algorithm is given by algorithm 2.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \tag{2.15}$$

Algorithm 2 SARSA Algorithm [3]

Require: discount factor γ, learning parameter α
  initialise Q arbitrarily
  for each episode **do**
    initialise s to the starting state
    choose an action $a \in A(s)$ based on Q and an exploration strategy
    repeat
      perform action a
      observe new state s' and received reward r
      choose an action $a' \in A(s')$ based on Q and an exploration strategy
      $Q(s,a) := Q(s,a) + \alpha \left(r + \gamma \cdot Q(s_{t+1}, a_{t+1}) - Q(s,a) \right)$
      $s := s'$
      $a := a'$
    until s' is a goal state
  end for

2.5.3 Actor-Critic Methods

Actor-Critic methods [37, 38] are also on-policy, the difference being that they learn the value function and the policy separately. The part learning the value function is the *critic* and the part learning the policy is the *actor*; the value function 'criticises' the actions taken by the policy (actor). This critique is given by the TD error (equation 2.16), a positive error means the action taken should be taken more often, otherwise it should be taken less. The TD error is used to update the *preference* of an action at a particular state (equation 2.17, β is the size of the update) [3, 22].

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.16}$$

$$p(s_t, a_t) \leftarrow p(s_t, a_t) + \beta\delta_t \tag{2.17}$$

There are two advantages of using actor-critic methods. Firstly, they require minimal computation when selecting actions; this can be particularly useful if there are many (or continuous) actions. Secondly they can learn stochastic policies [3, 22].

2.6 Continuous State and Action Spaces

The above methods for the most part focus on learning a policy in discrete state and action spaces. In these spaces the value function can often be represented as a lookup table when implemented, such that each possible state-action pair corresponds to a value. However many real world problems (such as the topic of this report) are in continuous state and action spaces. Representing these as a table would give huge tables that take a very long time to populate with correct values. Therefore, when dealing with continuous state and action spaces some method of approximation and generalisation must be used. This is because a precise state-action will often not have been visited before but a state-action extremely close may have. For example, the state 1.23 may not have been visited yet by 1.2301 has and so it must be possible to generalise between these nearly identical states. Because of the infinite number of states, visiting all state-action pairs, like the above methods require, is infeasible [3, 22].

Function approximation allows the value and policy functions to generalise to cases that have not been seen. A common method of function approximation is *Linear Function Approximation*. In linear function approximation, *features* are extracted from the state space. These features (ϕ), along with some linear parameters ($\theta \in \Theta$), are then used in the value approximation:

$$V_t(s) = \theta_t^T \phi(s) \tag{2.18}$$

2.6.1 Discretisation and Tile Coding

Tile coding is a method of doing the above mentioned feature extraction. Based on the Cerebellar Model Articulation Controller (CMAC) by Albus [39], tile coding partitions state space into *tiles* known as a *tiling*; each of these tiles represents a binary feature. Binary because any particular state is either in or out of a tile. Multiple tiling can be layered over one another (figure 2.4), and these tilings should overlap (not line up perfectly). As a state can only be in one tile in a tiling (tiling / feature function given in equation 2.19), using multiple tilings gives a number of features equal to the number of tilings [40]. The values of states (or state-action pairs) are then given by summing the weights associated with the active tiles (equation 2.20) [3].

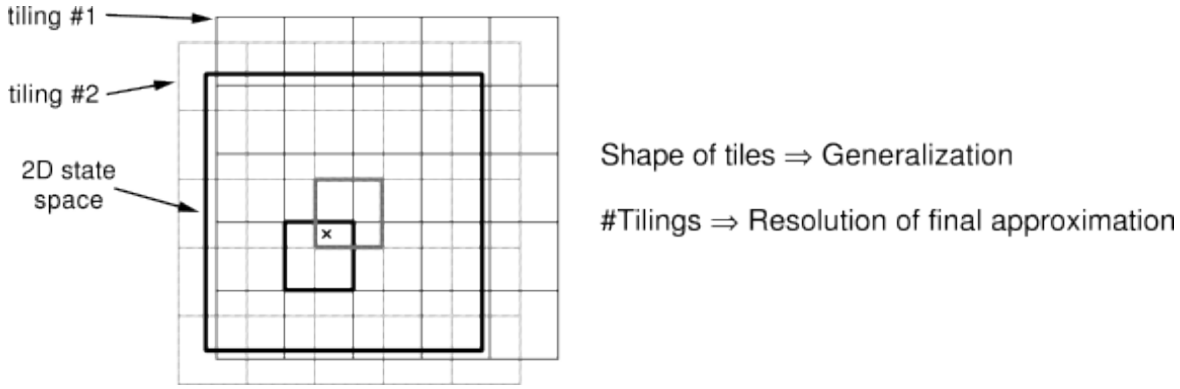


Figure 2.4: Two tilings in a two dimensional state space [3], a point in the state space labelled by x activates the two highlighted tiles.

$$\phi_i^j(s) = \begin{cases} 1 & \text{if } s \text{ is in tile } j \text{ in tiling } i \\ 0 & \text{otherwise} \end{cases} \tag{2.19}$$

$$V(s) = \sum_{i=1}^{N_{tilings}} \sum_{j=1}^{N_{tiles}} \phi_i^j(s)\theta_i^j \tag{2.20}$$

If the state space is large, tile coding can result in there being very many tiles to keep track of and out of these only a small subset of them will ever be hit. This can cause excessive memory usage and slow learning due to the *curse of dimensionality* [41]. This can be countered by a technique known as *hashing* [42]. Hashing reduces the memory usage by only representing the disjoint areas of the state space that have actually been visited, making the memory usage match the size of the task [3].

The tiles in the tilings can be any shape and size. Different tile shapes can give more generalisation in a particular direction in the state space, such as stripes of tiles generalising in the stripe direction and discriminating in the other. Uniform tile coding involves two parameters: tile width w and number of tilings t, giving the resolution of the tile coding w/t; the more tilings the more generalisation. To effectively tile code a problem these parameters must be adjusted to fit the problem as different parameters perform differently on different problems, with no general best choice. To find the best parameters one must experiment empirically; an automated on-line parameter choice algorithm has also been tested with success [43].

The idea of automating tile coding parameter choice can be extended further with the idea of *adaptive tile coding* [44]. Normal tile coding requires someone to define the tiling to be used in the experiment manually, then this tiling is used throughout the experiment. This fixed generalisation can be a limitation as reducing generalisation over time can improve performance [43]. In adaptive tile coding one starts with a few large tiles. These tiles can then be recursively split in two, increasing the number of tiles. Splitting is done when learning slows in a tile. Adaptive tile coding can outperform standard tile-coding but cannot outperform an optimally selected tile coding. Therefore the use of adaptive tile coding is better than using a bad tiling but less good that using the optimal tiling for a problem.

Adaptive tile coding can be built on further with *Evolutionary Tile Coding* [45]. Similarly to adaptive tile coding, evolutionary tile coding starts with just one tile. This tile is then split recursively when necessary. When and where to split, however, are determined by an

evolutionary algorithm. This technique can outperform both standard tile coding (as long as the optimal coding has not been engineered) and adaptive tile coding.

There are several other methods of discretising the state space which are very similar in nature to tile coding. The generalisation of tile coding technique is called *coarse coding* [3, 46], where a number of overlapping circles are placed over the state space in a manner similar to having many single tile tilings. Other methods are: using *radial basis functions* rather than tiles with their centres being located throughout the state space [3] and *Kanerva coding* [47] which is useful when the state space is extremely high dimensional.

The problem with discretising the state space with the above methods is that two states get mapped to the same tiles, so from a tile it it is not possible to tell what the exact state is; it is not injective ($\phi(s) = \phi(s') \not\Rightarrow s = s'$). This means that the problem after being discretised is partially observable, and therefore we are working on a *Partially Observable Markov Decision Process* (POMDP). The learning algorithms in section 2.5 rely of the assumption that they are working on MDPs rather than POMDPs, and therefore their proof of convergence does not hold, although in practice good results have been achieved [22].

2.6.2 Function Approximation Parameters

When using the above function approximation the linear function parameters θ must be updated. This update is given by equation 2.21; where α_t is the update step size, δ_t is the parameter error given by equation 2.22 for Q-Learning or equation 2.23 for SARSA. On policy TD algorithms will converge with linear gradient decent [29, 32]. Standard off-policy algorithms will not converge; however, modified off-policy algorithms have been developed which will converge [48] such as the GQ algorithm [49].

$$\theta_{t+1} = \theta_t + \alpha_t \delta_t \phi(s_t) \tag{2.21}$$

$$\delta_t = r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t) \tag{2.22}$$

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t) \tag{2.23}$$

2.6.3 Continuous Action Space

In addition to a continuous state space, many problems can also have continuous action spaces. The problem presented by continuous actions is that it becomes hard to select which is the best action to take; some form of approximation is needed. As with continuous state spaces, discretisation of actions via tile coding can be used [43, 50].

The problem with discretising the state and action spaces as above is that the size of the table used to store the discretised value functions can grow exponentially with the number of state and action variables. Making the discretisation coarser makes functionally different states or actions seem the same; control tasks need fine grained actions.

Van Hasselt and Wiering [51] introduced a method for dealing with continuous action spaces called *Continuous Actor Critic Learning Automaton* (CALCA). CALCA is a model-free on-line temporal difference learning algorithm based on actor-critic methods. Gaskett et al. [52] propose eight criterion they believe are necessary and sufficient for continuous state and action space problems where actions vary smoothly with smooth variations in state (no discretisation). Six algorithms are assessed against (and do not meet all) these constraints: Adaptive Critic Methods [53], CMAC Based Q-learning [50], Q-AHC [54], Q-Kohonen [55], Q-Radial Basis [56], Neural Field Q-learning [57].Their proposed method is called "Wire-fitted Neural Network

Q-Learning" [52], this method uses a single feed-forward neural network with a wire-fitter to fulfil their criteria.The algorithm inputs the state into the neural networks and performs the action with the highest Q value, thus moving to the next state. A new estimate for the state and action values are calculated from the previous and current state, the action, and the reward using the wire fitter. The new Q values are used to train the neural network. This algorithm performed well on their simulated tests.

2.7 Additional Techniques

This section describes techniques that can be used in addition to the methods described in the previous sections to improve the performance of reinforcement learning.

2.7.1 Eligibility Traces

Eligibility traces are a way of combining Monte Carlo and Temporal Difference learning methods combining the learning benefits of both. In MC methods, a reward is received at the end of an episode then all the values of state-actions visited to get to the terminal state are updated. In TD methods, state-action values are updated immediately after the reward is received when taking the action. Eligibility traces bring these two ideas together so that at a particular time, the value update is determined both by the previous value and the reward (as with TD), but also state-actions up to this point are updated as well (as with MC). Eligibility traces mean that the more recently a state-action had been visited, the more it affected the current state-action, thus the more it influenced the current reward. A state's eligibility decays over time, so when it has not been visited in a long time it has no eligibility [3].

Eligibility traces are implemented using a backwards view; each state-action has an additional variable $e_t(s,a)$ known as its eligibility trace (at time t). When an update occurs, the current state's eligibility is increased. All other eligibilities decrease by a factor of $\gamma\lambda$ where γ is the learning discount factor and $0 \leq \lambda \leq 1$ is the *trace decay parameter*. When $\lambda = 0$ the TD algorithms behave as if eligibility traces were not included; if $\lambda = 1$ then TD methods behave the same as MC [3].

Eligibility traces are incorporated into TD methods to make TD(λ) algorithms [32] by equation 2.24. δ_t is given by equation 2.22 for Q(λ) [33, 58] or equation 2.23 for SARSA(λ) [3].

$$Q(s,a) = Q(s,a) + \alpha\delta_t e_t(s,a) \tag{2.24}$$

The standard way of calculating the eligibility of a state is known as *accumulating traces* (equation 2.25). This method can cause slow learning if state-actions are visited more often than it takes their eligibility to decay to 0, resulting in their eligibility being greater than 1. A solution to this is to use *replacing traces* [59], where the current state is set to one, and all others are decayed (equation 2.26) [3].

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t \end{cases} \tag{2.25}$$

$$e_t(s,a) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t \\ 1 & \text{if } s = s_t \end{cases} \tag{2.26}$$

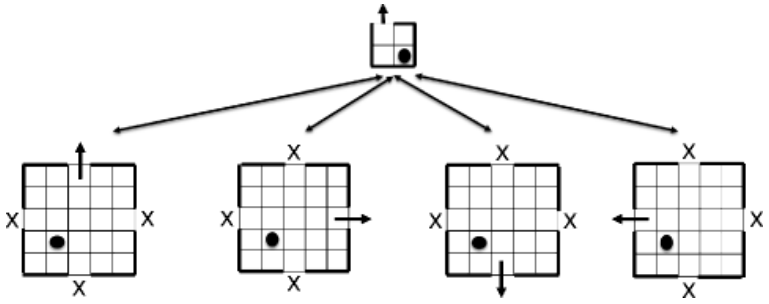


Figure 2.5: Four room problem hierarchy example [4]

2.7.2 Reward Shaping

In standard exploration the action to explore is, for the most part, chosen randomly. Additional information can, however, be used to guide the learner; this is known as *shaping* [60]. First used by Matarić [61] in the form of "progress estimators", shaping moves the reward closer to intermediate sub-goals having the effect that the agent is encouraged to search for the best policy more effectively.

If the reward shaping function is implemented arbitrarily, or without careful consideration in regards to the goal of the task, the agent may learn a policy which is optimal for the shaped rewards but not for the actual problem. Randløv and Alstrøm [62] successfully demonstrate the potential of reward shaping in improving reinforcement learning with an experiment in learning to ride a bike to a goal. However, they note that incorrect shaping lead their learner to ride in circles, achieving the maximum shaped goal if the shaping was badly formed.

Ng et al. [60] use potential-based reward shaping to mitigate the issue of bad shaping. In this method the shaping reward is combined with the reward form the environment by equation 2.27 and 2.28 where R' is the reward after shaping, R is the reward from the environment, F is the potentially shaped reward and $\Phi : S \to \mathbb{R}$ is the shaping function based on the state.

$$F(s_t, a, s_{t+1}) = \gamma\Phi(s_{t+1}) - \Phi(s_t) \tag{2.27}$$

$$R'(s_t, a, s_{t+1}) = R(s_t, a, s_{t+1}) + F(s_t, a, s_{t+1}) \tag{2.28}$$

2.7.3 Hierarchical Reinforcement Learning

Many reinforcement learning problems suffer from the curse of dimensionality [22], where the number of parameters to learn increases exponentially with the size of the state space [41]. Hierarchical reinforcement learning can reduce the size of the state space by utilising the underlying structure and independences in a problem; this is done by encoding this prior knowledge into the structure of the hierarchies [63, 64]. Each part of the hierarchy is a smaller reinforcement learning problem, with the reinforcement learning problem being learnt at each level separately producing the overall solution to the problem.

A simple example is the four room problem where the agent must learn to exit the rooms. Each room is the same (except the position of the door) and so the problem can be broken down into a hierarchy (see figure 2.5). The top level is learning movement between the four rooms and the lower level is learning how to exit an individual room [4].

The actions performed by a higher level RL problem are *temporally extended* or *abstract* actions. When activated, they invoke a sub RL problem which may perform many of its own actions

(over a period of time). By the use of these abstract actions the MDP problem becomes a *semi Markov Decision Process* (SMDP) [22, 65]. SMDPs differ from MDPs in that the transition and reward function contain a random variable which represents the amount of time it could take for an abstract action to complete. Sutton et al. [27] have used these techniques to speed learning between MDP and SMDP use. An implementation of Hierarchical Reinforcement Learning is the MAXQ algorithm by Dietterich [66].

3 Helicopter Control

In this section the problem domain of this project is detailed and previous work discussed. First general helicopter dynamics are described, followed by their representation in the simulator used for this project. Finally, previous applications of reinforcement learning to helicopter control are discussed, in both reality and simulation.

3.1 Helicopter Dynamics

There are many forces that act upon helicopters. Firstly, gravity is constantly pulling the helicopter down towards the ground; this is countered by the upward thrust created from the spinning main rotor generating lift. By applying torque to the main rotor to make it rotate, an anti-torque is exerted on the body of the helicopter, making it spin in the opposite direction. The tail rotor of a helicopter counters this rotation from the anti-torque by blowing air sideways. This then has the effect of pushing the helicopter sideways; so, to hover a helicopter must counter this small sideways force by leaning slightly to the side. The spinning nature of the blades means that to get the helicopter to move in one direction a force must be applied at 90 degrees to that direction; this is due to gyroscopic precession.

The above forces and helicopter dynamics can be abstracted to give three helicopter controls: cyclic, collective and anti-torque. Cyclic controls the main rotor blade pitch having the effect of tipping the helicopter forwards/backwards and side-to-side, causing the helicopter to move in those directions. Collective controls the angle of all the main rotor blades collectively (all at the same time) having the effect of increasing/decreasing the thrust produced by the blades, thus controlling vertical height and speed. Anti-torque peddles are used to control the tail rotor effecting the heading of the helicopter [18, 67, 68]. These forces and controls can be seen in figure 3.1.

3.2 Helicopter Simulator

A simulator is provided as part of the reinforcement learning competition's helicopter domain [16]. This simulator is based on RL-Glue: a framework facilitating communication between agents and environments for reinforcement learning experiments [69]. The competition software provides the environment of the experiment: MDPs, physical simulations and calculation of reward feedback. The goal of the helicopter is to hover at the origin; the reward, given by the environment from reinforcement learning competition software, is the sum over all the state variables of the squared difference between the state variable at a given time t and the target position (the origin). This is given by equation 3.1. This means each state should be as close to 0 as possible to gain maximum reward.

$$r_t = -\sum_{s \in S_t} s^2 \tag{3.1}$$

The helicopter has twelve continuous state observations and four continuous actions it can perform, shown in table 3.1, each with various input/output ranges. The state variables include the helicopter's velocity in each direction, the distance the helicopter is from the origin in each

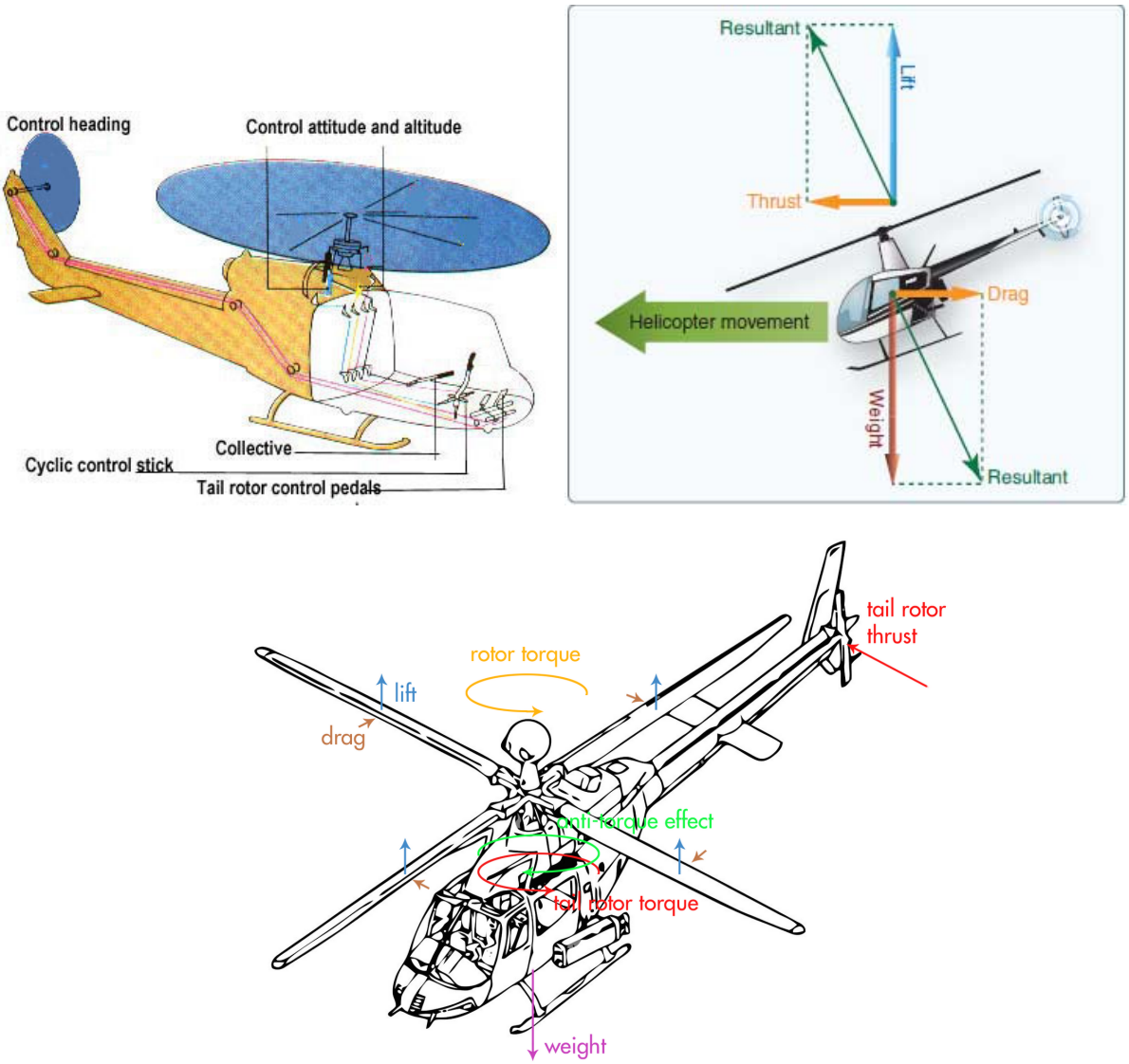


Figure 3.1: Helicopter controls [5] and the forces upon the helicopter [6, 7]

State		Range	Action		Range
Forward velocity	u	$-5..5$	Collective	c	$-1..1$
Sideways velocity	v	$-5..5$	Longitudinal Cyclic[1]	e	$-1..1$
Downward velocity	w	$-5..5$	Latitudinal Cyclic[2]	a	$-1..1$
Forward error	x	$-20..20$	Anti-torque	i	$-1..1$
Sideways error	y	$-20..20$			
Downward error	z	$-20..20$			
Angular rate around forward axis	p	$-12.566..12.566$			
Angular rate around sideways axis	q	$-12.566..12.566$			
Angular rate around vertical axis	r	$-12.566..12.566$			
Rotation around forward axis	q_x	$-1..1$			
Rotation around sideways axis	q_y	$-1..1$			
Rotation around vertical axis	q_z	$-1..1$			

Table 3.1: Helicopter Simulator state inputs and action outputs.

direction, the rate the helicopter is rotating around each axis and the degree of rotation around each axis. The four actions are the same as those described in the previous section.

To make the simulation more realistic additional effects are added to the state transitions in the helicopter environment. These include adding random noise to the helicopter's state to simulate inaccuracies in real sensors and the addition of wind which randomly pushes the helicopter. The wind and noise are not observed by the agent, so their effects cannot be learnt. The wind is particularly difficult as it augments the effect of the chosen action randomly with every step. The simulation, however, does no include the latency between applying an action and it taking effect that a real helicopter would have.

The helicopter simulator is episodic and has discrete time steps; each step is a tenth of a second. There are two end conditions for an episode. Firstly, the maximum length of an episode is $6,000$ time steps. The second is a failure condition; if any state variable exceeds its range (shown in table 3.1) the episode ends. This represents the helicopter reaching an unrecoverable state or crashing. At this point a reward equal to the maximum negative reward for the remainder of the total episode is received by the agent [16].

3.3 Reinforcement Learning for Helicopter Control

This section covers previous work on the use of reinforcement learning in the context of helicopter control. There are two areas of interest: the use of reinforcement learning with real helicopters and the use of reinforcement learning with regards to simulated helicopters.

3.3.1 Reinforcement Learning on Real Helicopters

Reinforcement Learning can used to learn the controls of real world helicopters [15, 70–75]. In doing so there are several challenges to be faced. Two of these challenges are that collecting data (exploration) can be costly (in terms of fuel) and dangerous (in terms of not wanting the physical helicopter to crash due to the exploration). Another difficulty is that not all of the environment outside the helicopter can be measured resulting in Partially Observable MDPs (POMDPs), which is exacerbated by the continuous nature of the environment.

[1] Forward-Backward control
[2] Left-Right control

Bagnell and Schneider [70] propose a policy search method with the goal of their experiments being to hover the helicopter or put it on a slowly varying trajectory. In their method, at each point in time an off-line simulation of the helicopter is run for n steps in a similar fashion to Monte-Carlo Learning. The result of this simulation updates a stored model and an action is chosen. The controller for computing the policy consists of two feed forward kernel based neural networks; the inputs are the observed variables and the outputs are adjustments to be made in terms of latitude and longitude. The model updates affect the weights in the network. This method was first tested on a simulator, then once confirmed to work was ported to a remote control helicopter. The real helicopter was able to fly in windy conditions better than a experienced pilot.

Ng et al. [15] experiment with getting a helicopter to hover using reinforcement learning; then performing aerobatic manoeuvres. Rather than using all 12 world coordinate variables they transform these into 8 body relative coordinates; reducing the state space significantly. In their experiments they first developed a model of the task they wanted the helicopter to perform (such as hover) by having an experienced pilot perform the task and recording the states and actions. The policy is defined by a neural network where the edge weights are what is to be learned. A reward function that penalises large distances, velocities and adjustments of actions is used. The Pegasus algorithm [76] was used which approximates utilities of policies (a function of the rewards estimated Monte-Carlo simulations at each time step using the derived model). Then hill climbing is used to estimate the optimal policy. Similar methods were again used by Ng et al. [72] for inverted helicopter flight.

A method similar to reinforcement learning known as *Apprenticeship Learning* [77] has also been applied to helicopter flight [74]. This is where an expert demonstration is used to guide the learning.

3.3.2 Reinforcement Learning on Simulated Helicopters

Work on reinforcement learning on simulated helicopters has also been conducted. The use of simulators allows algorithms and techniques to be tested without risk of damaging physical helicopters or their surroundings. Additionally many more tests can be carried out as simulation-time can progress faster than real-time. However, a simulation cannot have all the variables and fidelity of the real world. Many of the methods described in the previous section use internal simulations to decide on the best policy, so can benefit from research in simulations. The two papers reviewed in this section are entries into the reinforcement learning competitions of previous years. Both perform well, reaching the end of episodes without failing early.

Koppejan and Whiteson [78] use neuroevolutionary [79] reinforcement learning to achieve generalised helicopter control; on-line helicopter control and learning with no prior knowledge of the MDP. Neuroevolutionary reinforcement learning is a method where the rewards from traditional reinforcement learning are used to guide the structural evolution of a neural network. The neural network is used to define the policy; thus the policy is evolved. It is noted that only nine of the twelve inputs are necessary as the angular rate variables can be derived from the other nine.

Asbah et al. [80] explore two reinforcement learning methods to learn helicopter hovering. The first involves the use of controllability [81]; this is a method of guiding exploration so that only good areas of the state space are explored. The controllability of a state-action pair is inversely proportional to the variability of the TD error for that pair. An action is picked using a combination of the Q value and the controllability of the state (with some weighting); this can then be incorporated into reinforcement learning algorithms such as SARSA.

The second method used by Asbah et al. [80] is kernel-based reinforcement learning [82]. This

method approximates the continuous MDP with a finite MDP by the use of kernel functions. The kernel function maps the transition and reward functions onto the new MDP. KBRL is useful because it handles continuous state-action spaces and is guaranteed to converge to a unique solution no matter what its initialisation.

4 Helicopter Learner

This chapter covers the engineering aspects of this project. Firstly the aims of the project are defined, then the requirements of the helicopter learning system are defined. Based on theses requirements, the architecture and design of the helicopter learner software is derived. Finally, an experimental method using the software is presented.

4.1 Project Objectives

The aim of this project is to apply reinforcement learning techniques to the helicopter domain presented in chapter 3 and be, in effect, a pseudo entry into the Reinforcement Learning Competition 2014 [16]. Section 3.3 covered previous attempts at using reinforcement learning techniques for helicopter control; however, these used highly specialised algorithms to solve the problem. This project performs an exhaustive parameter study of fundamental and general purpose reinforcement learning algorithms described in chapter 2. The goal is to establish the performance of these algorithms and ascertain whether the heavily tailored algorithms of previous research are necessary to perform well in the helicopter domain. Furthermore, there is little research of the use of the basic algorithms for complex problems, particularly with regards to continuous action spaces, such as in this helicopter domain. This project aims to fill this void.

4.2 Requirements and Development Approach

Here the requirements of the helicopter learning software built during this project and used for the experiments in chapter 5 are presented. Before requirements can be elicited, stakeholders must be defined; these are people that have interest in the project and influence its requirements. A list of stakeholders can be seen in table 4.1.

Following identification of stakeholders, requirements can be elicited. Functional requirements for the project software can be found in table 4.2, non-functional requirements can be found in table 4.3.

For the implementation phases of this project an Agile development approach will be used. External libraries will be used (RL Competition Software and Reinforcement Learning Libraries) and a helicopter agent will be developed. The needs of the agent are likely to change over the course of the project by direction of the Researcher; experiments are undertaken with their results feeding back into development, such as implementing or adjusting learning algorithms. Therefore, the development methodology must be flexible to changes proposed by the Researcher and cope with unforeseen issues with the use of external libraries.

Agile development gives these flexibilities with its iterative approach. The goals of each agile sprint will be set by the Researcher. The first of which is to implement a baseline agent using RL-Glue and the RL Competition software. Following this, the goals will be to implement various reinforcement learning algorithms and techniques between experimental phases. The iterative agile development also allows for the addition of any other tasks needed to solve implementation problems.

Stakeholder	Description
Supervisor	This is the person who originally specified to the project. Throughout the project they provide feedback and guidance towards the direction of the project. They are also one of the markers of the project.
Researcher	This is the person performing the research segment of the project. They have a computer science background and knowledge of artificial intelligence and machine learning. They give the primary research direction of the project with guidance from the Supervisor and provide solutions to problems encountered.
Developer	This is the person who implements to software used for performing the research in the project as directed by the Researcher and the requirements. They also have a computer science background with programming experience.
Reinforcement Learning Competition	This project attempts to solve problems defined in the reinforcement learning competitions' helicopter domain. As such, competition software will be used and have influence.

Table 4.1: Project Stakeholders

ID	Requirement	Reasoning
FR1	The environment must provide the agent with an observed state of the environment at each time step.	Reinforcement Learning requires the agent be able to observe (partially) the current state of the environment.
FR2	The environment must provide the agent with a reward at each time step.	Reinforcement Learning requires the agent be shown a reward indicating how 'good' the last state-action pair was.
FR3	The environment must move from one state to another when given an action by the agent.	Reinforcement Learning requires the agent decide on an action and apply it to the environment.
FR4	Multiple reinforcement learning algorithms and techniques described in chapter 2 must be available to test in the simulator. This includes value estimation, storage and action selection.	This will provide a variety of experiments to perform.
FR5	The reinforcement learning parameters ε, α, γ and other algorithm specific parameters should be configurable.	This will allow for these parameters to be experimented with.
FR6	It must be possible to change the number of experiments run and their duration.	This will allow for experimentation with different numbers of episode and allow for an episode count that does not take too long to compute.
FR7	The system must provide experiment output in a format that is easily analysable	This will allow for analysis of the result of the experiments.

Table 4.2: Functional Requirements

ID	Requirement	Reasoning
NFR1	The agent should be able to learn how to hover	This is the primary goal of the reinforcement learning competition.
NFR2	The software should be modifiable and extensible	New reinforcement learning algorithms and techniques may need to be added in the future and throughout the project.

Table 4.3: Non-Functional Requirements

4.3 Helicopter Simulator Software

This section describes the architecture of the helicopter simulator and the design of the helicopter agent. Since this project is closely related to the Reinforcement Learning Competition, the learning agent must be integrated into the existing competition software. The software is Java based and, as described in section 3.2, provides the environment for the learning agent and uses RL-Glue to communicate with the agent.

There are two software components that require implementation for this project: a trainer and an agent. The trainer controls the experiment; it defines the number of episodes in the experiment, controls the learning and evaluation phases and interfaces with the environment via RL-Glue. The agent contains the learning algorithms. It communicates with the trainer (and thus, the environment) via RL-Glue. An abstract architecture diagram can be seen in figure 4.1 (top). The left package represents the Trainer; it contains the trainer and an RL-Glue control instance along with the helicopter environment package. The right package represents the Agent; an `AgentLoader` loads and initialises the `HelicopterAgent` which is an implementation of an `AgentInterface`. Details of the `HelicopterAgent` are in section 4.3.1

A simplified example of the interactions between the Trainer and the Agent can be seen in figure 4.1 (bottom). The Trainer and the Helicopter Agent run as separate processes and communicate via RL-Glue. The sequence diagram shows the initialisation of the experiment and single training phase (loop: number of episodes). In an experiment there will be multiple training phases with evaluation phases in between. Evaluation phases have the same form as training phases but prior to the phase the Trainer sends the message `agent_message("freeze-learning")` which causes the agent to not learn and to only take greedy actions (no exploration). At the end of the evaluation phases, the Trainer calls `agent_message("unfreeze-learning")` to re-enable learning of the agent for the next training phase.

4.3.1 Agent Software Design

The `HelicopterAgent` shown in figure 4.1 is not a single class but a class hierarchy for the various experiments performed in this project. A simplified class diagram showing this structure can be seen in 4.2. This is the structure after all iterations of agile development have been completed.

There are, in fact, four direct implementations of `AgentInterface` for this project: `HelicopterAgentJava`, `HelicopterAgentQ`, `HelicopterAgentSARSA` and `TileCodedAgent`. `HelicopterAgentJava` is an agent that does no learning and was provided as part of the Reinforcement Learning Competition software; instead it uses a hard coded PID policy. This hard coded policy, known as `agent_policy()` is capable of hovering the simulated helicopter

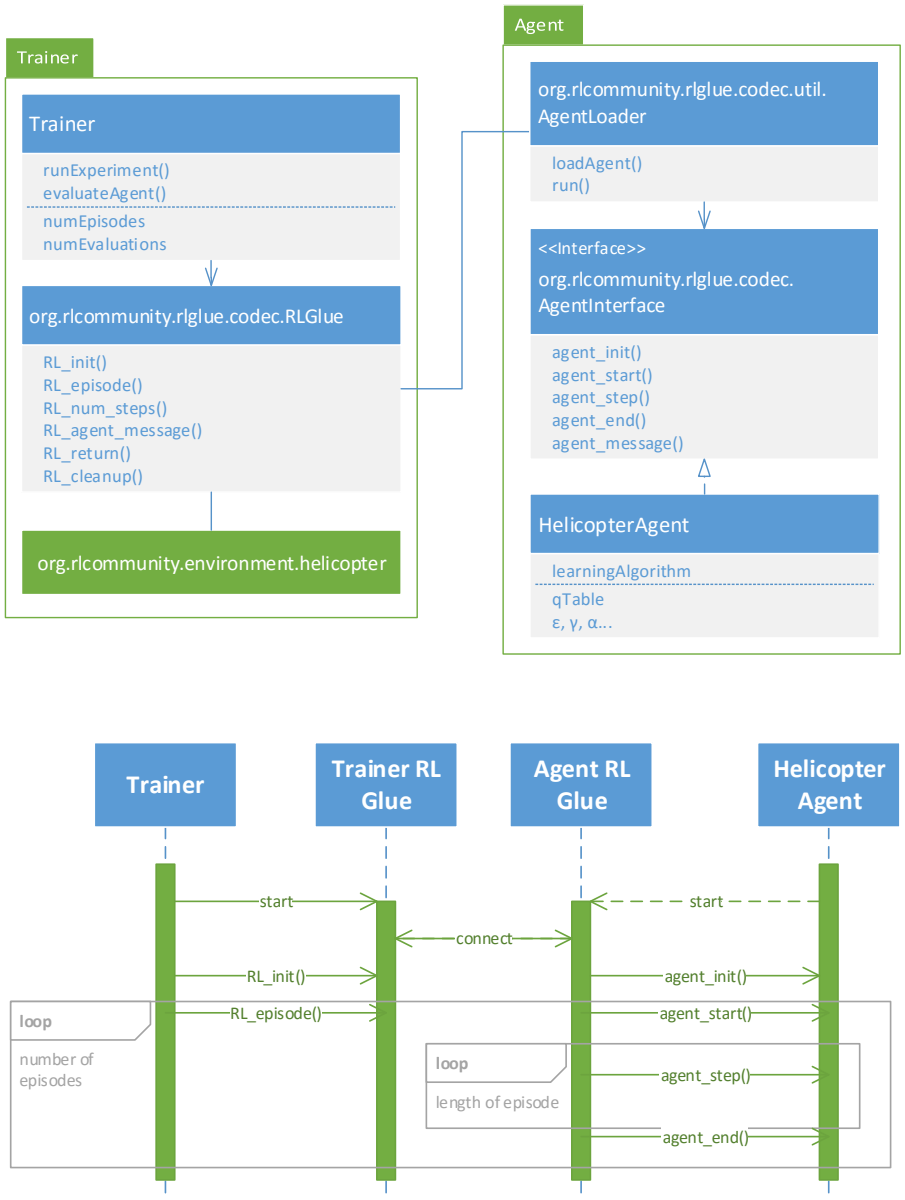


Figure 4.1: Helicopter Software Architecture (top) and Sequence Diagram (bottom)

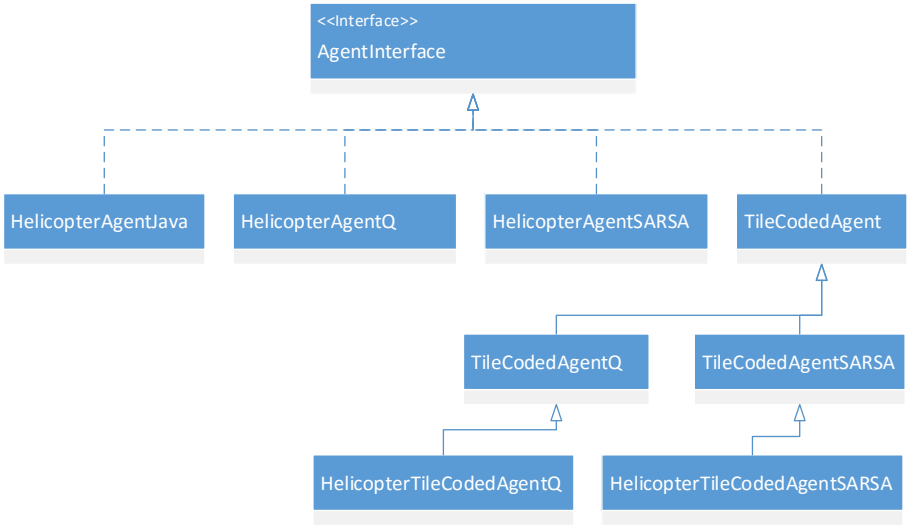


Figure 4.2: Helicopter Agent Class Diagram

for a full episode duration (6000 time steps) but is not optimal as much negative reward is still received. `agent_policy` is given by equations 4.1 to 4.4; h is a set of constant weights, the other symbols used are for the four actions and the state variables defined in table 3.1.

$$c = h_z z + h_w w + h_c \tag{4.1}$$

$$e = -h_x x - h_u u + h_{q_y} q_y + h_e \tag{4.2}$$

$$a = -h_y y - h_v v + h_{q_x} q_x + h_a \tag{4.3}$$

$$i = -h_{q_z} q_z \tag{4.4}$$

`HelicopterAgentQ` and `HelicopterAgentSARSA` are basic implementations of Q-Learning and SARSA with no consideration for the continuous nature of the states and actions. These are used as a baseline to compare the other learning methods. Both these classes are extended further with discretised version of themselves. The discretised classes contain a selection of hard coded discretisations of the state and action spaces, the purpose being to deal with the continuous nature of the problem and compare against the baseline and tile coding.

The majority of the code and implementation work for this project is in the `TileCodedAgent` type hierarchy. `TileCodedAgent` contains much of the RL-Glue functionality necessary to implement in the agent. It also contains base tile coding and agent functionality such as the Q-Table and action selection. `TileCodedAgent` is extended by four classes (only two shown in figure 4.2): `TileCodedAgentQ`, `TileCodedAgentSARSA`, `TileCodedAgentQ`λ, `TileCodedAgentSARSA`λ. Each of these implements a reinforcement learning algorithm: Q-Learning, SARSA, Q(λ) and SARSA(λ) respectively (the latter two including eligibility traces). At the next, and final, layer of the inheritance tree are agents corresponding to individual experiments. Only two are shown, however there is one class here for each of the experiments

presented in chapter 5. These classes implement any functionality, and provide initialisation, needed on an experiment specific basis.

Experiment parameters, such as ε, γ and α, are set in a configuration file `experiment-Config.ini` and are read into the program, in the class they are needed, at initialisation. Experiments can be quickly started by changing the parameters in plain text rather than recompilation. The configuration file and class structure means creation of new experiments is trivial as only a new base level class is needed, or parameters changed. Addition of new reinforcement learning algorithms and techniques is also simple (other than the design and implementation of the algorithm itself) by adding a class at the appropriate level in the type hierarchy.

4.3.2 Software Implementation and Challenges

The implementation of the Agent and Trainer must be compatible with the competition software which is implemented in C++ and Java. Java was chosen for the implementation of this project due to previous development experience in its use, the fact that it is easily portable between Mac OS X, Linux and Windows and that reinforcement learning libraries are available for use with Java [83, 84].

The helicopter domain of this project has a continuous state and action space. The continuous action space proved to be a particular difficulty as neither the Brown-UMBC Reinforcement Learning and Planning (BURLAP) [84] library nor the York Reinforcement Learning Library (YORLL) [83] support continuous action spaces; therefore, the library version of Q-Learning and SARSA could not be used. The learning algorithms from YORLL were re-implemented with support for continuous action spaces.

A further challenge is the high dimensionality of the domain: twelve state dimensions and four action dimensions, all continuous. Tile coding reduces the state space from being infinite to being finite; however, it is still large. Say there are ten tiles in each of the twelve state dimensions and four action dimensions and sixteen tilings over the states and actions. This gives $10^{12} * 16 = 16,000,000,000,000$ possible state tiles and $10^4 * 16 = 160,000$ possible action tiles. Additionally, any of the actions could be used with any of the states; each combination being given a weight in the learning algorithm as described in section 2.6.2. This is more tiles than the tile coding implementation in YORLL (originally by Sutton [85]) was designed to handle; a small modification of changing all internal uses of `int` to `long` was required to handle these large numbers.

The tile coding algorithm utilises hashing (described in section 2.6.1), meaning only tiles that have been visited need to be stored, thus significantly reducing the stored tiles and memory usage. However, memory usage is still extremely high as millions of state-action tile pairs are visited while learning. To utilise the hashing of the tile coding algorithm Java `HashMap`s were used for the Q-Table; however, these were found to be slow and to exacerbate the memory usage issue. An alternative map implementation, Koloboke Collections [86], was found that uses less memory and is higher performance when dealing with insertions and lookups due to its focus on primitive types (which the hashed tiles and tile values are).

4.4 Experimental Method

There are many experiment parameters involved in reinforcement learning: α, γ, ε, state tiles, state tilings, action tiles, action tilings, the algorithm used and λ when using eligibility traces. A methodical approach to finding the best values for these parameters must be devised. The method used is, for each learning method, to arbitrarily set all but one of the parameters;

then, vary the value of the single parameter over a series of experiments. The best performing parameter value (the value generating the highest reward) is then fixed and another parameter value is varied. This is repeated for all parameters for all learning methods. The value of γ (the discount factor) is fixed at 1 for all experiments; this is specified by the competition software and also helps to reduce the total number of experiments. A discount factor of 1 means that learning at every episode is of equal importance, rather than early learning taking greater precedence ($\gamma < 1$) [3]. Having $\gamma = 1$ is possible as all episodes are guaranteed to end.

This experimental procedure implies that all parameter choices are independent. This is not the case; for instance, the number of tiles and tilings are closely related and combined give a tiling resolution. The alternative option is an exhaustive parameter search, which is infeasible. For the arbitrary parameter selection, values known to be generally good choices are chosen as a starting point.

An episode count and an evaluation count must be decided on. After preliminary experimentation it was found that $20,000$ provided a balance between experiment duration (in terms of real world time) and learning with the learning curve flattened by the end of the experiment. Asbah et al. [80] (a previous entry in to the RL competition helicopter domain) experiment over $200,000$ episodes; their learning curves also start to flatten at around $20,000$ episodes.

The $20,000$ episodes are evaluated 200 times, giving an evaluation every 100 learning episodes. During an evaluation, learning is paused and only greedy actions are taken. 100 evaluations are averaged to give a single point on a learning curve graph for the experiment. This gives enough points on the curve to be able to visualise the learning and reduce the amount of noise. A moving average of 10 points is applied to the learning curve to aid in comparing different parameter values by smoothing the curve.

The varying reinforcement learning methods and parameter choices are evaluated primarily in terms of the average number of steps achieved by the agent rather than the total average reward gained over the evaluation episodes. This was chosen because of a number of factors. Firstly, no agent reaches the total of $6,000$ steps per episode; so all have a very large negative reward [16]. Secondly, the reward received can be highly variable, even when the same number of steps are reached.

The average steps are considered in multiple ways when evaluating which parameter choices and methods are most effective. Primarily the number of steps achieved over the final ten evaluations is used to evaluate one parameter choice over another. Other factors are: how early in the learning curve the agent's learning peaks, the stability/variability of the learning curve and the memory usage during learning. Standard error from the mean (SEM) is used on the reward gained (which is correlated with the steps archived) to determine whether one parameter choice is statistically significantly better performing than another.

The experiment time must also be considered. Experiments can range in computation from from a quarter of an hour to two hours, most being around half an hour. Due to this computation time, running many experiments is extremely time consuming; therefore, only a subset of possible parameter choices can be tested. The majority of experiments presented in this report were conducted over a dedicated two week period.

5 Experimentation and Results

In the following sections various experiments of applying reinforcement learning to the helicopter domain are carried out using the experimental method described in the previous section. Each section within this chapter presents a series of experiments using a different form of reinforcement learning starting with a baseline agent, manual discretisation then moving on to tile coding. Once tile coding parameters have been established, experiments with additional reinforcement techniques are performed, such as the use of eligibility traces and reward shaping.

5.1 Baseline Agents

In this first series of experiments three agents are tested. The first is `HelicopterAgentJava` which is a non-learning agent which follows a predetermined hard-coded policy. The second and third are Q-Learning and SARSA agents that have no consideration for the continuous nature of the helicopter domain. The purpose of these experiments is to firstly provide a comparison between the performance of a reinforcement learning agent and a hard-coded agent; secondly, to compare the most basic reinforcement learning methods against the more complex in the later experiments.

Figure 5.1 show the rewards gained from the hard coded policy that was provided with the competition software. As expected, there is no increase in reward over time because there is no learning in this agent. What can be seen, however, is the large variability in reward gained, even though all episodes reached the maximum of $6,000$ steps. The maximum reward received is approximately $-76,000$ and the minimum $-80,000$. These large negative rewards show that this hard-coded controller is not optimal, as much improvement in rewards is possible; the maximum reward possible is 0. The large variations is due to the non-deterministic nature of the environment: random noise on observations and random wind effects. In some cases the noise will cause a non-optimal action to be taken for the true state and in some cases the wind will move the agent further from the goal position, thus reducing the reward.

Figure 5.2 shows the learning curves of Q-Learning and SARSA for $\alpha = 0.1$ and $\varepsilon = 0.1$. These implementations have no regard to the continuous nature of the domain. Experiments of

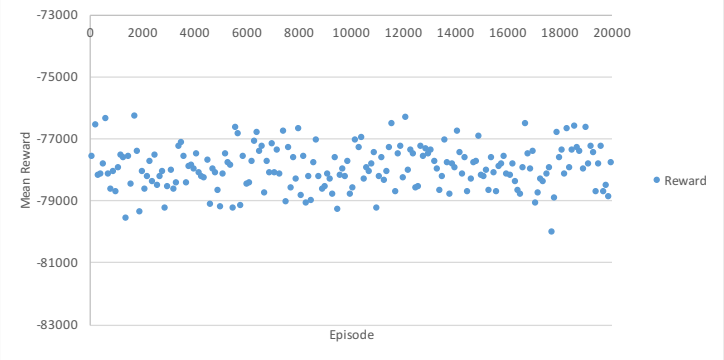


Figure 5.1: Reward gained from a non-learning hard-coded policy agent.

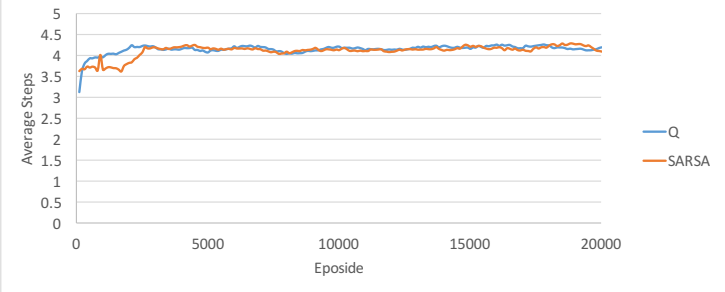


Figure 5.2: Learning curve of basic reinforcement learning implementation

varying α and ε values were carried out with no effect on the agent's ability to learn. It can be seen that after about 3,000 episodes the agent has learnt enough to last one additional step in the episode before crashing; no learning continues after this point however.

The reason for the lack of learning is that there is no generalisation. Generalisation is necessary for continuous domains as described in section 2.6. With a continuous state space there are an infinite number of states. Therefore, it is extremely unlikely that the same state will ever be visited multiple times. Thus, it is not possible for the agent to learn what action to take in each state. Reinforcement learning requires visiting every state-action pair infinitely often [3] or at least a large number of times, which will never happen in this case.

5.2 Basic Discretisation

The above results demonstrate that the continuous state and action spaces must be discretised in some way. Here this is done by partitioning the state and action spaces into separate areas. All the states within an area are generalised to being the same state. Once this has been done, the reinforcement learning algorithms continue in their discrete tabular fashion. The action space is reduced by only allowing a specific subset of the actions. The purpose of this experiment is to establish that discretisation of the state space is necessary and to compare this against with the more general method of tile coding. In these experiments $\alpha = 0.1$ and $\varepsilon = 0.1$; varying these parameters had negligible difference to performance as little is learnt by the agent.

Two state discretisations and three action discretisations were tested; their descriptions can be seen in table 5.1 and their learning curves for both Q-Learning and SARSA can be seen in figure 5.3. The two state discretisations were chosen as they allow comparison between many discrete states and few discrete states. The action discretisations 0 and 2 allow for the same analysis. Action discretisation 1 shows the effect of the agent taking the same action at every time step.

It can be seen in figure 5.3 that the learning curves of Q-Learning and SARSA are similar for each of the discretisations. Q-Learning performed slightly better in the two experiments for action discretisation 2. However, it should be noted that the performance of both learning algorithms are very unstable, with high variation in the number of steps reached in each test. This can be seen in the graphs by the sharp variations in the learning curves. This is most likely due to poor generalisation and is exacerbated by the random noise and wind from the environment. Furthermore, this discretisation method, unlike tile coding, does not allow for any hill climbing to the optimal solution.

An interesting observation from these learning curves is that the action discretisation has more effect over the shape of the learning curve than the state discretisation. For instance the

State	Description	Action	Description
0	States 0 to 8 rounded to the nearest integer, states 9 to 11 rounded to the nearest 0.1	0	Actions are rounded to the nearest 0.1
1	States are rounded such that they are either positive or negative	1	Actions are always 0.
		2	Actions are always ± 0.5

Table 5.1: Basic Discretisations

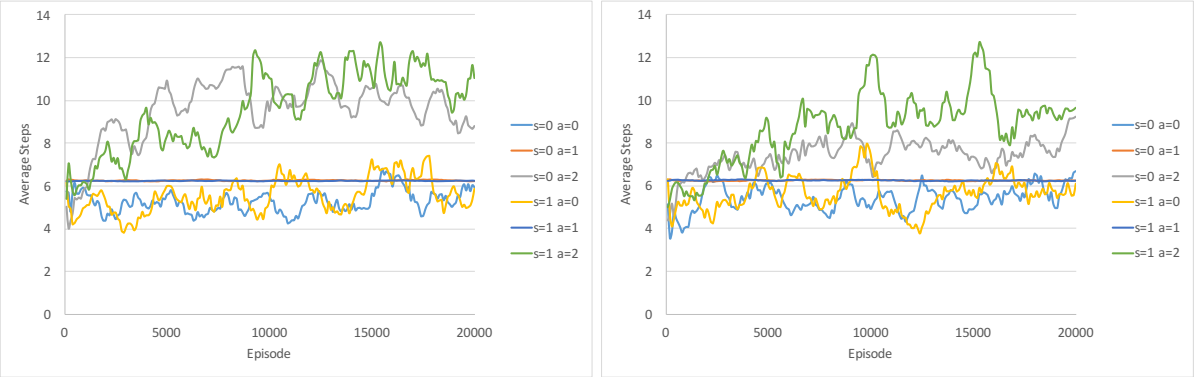


Figure 5.3: Learning curve of discretised Q-Learning (left) and SARSA (right)

action discretisation of the actions always being ± 0.5 performs the best whereas a uniformly discretised action space performs worst. This may be because the highly restricted action space consists of generally better performing actions and less exploration is necessary to find the good actions in the action space.

For the action discretisation where no action (all action variables are zero) is taken the performance is consistent throughout the experiment, reaching just over six steps consistently. This is higher than the performance of action discretisation 0 where the actions are uniformly discretised. This is because action discretisation 0 has a large action space to explore for every state and many of the possible actions have a detrimental effect on performance rather than a positive one. In effect, this experiment is showing that without assistance from poor action choices it takes just over six time steps for the helicopter to fall out of the sky. All these discretisations perform better than no discretisation.

5.3 Tile Coding

Here the experiments move on to the use of tile coding in the helicopter domain. Tile coding (explained in detail in section 2.6.1) is a method of generalisation for use in continuous state spaces. Tile coding is used rather than other generalisation methods, such as neural networks, as it (and variations of it) have been successfully applied to other domains [3, 13, 40, 43, 44] but has not been applied to the helicopter domain. In the helicopter domain not just the state space is continuous but the action space as well [16]. To deal with this, tile coding is applied to both states and actions. There are two ways to implement this. The first is to treat the action variables as additional state variables and combine them together when finding the tiles for a state-action pair. The second is to tile them separately then use the two sets of tiles together as a state-action pair.

Both these tiling methods give the same size state-action tile space. The second option allows

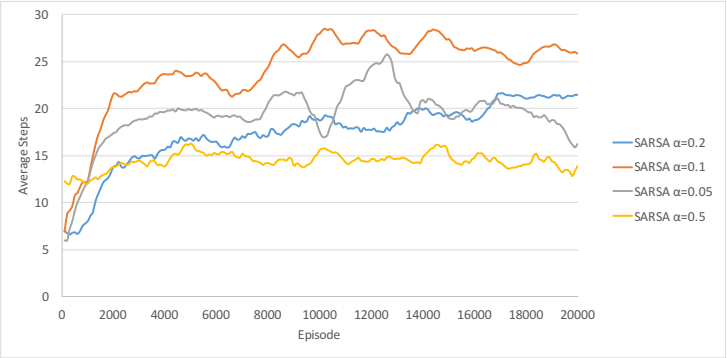


Figure 5.4: Varying learning rate (α) values

the states and actions to be tiled independently and differently. The second method was chosen; both tilings (states and actions) are implemented using the hashed tile coding function by Sutton [85]. The hashing capability of the function means that only visited state-action pairs need be stored. To utilise this, tile weights are stored in a `Map` of `Map`s. The inner map maps action tiles to state-action tile weights; the outer map maps state tiles to an action tile map.

Tile coding causes continuous problems to be converted from using an MDP to a POMDP where, once tiled, it is not possible to tell two close states apart [22]. The same is true when using tile coding on continuous action spaces; many actions may reside within the same action tile; so the action selected may not in fact be the optimal one within the tile area.

5.3.1 Finding Learning Rate

During the following experiments using tile coding, each parameter is searched independently, the first being the learning rate α. This parameter controls how much the immediate reward affects the value of a state-action pair; high α produces large updates, low α produces small updates. In other domains it has been found that α values too close to 0 or 1 perform badly, with the best being around 0.1 and smaller values aiding generalisation [14, 43].

As good values for the other parameters are not known at this stage, arbitrary values are used. These are: exploration rate $\varepsilon = 0.1$, $\#state\_tiles = 10$, $\#state\_tilings = 16$, $\#action\_tiles = 10$ and $\#action\_tilings = 16$. These values were found, during implementation testing, to support learning and perform better than the basic discretisation in the previous section.

Four values for the learning rate were tested $\alpha = \{0.05, 0.1, 0.2, 0.5\}$. The results of these experiments can be seen in figure 5.4 using the SARSA algorithm. The learning rate that gave the highest performance was $\alpha = 0.1$, which is statistically significantly better performing than other parameter values. Increasing the learning rate too high had the effect of slowing the learning. A higher learning rate means that the new information gained has a higher impact on the valuation of a state-action pair; therefore, if the learning rate is too high it can cause the valuations to fluctuate rapidly causing non-optimal actions to be selected. On the other hand, if the learning rate is too low the value updates will be small, meaning it takes more updates to differentiate one state from another, slowing learning.

The experiment was also run using the Q-Learning algorithm and it was found that $\alpha = 0.1$ is also the most effective learning rate for Q-Learning. Figure 5.5 compares Q-Learning with SARSA for $\alpha = 0.1$. It can be seen in the right hand graph that Q-Learning is unstable when tile coding is used in this domain with many very low performing outliers, whereas SARSA has relatively little variability. In the moving averaged graph (left) this manifests as a significantly more variable learning curve. It has been shown that standard on-policy algorithms (SARSA)

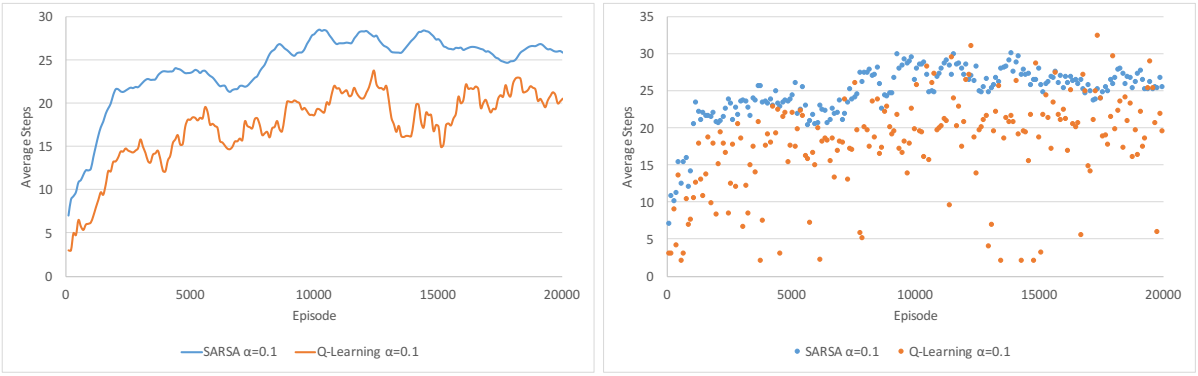


Figure 5.5: Comparison between Q-Learning and SARSA for $\alpha = 0.1$. Moving average (left) and raw data-points (right)

Figure 5.6: Varying the exploration probability ε

will converge when function approximation is used, whereas standard off-policy algorithms may not (Q-Learning) [32, 48]. This could be why Q-Learning has such high variability; the algorithm is diverging then re-learning as the episodes progress. Another explanation is that off-policy algorithms do not perform well in non-stationary environments (when the environment is constantly changing); on-policy algorithms are able to handle the changes better [22]. The random wind in the environment can be considered a non-stationary component of the environment; thus SARSA is able to deal with its effects better. The same behaviour was observed over all experiments in the remaining sections; therefore, only SARSA results are reported for the remainder of the report.

5.3.2 Finding the Exploration Probability

The next parameter to experiment with is the exploration probability ε. This parameter controls how often the agent will take exploratory (random) actions versus taking actions greedily (ones that are known to have a high value). This can be a key factor in well performing reinforcement learning as too little or too much exploration can slow learning. Furthermore, this perimeter must be found empirically as there are no general case heuristics to follow [3, 27].

Following on from the previous experiment, the value of $\alpha = 0.1$ is used as it gave the highest performing agent. All other parameters remain the same. Several exploration probabilities were tested $0 \leq \varepsilon \leq 0.5$ as well as decreasing ε linearly over the $20,000$ episodes of an experiment from 0.1 to 0.

Figure 5.6 shows the learning curves for $\varepsilon = \{0.01, 0.1, 0.5\}$ and decreasing ε. Initially the

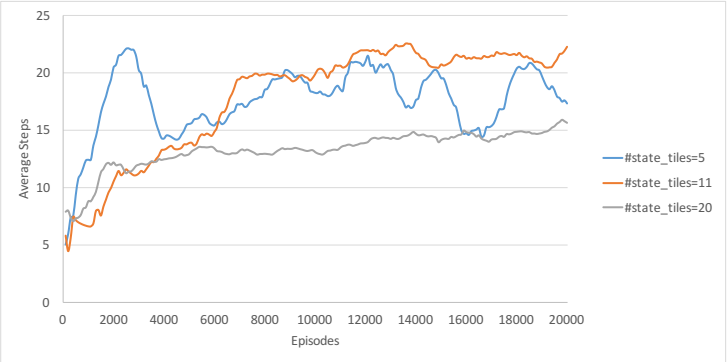


Figure 5.7: Varying the number of state tiles

learning rates of $\varepsilon = 0.1$ and decreasing ε were very similar as would be expected because their ε values would be very similar. Past $1,000$ episodes decreasing epsilon slowed the agent's learning because after this point the agent was not exploring new actions enough. It can also be seen that having exploration rates that are either two high ($\varepsilon = 0.5$) or too low ($\varepsilon = 0.01$) slows learning. This is because the agent does not exploit enough and therefore does not converge (high ε) or not enough exploration is done to find potentially good actions (low ε). $\varepsilon = 0.1$ is statistically significantly better performing than other parameter values.

5.3.3 Finding Tile Coding Parameters

The helicopter domain has an infinite number of states and actions as both the state space and action space are continuous. Tile coding is used to discretise this space, resulting in a finite number of states and actions. Tile coding introduces two additional parameters into reinforcement learning, the number of tiles and the number of tilings. Here this is done for both the state space and the action space. The chosen tile coding parameters can have a large impact on the performance of reinforcement learning with fewer larger tiles and more tilings giving improved generalisation. More generalisation can lead to improved performance initially but can reduce the maximum performance in the longer term [43].

As mentioned earlier, a tile coding algorithm by Sutton [85] is being used for the state space and action space separately. This leads to tiles of uniform shape throughout the spaces. The number of tiles in each dimension of the space is equal but the range of values are all different; so the state tiles are an irregular twelve-dimensional shape. The action space variables all have the same range so tiles are a regular four-dimensional shape.

The parameter variations in the following experiments have large impacts on the memory usage and computation time of the experiments. To put these increases or decreases into perspective, the average memory usage over the previous experiments in sections 5.3.1 and 5.3.2 is approximately 7GB and the computation time approximately twenty minutes per experiment.

5.3.3.1 Finding State Tiles

A series of experiments were carried out with only the number of state tiles being varied. $\alpha = 0.1$ and $\varepsilon = 0.1$ are carried over from the previous series of experiments. Tile counts ranging from 2 to 20 were tried; with $\#state\_tiles = \{5, 11, 20\}$ shown in figure 5.7 as these illustrate particular features of interest. Firstly, $\#state\_tiles = 11$ was found to be the highest performing agent; however, this result not statistically significant when compared to all tested parameter values such as $\#state\_tiles = 10$.

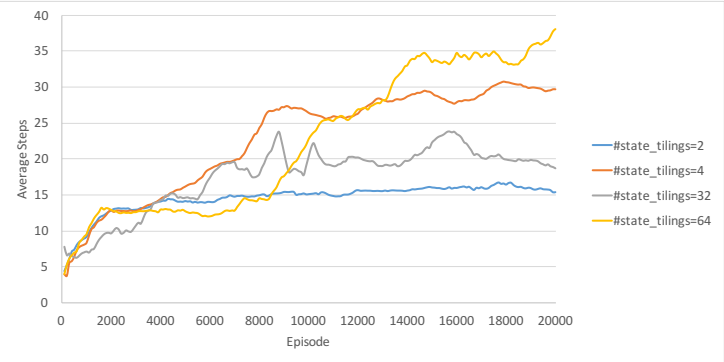


Figure 5.8: Varying the number of state tilings

Increasing the number of tiles had the effect of increasing the learning stability (the smoothness of the learning curve); whereas small numbers of tiles decreased stability. This can be seen by comparing the learning curves of 5 tiles and 20 tiles, where 5 tiles is significantly more variable. This is likely a manifestation of the tile coding property:

> "As the size of a tile increases, the probability that an agent will diverge from the optimal property also increases" [87]

It was also found that lower tile numbers tended to increase performance faster initially, then taper off earlier. This is has also been demonstrated by Sherstov and Stone [43].

5.3.3.2 Finding State Tilings

Using the highest performing number of state tiles ($\#state\_tiles = 11$) experiments were performed to determine the best number of tilings to use over the state variables. The number of tilings controls the number of binary features that will be extracted from the observed state. Increasing the number of tiles improves generalisation across tile boundaries [3, 43].

The number of tilings experimented with were all powers of 2 (1, 2, 4 etc.) up to 64. These tiling numbers are used as, although not necessary, powers of 2 are recommended for use with the tile coding algorithm [85]. Figure 5.8 shows the learning curves for $\#state\_tilings = \{2, 4, 32, 64\}$. It was found that too few and too many tilings generally decreased the maximum number steps achieved in each episode.

Increasing the number of tilings significantly increased the memory usages of the helicopter agent. Increasing the number of tiles also increased the memory usage; however, not to the same degree. The previous experiments on the number of state tiles saw memory usage increase from 1GB when two state tiles are used to 7GB when twenty state tiles are used. These experiments, however, saw variation from 0.5GB when one tiling was used to 80GB when sixty-four tilings were used. One may expect the number of tiles to have a greater effect on memory usage as increasing the number of tiles increases the total number of potential tiles exponentially whereas increasing tilings has a linear effect. The reason is due to the hashing of the tile coding function; even though there are more possible tiles when increasing the number of tiles, only a small proportion more are visited and stored. Whereas increasing the tilings does increase the total number of visited (and stored) tiles.

Additionally, increasing the number of tilings also increased the computation time. With one tiling an experiment lasts approximately five minutes. With sixty-four tilings however, an experiment lasts over two hours.

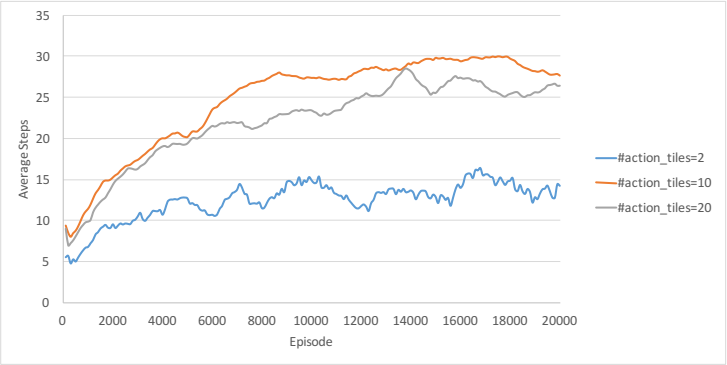


Figure 5.9: Varying the number of action tiles

Using sixty-four state tilings, although the highest performing, resulted in excessive memory usage and very high computation time. Due to this, the second highest performing $\#state\_tilings = 4$ will be used for the remainder of the experiments, which is statistically significantly better performing than the other parameter choices.

5.3.3.3 Finding Action Tiles

Using $\#state\_tiles = 11$ and $\#state\_tilings = 4$ experiments on the number of action tiles were performed. As with finding the number of state tiles, action tiles of 2 to 20 were tested. $\#action\_tiles = 10$ was found to be the highest performing, but not statistically significant. As with finding the state tiles, having too few tiles degraded performance and made the learning less stable. This is likely because the action that is being performed, although having a good value once tiled, is not necessarily the best action to take that is within the highest value tiles. More tiles means smaller tiles; therefore, the action within the tile is more likely to be representative of the tiles' value.

Increasing the number of tiles eventually hits a performance ceiling, at which point performance slowly decreases. This decrease may be due to a gradual over fitting with the smaller sized tiles. Figure 5.9 shows the learning curve for $\#action\_tiles = \{2, 8, 20\}$.

5.3.3.4 Finding Action Tilings

Using the highest performing number of action tiles a number of action tilings were tested. As with the state tilings the action tiles were tested in powers of 2 from 1 to 64. The performance pattern with varying the number of action tilings follows a similar pattern to that of varying the state tilings. Once again, too few or too many tilings resulted in degraded performance from the agent. Additionally, increasing the number of tilings increased the memory usage and computation time to a higher degree than varying the number of tiles. However, the effect of this was substantially less than with the state tilings due to the much smaller size of the action space compared to the state space.

Parameter	Value
α	0.1
ε	0.1
$\#state\_tiles$	11
$\#state\_tilings$	4
$\#action\_tiles$	10
$\#action\_tilings$	8

Table 5.2: Summary of reinforcement learning parameters

Learning curves for $\#action\_tilings = \{1, 8, 32\}$ can be seen in figure 5.10. In these experiments $\#action\_tilings = 8$ was found to be marginally better performing (but not statistically significant) than $\#action\_tilings = 16$ but had the added benefit of reduced run time and

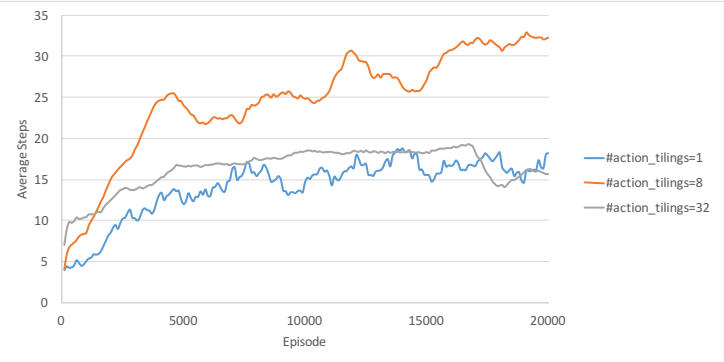


Figure 5.10: Varying the number of action tilings

memory usage due to its lower number of tilings. $\#action\_tilings = 8$ is, however, statistically significantly better than other parameter choices.

5.4 Eligibility Traces

The basic parameters of reinforcement learning with tile coding have now been established (summarised in table 5.2). Following this, experiments regarding enhancements to reinforcement learning can be performed. The first such enhancement is the inclusion of eligibility traces. Eligibility traces blur the line between temporal difference reinforcement learning methods and Monte-Carlo reinforcement learning methods; previously visited state-action pairs are updated at each time step rather than just the current pair. State-action pairs have an additional eligibility parameter which contributes to the state-action pair's value. The use of eligibility traces introduces an additional parameter: the trace decay parameter λ [3].

The use of accumulating eligibility traces with $\lambda = \{0.1, 0.5, 0.9\}$ were tested and compared against not using eligibility traces ($\lambda = 0$). Learning curves generated by these experiments can be seen in figure 5.11. Two observations can be made from these learning curves. Firstly, learning slows as higher values of λ are used. Secondly, higher values of λ causes less stable learning. These effects are possibly due to the highly noisy and random nature of the environment; a particularly noisy or windy state giving a bad reward causes past states to be undervalued by the back-propagation of the reward to these states, even though those states were not actually bad.

In addition to the alternative learning curves produced by the use of eligibility traces, runtime and memory usage must also be considered. Eligibility traces require back-propagation of rewards to all previous states-action pairs. In the helicopter domain (with tile coding) there are many thousands of state action pairs visited; thus, iterating over all state-action pairs to back-propagate the reward increases the agent's runtime significantly. To mitigate this only the last $2,000$ pairs have their eligibility and Q-value updated. $2,000$ past states strikes a balance between updating all visited states and runtime with the most historic pair in the list having virtually zero eligibility by the time it drops off. This gives a run time of approximately three hours (six times longer than without the use of eligibility traces). Memory usage is also increased as every state action pair has an eligibility in addition to a value.

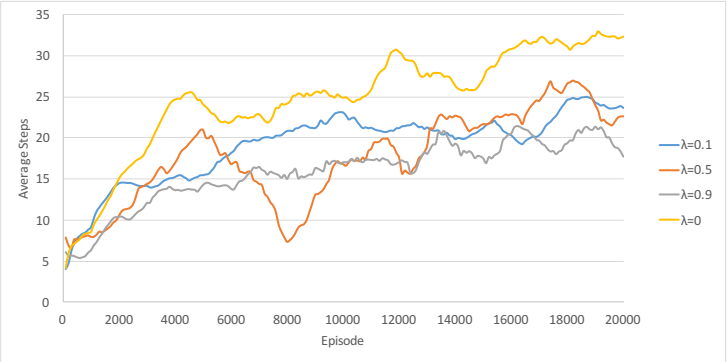


Figure 5.11: The use of Eligibility Traces

5.5 Reward Shaping

Reward Shaping is a technique that allows domain knowledge to be encoded into the reinforcement learning problem. This is done by manipulating the reward received from the environment. The domain knowledge is encoded by a potential function Φ that takes the state and gives a real value indicating (in the same manner as the reward) whether the state is good or bad in terms of the domain knowledge. The domain knowledge is then added to the reward from the environment [60, 62]. Reward shaping can lead to the optimal policy being shifted, decreasing performance. Potential based reward shaping [60] prevents this by incorporating the difference between the last and current state's Φ values rather than the Φ value itself.

The experiment here compares the performance difference between no reward shaping, potential based reward shaping and non-potential based reward shaping. The shaping value function for these experiments is given by equation 5.1. This function gives a positive reward if the agent is heading towards the goal (the origin) or a negative reward if the agent is heading away from the goal; it also scales with the magnitude of the velocity (how fast the agent is moving) and how far the agent is from the goal. Other potential functions are possible; time restrictions only allowed for the designing and evaluation of one.

$$\Phi(s) = -(s_u s_x + s_v s_y + s_w s_z) \tag{5.1}$$

Figure 5.12 shows the learning curves produced by these experiments. Non-potential based reward shaping did in this case reduce the performance of the agent. Potential based reward shaping increased the initial learning speed but not the maximum performance compared to to using reward shaping. This is possibly because the shaped reward helps the agent's Q-values reach their optimal values faster but does not provide enough additional information to make better decisions in the long term.

5.6 Problem Representation

In reinforcement learning problems the state perceived from the environment does not contain all the information possible about the environment [3]. In terms of the helicopter domain of this project, the state contains no information about the wind in the environment. The perceived state may, however, include information which can be derived from other elements in the state. This is true in the helicopter domain; not all positions, rotations and velocities are needed as one can be derived from the others [15, 78].

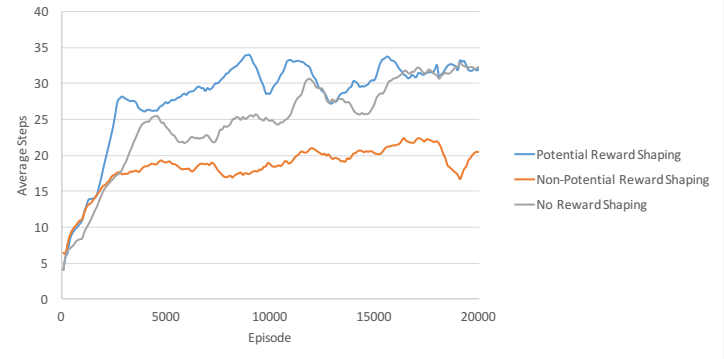


Figure 5.12: The use of Reward Shaping

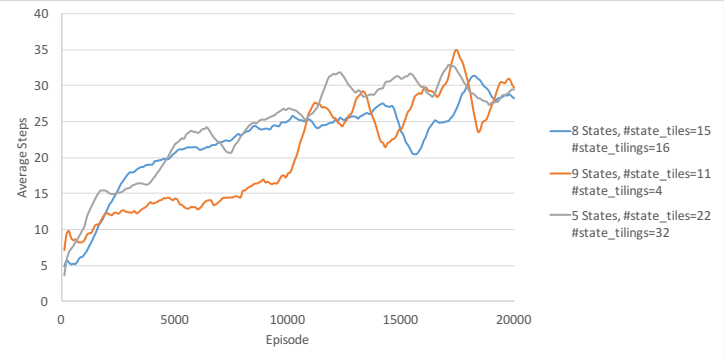


Figure 5.13: The effect of discarding some state variables.

Ng et al. [15] use eight states rather than the twelve available on their work with real helicopters. These states are: roll, pitch, the three velocities and the three angular velocities. Koppejan and Whiteson [78] use nine states in a previous RL Competition. The states they use are: x, y and z positions, roll, pitch and yaw and the three angular velocities.

These two problem representations were experimented with. As the state representations are different the tiling parameters previously established are not necessarily optimal any more. State tiles and tilings were reassessed using the same method as in sections 5.3.3.1 and 5.3.3.2. This resulted in the eight state representation using $\#state\_tiles = 15$ and $\#state\_tilings = 16$. For the nine state representation $\#state\_tiles = 15$ and $\#state\_tilings = 16$ resulted in the highest performance. The resulting learning curves can be seen in figure 5.13.

In addition to the eight and nine state representations a five state representation was also experimented with. This representation is the intersection of the other two, including: roll, pitch and the three spacial velocities. The highest performing tile coding parameters for this representation are $\#state\_tiles = 22$ and $\#state\_tilings = 32$. From the learning curves in figure 5.13 it can be seen that the agent still learns with all three of these reduced state representations. The rate of learning is, however, slightly slowed compared to the twelve state representation, meaning that the learning curve increases at a slower rate. By the end of the experiment the agent does, however, reach a similar number of steps to the original agent. The learning curves also get gradually less stable as the experiment progresses.

Even though reducing the state representation reduces the speed of learning it has other benefits. The primary benefit is the reduction in the size of the state space; there are significantly fewer possible state tiles with each reduction in state representation size due to the tile space size being exponential in the number of state dimensions. This has the effect of significantly

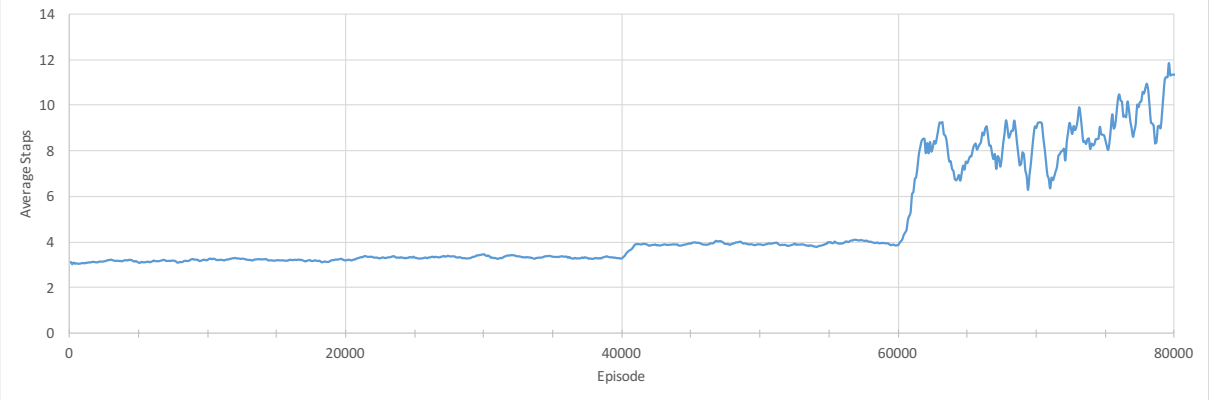


Figure 5.14: The learning curve of Hierarchical Reinforcement Learning

reducing the memory requirements of the agent and decreasing the execution time of the experiment. This decreased experiment time means that longer experiments can be run, allowing the agent to learn for more time steps and potentially achieving the same level of performance as with the original representation.

5.7 Hierarchical Reinforcement Learning

This experiment is the use of hierarchical reinforcement learning in the helicopter domain. Hierarchical reinforcement learning breaks a reinforcement learning problem into sub-problems which can be learnt independently. The sub-problems are smaller, simpler problems. They often have a reduced state space and can be easier problems for reinforcement learning to solve. They are then tied together to give a solution to the reinforcement learning task as a whole [22].

A particular challenge in hierarchical reinforcement learning is determining the method by which the problem is broken down into sub-problems. The breakdown for the helicopter domain used here is that each of the four action variables are learnt independently. In this experiment independent learning of the four action variables is done by performing 20,000 learning episodes where only one action variable is learnt at a time. The other action variables are set by the hard-coded agent policy.

This method gives a total of 80,000 learning episodes; every 20,000 episodes the variable being learnt changes. For the evaluation phases (that produce points on the learning curve) the individual action variables giving the highest value are chosen then combined to give a four variable action for the helicopter. If there is no best action a random one is chosen. This results in a seemingly flat learning curve for the first three action variables as there is always an action that has not been learnt at all yet. There is a small increase in performance for the third variable as three of four variables have been learnt. For the fourth, and final, action variable there is a significant increase in performance as all parts of the action have been learnt. Figure 5.14 shows this learning curve.

The agent does not perform as well as any of the previous agents experimented with. This is because the areas of the state space visited when learning each action variable independently are largely separate. When evaluating the agent, the area of the state space visited is again largely separate from the areas visited when learning each of the action variables. This happens because the actions chosen by the greedy policy are not (often) the same as those chosen by the hard-coded policy; resulting in applying actions that take the agent to a different, less well explored, area of the state space. This process could be performed iteratively, using the learnt

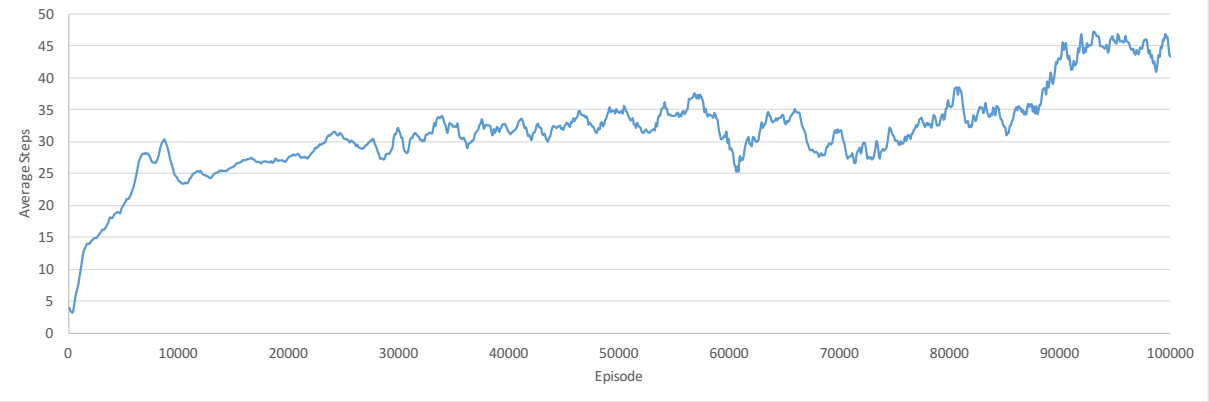


Figure 5.15: Learning curve of tile coded agent with an experiment duration of 100,000 episodes

values after the first iteration, which may result in continued learning up to and beyond that seen in the previous experiments.

5.8 Extended Experiment

A final experiment is the use of the highest performing parameters (shown in table 5.2) in a longer experiment to ascertain whether learning continues past 20,000 episodes. Asbah et al. [80] and Koppejan and Whiteson [78] use experiment durations longer than that used in this project: 200,000 episodes and 100,000 episodes respectively. This experiment tests the tile coded agent on 100,000 learning episodes, five times that of the previous experiments. It can be seen in figure 5.15 that learning does indeed continue past 20,000 episodes, reaching a maximum average performance of 46 steps. However, the stability of the performance appears to decrease over time. The performance exceeds that found by Asbah et al. [80] (38 steps at 200,000 episodes) in their agent with no added controllability. However, with controllability their agent performed better (80 steps at 100,000 episodes).

6 Conclusions

In this chapter the work presented in this project is summarised. Firstly, the agents developed here are compared against other work on autonomous helicopter control. Limitations of this work are outlined. Conclusions are drawn and potential future work is proposed.

6.1 Comparisons with other Helicopter Controllers

In this section the helicopter agents presented in this report is compared to previous work in autonomous helicopter control. The first comparison is against previous entries in the reinforcement learning competition helicopter domain. The tile coded SARSA agent developed in sections 5.3.3 compares very closely to the controllability agent developed for the RL Competition by Asbah et al. [80] when the controllability parameter is zero; this is equivalent to not using controllability. Controllability is similar to reward shaping except the shaping is done when choosing an action rather than when updating a state-action value; having the effect that it directs the exploration. A method of function approximation is used to handle the continuous states; however, the precise method and any necessary parameters are not given although tile coding is a likely choice. It is therefore expected that their SARSA agent and the SARSA agent presented here should perform similarly, and they do. When the controllability parameter is increased the learning speed increases (similarly to the use of reward shaping in section 5.5); however, final performance also increases with the use of controllability.

The other method presented by Asbah et al. [80] is the use of kernel-based reinforcement learning. This method performs significantly better than controllability and the agents presented in this project. By the end of the experiments the agent manages to last the full $6,000$ steps in an episode. This method builds an approximate finite MDP to model the continuous environment while learning; this MDP is used to better choose actions.

The use of some form of approximate model has major performance improvements over the completely model-free agents of this report. Forms of neural networks are often used to help define the agent's policy. Koppejan and Whiteson [78] use both fixed topology and evolutionary neural networks to define the agent's policy. Both cases use the rewards from the environment to guide the neural network weight assignment and, in the latter case, the neural network structure. All neural network methods presented perform very well, achieving rewards of the magnitude -10^4 to -10^2 rather than -10^7 for the agents in this report.

Both Asbah et al. [80] and Koppejan and Whiteson [78] use experiment durations much longer than that used in this project: $200,000$ episodes and $100,000$ episodes respectively. With an extended experiment duration of $100,000$ episodes the tile coded agent of this project outperforms the non-controllability agent, but is outperformed by the controllability agents, presented by Asbah et al. [80].

Neural networks with policy search methods have been used with reinforcement learning on real helicopters [15, 72]. These methods are capable of performing complex manoeuvres as well as just hovering. The tile coded agent of this report would likely perform worse if used in a real world environment. This is due to the added sensor and control delays and increased noise.

Several other machine learning methods have been used to learn helicopter controls other

than reinforcement learning; primarily involving the use of neural networks. Buskey et al. [88] use fuzzy associative memories that can be trained by a teacher in simulations to act as velocity and altitude controllers. Enns and Si [89] use neural networks combined with dynamic programming to learn and perform complex helicopter manoeuvres. It is also possible to create helicopter controller manually. This could be a basic PID controller or other techniques such as fuzzy gain scheduling [90].

6.2 Conclusion

This project explored the use of reinforcement learning on the domain of simulated helicopters. Unlike fixed wing aircraft, helicopter control is unstable; if corrective actions are not taken the helicopter will very quickly become unstable and fall from the sky. Helicopter control is a challenging reinforcement (and machine) learning problem due to a multitude of factors. Firstly, the state and action spaces are both continuous, so there are an infinite number of possible state-action combinations. Furthermore, there are many state and action dimensions, compounding the problem. Secondly, the environment is noisy and stochastic; noisy observations and random wind affect how actions are applied and states are observed. Finally, the entire environment is not observed and non-stationary; the wind effects are not known to the agent.

A series of experiments were undertaken establishing reinforcement learning parameters and the use of different reinforcement learning techniques. None of the reinforcement learning techniques tested reached performance comparable to that of a hard-coded controller. The controller remained airborne for the entirety of every episode (6,000 time steps), where as the maximum number of steps achieved in the experiments was 46.

If nothing is done about the continuous nature of the state and action spaces it is equivalent to doing random actions at every step, with no learning; this method gives an average of 4 steps reached per episode. A first step towards dealing with the continuous spaces is a fixed discretisation. This did enable some learning with a maximum performance of approximately 12 steps. It was also found that doing nothing (all action variables set to zero) at every time step gave better performance (consistently 6 time steps) than doing some actions showing that some actions quicken the helicopter's demise.

Tile coding has been used as a method of state and action space discretisation to allow for generalisation. Reinforcement learning with tile coding parameters were found in an iterative manner, assessing one parameter at a time. The highest performing parameters are summarised in table 5.2. The use of tile coding significantly improved the agent's ability to learn, increasing the number of time steps achieved to 40 with the best parameter choice found. Too few or too many tiles or tilings for both state and actions caused degraded learning. Q-Learning was found to be much less stable than SARSA and too large tiles caused increased divergence when learning.

Further reinforcement learning techniques were also tested. The use of eligibility traces was found to slow learning and decrease learning stability as the trace decay parameter was increased. Additionally, computation time and memory use increased significantly. Potential based reward shaping sped up initial learning but did not allow for an overall increase in performance. Altering the state representation mildly decreased performance but had significant computation speed and memory usage reductions. Finally, hierarchical reinforcement learning performed particularly badly as the state space areas explored independently were highly separate.

Although it has been shown that an agent can learn some helicopter control using reinforcement learning with tile coding, the performance is not satisfactory to use in real world situations. Each simulated time step is equivalent to a tenth of a second; so, the maximum

flight time achieved in these experiments is forty-six seconds (46 time steps). This is clearly not enough for real world use where there are additional challenges such as latency and the added cost of not wishing the helicopter to crash.

6.3 Future Work

There are several avenues of future work to expand on this project. In terms of further research; automated methods of obtaining optimal reinforcement learning and tile coding parameters could potentially lead to a higher performing agent as these parameters are interdependent [43]. Alternative tile patterns and different tile shapes in different dimensions may also give more appropriate generalisation for the problem. Adaptive [44] or evolutionary tile coding [45] could be used to find the optimal generalisation for the problem. Experiments using these methods would establish the use of the methods in real world problems and potentially increase the helicopter agent's ability to learn and control. Additional reward shaping functions and alternative hierarchical problem decompositions could also be tested. Furthermore, all these methods could be tested in combination.

This project required implementation of tile coding for continuous action spaces as neither YORLL nor BURLAP supported this. A future project may be to add this functionality to either or both these libraries as many real world problems have both continuous state and action spaces.

Bibliography

[1] T. M. Mitchell, "Reinforcement Learning," in *Machine Learning*. McGraw-Hill, 1997, pp. 367–387.

[2] D. Kudenko, "Multi-Agent Learning," p. 21, 2015. [Online]. Available: http://www-module.cs.york.ac.uk/maig/lectures/maig-learning.pdf

[3] R. S. Sutton and A. G. Barto, *Reinforcement Learning : An Introduction*. MIT Press, 1998. [Online]. Available: https://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html

[4] B. Hengst, "Hierarchical Reinforcement Learning," in *Encyclopedia of Machine Learning SE - 363*, C. Sammut and G. Webb, Eds. Springer US, 2010, pp. 495–502.

[5] Thai Technics, "Flight Direction Control," 2001. [Online]. Available: http://www.thaitechnics.com/helicopter/heli_control_3.html

[6] Flight Learnings, "Helicopter Forward Flight," 2016. [Online]. Available: http://www.danubewings.com/helicopter-forward-flight/

[7] Technological Design Engineering, "Forces acting on a helicopter," 2011. [Online]. Available: http://cesarvandevelde-constructieleer.blogspot.co.uk/2011/01/forces-acting-on-helicopter.html

[8] S. Russell and P. Norvig, "Learning," in *Artificial Intelligence: A Modern Apprach*, 3rd ed. Pearson, 2009, ch. 18-21, pp. 693–858.

[9] T. M. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

[10] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, pp. 237–285, 1996.

[11] G. J. Tesauro, "Temporal difference learning and TD-Gammon," *Communications of ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[12] C. Thiery and B. Scherrer, "Building Controllers for Tetris," *International Computer Games Association Journal*, vol. 32, pp. 3–11, 2009.

[13] S. Whiteson and P. Stone, "Evolutionary Function Approximation for Reinforcement Learning," *Journal of Machine Learning Research*, vol. 7, no. AI05-320, pp. 877–917, 2006.

[14] P. Stone, R. S. Sutton, and G. Kuhlmann, "Reinforcement Learning for RoboCup-Soccer Keepaway," *Adaptive Behavior*, vol. 13, no. 3, pp. 165–188, 2005.

[15] A. Y. Ng, H. J. Kim, M. I. Jordan, and S. Sastry, "Autonomous helicopter flight via Reinforcement Learning," *Advances in Neural Information Processing Systems 16*, vol. 16, pp. 363–372, 2004.

[16] P. Abbeel, A. Coates, and A. Ng. (2014) Helicopter - RL Competition 2014. [Online]. Available: https://sites.google.com/site/rlcompetition2014/domains/helicopter

[17] S. Whiteson, B. Tanner, and a. White. (2010) The reinforcement learning competitions. [Online]. Available: http://staff.science.uva.nl/~whiteson/pubs/whitesonaim10.pdf

[18] J. Seddon, *Basic helicopter aerodynamics*. BSP Professional Books, 1990.

[19] R. L. Finn and D. Wright, "Unmanned aircraft systems: Surveillance, ethics and privacy in civil applications," *Computer Law and Security Review*, vol. 28, no. 2, pp. 184–194, 2012.

[20] N. Sharkey, "Saying 'No!' to Lethal Autonomous Targeting," *Journal of Military Ethics*, vol. 9, no. 4, pp. 369–383, 2010.

[21] S. Russell and P. Norvig, "Intelligent Agents," in *Artificial Intelligence: A Modern Apprach*, 3rd ed. Pearson, 2009, ch. 2, pp. 34–61.

[22] M. Wiering and M. van Otterlo, *Reinforcement Learning: State of the Art*. Springer Verlag, jan 2012.

[23] S. D. Whitehead and L.-J. Lin, "Reinforcement learning of non-Markov decision processes," *Artificial Intelligence*, vol. 73, no. 1-2, pp. 271–306, 1995.

[24] R. Bellman, "A Markovian decision process," pp. 679–684, 1957.

[25] R. A. Howard, *Dynamic Programming and Markov Processes*. New York: MIT Press, 1960.

[26] R. E. Bellman, *Dynamic programming*, R. Corporation, Ed. Princeton: Princeton University Press, 1957.

[27] R. S. Sutton, D. Precup, and S. Singh, "Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning," *Artificial Intelligence*, vol. 112, no. 1, pp. 181–211, 1999.

[28] M. Wiering, "Explorations in Efficient Reinforcement Learning," Ph.D. Thesis, Universiteit van Amsterdam, 1999.

[29] R. S. Sutton, "Temporal Credit Assignment in Reinforcement Learning," Ph.D Thesis, University of Massachusetts, 1984.

[30] S. P. Singh and R. S. Sutton, "Reinforcement Learning with replacing elibibility traces," *Machine Learning*, vol. 22, pp. 123–158, 1996.

[31] B. Widrow, N. K. Gupta, and S. Maitra, "Punish/Reward: Learning with a Critic in Adaptive Threshold Systems," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 3, no. 5, 1973.

[32] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988.

[33] C. J. C. H. Watkins, "Learning from delayed rewards," Ph.D Thesis, University of Cambridge, 1989.

[34] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[35] G. a. Rummery and M. Niranjan, "On-line Q-Learning Using Connectionist Systems CUED / F-INFENG / TR 166 Abstract," Cambridge University, Tech. Rep. September, 1994.

[36] S. Singh, T. Jaakkola, M. L. Littman, and C. S. Ari, "Convergence Results for Single-Step On-Policy Reinforcement-Learning Algorithms," *Machine Learning*, vol. 39, no. 3, pp. 287–308, 2000.

[37] V. Konda and J. Tsitsiklis, "Actor-Critic Algorithms," *NIPS*, vol. 13, pp. 1008–1014, 1999.

[38] I. H. Witten, "An adaptive optimal controller for discrete-time Markov environments," *Information and Control*, vol. 34, no. 4, pp. 286–295, 1977.

[39] J. S. Albus, "A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC)," *Journal of Dynamic Systems, Measurement, and Control*, vol. 97, no. 3, pp. 220–227, sep 1975.

[40] R. Sutton, "Generalization in reinforcement learning: Successful examples using sparse coarse coding," *Advances in neural information processing systems*, no. October, pp. 1038–1044, 1996.

[41] R. Bellman and R. E. Bellman, *Adaptive Control Processes: A Guided Tour*, ser. 'Rand Corporation. Research studies. Princeton University Press, 1961.

[42] J. Shewchuk and T. Dean, "Towards Learning Time-Varying Functions With High Input Dimensionality," in *Proceedings of the Fifth IEEE International Symposium on Intelligent Control*, 1990, pp. 383–388.

[43] A. Sherstov and P. Stone, "Function approximation via tile coding: Automating parameter choice," *Abstraction, Reformulation and Approximation*, pp. 4–6, 2005.

[44] S. Whiteson, M. Taylor, and P. Stone, "Adaptive tile coding for value function approximation," University of Texas at Austin, Tech. Rep., 2007.

[45] S. Lin, R. Wright, and B. Rd, "Evolutionary Tile Coding : An Automated State Abstraction Algorithm for Reinforcement Learning," *Proceedings of the IEEE*, pp. 42–47, 2007.

[46] G. E. Hinton, "Distributed representations," Carnegie Mellon University, Tech. Rep., 1984.

[47] P. Kanerva, *Sparse distributed memory*. MIT press, 1988.

[48] R. S. Sutton and C. Szepesv, "A Convergent O (n) Temporal-difference Algorithm for Off-policy Learning with Linear Function Approximation," *Computing*, pp. 1–8, 1992.

[49] H. R. Maei and R. S. Sutton, "GQ(λ): A general gradient algorithm for temporal-difference prediction learning with eligibility traces," *Proceedings of the 3d Conference on Artificial General Intelligence AGI10*, pp. 1–6, 2010.

[50] J. C. Santamaria, R. S. Sutton, and A. Ram, "Experiments with Reinforcement Learning in Problems with Continuous State and Action Spaces," *Adaptive Behavior*, vol. 6, no. 2, pp. 163–217, 1997.

[51] H. Van Hasselt and M. a. Wiering, "Reinforcement learning in continuous action spaces," *Proceedings of the 2007 IEEE Symposium on Approximate Dynamic Programming and Reinforcement Learning, ADPRL 2007*, no. Adprl, pp. 272–279, 2007.

[52] C. Gaskett, D. Wettergreen, and A. Zelinsky, "Q -Learning in Continuous State and Action Spaces," *Advanced Topics in Artificial Intelligence*, vol. 1747, pp. 417–428, 1999.

[53] P. J. Werbos, "Approximate dynamic programming for real-time control and neural modeling," *Handbook of intelligent control: Neural, fuzzy, and adaptive approaches*, vol. 15, pp. 493–525, 1992.

[54] G. Rummery, "Problem solving with reinforcement learning," Ph.D. dissertation, Cambridge University, 1995.

[55] C. F. Touzet, "Neural reinforcement learning for behaviour synthesis," *Robotics and Autonomous Systems*, vol. 22, no. 3-4, pp. 251–281, 1997.

[56] J. M. Santos, "Contribution to the study and the design of reinforcement functions," Ph.D. dissertation, Universidad de Buenos Aires, Universite d'Aix-Marseille III, 1999.

[57] H.-M. Gross, V. Stephan, and M. Krabbes, "A Neural Field Approach to Topological Reinforcement Learning in Continuous Action Spaces," *Proc. 1998 IEEE World Congress on Computational Intelligence, WCCI'98 and International Joint Conference on Neural Networks, IJCNN'98*, vol. 3, pp. 1992–1997, 1998.

[58] J. Peng and R. J. Williams, "Incremental Multi-Step Q-Learning," *Machine Learning*, vol. 22, no. 1-3, pp. 283–290, 1996.

[59] S. P. Singh and R. S. Sutton, "Reinforcement Learning with replacing elibibility traces," *Machine Learning*, vol. 22, pp. 123–158, 1996.

[60] A. Y. Ng, D. Harada, and S. Russell, "Policy invariance under reward transformation: Theory and application to reward shaping," *ICML*, vol. 99, pp. 278–287, 1999.

[61] M. J. Matarić, "Reward functions for accelerated learning," *Proceed- ings of the Eleventh International Conference on Machine Learning. Morgan Kaufmann*, pp. 181–189, 1994.

[62] J. Randløv and P. Alstrøm, "Learning to Drive a Bicycle using Reinforcement Learning and Shaping," *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 463–471, 1998.

[63] R. Parr and S. Russell, "Reinforcement learning with hierarchies of machines," *Advances in neural information processing systems*, pp. 1043–1049, 1998.

[64] A. G. Barto and S. Mahadevan, "Recent Advances in Hierarchical Reinforcement Learning," *Discrete Event Dynamic Systems: Theory and Applications*, pp. 41–77, 2003.

[65] M. L. Puterman, *Markov decision processes : discrete stochastic dynamic programming*. New York: Wiley, 1994.

[66] T. G. Dietterich, "The MAXQ Method for Hierarchical Reinforcement Learning," *Proc. of the fifteenth international conference on machine learning*, no. c, pp. 118–126, 1998.

[67] J. G. Leishman, *Principles of Helicopter Aerodynamics*, 2nd ed. Cambridge University Press, 2006.

[68] Federal Aviation Administration, *Helicopter Flying Handbook*. U.S. Department of Transportation, 2012.

[69] B. Tanner and A. White, "RL-Glue: Language-Independent Software for Reinforcement-Learning Experiments," *Journal of Machine Learning Research*, pp. 2133–2136, 2009. [Online]. Available: http://glue.rl-community.org/wiki/Main_Page

[70] J. Bagnell and J. Schneider, "Autonomous helicopter control using reinforcement learning policy search methods," *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 2, pp. 1615—-1620, 2001.

[71] P. Abbeel, V. Ganapathi, and A. Y. Ng, "Learning Vehicular Dynamics, with Application to Modeling Helicopters," *Advances in Neural Information Processing Systems (NIPS)*, pp. 1–8, 2005.

[72] A. Y. Ng, A. Coates, M. Diel, V. Ganapathi, J. Schulte, B. Tse, E. Berger, and E. Liang, "Autonomous inverted helicopter flight via reinforcement earning," *Springer Tracts in Advanced Robotics*, vol. 21, pp. 363–372, 2006.

[73] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, "An application of reinforcement learning to aerobatic helicopter flight," *Education*, vol. 19, p. 1, 2007.

[74] A. Coates, P. Abbeel, and A. Y. Ng, "Apprenticeship learning for helicopter control," *Communications of the ACM*, vol. 52, no. 7, p. 97, 2009.

[75] P. Abbeel, A. Coates, T. Hunter, and A. Y. Ng, "Autonomous Autorotation of an RC Helicopter," *Springer Tracts in Advanced Robotics*, vol. 54, pp. 385–394, 2009.

[76] A. Y. Ng and M. Jordan, "PEGASUS: A Policy Search Method for Large MDPs and POMDPs," *Uai*, vol. 94720, pp. 406–415, 2000.

[77] P. Abbeel and A. Y. Ng, "Apprenticeship learning via inverse reinforcement learning," *Proceedings of the 21st International Conference on Machine Learning (ICML)*, pp. 1–8, 2004.

[78] R. Koppejan and S. Whiteson, "Neuroevolutionary reinforcement learning for generalized control of simulated helicopters." *Evolutionary intelligence*, vol. 4, no. 4, pp. 219–241, 2011.

[79] N. T. Siebel and G. Sommer, "Evolutionary Reinforcement Learning of Artificial Neural Networks," *International Journal of Hybrid Intelligent Systems*, vol. 4, no. 3, pp. 171–183, 2007.

[80] A. Asbah, C. Gehring, J. Pineau, and D. Precup, "Reinforcement Learning Competition : Helicopter Hovering with Controllability and Kernel-Based Stochastic Factorization," *Icml 2013*, 2013.

[81] C. Gehring and D. Precup, "Smart Exploration in Reinforcement Learning using Absolute Temporal Difference Errors," *Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 1037–1044, 2013.

[82] D. Ormoneit and S. Sen, "Kernel-Based Reinforcement Learning," *Machine Learning*, vol. 49, no. 2-3, pp. 161–178, 2002.

[83] P. Scopes, D. Kudenko, D. Kentse, F. Alias, K. Efthymiadis, and S. Devlin. (2013) York Reinforcement Learning Library. [Online]. Available: https://www.cs.york.ac.uk/rl/software.php

[84] J. MacGlashan. (2015) Brown-UMBC Reinforcement Learning and Planning (BURLAP). [Online]. Available: http://burlap.cs.brown.edu/

[85] R. S. Sutton. Tile Coding Software. [Online]. Available: http://webdocs.cs.ualberta.ca/~sutton/tiles2.html

[86] Chronicle Software. (2016) Koloboke Collections. [Online]. Available: http://chronicle.software/products/koloboke-collections/

[87] P. Scopes and D. Kudenko, "Theoretical Properties and Heuristics for Tile Coding," *ALA Workshop, AAMAS*, vol. 2014, 2014.

[88] G. Buskey, J. Roberts, and G. Wyeth, "Online Learning of Autonomous Helicopter Control," *Australasian Conference on Robotics and Automation*, no. November, pp. 27–29, 2002.

[89] R. Enns and J. Si, "Helicopter trimming and tracking control using direct neural dynamic programming," *IEEE Transactions on Neural Networks*, vol. 14, no. 4, pp. 929–939, 2003.

[90] B. Kadmiry, P. Bergsten, and D. Driankov, "Autonomous Helicopter Control Using Fuzzy Gain Scheduling," *IEEE International Conference on Robotics and Automation*, vol. 1, no. 1, pp. 2980–2985, 2001.

A Definitions

A.1 Keywords

Keyword	Definition
Supervised Learning	A category of machine learning techniques where sample inputs and outputs are presented to the machine for training
Unsupervised Learning	A category of machine learning techniques where underlying patterns in data are extracted without any training examples.
Reinforcement Learning	The topic of this report. This learning method sits somewhere between supervised and unsupervised learning.
Agent	The learning entity.
Environment	The world the entity is in.
State	The agent's perception of the environment.
Reward	Feedback from the environment of how well the agent is performing.
Value Function (V)	An indication of 'how good' a state is for an agent to be in.
Action Function (Q)	An indication of 'how good' an action is to take at a particular state.
Policy	A function defining which action to take when in which state.
Optimal Policy	The policy which generates the most reward over time.
Generalised Policy Iteration	The iterative process of evaluating then improving a policy.
Model-Based	A learning method which uses a complete model of the environment during the learning process.
Model-Free	A learning method which uses no model of the environment during the learning process, it builds its own.
Markov Decision Process (MDP)	The underlying structure of the environment. The current state only depends on the previous state and action used. No further historic information is needed.
Bootstrapping	Value estimates are based off other estimates.
Dynamic Programming	A Model-Based learning method.
Indirect / Model Based RL	Estimate a model by exploration, then apply DP algorithms on this.
Direct / Model Free RL	Estimating values for action without estimating a model.
Online RL	Learns by interacting with the environment and updating value estimates at each experience.
Episodic Task	The task has a defined end.
Off-Policy	Finds π^* while following the exploration policy π

On-Policy	The policy being followed is also the optimal policy being learnt.
Non-Stationary Environment	The environment is constantly changing over time.
Q-Learning	An off-policy TD learning algorithm.
SARSA	State-Action-Reward-State-Action: An on-policy TD learning algorithm.
Actor-Critic	An on-policy TD learning method.
Curse of Dimensionality	When using a state space with high dimensionally, the space becomes sparsely populated with data points.
Tile Coding	Dividing up a continuous state space to allow for generalisation.
Partially Observable MDP (POMDP)	When tile coding is used, the exact state within a tile can't be known. Therefore it is partially observable.
Adaptive Tile Coding	Automatically determine the number of tiles needed.
Evolutionary Tile Coding	Automatically determine the number of tiles needed using an evolutionary algorithm.
Eligibility Trace	A method of combining Monte Carlo and Temporal Difference learning methods.
Reward Shaping	A method of including domain knowledge to guide learning.
Hierarchical Reinforcement Learning	The process of breaking a problem down into a hierarchy of learning tasks.
Cyclic Control	Controls main rotor pitch. Causes forward-backward and side-to-side tipping and movement.
Collective Control	Controls all main rotor blade angles. Causes increased or decreased lift affecting movement speed and altitude.
Anti-Torque Control	Controls tail rotor. Prevents helicopter spinning and controls heading.
RL-Glue	A reinforcement learning framework. [69]
YORLL	The YOrk Reinforcement Learning Library [83]
BURLAP	The Brown-UMBC Reinforcement Learning and Planning library [84]
Koloboke Collections	A high performance, low footprint `Map` and `Set` library by Chronicle Software [86]

A.2 Symbols

Symbol	Definition
α	The learning rate of temporal difference learning algorithms. Determines how influential new information is in the state valuation.
γ	The discount factor. Lower discount factors mean earlier learning is more important than that learned in later episodes. Can be 1 for episodic tasks.
ε	The probability of exploration vs exploitation
ϕ	A discretisation (or tile coding) function
θ	A weight associated with a tile.

R_t	The return at a point in time. This is the (discounted) sum of all future rewards.
π	A policy.
π^*	The optimal policy.
t	A point in time.
s_t	The state the agent is in at time t.
$V^\pi(s)$	The value of a state when following the policy π.
$Q^\pi(s,a)$	The value of a state-action pair when following the policy π.
$V^*(s)$ or $V^{\pi^*}(s)$	The value of a state when following the optimal policy.
$Q^*(s,a)$ or $Q^{\pi^*}(s,a)$	The value of a state-action pair when following the optimal policy.
δ_t	The temporal difference (TD) error at time t.
$e_t(s,a)$	The eligibility of a state-action pair at time t.
λ	The trace decay parameter used in eligibility traces. $\lambda = 1$ is equivalent to Monte-Carlo learning, $\lambda = 0$ is equivalent to temporal difference learning.
Φ	A function used in reward shaping that maps a state to a real number indicating how good the state its.
$F(s_t, a_t, s_{t+1})$	A potentially shaped reward function.
$R(s_t, a_t, s_{t+1})$	The reward function from the environment.
$R'(s_t, a_t, s_{t+1})$	The reward function after shaping.