Single- and Multiobjective Reinforcement Learning in Dynamic Adversarial Games

BUDI KURNIAWAN

School of Engineering, Information Technology and Physical Sciences
FEDERATION UNIVERSITY AUSTRALIA

A dissertation submitted to Federation University Australia in accordance with the requirements of the degree of DOCTOR OF PHILOSOPHY in the School of Engineering, Information Technology and Physical Sciences.

MAY 2022

Abstract

This thesis uses reinforcement learning (RL) to address dynamic adversarial games in the context of air combat manoeuvring simulation. A sequential decision problem commonly encountered in the field of operations research, air combat manoeuvring simulation conventionally relied on agent programming methods that required significant domain knowledge to be manually encoded into the simulation environment. These methods are appropriate for determining the effectiveness of existing tactics in different simulated scenarios. However, in order to maximise the advantages provided by new technologies (such as autonomous aircraft), new tactics will need to be discovered. A proven technique for solving sequential decision problems, RL has the potential to discover these new tactics.

This thesis explores four RL approaches—tabular, deep, discrete-to-deep and multiobjective—as mechanisms for discovering new behaviours in simulations of air combat manoeuvring. It implements and tests several methods for each approach and compares those methods in terms of the learning time, baseline and comparative performances, and implementation complexity. In addition to evaluating the utility of existing approaches to the specific task of air combat manoeuvring, this thesis proposes and investigates two novel methods, discrete-to-deep supervised policy learning (D2D-SPL) and discrete-to-deep supervised Q-value learning (D2D-SQL), which can be applied more generally. D2D-SPL and D2D-SQL offer the generalisability of deep RL at a cost closer to the tabular approach.

Acknowledgements

I would like to acknowledge the guidance and advice provided to me by my supervisors:

- Dr. Peter Vamplew, Associate Professor at School of Engineering, Information Technology and Physical Sciences, Federation University Australia
- Dr. Michael Papasimeon from the Defence Science and Technology Group in the Australian Department of Defence.
- Dr. Richard Dazeley, Associate Professor at School of Information Technology, Deakin University
- Dr. Cameron Foale from School of Engineering, Information Technology and Physical Sciences, Federation University Australia

I would also like to thank Dr. Savin Chand for his advice on statistical analysis and I acknowledge that the research conducted as part of this thesis was supported by the Defence Science and Technology Group, Australia; the Defence Science Institute, Australia; and an Australian Government Research Training Program Fee-offset scholarship through Federation University Australia. The Centre for Multimedia Computing, Communications, and Artificial Intelligence Research (MCCAIR) at Federation University Australia contributed some of the computing resources.

Finally, I would like to thank my wife Wellia Lioeng for her love and support throughout years of my study.

DECLARATION

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

SIGNED: .... DATE: ....

List of Publications

Some elements of this thesis have appeared in these refereed publications.

- Kurniawan, B., Vamplew, P., Papasimeon, M., Dazeley, R., and Foale, C. (2019).
  An empirical study of reward structures for actor-critic reinforcement learning in air combat manoeuvring simulation.
  32nd Australasian Joint Conference on Artificial Intelligence.

- Kurniawan, B., Vamplew, P., Papasimeon, M., Dazeley, R., and Foale, C. (2020).
  Discrete-to-deep supervised policy learning: An effective training method for neural reinforcement learning.
  Adaptive and Learning Agents workshop at the 19th International Conference on Autonomous Agents and Multi-Agent Systems.

- Kurniawan, B., Vamplew, P., Papasimeon, M., Dazeley, R., and Foale, C. (2022).
  Discrete-to-deep reinforcement learning methods.
  Neural Computing and Applications, 34, 1713–1733 (2022).

Some of the experiments conducted for these papers are updated for this thesis. In addition, an open source framework for agent learning and testing called Spy RL was developed for this thesis and can be downloaded from `https://github.com/budi-kurniawan/spyrl`.

GLOSSARY

The following are important terms used in this thesis.

1. A **Markov decision process (MDP)** is a discrete-time stochastic control process for modelling decision making. Among others, it is used to model a reinforcement learning problem.

2. **Learning agent**, **engineered agent**, **scripted agent**. A learning agent is an entity that can sense its environment, process sensory input and produce effects in its environment, in order to improve its performance with respect to a goal. An engineered agent is programmatically controlled. A scripted agent is an engineered agent that follows a simple script, e.g. take action a_1 at t_m, action a_2 at t_n and so on.

3. **Model-based vs. model-free solutions**. A model-based method uses planning and a model of the environment. A model mimics the behaviour of the environment and predicts the next state and next reward given the agent's current state and selected action. Model-free solutions explicitly learn by trial and error. In addition to methods that fall into this dichotomy, there are reinforcement learning techniques that simultaneously learn by trial and error, learn a model of the environment and use the model for planning. This is explained in detail in Section 1.3 of Sutton and Barto (2018).

4. The **credit assignment problem** is the problem of not knowing which past action was responsible for a reward or punishment received due to the fact that a non-zero reward is often sparse and delayed (an action taken at the current timestep is only rewarded after a multitude of future actions). In addition, not just a single action but rather a combination of actions carried out in the appropriate circumstances may be responsible for a reward.

5. **Curse of dimensionality** is a term that describes how hard it is to achieve convergence in high-dimensional state spaces due to the sheer number of states that need to be visited for the learning to be adequate.

6. **Optimal and stable policies**. An optimal policy is one that returns the highest possible cumulative rewards. There may be multiple optimal policies for a problem. A stable policy is achieved if further learning no longer changes the outcome of the policy.

7. **Online vs. offline learning**. Online learning refers to learning conducted at the same time as data is collected. Most reinforcement learning techniques are online learning. Offline reinforcement learning uses previously collected data to learn a policy, without requiring further interaction with the environment. For example, discrete-to-deep supervised policy learning, a novel method proposed in this thesis, is offline learning as it uses previously collected data.

8. **On-policy vs. off-policy learning**. On-policy refers to a learning method whereby the target policy (the policy being updated) is the same as the behaviour policy (the policy used to generate the data used in training). In off-policy methods, the target policy is different from the behaviour policy, implying there are multiple policies involved.

9. **Deterministic vs. stochastic policies**. A deterministic policy always selects the same action given the same state. A stochastic policy specifies probabilities for each action and therefore does not always selects the same action for the same state.

10. A **bootstrapping method** is an RL method that bases its update in part on an existing estimate. TD(0) methods such as Sarsa and Q-learning are bootstrapping methods.

11. **Episodic vs. continuing/non-episodic tasks**. In many MDP problems, there is a natural way of terminating learning, normally when the agent reaches a goal state or another terminal state. In such cases, the duration between when learning starts and when it is terminated is called an episode and such a problem is called an episodic task. MDP problems with no natural way of ending a learning session are called continuing or non-episodic or infinite-horizon tasks. To learn a good policy, an RL agent must normally go through many episodes, sometimes millions of episodes.

12. **Value-based vs. policy-based learning**. A value-based or action-value method is an RL method that learns the values of actions and whereby actions are selected based on their estimated action values. Policy-based methods learn a parameterised policy and are able to do so without a value function. The advantage of the policy-based approach is that learning may result in a policy that is stochastic. The difference between value-based and policy-based learning is explained in more detail in Chapter 2 of this thesis.

13. **Tabular methods vs. approximate solution methods**. These are ways of implementing a model-free RL method. Tabular methods use a table to store values or parameters and are suitable for small scale or toy problems. Approximate solution methods use a function approximator, are more complex than tabular methods, and can accommodate large scale state and action spaces.

TABLE OF CONTENTS

List of Figures

Figure **Page**

List of Tables

Chapter 1

Introduction

This thesis applies reinforcement learning (RL) techniques to address dynamic adversarial games in the context of air combat manoeuvring simulation. It proposes two novel methods—discrete-to-deep supervised policy learning (D2D-SPL) and discrete-to-deep supervised Q-value learning (D2D-SQL)—and uses both single- and multiobjective RL (MORL) techniques. Single objective RL has been the dominant topic in research in RL and relatively little has been done in terms of MORL (Vamplew et al., 2011). Considering many real-world problems often have conflicting objectives, MORL has the potential to be transformative in many areas, including in air combat manoeuvring simulation.

To compare the performance of the D2D and MORL methods, this thesis builds two baselines comprising tabular RL methods and deep RL methods. This thesis implements and tests several D2D and MORL methods and compares them with the baseline methods in terms of the learning time, baseline and comparative performances, and implementation complexity.

1.1 Motivations

Air combat simulation is an important domain because it eliminates risks to human pilots and expedites training. It also provides significant economic motivations considering a fighter jet costs between \$50 million and \$250 million (Mizokami, 2017).

Modelling the air combat manoeuvring domain is an extremely complex application of constructive simulations because it is highly dynamic, adversarial, continuous, partially observable and involves multiple objectives. Here *constructive* means simulations without human players, that is an agent versus agent setting.

Traditional approaches for modelling air combat manoeuvring used agent programming methods which relied on significant domain knowledge to be manually encoded into the simulation environment (Burgin and Fogel, 1975; Burgin and Sidor, 1988). Today, in order to maximise the advantages provided by new technologies (such as autonomous aircraft), new tactics will need to be discovered. Artificial intelligence and machine learning have long been used for this purpose.

A subfield of machine learning, RL has the potential to discover these new tactics because it has proven useful for solving sequential decision making problems. RL has been used to train computer agents to play ATARI games using only pixel inputs (Mnih et al., 2015) and is the technology behind Google DeepMind's AlphaGo Zero (Silver et al., 2017) that beat the world champion of Go without human guidance or domain knowledge beyond game rules. RL has also been chosen to solve a wide variety of problems including autonomous driving (Shalev et al., 2016), natural language processing (Fang et al., 2017b), IoT (Gai and Qiu, 2018) and stock trading (Wu et al., 2020).

In the air-combat domain, RL has been used in recent studies utilising deep RL, hierarchical RL and multiobjective RL. Examples of deep RL-based air-combat research include Zhang et al. (2018); Yang et al. (2019a); Wang et al. (2020); Soyluoğlu (2021), who used DQN; Ma et al. (2018), who used double DQN, and Yang et al. (2019b); Li et al. (2019); Kong et al. (2020); Källström and Heintz (2020), who used DDPG. Examples of hierarchical RL being used in the domain include Pope et al. (2021) and Sun et al. (2021). The first published research on the use of multiobjective RL to address an air-combat problem is by Källström and Heintz (2019). All these techniques are discussed in more detail in Chapter 2.

There are two motivations of this thesis, one related to the air combat manoeuvring domain and one related to RL.

- To investigate whether RL, which is under-explored in this domain, can be used to discover novel tactics for air combat manoeuvring problems without requiring significant amounts of human knowledge and effort as is the case for conventional agent programming methods.
- To investigate which RL methods are effective in solving air combat manoeuvring problems, how tabular (table-based) methods compare with deep RL methods, how learning time can be improved and whether MORL performs better than single-objective RL.

1.2 Scope

The following is the scope of this research.

- Only air combat manoeuvring involving two aircraft in one-versus-one scenarios is evaluated. Furthermore, both aircraft can only move in two-dimensional space, i.e. both flying at a fixed altitude. Other aspects of air combat, such as weapon assignment and sensors, are not within the scope of this thesis.

- The research uses Ace Zero, an air combat simulator from Defence Science and Technology Group, Australia, so the thesis can focus on the RL algorithms developed. In future research, however, different simulators can be used.
- This research evaluates and implements tabular and approximate solution methods. For the latter, the approximators can take many forms, however in this research only neural networks are used as approximators. In other words, only approximate solution methods that fall into deep RL methods are considered.
- Due to time and computing resource constraints, the number of learning episodes are limited to 200,000 for tabular methods and 20,000 for deep RL methods. As an illustration, training a single-objective deep RL agent for 20,000 episode takes about 44 hours.

Figure 3.2 shows a one-versus-one air combat manoeuvring scenario. In this thesis, the following state variables are used.

- the range (distance) between the two aircraft,
- the attack angle (AA), and
- the antenna train angle (ATA)

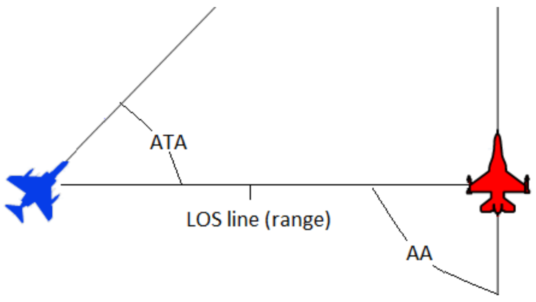


Figure 1.1: Aircraft relative geometry

By convention blue is used to depict the aircraft controlled by the subject agent and red to represent the opposing aircraft. The angles in Figure 3.2 are shown from Blue's point of view. The aircraft centres of mass are connected by the line of sight (LOS) line, which is also used to calculate the range between the two aircraft. The attack angle (AA) is the angle between the LOS line and the tail of the red aircraft. The antenna train angle (ATA) is the angle between the nose of the blue aircraft and the LOS line. Both the AA and ATA help the pilot make manoeuvring decisions. The value of the AA and ATA is within $0° \pm 180°$. By convention, angles to the right side of the aircraft are considered positive and angles to the left negative (McGrew et al., 2010).

1.3 Thesis Questions

The main question this thesis tries to answer is *"How can dynamic adversarial games problems, in the context of air combat manoeuvring simulation, be addressed using single- and multiobjective RL algorithms?"*

Specifically, it attempts to find answers to the following questions:

1. *To what extent are the speed of reinforcement learning and the quality of policies affected by any of the following factors?*
    - *the structure of the reward function*
    - *the state-space representation*
    - *the choice of the learning algorithm*
2. *Is it possible to combine RL and supervised learning techniques to improve learning time and agent experience?*
3. *To what extent does including multiple objectives result in the discovery of better policies?*

1.4 Contribution

This thesis contributes to the following research areas:

- The design of reward signals to drive learning in the air combat manoeuvring domain and corresponding evaluations that determine the most effective signals.
- Novel methods that combine RL and supervised learning that accelerate deep RL and outperform baseline algorithms, presented in Chapter 6. Considering that training in RL is a very expensive and resource-extensive operation, these novel methods that shorten learning time are the most important contribution of this thesis.
- Multiobjective RL agents that outperform single-objective agents, presented in Chapter 7. This is a very important contribution as this proves the relatively recent invention that is multiobjective RL can outperform the performance of single-objective RL without actually increasing the learning time.
- An open-source framework for training and testing single- and multiobjective RL agents.

1.5 Thesis Structure

The rest of this thesis is organised as follows.

- **Chapter 2: Reinforcement Learning** introduces the basics of RL, explains challenges in applying RL methods to sequential decision problems and deals with the two types of model-free RL algorithms, the tabular methods and the approximate solution methods. After a brief survey of the current progress in RL, the chapter discusses deep RL and MORL and presents studies that use RL to solve air combat simulation problems.

- **Chapter 3: Methodology** discusses the many aspects of how this research project was conducted, from the simulator and reward structures to baselines and reproducibility. This methodology is applied in the experiments discussed in Chapters 4, 5, 6 and 7.
- **Chapter 4: Tabular Baseline Methods** discusses the two tabular methods, Watkins's Q(λ) and table-based actor-critic with eligibility traces (ACET). It presents the learning times, learning curves and performance of both methods for addressing the air combat manoeuvring problem and compares their results. It also presents an empirical study of the effects of reward structures on learning. In addition, this chapter answers the first thesis question by presenting the structure of the reward function and the state-space representation.
- **Chapter 5: Deep Reinforcement Learning Baseline Methods** discusses the deep RL methods used as baselines: deep Q-network (DQN), DQN with a target network (DQN-wTN), double DQN and proximal policy optimisation (PPO). It presents the learning curves and performance of the methods and compares their results.
- **Chapter 6: Discrete-to-Deep Methods** discusses two novel techniques called discrete-to-deep supervised policy learning (D2D-SPL) and discrete-to-deep supervised Q-value learning (D2D-SQL). D2D-SPL combines RL and supervised learning and uses tabular ACET. D2D-SQL combines RL and supervised learning and uses Watkins's Q(λ) RL technique. This chapter compares the learning times and performances of the new algorithms with those of several popular deep RL methods. The novel methods in this chapter provide answers to the second thesis question regarding the possibility of combining RL with supvervised learning to improve learning time and agent experience.
- **Chapter 7: Multiobjective Reinforcement Learning Methods** discusses the performance of MORL techniques used in this thesis and how they can outperform single-objective RL agents used to address the same problem. This chapter also answers the third thesis question concerning to which extent multiple objectives result in the discoveries of better policies.
- **Chapter 8: Evaluation** evaluates the learning times, learning curves and indirect and direct performance comparisons. In indirect comparison, the performance of RL agents are compared against baselines obtained from engineered agents. In direct comparison, the RL agents are directly pitted against those engineered agents.
- **Chapter 9: Conclusions** presents the conclusions of this thesis, answers the thesis questions and discusses studies that can be done in the future as extensions to the work conducted for this thesis.

- **Appendix A: Hardware and Software Used** explains the hardware and software used in this thesis. Included in this appendix is the approach taken when running experiments.
- **Appendix B: Baselines** explains the use of engineered agents as baselines for indirect performance comparison.
- **Appendix C: Spy RL Framework** explains why existing RL frameworks lack important features for conducting proper scientific experiments. It then presents a new framework that offers features for solving common problems in RL learning and testing, such as automatically creating multiple best policies and not just last-episode policies, generating average score graphs over many trials, support for plug-and-play components and so on.
- **Appendix D: Solving Gridworld and Cartpole** compares the performance of Q-learning, Watkins's Q(λ) and tabular ACET to solve two classic RL problems to support the argument that some RL methods are suitable for solving continuous RL problems and others are effective in solving discrete RL problems.
- **References**

Chapter 2

Reinforcement Learning

This chapter introduces reinforcement learning (RL) and discusses RL techniques that have been used in air combat simulation. This chapter starts by explaining what RL is and presenting the families of RL algorithms and how they can be implemented using the approximate solutions methods. It then explains two topics that are of recent research interests in RL, deep RL and multiobjective RL (MORL). It concludes with a discussion of RL applications in the air combat simulation and how they compare to approaches proposed in this thesis.

2.1 What Is Reinforcement Learning?

RL is a machine learning technique for solving sequential decision problems. A sequential decision problem arises when the solver must take a sequence of actions and each action selection is decided based on the conditions at the time the action is taken. Playing a computer game is an example of a sequential decision problem. From the time the game is started, the player must continually make decisions as to what to do at every timestep (whether to go left, jump, move right, etc) until the game is terminated. Another example of a sequential decision problem is driving a car from point A to point B. If a computer program is to solve a driving problem, it would select an action (whether to turn left/right, slow down, accelerate or stop) at every timestep until it arrives at its destination.

Inspired by animal learning, RL is a learning method whereby the learner (called the agent) is neither given instructions nor told what to do (Sutton and Barto, 2018). Instead, the agent learns by trial and error and receives feedback in the form of a reward every time the agent selects and executes an action. Thanks to the rewards, the agent is able to alter its behaviour. A newborn giraffe learning to walk and a mouse navigating a maze to find an exit are examples of RL, and so is a computer learning to play chess by playing against itself. By contrast, a learning process whereby the learner is given direct instructions, such as when a young adult studies biology at school, is not RL. Learning programming from a book is also not RL.

Learning in RL is an interaction between the agent and the environment. In the case of RL in animals, the environment could be mother nature or a laboratory setting (if it occurs as part of a study). In the case of a computer agent, the environment could be a software module or a physical environment such as a robot or a self-driving car plus some software. The environment provides or generates rewards following the executions of actions by the agent. An interesting characteristic of RL is delayed reward, meaning an action the agent takes may not be rewarded immediately but at some point in the future. Sutton and Barto (2018) argue that trial-and-error search and delayed reward are the two most important distinguishing features of RL.

The formulation of the environment, state space, action space and reward function is a critical step in solving an RL problem. The details of how this is achieved for the problems addressed in this thesis are given in Chapter 3.

2.1.1 Markov Decision Processes

RL problems are formalised as Markov Decision Processes (MDPs). In an MDP, the system that the agent interacts with is called the environment. An MDP is a tuple $M = \{\mathscr{S}, \mathscr{A}, \mathscr{T}, \gamma, \mathscr{R}\}$. The elements of the tuple are as follows.

- $\mathscr{S}$. The set of possible states of the environment. In most cases each of the states consists of multiple features or state variables.
- $\mathscr{A}$. The set of actions that the agent can choose from at each timestep.
- $\mathscr{T}$. The set of transition probabilities. Each element of $\mathscr{T}$ is the probability of transitioning to a state s' from a state s upon taking an action a. It is a model of the environment's dynamics.
- γ. The discount rate or the discount factor, where $\gamma \in (0,1]$. The discount rate is explained below.
- $\mathscr{R}$. A reward distribution function conditioned on s', s and a.

The output of an MDP algorithm is a policy π that maps states to actions. A policy is a prescription that tells the agent what action to take while in a certain state. An MDP with finite numbers of states, actions and rewards is called a finite MDP.

2.1.2 The Learning Process

An MDP problem is defined by first specifying the set of states and the set of actions, followed by the start state or a distribution over a set of start states and the reward function R. The agent learns by selecting an action for each state it is in, based on the algorithm employed. A timestep is an occasion in which the agent selects an action.

In many MDP problems, there is a natural way of ending a learning session, normally when the agent reaches a goal state or another terminal state. In such cases, a learning session is called an *episode* and such a problem is called an episodic task. MDP problems with no natural way of ending a learning session are called continuing or non-episodic or infinite-horizon tasks.

Figure 2.1 shows a diagram that illustrates the learning process in an MDP system.

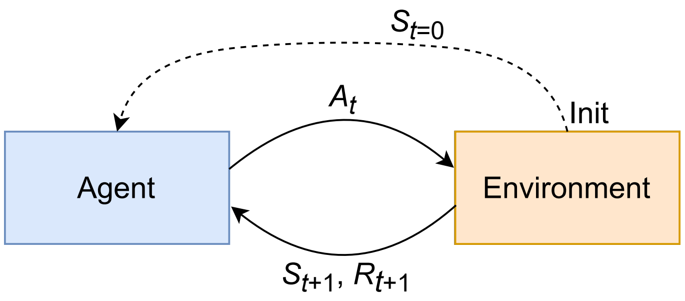


Figure 2.1: Learning in an MDP system

Learning starts at timestep 0 with the environment informing the agent of the state it is in. Based on this information, the agent selects an action. To generalise, the state the agent is in at timestep t is referred to as S_t and the action it takes as A_t. Responding to A_t, the environment increments the timestep and tells the agent that it is now in state S_{t+1} as well as gives the agent a reward R_{t+1}. The agent notes the reward and selects another action A_{t+1}. The process is repeated until the agent reaches a goal state or another non-goal terminal state. Note that after an action selection, the agent may transition to the same state it was in. Assuming the timestep t starts from 0, there is no R_0. The first reward will be R_1 when $t = 1$.

The sequence or trajectory that the learning process generates is like this:

$$S_0 \to A_0 \to R_1 \to S_1 \to A_1 \to R_2 \to S_2 \to A_2 \to R_3 \to ...$$

The set of transition probabilities, which form the model of the environment, are values given by this formula.

$$p(s', r|s, a) := Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{2.1}$$

This defines the probability of the agent landing on state s' and receiving reward r upon taking action a while in state s. The transition probability for each state-action-state-reward tuple is not always one. That is, taking the same action while in a given state does not always produce the same result.

The sum of all probabilities of the agent landing on any state with any reward upon taking action a while in state s is 1.

$$\sum_{s \in S} \sum_{r \in R} p(s', r|s, a) := 1 \tag{2.2}$$

In each timestep, the objective of the agent is to get the maximum cumulative reward, not only the maximum reward for the current timestep. In an episodic task, the simplest way to calculate the maximum future reward is to sum all the rewards in all future timesteps.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{2.3}$$

where R_T is a terminal timestep.

In continuing tasks, G_t in the previous formula can easily lead to infinity. Therefore, for continuing tasks a discounting concept is introduced to calculate the total future reward:

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

where γ is a parameter, $0 \le \gamma \le 1$, called the discount rate.

The reward functions for episodic and continuing tasks can be unified using this formula, where setting γ to 1 produces Equation 2.3 and setting γ to less than one yields Equation 2.4.

$$G_t = \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k \tag{2.5}$$

An agent is said to learn successfully if the resulting policy changes and improves during learning, and eventually converges to a stable, optimal (or near-optimal) solution.

2.1.3 Challenges in RL

Non-trivial RL problems cannot be solved easily due in part to the credit assignment problem and in part to the curse of dimensionality. The credit assignment problem is the problem of not knowing which past action was responsible for a reward or punishment received. This is because a non-zero reward is often sparse (only a handful of states present a non-zero reward) and delayed (an action taken at the current timestep is only rewarded after a multitude of future actions). In addition, not just a single action but rather a combination of actions carried out in the appropriate circumstances may be responsible for a reward.

As for the curse of dimensionality, convergence is hard to achieve in high-dimensional state spaces. A very simple RL problem, such as a 4x4 gridworld, can be solved in a few dozen episodes (Sutton and Barto, 2018). However, if the gridworld is enlarged to, say, 100x100, then things suddenly get more interesting as there are now 10,000 states. It is no longer trivial for the agent to reach the goal. In fact, experiments have proved that it may take tens of thousands of episodes to obtain an optimal policy. An example of a high state space system is the backgammon engine described in (Tesauro, 1995). This system has 10^{20} states.

2.2 Families of RL Algorithms

Generally, RL algorithms are either model-based or model-free. Model-based methods involve planning and use a mathematical model that mimics the environment or allows inferences to be made about how the environment will behave. Model-free methods do not use a model and are based on trial and error, which can be viewed as the opposite of planning (Sutton and Barto, 2018). Many model-based RL solutions are dynamic programming algorithms and a thorough discussion of this technique can be found in Kaelbling and Littman (1996). In the absence of a model, model-free methods must learn from their experience in interacting with the environment. This thesis only deals with model-free algorithms.

Model-free algorithms can be grouped into three classes:

- value-based methods
- policy search methods
- hybrid approaches, which combine value-based and policy search methods. The most popular hybrid approach is actor-critic, which is used in this thesis and explained below.

2.2.1 Value-Based Methods

A value-based or action-value method is an RL method that learns the values of actions, and whereby actions are selected based on their estimated action values (Sutton and Barto, 2018). In this learning method, the agent wishes to take an action that will maximise the cumulative future rewards, and not one that simply returns the greatest reward for the current timestep. To this end, value-based algorithms dictate that the agent calculate a value function for each state or state-action pair. A value function measures the worthiness of being in a state s or taking an action a while in a state s. The value of a state is the expected return (expected cumulative future discounted reward) starting from that state.

Types of Value Functions

There are two types of value functions. The first is the state-value function for a policy π:

$$v_\pi(s) := \mathbf{E}_\pi\left[G_t|S_t = s\right] = \mathbf{E}_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k | S_t = s\right], \text{ for all } s \in \mathscr{S} \tag{2.6}$$

where $\mathbf{E}_\pi$ is the expected value, S includes all nonterminal states and G_t is specified in (2.5).

The second is the action-value function (or quality function) for a policy π:

$$\begin{aligned} q_\pi(s,a) &:= \mathbf{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbf{E}_\pi\left[\sum_{k=t+1}^{T} \gamma^{k-t-1} R_k | S_t = s, A_t = a\right], \text{ for all } s \in \mathscr{S} \end{aligned} \tag{2.7}$$

$v_\pi(s)$ is the value of state s under policy π and refers to the expected return when starting in state s and following π henceforth. Similarly, $q_\pi(s,a)$ is the value of taking action a while in state s under policy π, and refers to the expected return starting from s, taking action a, and thereafter following policy π (Sutton and Barto, 2018). $v_\pi(s)$ can be defined by maximising $q_\pi(s,a)$:

$$v_\pi(s) := \max_a q_\pi(s,a) \tag{2.8}$$

Because the expected return for state s at timestep t is the sum of the current reward and the discounted expected return for state s at timestep $t+1$, $v_\pi(s)$ can be written in this recursive form:

$$v_\pi(s) := \mathbf{E}_\pi[G_t|S_t = s] = \mathbf{E}_\pi[R_{t+1} + \gamma G_{t+1}], \text{ for all } s \in \mathscr{S} \tag{2.9}$$

The choice between a $v_\pi(s)$ and a $q_\pi(s)$ depends on whether or not a model is accessible to the agent. If a model is accessible, state values can be used to determine a policy because the algorithm can look ahead one step and find which action returns the best mix of reward and next state. Without an accessible model, the value of each action must be explicitly estimated in order for the state values to be useful in suggesting a policy (Sutton and Barto, 2018).

Considering that the expected value of a random variable is the average over all possibilities weighted each by its probability of occurring, $v_\pi(s)$ can also be written as:

$$v_\pi(s) := \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma \mathbf{E}\pi\left[G_{t+1}|s_{t+1} = s'\right]\right] \tag{2.10}$$

where $\pi(a|s)$ is the probability that action a is selected while the agent is in state s. The last equation is a sum over all values of variables a, s' and r. Merging the last two sums in the previous equation produces the Bellman equation for state values:

$$v_\pi(s) := \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\left[r + \gamma v_\pi(s')\right], \text{ for all } s \in \mathscr{S} \tag{2.11}$$

If the environment's dynamics are completely known, then the solution for $v_\pi(s)$ is a system of $|\mathscr{S}|$ simultaneous linear equations for $|\mathscr{S}|$ unknowns, where $|\mathscr{S}|$ is the number of nonterminal states. In practice, it is often the case that the environment's exact dynamics are not known and an estimate V of v_π or an estimate Q of q_π has to be used. The agent then learns so that the estimates for the state or action values converge to the true values. The computation of the value function to find a fixed arbitrary policy is called policy evaluation or the prediction problem. In addition, there is the control problem, which is an attempt to find an optimal policy. The prediction and control problems are two problems that every RL algorithm seeks to solve.

An optimal policy, denoted π^*, is a policy with an expected return equal to or greater than the expected returns of all other policies for all states. An MDP problem may have more than one optimal policy. The state-value function that makes a policy an optimal policy is called the optimal state-value function, denoted v^*. The action-value function that causes a policy to be an optimal policy is called the optimal action-value function, denoted q^*. Both v^* and q^* are defined as follows.

$$v^*(s) := \max_{\pi} v_{\pi}(s), \text{ for all } s \in \mathscr{S} \tag{2.12}$$

$$q^*(s,a) := \max_{\pi} q_{\pi}(s,a), \text{ for all } s \in \mathscr{S} \text{ and } a \in \mathscr{A} \tag{2.13}$$

$v^*(s)$ can be written in these two forms of the Bellman optimality equation (BOE) for v^*.

$$v^*(s) := \max_{a} \mathbf{E}\left[R_{t+1} + \gamma v^*(s_{t+1}) | S_t = s, A_t = a\right] \tag{2.14}$$

$$v^*(s) := \max_{a} \sum_{s',r} p(s',r|s,a)\left[r + \gamma v^*(s')\right] \tag{2.15}$$

The BOE for q^* is

$$\begin{aligned} q^*(s,a) &:= \mathbf{E}\left[R_{t+1} + \gamma \max_{a} q^*(s_{t+1},a') | S_t = s, A_t = a\right] \\ &= \sum_{s',r} p(s',r|s,a)\left[r + \gamma \max_{a'} q^*(s',a')\right] \end{aligned} \tag{2.16}$$

The BOE assumes that the environment's dynamics are known precisely, that there are sufficient computing resources to solve it, and that the problem at hand meets the Markov property. Due to the difficulty in meeting these assumptions, the BOE is hardly useful except in trivial problems. Nevertheless, it is still important to study the BOE because many RL algorithms can be viewed as approximately solving the BOE. For example, Q-learning, an algorithm that will be used in this project, is an RL algorithm that approximates q^*.

Temporal Difference Learning and Q-Learning

Temporal difference (TD) learning is a type of value-based learning. The simplest TD method, called TD(0) or one-step TD, updates $V(S_t)$ at $t+1$. In other words, it only needs to wait until the next timestep to update the estimated value function. Its update rule is as follows.

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{2.17}$$

where $R_{t+1} + \gamma(S_{t+1})$ is the target of the TD update and $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ the TD error (δ_t). The TD error measures the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_t + 1)$. α is a small positive fraction called the step-size parameter, which determines the rate of learning.

The update rule for $V(S_t)$ intuitively means that at each timestep the value of the current state S_t is updated by "backing up" the value of the next state S_{t+1} so that the value of the current state will be closer to the value of the next state by a fraction of the way toward the value of the next state. The update rule does not apply to terminal states, whose state and action values are always zero.

Further, TD(0) is called a *bootstrapping* method because it bases its update in part on an existing estimate. TD(0) is guaranteed to converge to v_π provided every state is visited an infinite number of times (Sutton and Barto, 2018).

The TD(0) algorithm is given in Algorithm 1. Here S' is used to denote the next state S_{t+1} and $\mathscr{S}^+$ means all states including terminal states. The mechanism for action selection in Line 5 differs from one algorithm to another.

Algorithm 1: TD(0) (Sutton and Barto, 2018)

```
   Input: the policy $\pi$ to be evaluated
   Parameter(s): step size $\alpha \in (0,1]$, small $\epsilon > 0$
1  Init $V(s)$, for all $s \in \mathscr{S}^+$, arbitrarily except $V(terminal) = 0$;
2  for each episode do
3      Initialise $S$;
4      for each step of episode or until S is terminal do
5          A $\Leftarrow$ action given by $\pi$ for $S$;
6          Take action $A$, observe $R$, $S'$;
7          $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$;
8          $S \leftarrow S'$;
9      end
10 end
```

Two of the most popular variants of TD are Sarsa and Q-learning. The name Sarsa was given by Sutton and Barto (2018), but the Sarsa algorithm was originally proposed by Rummery and Niranjan (1994), who called it "modified connectionist Q-learning." Sarsa is not used in this thesis, so it will not be explained further. On the other hand, Q-learning is one of the tabular methods used in this research so it is discussed in this section.

Proposed by Watkins (1989), Q-learning is a TD learning control method that estimates the value of $Q^*(s,a)$, the optimal action-value function. The update rule for Q-learning is as follows.

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t,A_t)\right] \tag{2.18}$$

where $Q(S_t,A_t)$ is the current estimate of $Q^*(s,a)$. The update rule clearly indicates that Q-learning is a bootstrapping method. Q-learning is presented in Algorithm 2.

Line 5 of Algorithm 2 is the action selection step whereby the agent selects an action based on a certain strategy. ϵ-greedy is a strategy often used in Q-learning and other value-based methods. In this strategy, the agent selects the action with the maximum value (the greedy action) most of the time and a random action the rest of the time. The random action is chosen with probability ϵ. This is explained further in Section 2.3.

Line 7 of Algorithm 2 shows that the action-value update function includes the Q-value for the greedy action from the subsequent state. As the greedy action may not be the policy that

Algorithm 2: Q-learning (Watkins, 1989)

```
Parameter(s): step size $\alpha \in (0,1]$, small $\epsilon > 0$
Result: Q
1  Init. $Q(s,a)$, for all $s \in \mathscr{S}^+$, $a \in \mathscr{A}(s)$, arbitrarily, $Q(terminal,.) = 0$;
2  for each episode do
3      Initialise $S$;
4      for each step of episode or until S is terminal do
5          Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy);
6          Take action $A$, observe $R$, $S'$;
7          $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$;
8          $S \leftarrow S'$;
9      end
10 end
```

is actually followed by the agent, Q-learning is an off-policy method. Off-policy methods are discussed in detail in Section 2.4

2.2.2 Policy Search Methods

The value-based class of methods learn the values of actions and choose actions based on their estimated action values. These methods are easy to use, but come with two drawbacks. First, the policy obtained is deterministic, whereas in some contexts optimal policies may be stochastic. Second, learning can be prohibitively long for complex problems involving large state and action spaces (Sutton et al., 2000).

Policy search methods learn a parameterised policy without a value function. Peters and Neumann (2015) group policy search methods into policy gradient methods, weighted maximum likelihood approaches and other extensions. This thesis is focused on policy gradient methods, the most popular policy search methods, and therefore this section only discusses this category of policy search methods.

First introduced by Barto et al. (1983), policy gradient methods learn a parameterised policy and are able to do so without a value function. The advantage of the policy gradient approach is that learning may result in a policy that is stochastic. In addition, policy gradient methods can be more effective than value-based methods in solving problems with high-dimensional state-action spaces (Kurniawan et al., 2020).

In a policy gradient method, a policy is the probability that action a is taken when the agent is in state s at time t with parameter vector $\boldsymbol{\theta}$. In other words,

$$\pi(a|s,\boldsymbol{\theta}) = \Pr\{A_t = a|S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta}\} \tag{2.19}$$

If the action space is discrete and not too large, an exponential soft-max distribution can be used in action selection,

$$\pi(a|s,\boldsymbol{\theta}) \doteq \frac{e^{h(s,a,\boldsymbol{\theta})}}{\sum_b e^{h(s,b,\boldsymbol{\theta})}}, \tag{2.20}$$

where

$$h(s,a,\boldsymbol{\theta}) = \boldsymbol{\theta}^{\intercal}\mathrm{x}(s,a), \tag{2.21}$$

where $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and $\mathrm{x}(s,a) \in \mathbb{R}^{d'}$ is the feature vector.

2.2.3 Actor-Critic

First introduced by Barto et al. (1983), the actor-critic algorithm learns the policy parameter based on the gradient of some scalar performance measure with respect to the policy parameter, and therefore belongs to the policy-gradient family of methods. However, actor-critic also uses a value function to learn a policy parameter. The variant of actor-critic used in this thesis is called actor-critic with eligibility traces (ACET), a discussion of which can be found in Chapter 13 of Sutton and Barto (2018).

Actor-critic learning may result in a stochastic policy, whereby the probability that action a is taken at time t when the agent is in state s at time t with parameter vector $\boldsymbol{\theta}$ is given by (2.19).

Actor-critic is presented in Algorithm 3. The policy π is the actor and is used to select actions. The estimated value function $\hat{v}$ is the critic, which criticises the actions selected by the actor. The critique takes the form of a TD error, a scalar signal that is the sole output of the critic and drives all learning in both the actor and the critic (Sutton and Barto, 1998).

In Algorithm 3 the TD error δ is calculated in Line 8. This error is used in the update of the estimated state-value function's parameter vector w (Line 9) and the update of the policy's parameter vector $\boldsymbol{\theta}$ (Line 10).

Algorithm 3: Actor-critic for episodic problems (Sutton and Barto, 2018)

```
Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
Input: a differentiable state-value function parameterization $\hat{v}(s,\mathrm{w})$
Parameter(s): step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathrm{w}} > 0$
1  Initialise policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathrm{w} \in \mathbb{R}^{d}$ (e.g, to 0) ;
2  for each episode do
3      Initialise $S$ (first state of episode);
4      $I \leftarrow 1$ ;
5      for each step of episode or until $S$ is terminal do
6          $A \sim \pi(.|S,\boldsymbol{\theta})$ ;
7          Take action $A$, observe $S'$, $R$;
8          $\delta \leftarrow R + \gamma\hat{v}(S',\mathrm{w}) - \hat{v}(S,\mathrm{w})$        (if $S'$ is terminal, $\hat{v}(S',\mathrm{w}) \doteq 0$) ;
9          $\mathrm{w} \leftarrow \mathrm{w} + \alpha^{\mathrm{w}}\delta\nabla\hat{v}(S,\mathrm{w})$ ;
10         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}} I\delta\nabla\ln\pi(A|S,\boldsymbol{\theta})$;
11         $I \leftarrow \gamma I$;
12         $S \leftarrow S'$;
13     end
14 end
```

2.3 Exploitation vs. Exploration

For each state during learning, the action with the maximum value is called the greedy action. If the agent selects a greedy action, the agent is said to be exploiting its current knowledge of the action values. On the other hand, if the agent selects a non-greedy action, the agent is exploring. Exploitation earns the agent the maximum reward at one timestep, however exploration may yield a greater total reward in the long run because a suitable level of exploration reduces the risk of the agent converging to a sub-optimal policy (Sutton and Barto, 2018).

Balancing between exploration and exploitation has been researched for decades (Thrun, 1992; Coggan, 2001; Ishii et al., 2002; Fruit et al., 2018). One common practice, though, is to exploit most of the time and explore once in a while, say by having the exploration and exploitation ratio to equal a small fraction ϵ. This method of action selection that balances exploitation and exploration this way is called ϵ-greedy. ϵ does not need to be constant. In fact it is common practice to use a bigger ϵ at initial timesteps and gradually reduce it as learning progresses. Using this strategy, the agent is encouraged to explore more at the beginning of learning and explore less as the policy stabilises.

2.4 On-Policy vs. Off-Policy Learning

In terms of how the policy is updated, all RL methods are either on-policy or off-policy. On-policy refers to a learning method whereby the target policy (the policy being updated) is the same as the behaviour policy (the policy used to generate the data used in training). In other words, an on-policy learning algorithm estimates the value function of the policy generating the data. In off-policy methods, on the other hand, the target policy is different from the behaviour policy, implying there are multiple policies involved. Generally on-policy is simpler. Off-policy, on the other hand, can be more effective but may be slower to converge, and can even diverge, when used in conjunction with function approximation (Baird, 1995; Tsitsiklis and Van Roy, 1997).

2.5 Approximate Solution Methods

An RL algorithm can be implemented as a tabular method or as an approximate solution method. The decision whether to implement an algorithm as a tabular or approximate solution method largely depends on the problem to solve. Tabular methods use a table to store state/state-action values or policy parameters and are suitable for small scale or toy problems. When the state space is large, using a table for storing state/state-action values can be prohibitively expensive in terms of memory usage as well as the time and data needed to fill and update the table accurately. Also, when there are a lot of states, only a subset of the state space can be visited during training. The problem then is that of generalisation, i.e. Can we use in production the available state values for states that were never visited during training?

This kind of generalisation, called function approximation, has been studied extensively in supervised learning, so there are already supervised learning generalisation methods that can be combined with RL algorithms. Generally, supervised learning methods that learn efficiently from incrementally acquired data may potentially be suitable for RL (Sutton and Barto, 2018). This section discusses the approximate solution methods in general.

2.5.1 Overview

Unlike in the tabular case where the approximate value function is represented as a table, in function approximation in RL, the value function is represented as a parameterised function with a weight vector $\mathbf{w} \in \mathbb{R}^d$, where $d \ll |\mathcal{S}|$ (the dimensionality of $\mathbf{w}$ is much less than the number of states). Because the number of states is much higher than the number of weights, updating a state affects many other states and it is impossible for all the states to have correct values.

In the approximate method, the value function is replaced with an approximate function, which is approximation of the actual value function conditioned on both the states and weight vector:

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s) \tag{2.22}$$

The approximate function can be a linear function of the weight vector or a non-linear function such as an artificial neural network. A comprehensive treatment of different types of approximate functions can be found in Chapter 9 of (Sutton and Barto, 2018).

Learning in the approximation case of RL occurs exactly the same as in the tabular case, however its objective is to continuously update a weight vector **w** rather than the entries of a table. In other words, the objective is to find an optimal weight vector **w** instead of $v^*(s)$ or $q^*(s,a)$. Once **w** is obtained, $\hat{v}(s)$ and $\hat{q}(s,a)$ can be calculated.

Now, the problem has shifted to that of supervised learning. However, one question lingers: In supervised learning we are given input-output pairs and use them to find a weight vector **w**, how do we get input-output pairs from an MDP system?

Recall that in the tabular case, every RL method has an update rule that shifts the estimates of state values or state-action values closer to the actual values. For example, the TD(0) method has the following update rule to update $V(S_t)$ at $t+1$.

$$V(S_t) \leftarrow V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \tag{2.23}$$

where $R_{t+1} + \gamma(S_{t+1})$ is the target of the TD update and $R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ the TD error (δ_t). The TD error measures the difference between the estimated value of S_t and the better estimate $R_{t+1} + \gamma V(S_t + 1)$. α is a small positive fraction called the step-size parameter, which determines the rate of learning.

To convert an RL problem to a supervised learning problem, the current value function can be viewed as input and its better estimate as output. Therefore, TD(0) can use the following as input-output pairs:

$$S_t \rightarrow R_{t+1} + \gamma V(S_{t+1}) \tag{2.24}$$

Because only an estimate of V is available rather than V itself and because the estimate is a function of **w**, S_t can be written as

$$S_t \rightarrow R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) \tag{2.25}$$

Now that the input-output pairs for the RL problem have been formulised, a suitable supervised learning algorithm can be chosen to solve it. Of the many supervised learning methods, those based on stochastic gradient descent (SGD) are most widely used and suitable for online RL. SGD is an optimisation technique for finding the values of parameters of a function f that minimises a cost function. In this case, the parameters are the forementioned weight vector **w**, f is the value function of the RL problem, and the cost function is often chosen to be the following mean squared value error.

$$\overline{VE}(w) \doteq \sum_{s\in S} \mu(s)[v_\pi(s) - \hat{v}(s,\mathbf{w})]^2 \tag{2.26}$$

The square root of $\overline{VE}$ measures how much the approximate values differs from the actual values and $\mu(s)$ is a state distribution that specifies how important the error at each state, The sum of $\mu(s)$ for all states is equal to 1 and oftentimes the number of visits a state receives divided by all visits is used as the value of $\mu(s)$(Sutton and Barto, 2018).

Using SGD to update **w** at each timestep, here is the update rule for **w**.

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t - \frac{1}{2}\alpha\nabla[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)]^2 \tag{2.27}$$

$$= \mathbf{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t,\mathbf{w}_t)]\nabla\hat{v}(S_t,\mathbf{w}_t) \tag{2.28}$$

where α is a positive step-size parameter. Equation 2.28 is not usable in TD(0) or other bootstrapping methods, where an update is based in part on an existing estimate, because $v_\pi(S_t)$ is unknown. However, an approximation of $v_\pi(S_t)$ can be calculated using its bootstrapping estimate, denoted U_t:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[U_t - \hat{v}(S_t,\mathbf{w}_t)]\nabla\hat{v}(S_t,\mathbf{w}_t) \tag{2.29}$$

Due to the fact that U_t is used in lieu of $v_\pi(S_t)$, TD(0) is not a gradient-descent method (Barnard, 1993). Rather, it is a semi-gradient method. Algorithm 4 shows the semi-gradient TD(0) algorithm for estimating $\hat{v} \approx v_\pi$ (Sutton and Barto, 2018).

Algorithm 4: Semi-gradient TD(0) algorithm for estimating $\hat{v} \approx v_\pi$ (Sutton and Barto, 2018)

```
   Input: the policy $\pi$ to be evaluated
   Input: a differential function $\hat{v} : \mathscr{S}^+ \text{x } \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(terminal, \cdot) = 0$
   Parameter(s): step size $\alpha > 0$
1  Initialise value function weights $w \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = 0$);
2  for each episode do
3      Initialise $S$;
4      for each step of episode or until $S$ is terminal do
5          Choose A $\sim \pi(\cdot|S)$;
6          Take action $A$, observe $R$, $S'$;
7          $\mathbf{w} \leftarrow \mathbf{w} + \alpha[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})]\nabla\hat{v}(S,\mathbf{w})$;
8          $S \leftarrow S'$;
9      end
10 end
```

2.5.2 Disadvantages

The advantage of using an approximate solution method is the ability to scale to high-dimensional problems. However, there are accompanying disadvantages. First, compared to tabular methods, each episode of an approximate solution method takes longer due to the higher computational cost of evaluating and updating the approximation, especially when one or more neural networks are used as the approximator. Second, while some table-based algorithms such as Q-learning have been proven to converge to an optimal solution, the same cannot be said of approximate solution methods. Approximate solution methods may also fail due to several causes including overestimation (Thrun and Schwartz, 1993; van Hasselt et al., 2016).

2.6 Deep Reinforcement Learning

The approximate solution method may use a non-linear approximate function in the form of a neural network. Neural networks have long been used in supervised learning and recent big success stories in supervised learning have been fuelled by *deep learning*, a set of supervised learning techniques powered by many-layer neural networks or deep networks. Deep learning has been successfully applied to various fields such as computer vision, natural language processing, speech and audio recognition, machine translation and bioinformatics (Voulodimos et al., 2018; Socher et al., 2012; Noda et al., 2015; Liu and Zhang, 2018; Min et al., 2017). Deep learning has also been extended to reinforcement learning, resulting in a new subfield of study called deep reinforcement learning (deep RL). Before the term ‘deep learning’ became popular, RL methods that used neural networks were called neural RL. Neural RL can be referred to as deep RL if the neural network used is sufficiently deep. However, there is not yet a consensus as to how many layers a neural network has to have in order to qualify as “deep” (Goodfellow et al., 2016). Today, neural RL and deep RL are used interchangeably, but deep RL is more popular.

The first success story of neural RL was that of TD-Gammon, a computer program that plays backgammon (Tesauro, 1995). TD-Gammon learned a policy by playing against itself. To get to the level where it could beat another computer-based backgammon player, it self-played 300,000 games. To make it even more skilful to the point it could defeat a grandmaster, it self-played 1.5 million games. After this initial success, however, researchers attempting to exploit neural networks to solve large-scale RL problems were faced with repeated failures that resulted in a move away from neural RL, as reported in Boyan and Moore (1995). Some consider TD-Gammon a one-time exception and believe the success of TD-Gammon was not due to the RL algorithm used, but because of the setup of co-evolutionary self-play biased by the dynamics of backgammon (Pollack and Blair, 1997).

The inability of neural networks to act as good function approximators in many RL studies is mainly attributed to the fact that samples generated in RL learning are correlated and not independently and identically distributed. Additionally, the neural network often forgets what it has learned, forcing it to re-learn when it sees a sample that has previously been presented to it, something known as the relearning problem (Lin, 1993). There are two ways to overcome these shortcomings, by using experience replay (Lin, 1993) or by employing multiple agents working asynchronously to update a centralised network.

Experience replay is an interesting invention because it inspired successful large-scale neural RL cases that came many years later. An experience is a tuple (s, a, r, s'), where s is the state the agent is in while taking action a, resulting in reward r and a new state s'. Experience replay is a method whereby a collection of experiences are presented repeatedly to the learning algorithm as if the learning agent experienced it again and again. Two benefits of experience replay are the expedition of the process of credit assignment and the opportunity for the agent to refresh what it has learned (Lin, 1993).

Neural Fitted Q Iteration (NFQ) (Riedmiller, 2005) is a method that uses experience replay. It is based on a multi-layer perceptron (MLP) and its principle idea is to collect all experience tuples within an episode and present the collection at the end of the episode to train the MLP rather than after each timestep. Targets are generated using a cost function. This is repeated for a preset number of episodes and the experience collection at each episode includes all the collections from prior episodes (Riedmiller, 2005).

Another technique that is based on experience replay is deep Q-network (DQN), which was used to train agents to play ATARI games. Based on Q-learning, DQN stores incoming samples in a buffer and randomly selects a subset of these samples at every timestep to train the neural network (Mnih et al., 2013). The same team extended DQN by introducing a second network called the target network to better de-correlate samples. At the beginning of a learning episode, the primary network is cloned and the clone is used to produce targets for the primary network. Every n timesteps the weights of the target network are updated using those of the primary network. This technique is known as DQN with a target network (DQN-wTN) (Mnih et al., 2015) and sometimes is called Nature DQN, after the journal that published it. For example, the term Nature DQN is used in Pritzel et al. (2017). Some scholars do not distinguish between DQN and DQN-wTN and often refer to DQN-wTN as DQN, as can be seen in van Hasselt et al. (2016). The success of DQN-wTN can be attributed to two innovations: 1. off-policy training with samples from a replay buffer to minimise correlations between samples; 2. The use of a target Q-network to give consistent targets during temporal difference backups. Both DQN and DQN-wTN are used in this thesis and explained in detail in Chapter 5.

Other derivatives of DQN include a parallel implementation called Gorila (Nair et al., 2015), double DQN (van Hasselt et al., 2016), prioritised experience replay (Schaul et al., 2015), dueling D-DQN (Wang et al., 2015) and distributed prioritised experience replay (Horgan et al., 2018). Rainbow, another variant of DQN, combines features in other DQN variants and proves able to improve performance in some ATARI games (Hessel et al., 2018).

Another algorithm that was inspired by DQN is deep deterministic policy gradient (DDPG) (Lillicrap et al., 2015). Unlike DQN which is used to solve problems with discrete action spaces, DDPG is designed to deal with problems with continuous action spaces. Because finding the greedy policy in a continuous space requires that the action be optimised at every timestep, which would be too slow in the case of a nontrivial action space, there is no easy way of applying

Q-learning to continuous action spaces. DDPG therefore uses an actor-critic algorithm based on the DPG algorithm (Silver et al., 2014), which maintains a parameterised actor function representing the policy and a critic learned using the Bellman equation as in Q-learning. DDPG uses neural network function approximators like those found in DQN-wTN and experience replay to de-correlate training samples. DDPG was tested with ATARI games and was able to solve those problems within 2.5 million steps of experience, a factor of 20 fewer steps than DQN requires for good Atari solutions.

Another deep RL technique that is based on policy gradient is proximal policy optimisation (PPO) (Schulman et al., 2017). PPO utilises a gradient estimator and maximises a "surrogate" objective, and then calculates the probability ratio of the new update and the old update and clips the ratio so that large updates are ignored and replaced with its clipped version.

Algorithms that capitalise on asynchronous parallel agents include asynchronous advantage actor-critic (A3C) (Mnih et al., 2016) and actor-critic with experience replay (ACER) (Wang et al., 2016). ACER utilises both experience replay and asynchronous parallel agents to improve A3C's sample efficiency.

Both experience replay and asynchronous parallel agents have proven effective but have weaknesses as well. Experience replay with a large buffer can hurt performance and both small and large replay buffers can hurt the learning process (Zhang and Sutton, 2017). Asynchronous parallel agents are sample inefficient and require a complex application architecture (Wang et al., 2016).

A giant leap in deep RL is DeepMind's AlphaGo, an RL agent that beat 18-time Go world champion Lee Sedol in 2016. AlphaGo uses two deep neural networks, one for evaluating board positions and one for selecting moves. The deep neural networks were trained using a combination of supervised learning from human experts and RL from self-play (Silver et al., 2016). A year later, the AlphaGo researchers created AlphaGo Zero, which is based solely on RL and was trained without human data, guidance or domain knowledge beyond game rules. AlphaGo Zero trained faster than and beat AlphaGo 100-0 (Silver et al., 2017).

2.7 Multiobjective Reinforcement Learning (MORL)

Most research on sequential decision-making problems has been focused on single-objective settings. However, there are also such problems with multiple, often-conflicting, objectives. For instance, an autonomous vehicle needs to arrive at the destination as soon as possible and at the same time minimise fuel usage, which requires the vehicle travelling at a constant speed. If the objectives are directly related or completely independent, the objectives can be easily combined into a single objective and the problem can then be transformed into a single-objective RL problem. However, for problems with conflicting objectives, any policy must either maximise only one objective or represent a trade-off between the conflicting objectives (Vamplew et al., 2011). Multiobjective problems do not necessarily have a single optimal policy and are best formulated as multiobjective Markov decision processes (MOMDP).

Roijers et al. (2013) identify three scenarios where sequential decision-making problems cannot be reduced to single-objective settings:

1. The unknown weights scenario, where the weights $\boldsymbol{w}$, used as parameters to a function f for scalarising a multiobjective reward vector into a scalar value, does not exist.

2. The decision support scenario, where scalarisation is infeasible throughout the entire decision-making process because of the difficulty of specifying $\boldsymbol{w}$, or even the scalarisation function f.

3. The known weights scenario, where $\boldsymbol{w}$ is known at the time of planning or learning, thus making scalarisation possible and feasible. However, such a scenario may be undesirable because of the difficulty of defining a single-objective MDP with additive returns such that, for all policies and states, the expected return equals the scalarised value.

All MORL algorithms seek to find a policy that produces suitable compromises between the multiple objectives of the task. A 'good' compromise can be defined in terms of Pareto dominance. MORL is a generalization of single-objective RL, with the environment formulated as a multiobjective MDP (MOMDP) = $\{\mathscr{S}, \mathscr{A}, \mathscr{T}, \gamma, \mathbf{R}\}$, which differs from the single-objective case in the reward function. In MOMDP the reward function yields a vector instead of a scalar and can be written as:

$$\mathbf{R}(s,a,s') = [R_1(s,a,s'),...,R_m(s,a,s')] \tag{2.30}$$

where m is the number of objectives.

One motivation for MORL is to convert a single-objective problem multiobjective to make the original problem easier to solve. This is known as multiobjectivisation and is a known technique in both RL and optimisation. Multiobjectivisation works by decomposing the single objective into multiple objectives or by adding new objectives. Several different techniques include multiobjectivisation by decomposition of scalar cost functions (Handl et al., 2008) and multiobjectivisation by reward shaping (Brys et al., 2014).

There are two classes of MORL algorithms (Vamplew et al., 2011):

1. single-policy. The most common type of MORL today, the single-policy method aims to learn a single policy that is 'optimal' in some way. At first glance, single-policy MORL methods are similar to single-objective RL, but there is a fundamental difference between the two. The single-objective RL algorithm is guaranteed to have an optimal policy or several optimal policies with identical performance. In the multiobjective case, there may exist multiple Pareto policies that are all optimal, so the algorithm must decide which policy is the preferred policy.

2. multipolicy. The multipolicy method attempts to find a set of policies that approximate the Pareto front.

Several single-objective RL methods have been extended to solve MORL problems. One such method is multiobjective Q-learning, which extends Q-learning. To solve a MORL problem, each objective is assigned a separate Q-table and all the Q-tables can be written as a vector and updated using this rule:

$$\mathbf{Q}(s,a) \leftarrow \mathbf{Q}(s,a) + \alpha\delta \tag{2.31}$$

$$\delta_i = \mathbf{R}_i(s,a,s') + \gamma max_{a'}\mathbf{Q}_i(s',a') - \mathbf{Q}_i(s,a) \tag{2.32}$$

In multiobjective Q-learning, greedy action selection is determined by applying a multiobjective selection mechanism to the vector values for the actions.

2.8 Reinforcement Learning for Air Combat Simulation Problems

Air combat simulation is an important domain because it eliminates risks to human pilots and expedites training. In addition, there are significant economic motivations for using air-combat simulators, given that a fighter jet costs between $50 million and $250 million (Mizokami, 2017).

Modelling air combat is usually divided into within-visual-range (WVR) and beyond-visual-range (BVR). WVR, often referred to as dogfight, is air combat whereby the opponents can see each other and try to manoeuvre their noses into position to shoot the other down. BVR air combat involves two opposing teams of aircraft located at large distances from each other (hundreds of kilometres) and operating in large air spaces (thousands of square kilometres). This type of combat is heavily reliant on active radar homing missiles with ranges of approximately 50 kilometres (Floyd et al., 2017). WVR air combat can be split into one-vs-one (1v1) and n-vs-m (NvM) models (Shaw, 1985). This thesis research only involves the 1v1 WVR model and focuses on constructive air combat simulations, which are used for operations research and in which there are no human players. In other words, it is an agent versus agent setting.

This section presents prior studies that used RL to address air-combat manoeuvring problems.

2.8.1 Temporal-Difference Techniques

The earliest study applying RL in the pilot decision making domain is probably that of Azak and Bayrak (2008). In their paper, they addressed the threat evaluation problem using a neural network and the weapon assignment (WA) problem using RL. Solving a WA problem means trying to find an optimal assignment of a set of weapons to a set of targets in order to maximise damage to the opposition. For the WA problem, Azak and Bayrak (2008) simplified the state space into 16x16 grids and chose Q-learning as their method, which they modified to allow agents to learn how to coordinate with each other. Their work was later extended by Mouton et al. (2011) who conducted two experiments, one using Q-learning and another using Monte-Carlo control with exploring starts (MCES), to address the WA problem. The state space in Mouton et al.'s tests was greatly simplified into a 71x71 grids and they were able to achieve some success.

Another early study is that of Vinberg (2009), who modified continuous-time continuous-space algorithms proposed by Doya (2000). Doya's methods in turn are extensions to discrete-time discrete-space TD(0), TD(λ) and actor-critic algorithms. The main objective of Vinberg's learning agent is to shoot down two opposing aircraft. The agent gets a positive reward upon successfully downing an opponent and a negative reward upon a miss and if it itself is shot. Whereas exploration in RL methods is usually random, in this system the exploration phase of agent learning followed predefined rules, with the aim of expediting learning. The study, which used a neural network as a function approximator and was restricted to beyond visual range (BVR) 1v1 fights, yielded positive results with the learning agent able to outperform the systems that supplied rule-based instructions used in the exploration phase of the learning.

After these preliminary works, the trend continued with some researchers using Q-learning or some modifications of it to solve air combat simulation problems. Lee and Bang (2012) used Q-learning to help aircraft evade missiles moving in a horizontal plane while modelling the aircraft equations of motions as a simplified fourth-order point mass. Teng et al. (2012) combined Q-learning and a self-organising neural network to develop an RL policy for outmanoeuvring an opponent in 1v1 air combat. Their TD-FALCON system was able to automatically extract the state space and action space as the domain knowledge to bootstrap reinforcement learning and to discover new tactics to consistently outmanoeuvre its adversary. Bilgin and Kadioglu-Urtis (2015) applied Q-learning with eligibility traces, a technique known as Watkins' Q(λ) (Watkins, 1989), to a multiagent aerial manoeuvring problem by training a pursuer and an evader. Fang et al. (2017a) combined Q-learning and a behaviour tree to train a fighter agent to decide whether to patrol, attack an enemy, turn around or flee. Their strategy was to use Q-learning to obtain a Q-table and then insert the Q-table in the behaviour tree.

2.8.2 Deep Reinforcement Learning

After Mnih et al. (2013, 2015) started the deep RL era by showing that deep networks could be used to solve complex RL problems, researchers began using deep RL methods, such as DQN, DDPG and PPO, to address the pilot decision making problem. Liu and Ma (2017) used a deep network in a 1v1 combat. Their state space was a continuous three-dimensional space where a state in their system consisted of twelve state variables. The first six state variables were the x and y coordinates, velocity, vertical direction, yaw and roll angles of the learning agent and the second six state variables were the same attributes of the opposing agent. Their action space consisted of five actions (turn-right-up, turn-right-down, turn-left-up, turn-left-down, cruise) and their agents learned by trying to outmanoeuvre an opponent that was moving according to the Minimax algorithm. Their results show that the DQN agent outperformed the Minimax-based pilot. Other researchers that used DQN to address the air combat problem include Zhang et al. (2018), Yang et al. (2019a), Wang et al. (2020) and Soyluoğlu (2021). The latter compared the performance of DQN and PPO.

Ma et al. (2018) used double DQN in a two-dimensional air combat system to train agents to outmanoeuvre opponents. A state in their solution consisted of five state variables, the x and y positions, velocity, heading and bank angle of the aircraft and a sum-tree structure was used for memory buffer and prioritised experience replay to sample transitions. When compared to Minimax and Monte Carlo Tree Search (MCTS), their solution beat both competing methods by a large margin.

Even though DQN and its variants can be used to solve high-dimensional RL problems, they can only deal with problems with discrete action spaces. For air combat systems with continuous action spaces, a different approach is needed. Yang et al. (2019b) used DDPG (Lillicrap et al., 2015) to solve a 1v1 air combat manoeuvring problem for their continuous action-spaced system. Their state space contained eight state variables: the angle between the agent's velocity vector and line of sight, the angle between the opponent's velocity vector and line of sight, the angle between the two aircraft's speed vectors, the distance between the two aircraft, the agent's velocity, the difference between the agent's and the opponent's velocities, the agent's altitude, the difference in altitude between the two aircraft. Their action space contained three components: the load factor along the velocity direction representing the thrust/resistance of the aircraft, the load factor along the pitch direction and the roll angle around the velocity vector. Training agents against an aircraft that started from a random position and flew in a straight line, their DDPG solution was able to learn good continuous air combat manoeuvring policies.

Other researchers that used DDPG to solve air combat problems include Li et al. (2019); Kong et al. (2020); Källström and Heintz (2020). Li et al. (2019) proposed an autonomous manoeuvring decision algorithm based on DDPG to train UAV agents to generate efficient manoeuvres under endpoint constraints in an interactive environment. They were able to develop a desirable manoeuvring decision policy using their algorithm. Kong et al. (2020) modelled air combat as a state-adversarial Markov decision process (SA-MDP) to introduce observation errors of UAVs. Their objective was to make the strategy network of air combat manoeuvre more robust in the presence of those observation errors. They used maximum entropy (MaxEnt) inverse RL to design a shaping reward to accelerate the convergence of their state-adversarial deep deterministic policy gradient (SA DDPG) algorithm.

Källström and Heintz (2020) used a version of DDPG for the multiagent environment called multiagent DDPG. They defined two problems called Aerial Reconnaissance with Sparse Rewards and Airstrike with Adjustable Risk Taking. In the aerial reconnaissance problem, two agents try to visit three Points of Interest (POI) as quickly as possible and both must coordinate their actions to achieve their goal. The airstrike problem involves a fighter jet that must navigate to a target location protected by an air defence system and a second aircraft that is equipped with a jamming pod. The two aircraft must collaborate to evade the air defence system.

Vlahov et al. (2018) and Zhang et al. (2020) evaluated and tested various RL and non-RL algorithms in an AFSIM environment. The RL methods used included Q-learning, asynchronous advantage actor-critic (A3C) (Mnih et al., 2016) and proximal policy optimisation (Schulman et al., 2017). They tried to address two problems. The first problem is a one-dimensional problem involving an agent piloting a fighter jet and a surface-to-air missile (SAM). The objective was to shoot down the SAM and avoid getting shot. The second problem is a two-dimensional problem involving a similar agent, a target fighter and a SAM. Its aim was to shoot down the target fighter without getting shot by the SAM. They reported that PPO was the only RL method that could solve the problems satisfactorily.

2.8.3 Approximate Dynamic Programming

Introduced by Bellman (1952), dynamic programming (DP) refers to a family of algorithms for computing optimal policies given a perfect model of the environment as a Markov decision process (MDP). The idea of DP is similar to that of reinforcement learning, which is to use value functions to organise and structure the search for good policies (Sutton and Barto, 2018). Classic DP algorithms are of little use to solve real-world problems because of their requirements for a perfect model, the exponential growth of the state space (the curse of dimensionality) and great computational expense (McGrew et al., 2010).

Approximate dynamic programming (ADP), a derivative of DP, can reduce the computation requirements and solve the curse of dimensionality by using a continuous function to approximately represent the future reward over the state space (Bertsekas and Tsitsiklis, 1996), similar to the way approximate solution methods in RL tackle the curse of dimensionality that tabular RL methods are unable to solve.

ADP has been proven useful in the field of air combat simulation, as demonstrated by McGrew et al. (2010), who presented a formulation of a level flight, fixed velocity, two-dimensional one-on-one air combat manoeuvring problem using an ADP approach to compute an efficient approximation of the optimal policy. They developed an air combat system with an eight-feature state consisting of the x and y positions, heading and bank angles of both the friendly (blue) and adversary (red) aircraft. The features of the state were used in the approximation architecture to estimate the value function. Their approach successfully produced policies that could be used by the blue aircraft to attain and maintain an offensive position behind the red aircraft. This paper also defines the McGrew score that is used as part of the reward function in the solutions proposed in this thesis.

2.8.4 Hierarchical Reinforcement Learning

Hierarchical RL is an RL technique that aims to combat the curse of dimensionality by dividing a complex task into smaller tasks. One of the advantages of hierarchical RL is transfer learning to use previously learned skills and subtasks in new tasks. A unique characteristic of hierarchical RL is that the system allows action sampling at every N timesteps instead of every timestep as in traditional RL, thus enabling the agent to complete a complex operation that takes more than a single timestep. Because of this multi-timestep requirement, most hierarchical RL approaches are modelled as a semi-Markov decision process (SMDP) instead of a Markov decision process (MDP). In an SMDP, the amount of time between one action selection and the next is a random variable (Dayan and Hinton, 1993; Barto and Mahadevan, 2003).

A number of techniques based on hierarchical RL have been applied to the air combat domain. Pope et al. (2021) combined a hierarchical architecture with maximum-entropy RL, integrating expert knowledge through reward shaping and supporting modularity of policies. Their approach is based on the metalearning shared hierarchies (MLSH) method (Frans et al., 2018), which attacks the problem by subdividing the task into subtasks. In MLSH the agent learns subpolicies for the subtasks as well as a master policy whose job is to select one of several subpolicies for the given state. Pope et al. (2021) used JSBSim (Berndt, 2004), an open source flight simulator, as their RL environment and their state space includes information about the agent's and the opponent's aircraft, aerodynamics, positions and altitudes. Their action space is continuous and maps to the inputs of the F-16's flight control system. The goal of the agent is to position the opponent within its weapons engagement zone (WEZ), which is defined as the locus of points lying within a spherical cone of two degree aperture extending out of the aircraft's nose. No missiles are shot, but attaining this geometry is called a gun snap and may earn the agent a reward, which is calculated as the amount of the potential damage that can be inflicted upon the opponent when the latter is within the WEZ. A non-zero reward is awarded when the opponent is within the agent's WEZ and their distance is between 500 and 3,000 feet within each other. A learning episode starts with both aircraft having full health and ends when either aircraft reach zero health or once the duration of the simulation reaches 300 seconds. Pope et al. (2021) divides the air combat task into these three subtasks:

- control zone (CZ). The CZ policy will try to attain a pursuit position behind the opponent that makes escape difficult for the latter.
- aggressive shooter (AS). The AS policy is encouraged to take aggressive shots from the side and head on.
- conservative shooter (CS). The CS policy is similar to the AS policy, but values gun snaps from near and far equally.

In addition to these subpolicies, the agent used by Pope et al. (2021) learns a master policy that determines which subpolicy to activate at each action selection. The policy selector is similar to the one introduced by Comanici and Precup (2010).

In another study, Sun et al. (2021) proposed the multiagent hierarchical policy gradient (MAHPG) algorithm for learning various strategies and transcending expert cognition by self-play, where a hierarchical decision network was adopted to deal with the complicated and hybrid actions and resembled a hierarchical decision-making ability of a human's. They reported that MAHPG outperformed the state-of-the-art air combat methods in terms of both defence and offence capabilities.

2.8.5 Multiobjective Reinforcement Learning

All the aforementioned studies are single-objective RL. The first published research on the use of multiobjective reinforcement learning (MORL) to solve a pilot training problem is by Källström and Heintz (2019). Using a simulator that is part of the Saab Gripen Flight Training Simulators, they devised a simple system whereby a learning agent must fly from an initial position to a target and avoid an enemy defence system located between the initial position and the target. They defined three objectives in their MORL project:

- Fly towards the target
- Reach the target as fast as possible
- Avoid the adversary's air defence system

Each of the three objectives carried a reward and consequently the environment returned a vector of three rewards at each timestep. They then scalarised the rewards using the weights [1, θ, 1 - θ] and ended up with a single scalar. Having had a single reward, they used a single-objective algorithm, in this case DQN, to learn a policy. By varying the value of θ, the agent learned different policies.

2.9 Discussion

Prior studies have largely focused on showing that the proposed approach could be applied to air combat simulation. These studies have proven that AI and RL in particular can be used to address the air combat manoeuvring problem. The successes shown by those studies are also justification for this thesis to use RL to address the problems. This thesis provides comparisons across a set of varying approaches, such as tabular vs. deep RL methods, different reward structures, and single- vs. multiobjective RL methods, to identify which combination provides the most effective means of developing air combat agents. For the tabular approach, this research compares Q(λ) and a policy-gradient method called actor-critic, using a novel tactics discovery problem. Additionally, the role of reward structures on the learning speed of actor-critic is investigated. This thesis also fills significant gaps in the literature, addressing two major areas that have been overlooked:

- Almost all the previous works on air-combat simulation focused on single-objective settings. One of the approaches conducted in this thesis uses multiobjective RL algorithms.
- Two principal categories of RL methods are tabular and approximate solution methods. The latter include deep RL. This thesis bridges the tabular and deep RL methods by proposing discrete-to-deep RL methods that are fast to learn as in tabular methods but that have generalisability of deep RL methods.

This work integrates with and builds upon the broader literature as follows:

- The reward function used in this thesis is adapted from a study by McGrew et al. (2010), who also addressed a two-dimensional one-on-one air combat manoeuvring problem. McGrew et al. (2010), however, used ADP instead of model-free RL approaches used in this thesis.
- This thesis uses a simulation technology called Ace Zero (ACE0), which was designed and developed by DSTG, Australia (Papasimeon and Benke, 2021). The simulator has been used by other scholars to address air combat and aerial manoeuvring problems. None of them used RL. Ramirez et al. (2017, 2018) used planning strategies, Masek et al. (2018) used finite state machines, Lam et al. (2019) used genetic algorithms and Hossam et al. (2020) used OptiGAN.
- Azak and Bayrak (2008) and Mouton et al. (2011) used Q-learning in their research in air-combat manoeuvring. However, their state spaces, 16x16 and 71x71 grids, respectively, are relatively small. By contrast, all the RL algorithms evaluated and implemented for this thesis use a continuous state space either as is or discretised into 14,000 states.
- Bilgin and Kadioglu-Urtis (2015) used Q(λ), which is one of the many algorithms used in this thesis. However, there are differences between the study and this thesis. First, their agents were only trained with 100 episodes each, as opposed to 20,000 to 200,000 episodes in this thesis. Second, while they tried to solve a similar air-combat manoeuvring problem, they focused on a multiagent environment that supported a pursuer and an evader. On the other hand, this thesis aims to train agents that try to pursue an opponent. Third, their study was concentrated on one single-objective tabular RL algorithm, whereas this thesis uses both single-objective and multiple-objective algorithms, implemented both as tabular and deep RL methods.
- Lee and Bang (2012) used Q-learning to help aircraft evade missiles moving in a horizontal plane while modelling the aircraft equations of motions as a simplified fourth-order point mass. The problem they tried to deal with is different from the problem this thesis attempts to address, which is to train agents to outmanoeuvre an opponent rather than to evade missiles.

- Teng et al. (2012) combined Q-learning and a self-organising neural network to develop an RL policy for outmanoeuvring an opponent in 1v1 air combat. Some of the approaches used in this thesis also use a combination of Q-learning and a neural network in deep RL methods called DQN (Mnih et al., 2013), DQN-wTN (Mnih et al., 2015) and double DQN (van Hasselt et al., 2016). The difference between the technique proposed in Teng et al. (2012) and DQN and its variants is the use of experience replay in the latter to help de-correlate samples. DQN and its variants were explained in Chapter 2.

- Fang et al. (2017a) combined Q-learning and a behaviour tree to train a fighter agent to decide whether to patrol, attack an enemy, turn around or flee. Q-learning with eligibility traces is one of the RL methods evaluated and tested in this thesis, however it is used without a behaviour tree.

- Kokolakis et al. (2020) use a variant of actor-critic to address the problem of tracking an actively evading target by employing a team of coordinating unmanned aerial vehicles while also learning the level of intelligence for appropriate countermeasures. This thesis also uses actor-critic, but addresses a different problem which is the one-on-one fight.

- Soyluoğlu (2021) used a reward function that is based on the McGrew score and compared DQN and PPO in addressing the air combat manoeuvring problem, a comparison that is also performed in the deep RL approach in this thesis. However, their study is limited to a small number of experiments and does not extend to other approaches used in this thesis, such as discrete-to-deep and MORL methods.

- Yang et al. (2019b) trained their agents against an aircraft that started from a random position and flew in a straight line, a similar strategy used in the learning of agents in this thesis. However, they used a continuous action space whereas the experiments in this thesis all use a discrete action space. They also used DDPG as their learning method, whereas this thesis uses various learning algorithms that do not include DDPG. In addition, they did not test their agents against opponents that had not been seen during training, whereas this thesis uses resulting policies in fights against never-before-seen engineered agents specifically programmed for air combat.

- A subset of this research is similar to Yang et al. (2019a), which used DQN to train RL agents to outmanoeuvre adversaries. Their research project utilised a relatively high numbers of actions and state variables, which made learning more difficult. They resorted to a form of curriculum training called basic-confrontation, which essentially mean they trained their agents using the easiest tasks first and gradually increased the difficulty of the tasks. To be sufficiently trained, their agents required about a million training steps. This thesis uses fewer actions and state variables and a different reward function. These three factors make the agents in this thesis project learn faster, only requiring a maximum of 200,000 learning steps.

- A number of scholars have done research on multiagent aerial manoeuvring. Källström and Heintz (2020) used multiagent DDPG to address two problems, aerial reconnaissance with sparse rewards and airstrike with adjustable risk taking. Zhang et al. (2022) investigate the target pursuit-evasion game in the obstacles environment and construct multiagent coronal bidirectionally coordinated with target prediction network with an extension to multiagent deep deterministic policy gradient (MADDPG) formulation. Yehoshua et al. (2021) develop a decentralised multiagent deep RL method to coordinate a group of drones to locate a set of static targets in an unknown area. Their learning algorithms are not used in this thesis and their problems are different from the air combat manoeuvring problem in this thesis.

- Zhang et al. (2020) evaluated and tested Q-learning, proximal policy optimisation (PPO) and other methods in an air combat environment. They tried to address a one-dimensional problem and a two-dimensional problem. Even though Q-learning and PPO are also used in this thesis, their problems are different from the problem formulated in this thesis.

- One aspect of this study is similar to optimistic initialisation, which is discussed by Sutton and Barto (2018) and investigated by Grześ and Kudenko (2009). The two previous studies used SARSA, whereas this thesis uses actor-critic.

- Thus far, Källström and Heintz (2019) is the only air-combat research that utilised MORL. Their study and this thesis differ in the problems to be dealt with. The aim of this thesis is to train RL agents that can outmanoeuvre an opponent, whereas their objective is to train an RL agent to fly to a target while avoiding a threat (the enemy's defence system). The experiments for this thesis involve two opposing aircraft whereas their experiments involved a single aircraft. In terms of the algorithms used, they only used DQN that was based on a single network, whereas this thesis evaluates and tests multiple MORL algorithms including multiobjective DQN with a shared network and multiobjective DQN with multiple networks. Another difference between this thesis and their research is that this thesis trains the agents using a simple scripted agent but tests with much more advanced agents, whereas their research trains and tests using the same target-threat layouts.

2.10 Summary

This chapter provides a literature review of RL in general and includes studies that applied RL techniques in the air combat manoeuvring field.

This thesis evaluates and implements two categories of methods as baselines, the tabular method and the deep RL method. Two tabular methods, a version of Q-learning that employs eligibility traces called Q(λ) (Watkins, 1989) and ACET, are chosen so that there are representations of both value-based and policy-gradient methods. Chapter 4 discusses these methods and their results in detail. Four deep RL methods are evaluated in this thesis and presented in Chapter 5: DQN, DQN-wTN, double DQN and PPO. DQN is chosen because it is one of the most important RL techniques today as it started the era of deep RL. DQN-wTN and double DQN are selected because they provide interesting comparison to the original DQN. PPO is included because it is one of the latest deep RL methods and has been reported to perform quite well (Schulman et al., 2017).

Some of the selected tabular and deep RL methods are extended to cater for multiple objectives to address the problem of air combat manoeuvring simulation. The results of these MORL techniques are presented in Chapter 7.

Chapter 3

Methodology

This chapter presents the methodology for this thesis, discussing elements common to all the experiments reported in the later chapters, including the simulation environment used to run the experiments, the reward structure, as well as the learning algorithms and testing methods. Individual variations from this general approach will be described in the respective chapters. The hardware and software used are explained in Appendix A and the source code for the thesis project, except parts that are restricted, can be found at https://github.com/budi-kurniawan/phd.

3.1 Experimental Setup

3.1.1 Simulation Environment

This research aims to train virtual agents (pilots) that can excel in simulated air combat manoeuvring. The simulator used for this thesis is Ace Zero, a Python-based software application developed by Defence Science and Technology, which is part of Australia's Department of Defence. Ace Zero supports three-dimensional space, but this thesis is based on two-dimensional space. The reason for using two-dimensional space is to restrict the size of the action space and complete testing of algorithms more quickly. Ace Zero has been used in other research projects including in Ramirez et al. (2018), Masek et al. (2018), Lam et al. (2019) and Kurniawan et al. (2019).

A software-based simulator like Ace Zero is essentially a game engine that executes in a loop moving virtual aircraft in units of time called *ticks*. In this thesis the simulated time interval dt for each tick is specified to be 0.5 second. If an aircraft starts at 2D coordinate (0, 0) with a velocity of 100 metres/second along the positive X axis, after a tick it will be at (50, 0) and after another tick its position will be at (100, 0). In 2D space, the location and orientation of a fighter at each tick can be written as a (x, y, ψ) coordinate, where x and y indicate a point in the Cartesian coordinate system and ψ the orientation of the aircraft relative to the positive X axis. For example, (0, 0, 90) refers to the origin and a heading that is parallel to the positive Y axis. This notation is used throughout this thesis.

Ace Zero supports any number of aircraft, but in this thesis only two fighters are used. Each fighter is controlled by a pilot or agent that pilots the aircraft by choosing, at every tick, a command or a set of commands that Ace Zero supports. By convention, Viper is the name of the agent being trained or tested and Cobra the name of its opponent. In diagrams, Viper is painted in blue and Cobra in red, and therefore in this thesis Viper is often referred to as Blue and Cobra as Red. Figure 3.1 shows a class diagram of Ace Zero. The ACEZero class, the entry point to the simulator, opens a scenario file and passes the information to a MultiAgentSimulation. The scenario file, which is a JSON file, contains hyperparameters such as the value of dt, the class names for Viper and Cobra and the class name for an umpire. The umpire specifies a condition that terminates the loop in the simulator. The agent contains a program or algorithm that controls the fighter. The simulation starts when the run method of the MultiAgentSimulation is invoked. This method starts a loop that calls the MultiAgentSimulation's tick method at each iteration and that is terminated after the condition specified by the umpire is met, usually after a specified number of ticks have been executed.

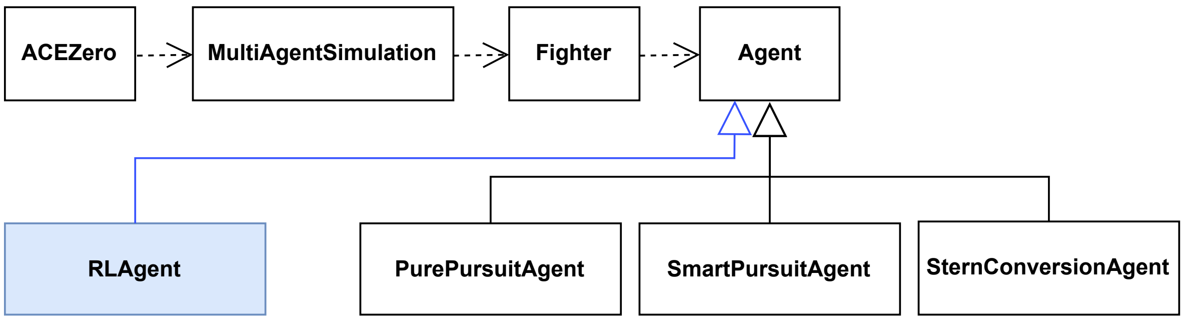


Figure 3.1: Ace Zero class diagram

The diagram also shows three agent implementations that are used in this thesis: Pure Pursuit, Smart Pursuit and Stern Conversion. These agents, all derived from the Agent class, are engineered agents as they are controlled programmatically to follow fixed, hand-crafted policies. These agents are explained in detail in Appendix B.

All the reinforcement learning (RL) agents used in this thesis are also subclasses of the Agent class. Unlike engineered agents, RL agents learn a policy and are not programmatically controlled.

In RL the term timestep is used to refer to a tick. Therefore, at every timestep, the fighters' tick methods are called and their state variables updated. The state variables that are used in this research are explained in the next section.

In order for Ace Zero to work in an RL context, a number of adjustments need to be made because Ace Zero is not the entry point to the RL system. Rather, Ace Zero is part of an environment that constantly communicates with the learning agent. As explained in Chapter 2, an RL program starts by telling the agent its initial state and then enters a loop. At each iteration the agent selects an action that is passed to the environment, which responds by telling the agent the reward it has received and the next state. Each iteration is a timestep, which corresponds to a tick in Ace Zero.

An interface was written to wrap Ace Zero in an RL environment called AceZeroEnvironment, which is similar to environments in OpenAI Gym framework (Brockman et al., 2016), a de-facto standard for developing and benchmarking RL algorithms. Among others, a Gym environment must provide reset and step methods. The reset method is called once when learning is to commence, and is used to reset states to their initial values. The step method is invoked at every timestep. The reset method of AceZeroEnvironment creates an instance of ACEZero, passing a scenario file, but does not call the MultiAgentSimulation's run method. The step method of AceZeroEnvironment calls the tick method of the MultiAgentSimulation. The reason the MultiAgentSimulation's run method is not used in this system is so that simulation will be controlled by the RL program and not by the ACEZero object. Thanks to this interface, Ace Zero can be plugged into any code that has been written to run a Gym application. This interface can be easily reused in any future research extending this thesis.

3.1.2 State Space Representation

This thesis uses a continuous state space that consists of four state variables.

- R, the distance between the two aircraft
- AA, the attack angle (from -180 to 180)
- ATA, the antenna train angle (from -180 to 180)
- $delta\_v$, the speed difference between the two aircraft (-125 to 125)

3.1.3 Action Space Representation

Table 3.1 lists all the commands supported by Ace Zero. At any timestep, an agent can select up to one command from each category. In other words, it can issue up to three commands at a time.

Category	Command	Description
A	SetFlyLevelCmd	Sets the desired pitch (theta_c) and roll (phi_c) angles to 0.0
A	SetPitchAngleCmd	Sets the desired pitch (theta_c) angle. Causes the aircraft to ascend or descend
A	SetAltitudeCmd	Sets the desired altitude (z_c) with a desired pitch (theta_c) angle
B	SetSpeedCmd	Sets the desired aircraft speed (v_c)
C	SetHeadingCmd	Sets the desired aircraft heading (psi_c)
C	SetWaypointCmd	Sets the desired waypoint to fly to (x_c, y_c)

Table 3.1: Ace Zero commands

The action space used in this thesis uses a subset of Ace Zero commands and is made up of these five actions that enable navigation on a two-dimensional plane:

- 10° turn to the left
- 10° turn to the right
- increase speed by 10%
- reduce speed by 10%
- do nothing

3.1.4 Reward Structure

During learning, the step method of AceZeroEnvironment is invoked at every timestep, passing an action selected by the learning agent. Calling the step method will invoke the MultiAgentSimulation's tick method, which in turn calls the fighters' tick methods, updating the states of both aircraft. Upon selecting an action at each timestep, an RL agent receives a reward from the environment to indicate how good the selected action is. In all of the experiments in this thesis, instead of defining winning or losing regions, the McGrew score (McGrew et al., 2010) is adopted to reward the agent. The McGrew score is commonly used in the literature and explained below.

To outmanoeuvre the opponent, the agent's aircraft needs to be in specific relative geometry to the opponent. Figure 3.2 shows such geometry that consists of the following components.

- the range (distance) between the two aircraft,
- the attack angle (AA), and
- the antenna train angle (ATA)

As mentioned in the previous section, by convention blue is used to depict the aircraft controlled by the subject agent and red to represent the opposing aircraft. In this thesis the aircraft are simply referred to as Blue and Red, respectively. The angles in Figure 3.2 are shown from Blue's point of view.

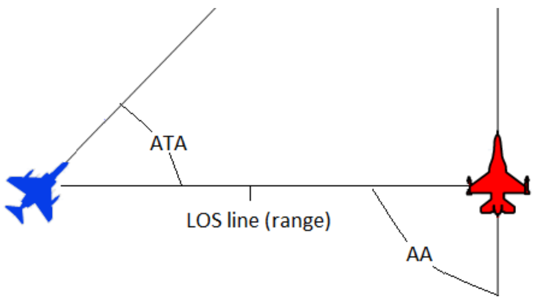


Figure 3.2: Aircraft relative geometry

The aircraft centres of mass are connected by the line of sight (LOS) line, which is also used to calculate the range between the two aircraft. The attack angle (AA) is the angle between the LOS line and the tail of the red aircraft. The antenna train angle (ATA) is the angle between the nose of the blue aircraft and the LOS line. Both the AA and ATA help the pilot make manoeuvring decisions. The value of the AA and ATA is within $0° \pm 180°$. By convention, angles to the right side of the aircraft are considered positive and angles to the left negative (McGrew et al., 2010).

Quantitatively, the McGrew score (McGrew et al., 2010) can be used to measure how favourable an agent's position is relative to the opponent. The higher the McGrew score, the better. The McGrew score (S_M) consists of two components, McGrew angular score (A_M) and McGrew range score (R_M), which together incorporate the AA, ATA and range.

$$S_M = A_M R_M \tag{3.1}$$

The McGrew angular score is defined as follows.

$$A_M = \frac{1}{2}\left[\left(1 - \frac{AA}{180°}\right) + \left(1 - \frac{ATA}{180°}\right)\right] \tag{3.2}$$

where AA and ATA are in degrees. The maximum possible value of $A_M = 1$ is achieved when $AA = ATA = 0$, when the aircraft is directly behind the tail of the opponent, facing towards it.

The McGrew range score is defined as this.

$$R_M = \exp\left[-\frac{|R - R_d|}{k \times 180°}\right] \tag{3.3}$$

where R is the current range of the two aircraft and R_d the desired range. R_d, the midpoint between the minimum gun range (500 feet = ≈153m) and the maximum gun range (3,000 feet = 914m), is determined to be 380m ((914-153)/2) (Shaw, 1985). The value of k, the hyperparameter scaling factor, determines the width of the function peak around R_d. The larger the value of k, the bigger the spread. A small value of k dictates that a high McGrew range score can only be achieved if the two aircraft are very close to the desired range. By default Ace Zero sets k to 5 and this is the value used throughout this thesis.

The McGrew score ranges from 0.0 to 1.0 (inclusive). Figure 3.3 shows the McGrew scores of Blue and Red pairs in various layouts. In all of them, Blue and Red are spaced 380 metres from each other. Figure 3.3A shows a position where Blue is trailing Red and both are flying in the same direction ($AA = ATA = 0°$). Figure 3.3B shows Blue and Red at 45°($AA = 0$, $ATA = 45°$) and Figure 3.3C shows them at 90°($AA = 0°$, $ATA = 90°$). Figure 3.3D shows Red is following Blue ($AA = ATA = 180°$). The McGrew scores for the positions are 1.00, 0.82, 0.65 and 0.00, respectively.

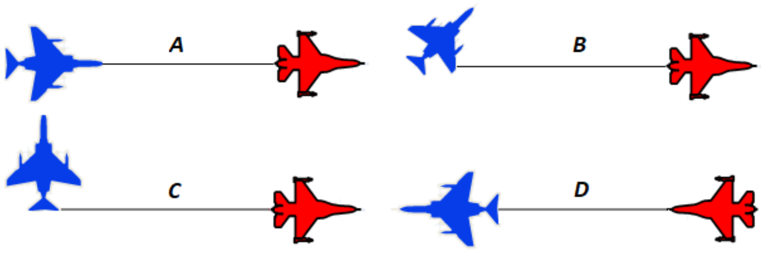


Figure 3.3: McGrew scores for R=380m (A=1.00, B=0.82, C=0.65, D=0.00)

3.2 Learning Algorithms

Four categories of RL methods are used in this thesis to address the air combat manoeuvring problem.

- Tabular approach, consisting of two tabular methods Q(λ) and actor-critic with eligibility traces (ACET). This is discussed in Chapter 4.
- Deep approach using four deep RL methods: deep Q-network (DQN), DQN with a target network, double DQN and proximal policy optimisation. This is discussed in Chapter 6.
- Discrete-to-deep approach, using two novel algorithms, discrete-to-deep supervised policy learning and discrete-to-deep supervised Q-value learning. This is discussed in Chapter 6.
- Multiobjective approach using four multiobjective RL methods: multiobjective Q(λ), multiobjective ACET, multiobjective DQN with a shared network and multiobjective DQN with multiple networks. This is discussed in Chapter 7.

The tabular and deep approaches are used as baselines, whereas the discrete-to-deep and multiobjective approaches are thesis contributions.

3.3 Experimental Design

One of the factors explored in this thesis is the impact which the starting positions used during training have on the learning of the agent. Six initial positions were identified. They are five fixed initial positions with small random variations (code-named 001, 002, 003, 004 and 005) and a fully random initial position. Figure 3.4 shows the fixed initial positions. In all the fixed initial positions, Blue always starts at (0, 0, 0), that is at the origin with its heading parallel to the positive X axis, and Red's location and heading vary according to Table 3.2.

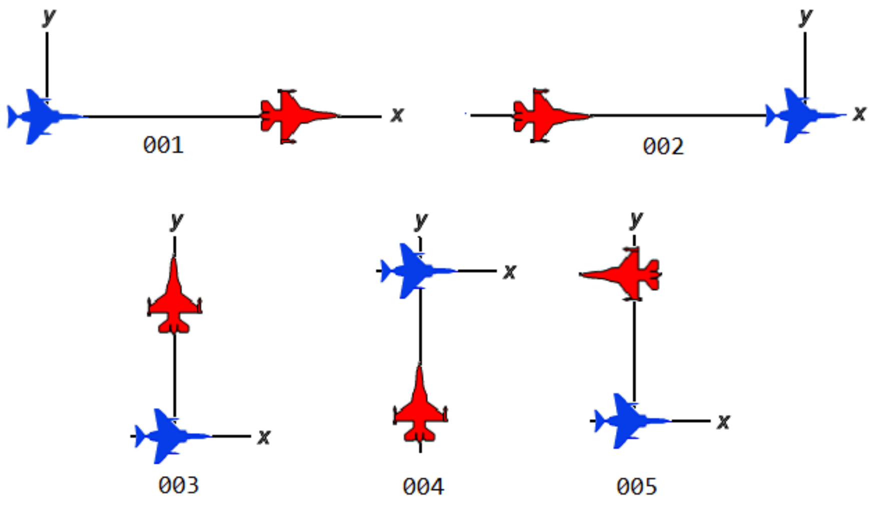


Figure 3.4: Initial positions 001, 002, 003, 004 and 005 used for learning

Name	Blue	Red
001	(0, 0, 0)	(1500, 0, 0)
002	(0, 0, 0)	(-1500, 0, 0)
003	(0, 0, 0)	(0, 1500, 90)
004	(0, 0, 0)	(0, -1500, 90)
005	(0, 0, 0)	(0, 1500, 180)

Table 3.2: Locations and headings for fixed initial positions used for learning

After the two aircraft are placed in their initial positions, but before learning starts, Blue's location and heading are varied by first generating a random number r that falls within [-0.5..0.5] and using it to alter x, y and ψ.

$$x = x + r * 10$$

$$y = y + r * 2$$

$$\psi = \psi + r * 2$$

For the fully random initial position, Blue is placed at (0, 0, 0) and x, y and ψ are randomly generated for Red where $-1600 \leq x \leq 1600$, $-1600 \leq y \leq 1600$ and $-180 < \psi \leq 180$.

3.4 Testing, Metrics and Evaluation

The RL agents in this thesis are trained against a simple agent that flies in a straight line at a constant speed and tested against more complex and sophisticated baseline agents. This is to measure whether the RL algorithms are able to generalise to situations that were not seen during learning. To measure the performance of the RL agents in this study, two kinds of tests are conducted, indirect and direct performance comparisons.

3.4.1 Indirect Performance Comparison

In indirect performance comparison, the performance of the agents is compared with the performance of engineered agents facing the same opponent. The three engineered agents that come with Ace Zero—the Pure Pursuit agent, the Smart Pursuit agent and the Stern Conversion agent—are used as baseline agents. However, because the Smart Pursuit agent is the best performing engineered agent, only this engineered agent is used in indirect performance comparison. How the performance of the baseline agents is measured is discussed in Appendix B.

A test suite called Basic Test Suite is used in indirect performance comparison. The test suite comprises 144 test cases. In each test case Blue follows an RL policy and Red flies in a straight line at a constant speed of 125 metres/second. Each test case differs in the starting flying directions of both aircraft. In the first case, Blue's initial flying direction (psi) is the same as Red's. In the next test case, Red's psi is incremented by 30°. Increasing Red's psi twelve times results in twelve different test cases and brings Red to the same psi as the first test case, upon which Blue's psi is incremented by 30°. Repeating the sequence twelve times gives 12 * 12 = 144 test cases.

Park et al. (2016) and Burgin and Sidor (1988) categorise relative positions in 2D air combat simulation into four situation classes: offensive, defensive, neutral and head-on. These situation classes are shown in Figure 3.5. The classification assumes perfect symmetry, i.e. Blue should take the same action when Red is at 10° and at -10° as long as both aircraft are at the same distance R at either angle.

Figures 3.6 and 3.7 show the 144 initial positions used in some of the tests conducted for this thesis. Each initial position is labelled with a number and situation class: o for offensive, d for defensive, n for neutral and h for head-on. For example, Test 1 where Blue is pursuing Red is categorised as offensive from Blue's perspective.

An agent that selects actions randomly, a Random agent, is introduced to compare how well an RL agent fares. The Random agent is instantiated by passing to it the number of actions N and a random seed, which it uses to create a pseudorandom number generator. At every timestep the Random agent generates an integer between 0 and $N-1$ (inclusive) to randomly select an action. In this thesis every experiment is run ten times and when the Random agent is involved, it too is run in ten trials using a different random seed for each trial.

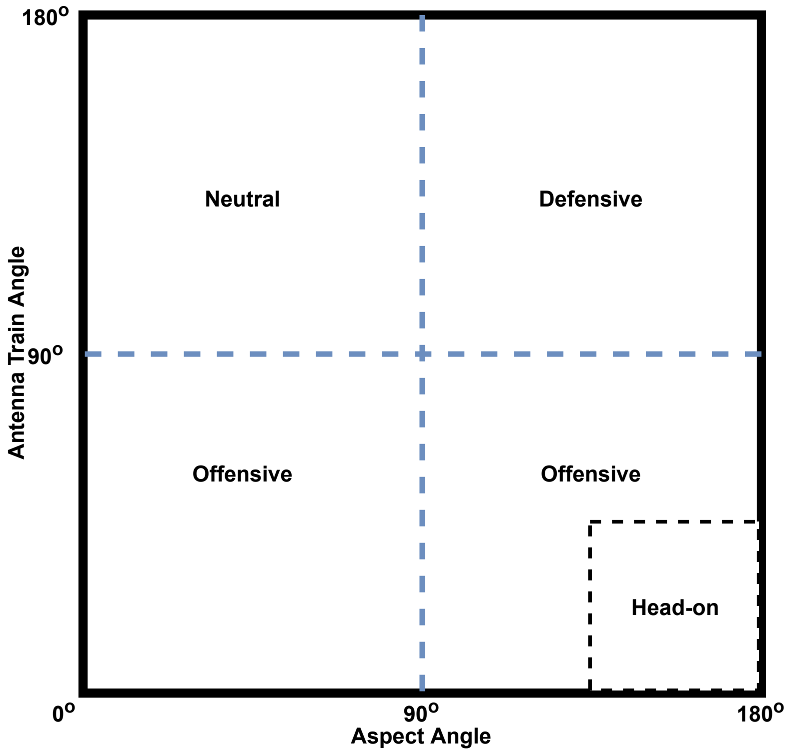


Figure 3.5: Situation classification of Blue relative to Red

3.4.2 Direct Performance Comparison

In direct performance comparison, RL agents are compared against restricted engineered agents, full-blown engineered agents and against each other. Direct performance comparison results are presented in Chapter 8.

Comparing against Restricted Engineered Agents

The engineered agents have a continuous action space and are physically superior to the RL agents, which have only five actions in their discrete action space so that the RL methods used in this thesis could be applied within a reasonable time-frame given the computation resources available. For example, an RL agent can often turn by 10° to the left and to the right. By contrast, the engineered agents can turn by the 0 to 90°. Also, an RL agent is restricted to a 10% speed change where as the engineered agents have no such a limit.

To make fairer comparison, clones of engineered agents are created that impose the same physical limits as the RL agents. The clones are called restricted engineered agents.

Comparing against Full-Blown Engineered Agents

In this benchmarking, the performance of the RL agents is compared to that of the physically superior engineered agents.

3.4.3 Data Collection

For each of the four RL approaches implemented and tested in this thesis, the following data are collected:

- learning time,
- learning curve (measured as the average reward per episode),
- performance against the baseline (measured by comparing the total McGrew scores of the agent and that of the baseline)

Tables 3.3 to 3.6 list all the agents used in this thesis. For each policy learning, a learning curve is generated that shows if an agent has successfully learned or failed to learn. The reason why tabular method agents learn for 200,000 episodes and the deep RL agents learn for 20,000 episodes is because the computational cost per episode of a deep RL method is about ten times the cost of a tabular method. Therefore, all the agents, except the discrete-to-deep methods, spend similar wall-clock times. In Table 3.5 the values in the Number of Episodes column refer to the number of learning episodes conducted in the reinforcement phase to obtain off-policy data. This will be explained further in Chapter 7.

Method	Initial Position	Number of Episodes	Agent Name
Q(λ)	001	200,000	ql-001
Q(λ)	002	200,000	ql-002
Q(λ)	003	200,000	ql-003
Q(λ)	004	200,000	ql-004
Q(λ)	005	200,000	ql-005
Q(λ)	random	200,000	ql-random
ACET	001	200,000	ac-001
ACET	002	200,000	ac-002
ACET	003	200,000	ac-003
ACET	004	200,000	ac-004
ACET	005	200,000	ac-005
ACET	random	200,000	ac-random

Table 3.3: Tabular method agents

Method	Initial Position	Number of Episodes	Agent Name
DQN	001	20,000	dqn-001
DQN	002	20,000	dqn-002
DQN	003	20,000	dqn-003
DQN	004	20,000	dqn-004
DQN	005	20,000	dqn-005
DQN	random	20,000	dqn-random
DQN-wTN	001	20,000	dqnwtn-001
DQN-wTN	002	20,000	dqnwtn-002
DQN-wTN	003	20,000	dqnwtn-003
DQN-wTN	004	20,000	dqnwtn-004
DQN-wTN	005	20,000	dqnwtn-005
DQN-wTN	random	20,000	dqnwtn-random
Double DQN	001	20,000	ddqn-001
Double DQN	002	20,000	ddqn-002
Double DQN	003	20,000	ddqn-003
Double DQN	004	20,000	ddqn-004
Double DQN	005	20,000	ddqn-005
Double DQN	random	20,000	ddqn-random
PPO	001	20,000	ppo-001
PPO	002	20,000	ppo-002
PPO	003	20,000	ppo-003
PPO	004	20,000	ppo-004
PPO	005	20,000	ppo-005
PPO	random	20,000	ppo-random

Table 3.4: Deep RL agents

Method	Initial Position	Number of Episodes	Agent Name
D2D-SPL	001	20,000	d2dspl-001
D2D-SQL	001	20,000	d2dsql-001
D2D-SPL	random	50,000	d2dspl-random
D2D-SQL	random	50,000	d2dsql-random

Table 3.5: Discrete-to-deep agents

Reward Builder	Method	Number of Episodes	Agent Name
RB-001	MO Q(λ)	200,000	mo-ql-random-rb001
RB-001	MO ACET	200,000	mo-ac-random-rb001
RB-001	MO DQN (shared)	20,000	mo-dqn-shared-random-rb001
RB-001	MO DQN (multinetworks)	20,000	mo-dqn-mnn-random-rb001
RB-002	MO Q(λ)	200,000	mo-ql-random-rb002
RB-002	MO ACET	200,000	mo-ac-random-rb002
RB-002	MO DQN (shared)	20,000	mo-dqn-shared-random-rb002
RB-002	MO DQN (multinetworks)	20,000	mo-dqn-mnn-random-rb002
RB-003	MO Q(λ)	200,000	mo-ql-random-rb003
RB-003	MO ACET	200,000	mo-ac-random-rb003
RB-003	MO DQN (shared)	20,000	mo-dqn-shared-random-rb003
RB-003	MO DQN (multinetworks)	20,000	mo-dqn-mnn-random-rb003
RB-004	MO ACET	200,000	mo-ac-random-rb004
RB-005	MO ACET	200,000	mo-ac-random-rb005
RB-006	MO ACET	200,000	mo-ac-random-rb006
RB-007	MO ACET	200,000	mo-ac-random-rb007

Table 3.6: Multiobjective RL agents, all learning from random initial positions

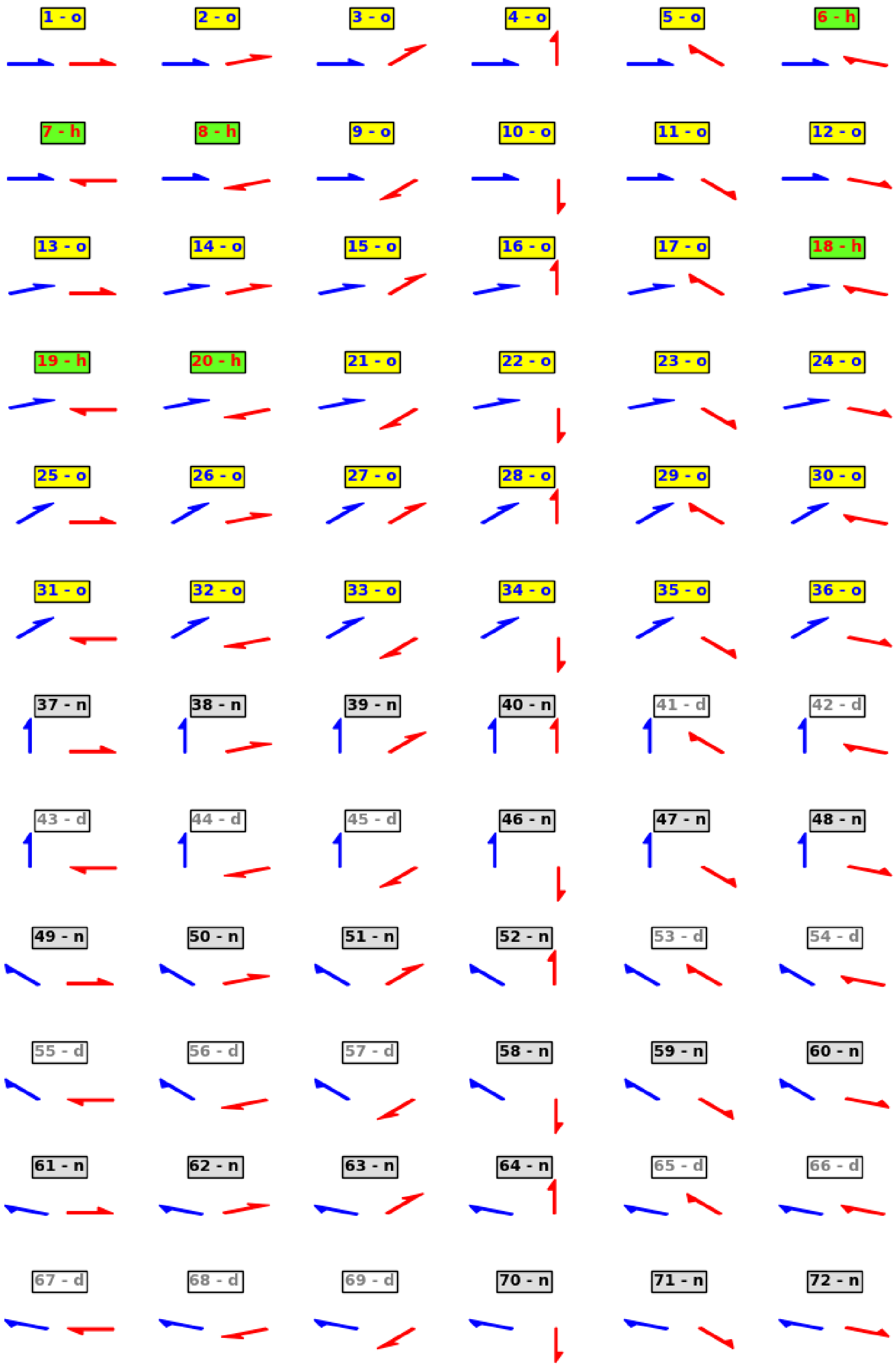


Figure 3.6: Initial positions and situations 1 to 72, the suffixes d, h, n and o combined with label colours white, green, grey and yellow indicate situation classes defensive, head-on, neutral and offensive, respectively

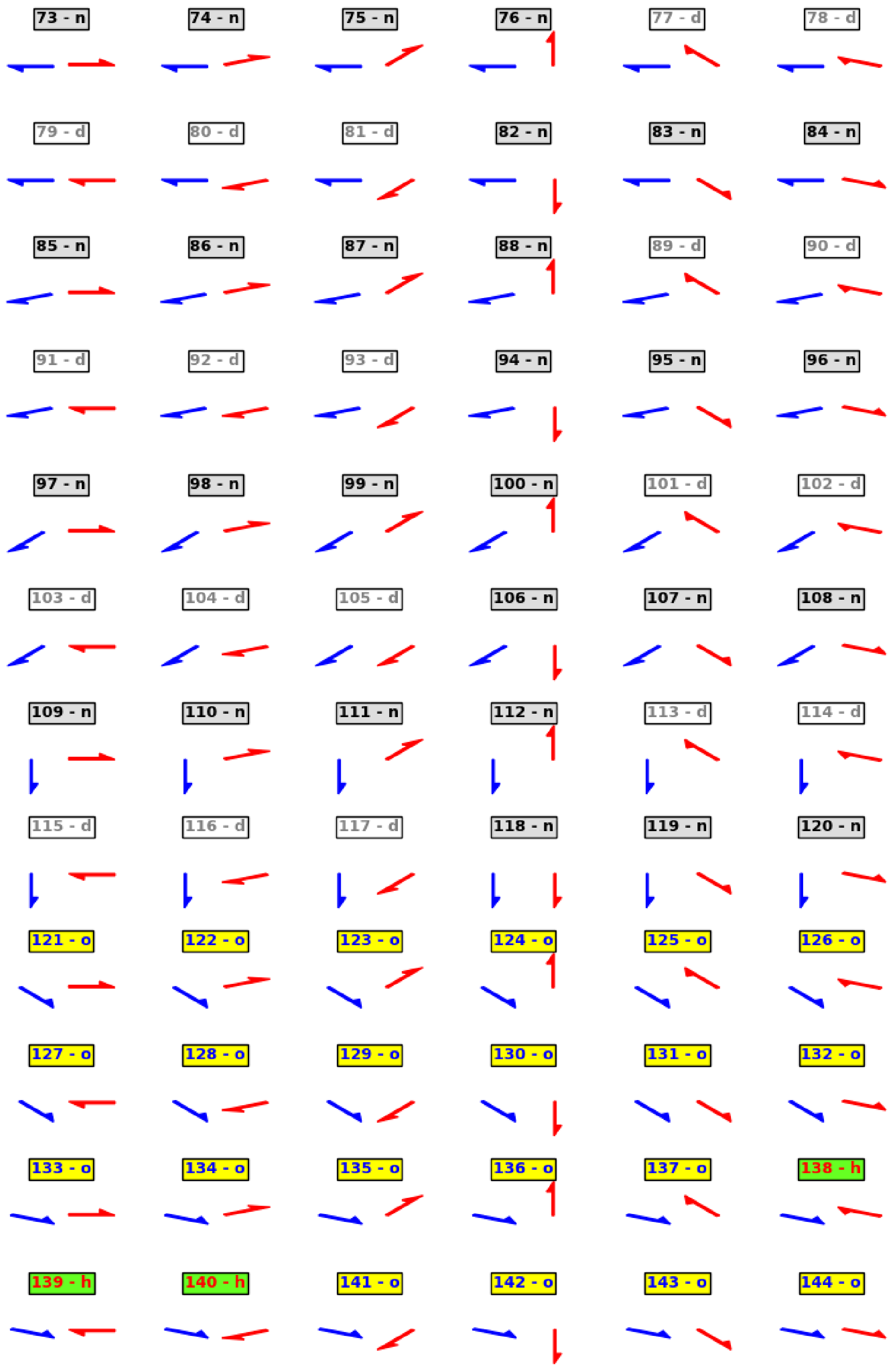


Figure 3.7: Initial positions and situations 73 to 144, the suffixes d, h, n and o combined with label colours white, green, grey and yellow indicate situations classes defensive, head-on, neutral and offensive, respectively

3.5 Statistical Analysis

Because deep RL requires a vast amount of computing power, only ten results are available for each RL method evaluated, making it hard to determine if the results are normally distributed. Because of this, the ten results are resampled using the bootstrap method (Efron, 1979) to generate 10,000 resamples and calculate a 95% confidence interval, which is a range of values that provide 95% confidence that the interval contains the true mean of the population. Confidence intervals can also be used for statistical analysis. The confidence intervals of two RL methods can be compared to conclude whether or not the performance difference between the two RL methods is statistically significant. If two confidence intervals overlap, the performance difference between two RL methods is not statistically significant. If they do not overlap, the difference is statistically significant. In this thesis, bootstrapping is done using SciPy's bootstrap function (Virtanen et al., 2020).

3.6 Other Considerations

3.6.1 Reproducibility

To make sure a test can be reproduced, every agent must use its own pseudorandom number generator. Reproducing a test result is then a matter of seeding it with the same seed previously used.

3.6.2 Reinforcement Learning Framework

A new RL framework called Spy RL was developed for this thesis. This framework is presented in Appendix C.

3.7 Summary

This chapter presents the methodology for this thesis, including the elements common to all the experiments, the simulation environment, the reward structure, the learning algorithms and the testing methods. The next chapter, Chapter 4, discusses the tabular approach, the first of the four approaches used in this thesis.

Chapter 4

Tabular Baseline Methods

Tabular or table-based reinforcement learning (RL) methods are easy to implement and can solve sequential decision problems with discrete or continuous state spaces. In this chapter two tabular methods, Watkins's Q(λ) (Watkins, 1989) and actor-critic with eligibility traces (ACET) (Barto et al., 1983; Sutton and Barto, 2018), are used to address the air combat manoeuvring problem. The two algorithms are explained at the beginning of this chapter, followed by a discussion of the setup used for the experiments. The results, which are given in Section 4.3, reveal how the two methods perform. The results are used as baselines for discrete-to-deep and multiobjective methods in the upcoming chapters.

4.1 Overview of the Methods Used

This chapter compares the performance of two tabular RL methods, Q(λ) and ACET, in addressing the air combat manoeuvring problem. A common theme underlying Q(λ) and ACET is eligibility traces. Q(λ) is Q-learning with eligibility traces and ACET is actor-critic with eligibility traces. Eligibility traces can speed up learning by reducing the severity of the credit assignment problem (Singh and Sutton, 1996). This is the problem of not knowing which past action was responsible for a reward or punishment received in a future step due to the fact that a non-zero reward is often sparse and delayed (an action taken at the current timestep is only rewarded after a multitude of future actions). In addition, not just a single action but rather a combination of actions carried out in the appropriate circumstances may be responsible for a reward. Eligibility traces can be used in conjunction with any tabular method and are harder to combine with an approximate solution method. Nonetheless, studies have been done to research the effects of eligibility traces in approximate solution methods and even deep RL (van Seijen and Sutton, 2014; Harb and Precup, 2017; Daley and Amato, 2019).

When eligibility traces are employed, all previously visited states or previously taken state-action pairs within the same episode are memorised and when the agent reaches a state that grants a non-zero reward, all the previous states or state-action pairs will also get a fraction of the reward. The further in time a state or state-action is from the reward-generating event, the smaller the reward. Generally, for each step in time, the reward is attenuated by a trace-decay parameter $\lambda \in [0,1]$.

4.1.1 Q(λ)

Q(λ) is an extension to Q-learning (Watkins, 1989), a simple algorithm that until now is one of the most popular RL methods, if not the most popular. Watkins's Q(λ) (Watkins, 1989) was the first attempt to combine Q-learning with eligibility traces, even though it is not the only one. For instance, there is another version of Q(λ) called Peng's Q(λ) (Peng and Williams, 1994). The name Q(λ) is consistent with TD(λ), which are TD(0) techniques that incorporate eligibility traces (Sutton, 1984). For the rest of this chapter, the term Q(λ) refers to Watkins's Q(λ).

Algorithm 5 shows a rewrite of the original Q(λ). The rewrite improves the computational efficiency of the original algorithm by eliminating the need for a' (the action taken at the next step) and speeding up learning by remembering all the states that have been visited in an episode and by only updating traces for those states instead of all the traces.

A data structure $e(s,a)$ is used to note all previously taken state-actions. For each timestep, the e value for the current state-action is incremented by one. For all other state-actions, they are either reduced by λ if the agent took a greedy action, or otherwise reset to zero.

Depending on how the $e(s,a)$ value for the current state-action is updated, the traces are either called accumulating traces or replacing traces. If accumulating traces are used, $e(s,a)$ is incremented by one every time a non-zero reward is granted. If $e(s,a)$ is set to 1 instead, the traces are called replacing traces. Accumulating traces are due to Sutton (1984) and replacing traces were introduced by Singh and Sutton (1996).

In Watkins's Q(λ), all traces are reset whenever an exploratory action is taken. Also, in this thesis replacing traces are used and the following are the values of the parameters: ϵ=0.2, α=0.7, γ=0.95, λ=0.9.

4.1.2 Actor-Critic with Eligibility Traces (ACET)

Actor-critic algorithms are some of the earliest RL methods (Witten, 1977; Sutton, 1984). First introduced by Barto et al. (1983), the actor-critic algorithm is a policy gradient method that also uses a value function to learn the policy parameter. Actor-critic learns the policy parameter based on the gradient of some scalar performance measure with respect to the policy parameter. Actor-critic can be implemented as a tabular method or an approximate solution method. The variant of actor-critic used in this chapter is a table-based method called actor-critic with eligibility traces (ACET) (Sutton and Barto, 2018).

Algorithm 5: Enhanced Q(λ) with replacing or accumulating traces

```
   Parameter(s): step size $\alpha \in (0,1]$, discount-rate $\gamma \in (0,1]$, decay rate $\lambda \in (0,1]$, small $\epsilon > 0$
   Result: Q
1  Init. $Q(s,a)$ arbitrarily and $e(s,a) = 0$ for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, $Q(terminal,.) = 0$;
2  for each episode do
3      Initialise $S$;
4      $Visited = []$;
5      for each step of episode or until S is terminal do
6          Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\epsilon$-greedy);
7          Take action $A$, observe $R$, $S'$;
8          $\delta \leftarrow R + \gamma \max_a Q(S',a) - Q(S,A)$;
9          $Q(S,A) \leftarrow Q(S,A) + \alpha\delta$;
10         $greedy = Q(S,A) == \max_a Q(S,a)$;
11         if $greedy$ then
12             for all $(s,a) \in Visited$ do
13                 $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$;
14                 $e(s,a) \leftarrow \gamma\lambda e(s,a)$;
15             end
16         else
17             for $(s,a)$ in $Visited$ do
18                 $e(s,a) \leftarrow 0$;
19             end
20             Delete all elements in $Visited$;
21         end
22         $e(S,A) \leftarrow 1$ (replacing traces) or $e(S,A) \leftarrow e(S,A) + 1$ (accumulating traces);
23         If $(S,A) \notin Visited$, add $(S,A)$ to $Visited$;
24         $S \leftarrow S'$;
25     end
26 end
```

Algorithm 6 shows the pseudocode of ACET. It is an extension of the actor-critic algorithm discussed in Chapter 2.

Algorithm 6: Actor-critic with eligibility traces for episodic problems (Sutton and Barto, 2018)

```
   Input: a differentiable policy parameterization $\pi(a|s,\boldsymbol{\theta})$
   Input: a differentiable state-value function parameterization $\hat{v}(s,\mathrm{w})$
   Parameter(s): step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathrm{w}} > 0$
   Parameter(s): trace decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathrm{w}} \in [0,1]$
1  Initialise policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and state-value weights $\mathrm{w} \in \mathbb{R}^{d}$ (e.g, to 0) ;
2  for each episode do
3      Initialise $S$ (first state of episode);
4      $\mathrm{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ ($d'$-component eligibility trace vector) ;
5      $\mathrm{z}^{\mathrm{w}} \leftarrow \mathbf{0}$ ($d$-component eligibility trace vector) ;
6      $I \leftarrow 1$ ;
7      for each step of episode or until $S$ is terminal do
8          $A \sim \pi(.|S,\boldsymbol{\theta})$ ;
9          Take action $A$, observe $S'$, $R$;
10         $\delta \leftarrow R + \gamma\hat{v}(S',\mathrm{w}) - \hat{v}(S,\mathrm{w})$      (if $S'$ is terminal, $\hat{v}(S',\mathrm{w}) \doteq 0$) ;
11         $\mathrm{z}^{\mathrm{w}} \leftarrow \gamma\lambda^{\mathrm{w}}\mathrm{z}^{\mathrm{w}} + \nabla\hat{v}(S,\mathrm{w})$ ;
12         $\mathrm{z}^{\boldsymbol{\theta}} \leftarrow \gamma\lambda^{\boldsymbol{\theta}}\mathrm{z}^{\boldsymbol{\theta}} + I\nabla\ln\pi(A|S,\boldsymbol{\theta})$ ;
13         $\mathrm{w} \leftarrow \mathrm{w} + \alpha^{\mathrm{w}}\delta\mathrm{z}^{\mathrm{w}}$ ;
14         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\delta\mathrm{z}^{\boldsymbol{\theta}}$;
15         $I \leftarrow \gamma I$;
16         $S \leftarrow S'$;
17     end
18 end
```

4.2 Experimental Setup

Experimental setup for RL includes two or three components that should be chosen carefully as they can have big impacts on the success of learning and agent performance. The first component is state discretisation, which converts continuous states to discrete states and is only required when a tabular method is used to solve a problem with a continuous state space. The second component is the action space, which contains the actions that an agent can select. The third component is the reward function. State discretisation is a module in the agent and both the action space and the reward function are part of the RL environment. The three components are explained in the following subsections.

4.2.1 State Discretisation

Both the Q(λ) and ACET methods implemented for this research project are tabular methods. As such, when trying to solve a continuous-state problem such as the air combat manoeuvring problem, observations need to be discretised by the agent. Several discretisation strategies were tested for Ace Zero and the one that returns the best results is the strategy that discretises the continuous space into 14,000 discrete states. The state discretisation strategy is based on these

Ace Zero state variables that are discussed in Chapter 3:

- R, the distance between the two aircraft
- AA, the attack angle (from -180 to 180)
- ATA, the antenna train angle (from -180 to 180)
- $delta\_v$, the speed difference between the two aircraft (-125 to 125)

The value of R falls into one of fourteen classes 0..13. Its value is calculated from the distance:

$$R = \min(13, \text{int}(distance \div 200))$$

The min function effectively groups together distances over 2,600 metres in class 13. There are two reasons for applying the min function to the state discretisation. The first reason is to limit the number of discretised states. The second reason is due to the fact that once the distance between the two aircraft is relatively large, the only actions that will yield positive rewards are those that shorten the distance and therefore there is no point in splitting that area into more discrete states as those additional discrete states would have the same state-action values.

The values of AA and ATA are grouped into ten regions according to the rules in Table 4.1. The value of $delta\_v$ is grouped into ten regions according to the rules in Table 4.2.

Finally, a discrete state ds is obtained using this formula:

$$ds = 1000\,R + 100\,delta\_v \text{ region} + 10\,AA \text{ region} + ATA \text{ region} \tag{4.1}$$

Angle (x)	Region
-10 $\leq x \leq$ 10	0
10 $< x \leq$ 30	1
-30 $\leq x <$ -10	2
30 $< x \leq$ 50	3
-50 $\leq x <$ -30	4
50 $< x \leq$ 70	5
-70 $\leq x <$ -50	6
70 $< x \leq$ 90	7
-90 $\leq x <$ -70	8
$x <$ -90 or $x >$ 90	9

Table 4.1: AA and ATA discretisation rules. All figures are in degrees

delta_v (x)	Region
$x > 20$	0
$15 < x \leq 20$	1
$10 < x \leq 15$	2
$5 < x \leq 10$	3
$0 < x \leq 5$	4
$-5 < x \leq 0$	5
$-10 < x \leq -5$	6
$-15 < x \leq -10$	7
$-20 < x \leq -15$	8
$x \leq -20$	9

Table 4.2: $delta\_v$ discretisation rules. All figures in metres/second.

4.2.2 The Action Space

The action space used in this chapter is the same as that discussed in Section 3.1.3. It is made up of these five actions:

- 10° turn to the left
- 10° turn to the right
- increase speed by 10%
- reduce speed by 10%
- do nothing

4.2.3 The Reward Function

As the reward provides the only feedback to the RL agent, it is well known that the specification of the reward will be one of the major factors influencing the learning behaviour of the agent (Sutton and Barto, 2018). Therefore, as part of this thesis research, a preliminary study was carried out to identify the most suitable reward structure for the air combat manoeuvring problem (Kurniawan et al., 2019). In particular the study investigated the impact of the range of the reward values on agent learning, as it has previously been established that learning speed and exploration can be impacted by whether the initial action values are optimistic or pessimistic with respect to the true returns available to the agent (Sutton and Barto, 2018). The agents in the study use the five actions discussed in Section 4.2.2 but discretise the continuous state space into 1,400 discrete states instead of 14,000. The discretisation strategy is similar to the one discussed in Section 4.2.1, but without $delta\_v$. To be precise, the study compares reward functions A, B and C:

A: Reward = McGrewScore

B: Reward = McGrewScore - 0.5

C: Reward = McGrewScore - 1.0

Three ACET agents learn a policy using reward functions A, B and C, respectively for 20,000 episodes. During a learning session, the agent (Blue aircraft) always starts from (0, 0, 0°) and the opponent (Red aircraft) from (1500, 300, 50°) and both have the same initial speed of 125m/s. The initial positions, headings and speeds of both aircraft are the same for all episodes and each of the three agents is run independently against an identical opponent.

The result shows that the reward function plays a role in learning speed. The three agents aim to get the highest rewards possible but do so at different learning speeds. The experiments are run ten times using different random seeds and Tables 4.3, 4.4 and 4.5 show the average rewards after 1,000, 2,000 and 5,000 episodes, respectively. The tables show that Agent A learns the slowest, due to having pessimistic initialisation. As every timestep returns a reward that is higher than its initial value, Agent A will tend to follow whichever actions it selects first, rather than seeking better alternatives. Agent B is the fastest learner. Having balanced initialisation that allows it to get a lower and higher reward than its initial values may encourage Agent B to learn faster. However, shifting the reward function further, as is done to Agent C, does not result in faster learning.

	Agent A	Agent B	Agent C
Trial 1	0.082	0.083	0.083
Trial 2	0.058	0.089	0.092
Trial 3	0.060	0.086	0.081
Trial 4	0.074	0.080	0.072
Trial 5	0.064	0.088	0.091
Trial 6	0.062	0.079	0.086
Trial 7	0.069	0.098	0.082
Trial 8	0.082	0.085	0.086
Trial 9	0.093	0.094	0.097
Trial 10	0.067	0.097	0.092
Average	0.071	0.088	0.086

Table 4.3: Average reward for 1,000 episodes

Table 4.6 shows the t-test results for all the agents. After 1,000 episodes, the differences in performance between agents A and B is statistically significant (2-tailed t-test, $p < 0.01$), however there is no significant difference between agents B and C. After 2000 and 5000 episodes there are significant differences between Agents A-B and Agents B-C ($p < 0.05$). The same cannot be said about Agents A-C, which in all cases show results that are not statistically different.

The charts in Figure 4.1 show the learning speeds for the first 1,000 episodes for the three agents. Agent B gets better rewards than Agent A in most of the early episodes and maintains a 20% lead against Agent A (0.241 vs. 0.207) throughout the ten trials. Agent C initially does better than Agent B for the first 100 episodes, but does not maintain the same learning speed

	Agent A	Agent B	Agent C
Trial 1	0.109	0.141	0.128
Trial 2	0.081	0.162	0.122
Trial 3	0.088	0.138	0.109
Trial 4	0.099	0.123	0.117
Trial 5	0.105	0.154	0.123
Trial 6	0.098	0.109	0.130
Trial 7	0.126	0.149	0.118
Trial 8	0.134	0.138	0.129
Trial 9	0.121	0.134	0.170
Trial 10	0.127	0.148	0.118
Average	0.109	0.140	0.126

Table 4.4: Average reward for 2,000 episodes

	Agent A	Agent B	Agent C
Trial 1	0.213	0.227	0.218
Trial 2	0.175	0.249	0.226
Trial 3	0.164	0.283	0.211
Trial 4	0.195	0.215	0.209
Trial 5	0.205	0.257	0.201
Trial 6	0.174	0.214	0.242
Trial 7	0.198	0.262	0.224
Trial 8	0.261	0.233	0.245
Trial 9	0.210	0.229	0.260
Trial 10	0.277	0.236	0.188
Average	0.207	0.241	0.222

Table 4.5: Average reward for 5,000 episodes

	Agents A-B	Agents B-C	Agents A-C
1,000 episodes	0.00041	0.29504	0.07712
2,000 episodes	0.00030	0.04075	0.35627
5,000 episodes	0.01196	0.04069	0.38051

Table 4.6: T-test results

and ends up having lower average scores than Agent B. The charts show the average reward received in each episode over ten trials for each agent. The colours indicate the percentiles of the trial outcomes at each episode.

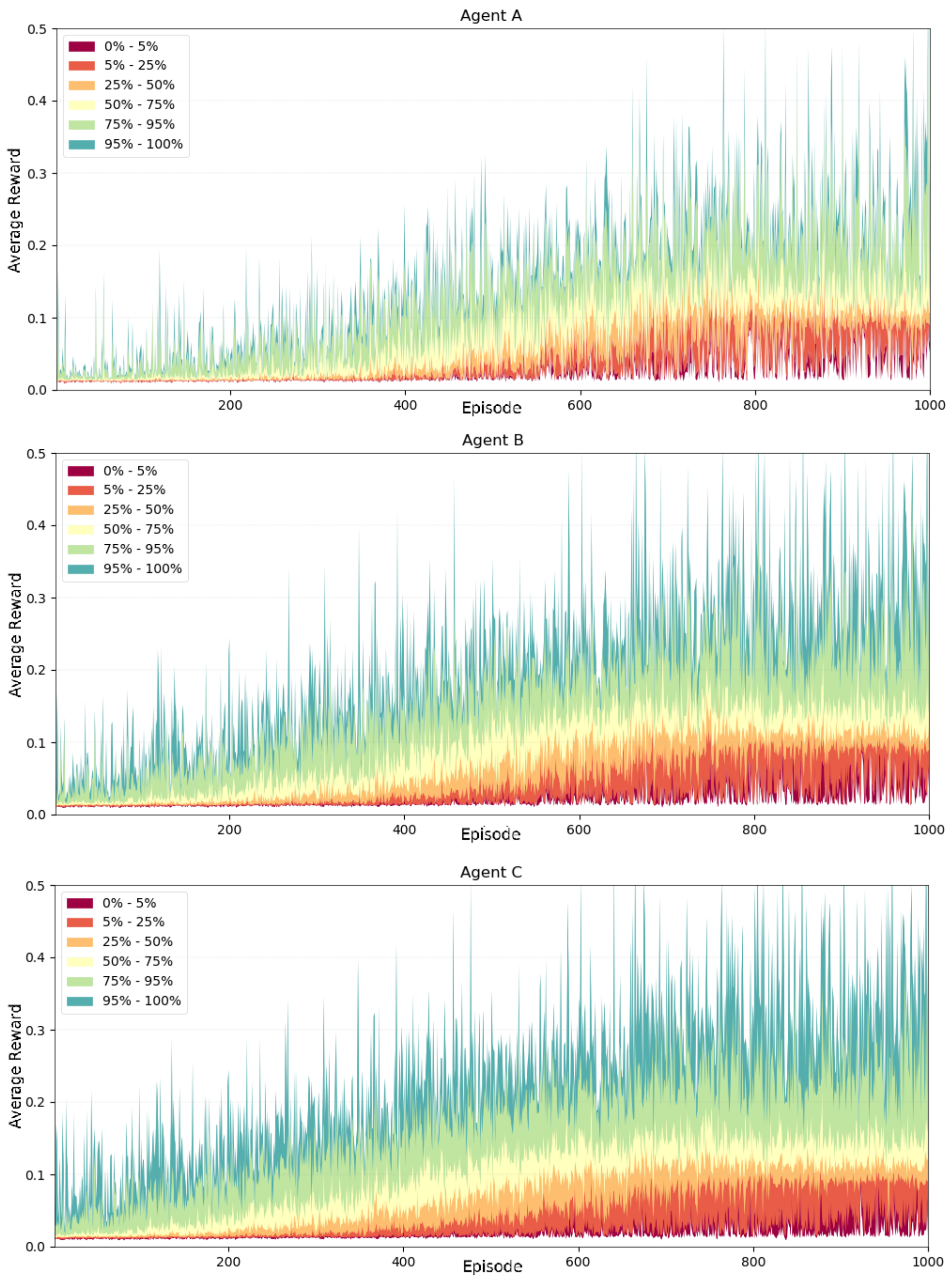


Figure 4.1: Reward received per episode over the ten trials of each agent. Colours indicate percentiles of the trial outcomes at each episode

The study shows that instead of using the McGrew score outright as the reward function, adding a negative offset as follows allows the agent to learn faster:

$$\text{Reward} = \text{McGrewScore} - 0.5 \tag{4.2}$$

Due to these results, Equation (4.2) is chosen as the reward function throughout this thesis.

4.3 Results

Six Q(λ) agents and six ACET agents are trained by piloting the blue aircraft, facing a red opponent that flies in a straight line at a constant speed. Five Q(λ) agents and five ACET agents start from fixed initial positions 001, 002, 003, 004 and 005, respectively, and each agent learns for 200,000 episodes that each consists of 700 timesteps. As detailed in 3.2, the initial position at the beginning of every episode is slightly randomised so no two episodes have the same initial position. The Q(λ) agents are code-named ql-001, ql-002, ql-003, ql-004 and ql-005. The ql-001 agent learns using the 001 initial position and the ql-005 agent learns using the 005 initial position. Similarly, the five ACET agents are code-named ac-001, ac-002, ac-003, ac-004 and ac-005.

The sixth Q(λ) agent and the sixth ACET agent learn using random initial positions for 200,000 episodes and are code-named ql-random and ac-random, respectively.

The experiments involving the twelve agents were repeated ten times using a set of ten random seeds. Every agent uses the same set of random seeds. All the agents use the discretisation strategy, action space and reward function explained in the previous section.

At each timestep, the Q(λ) agents use this linearly-decaying ϵ-greedy function to determine the probability of taking a random action:

$$\epsilon = c\frac{N-n}{N-1}$$

Here N is the number of episodes set for the learning session, n the current episode and c a small positive constant that controls the degree of exploration. The value of the first episode is c and the value of the last episode is 0. For the Q(λ) experiments in this chapter, c is set to 0.2. The ϵ-greedy function is used in action selection by generating a random number. If the random number for the timestep is smaller than ϵ, then the agent will explore, i.e. choose a random action. Otherwise, the agent will exploit. The ACET agents make the explore/exploit decision using the softmax function.

4.3.1 Learning Curves

Figure 4.2 show the learning curves of Q(λ) and ACET agents that learned by starting from 001, 002, 003, 004, 005 and random initial positions. The legend in each graph shows the agent's code-name and the average score over all episodes. All learning curves are obtained by running the agent with ten different random seeds. Every 250 consecutive data points averaged into one new data point. The darker line shows the mean of the new data points over seeds. The shaded area represents the mean ± the standard deviation. The values in the legend show the averages for all trials. Because the Ace Zero reward has been offset by -0.5, the average rewards for the learning curves are increased by 0.5 to reflect the original McGrew scores.

The low average scores for the Q(λ) learning curves indicate that the agents failed to learn. By contrast, all the ACET agents, except the one starting from fixed initial position 005, managed to learn.

A learning curve that increases steadily indicates a successful learning agent. The average reward over all the episodes can be used to measure how fast the agent learns. Table 4.7 shows the average reward for each starting position for both Q(λ) and ACET agents. The table shows that starting positions 001, 002 and Random allow both agents to learn the fastest.

It is not surprising that the agents learn relatively fast when starting from position 001. After all, Blue starts right behind Red, a layout that gives Blue a relatively high reward. Blue just needs to maintain its position throughout all the 700 timesteps in each episode.

Initial Position	Q(λ)	ACET
001	0.0387	0.1284
002	0.0182	0.1308
003	0.0042	0.0967
004	0.0073	0.0442
005	0.0030	0.0035
Random	0.0204	0.1332

Table 4.7: Average rewards over 200,000 episodes for Q(λ) and ACET agents for different initial positions

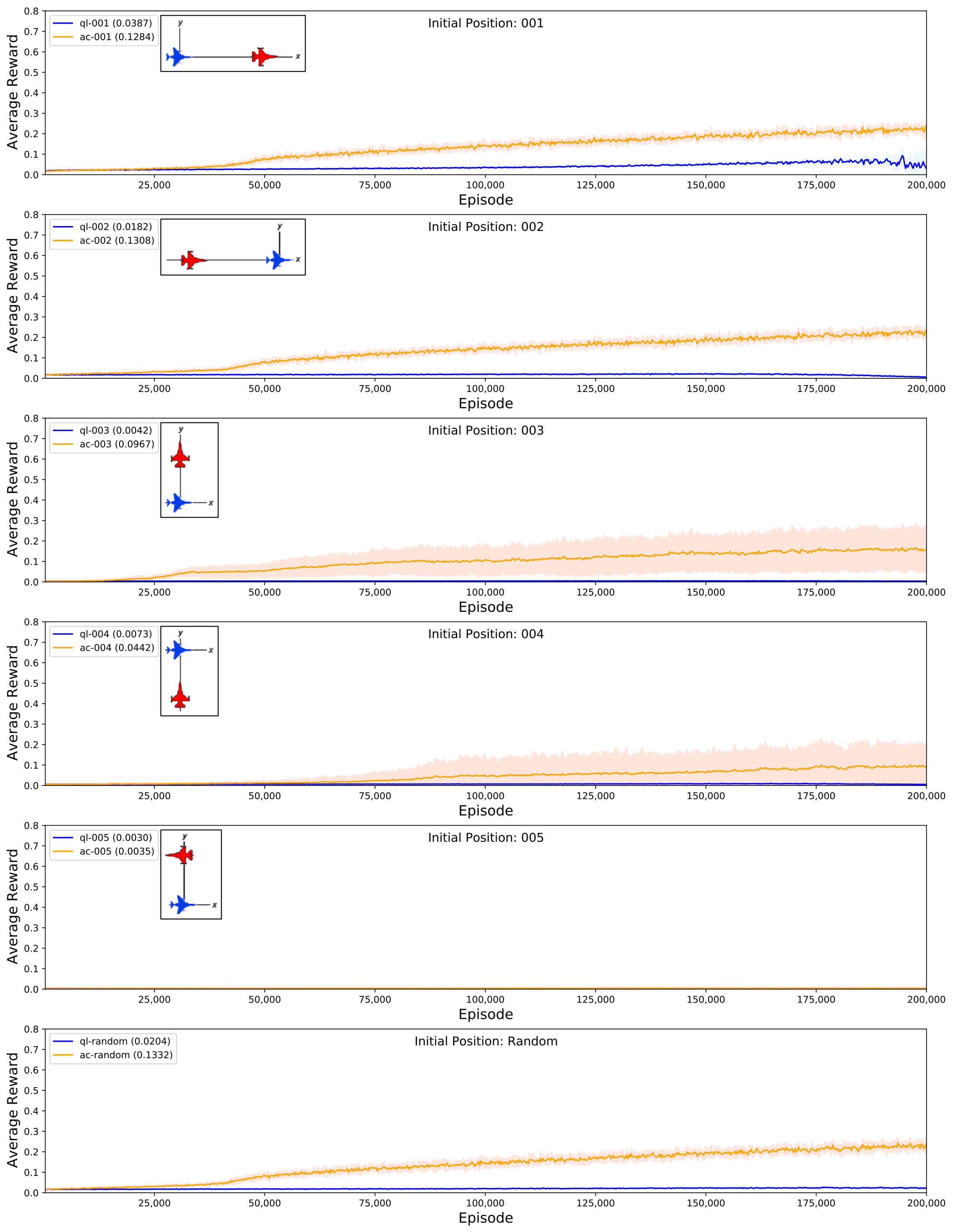


Figure 4.2: Q(λ) and ACET learning curves for different initial positions. The curves are obtained by running the agents with ten different random seeds and averaging every 250 consecutive data points into a new data point. The darker line shows the mean of the new data points over seeds. The shaded area represents the mean $\pm$ the standard deviation. The values in the legend show the averages for all trials and the average rewards have been increased by 0.5.

Position 002 starts Blue right behind Red and this layout initially gives Blue the lowest reward. However, through exploration Blue soon realises that it can make a left or a right to get behind Red and start earning higher rewards. The fact that Red flies in a straight line toward it really helps Blue outmanoeuvre Red.

According to the values in Table 4.7, Position 003 presents a bigger challenge to Blue since the average rewards for both agents when starting from this position are slightly lower than when they start from position 001 or 002. However, this layout gives Blue relative success and this is because Blue just needs to explore to the left to start trailing Red and start obtaining higher rewards.

Starting from initial position 004 makes it harder for Blue to get behind Red. One way to do this is to accelerate and then make a 270° turn to the right. Alternatively, it could brake and make a left. This challenge is reflected by lower average rewards for this position in Table 4.7.

Initial position 005 is the hardest initial position because Blue and Red start by flying away from each other. By the time Blue realises it has to make a U-turn to earn higher rewards, the episode is nearly terminated and its position reset. Both the Q(λ) and ACET agents do not show any sign of learning.

The random initial positions present the highest rewards for Blue. Even though Blue can be started at the most challenging layout, it is definitely helped by the fact that most of the time Blue starts from a closer range (0-1600 metres) from Red than when it follows one of the fixed initial positions which always places it 1,500 metres away from Red.

4.3.2 Performance

The indirect performance test results of the Q(λ) and ACET agents against the Smart Pursuit baseline on the 144 test scenarios are resampled using the bootstrap method as explained in 3.5 and the 95% confidence intervals generated from the resamples are shown in Figures 4.3 and 4.4, respectively. The horizontal red lines in both figures show the baselines from the Smart Pursuit engineered agent used in this thesis. Included in the charts is a random agent that was not trained and takes a random action at every timestep. Figure 4.3 shows the poor performance of the Q(λ) agents, in which they underperform the baseline and are not better than the random agent in three of four initial dispositions. In contrast, some of the ACET agents beat the baseline in three of the four initial dispositions. The most performant ACET agents are ac-001-200K, ac-002-200K and ac-random-200K, which learned from starting positions 001, 002 and random, respectively. The difference in performance of these three agents and the rest of the agents is statistically significant. Factoring in the learning curves of all the agents, it is clear there exists a positive correlation between learning success and performance. Note that Figure 4.4 uses a scale for the vertical axis that is different from that in Figure 4.3 to better show the improvement of ACET over Q(λ).

The finding that ACET is better than Q(λ) at handling a continuous-spaced problem such as the air combat manoeuvring problem is consistent with another experiment involving a continuous state space, which is presented in Appendix B. It could be an indication that Q(λ) is not suitable for problems with a continuous state space.

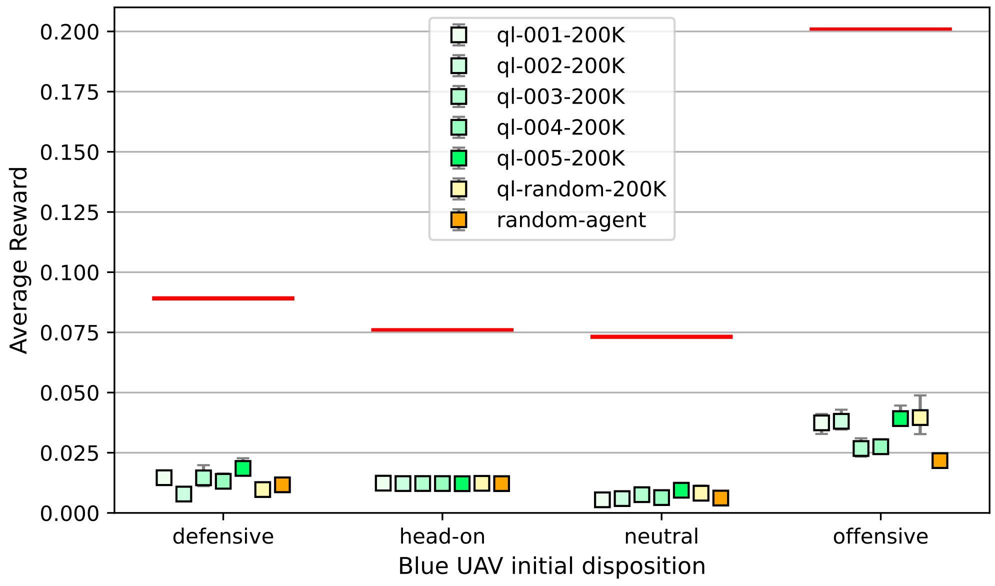


Figure 4.3: The performance of the Q(λ) agents against the Smart Pursuit baselines, which are indicated by the red horizontal lines. Each vertical bar shows a mean and a 95% confidence interval.

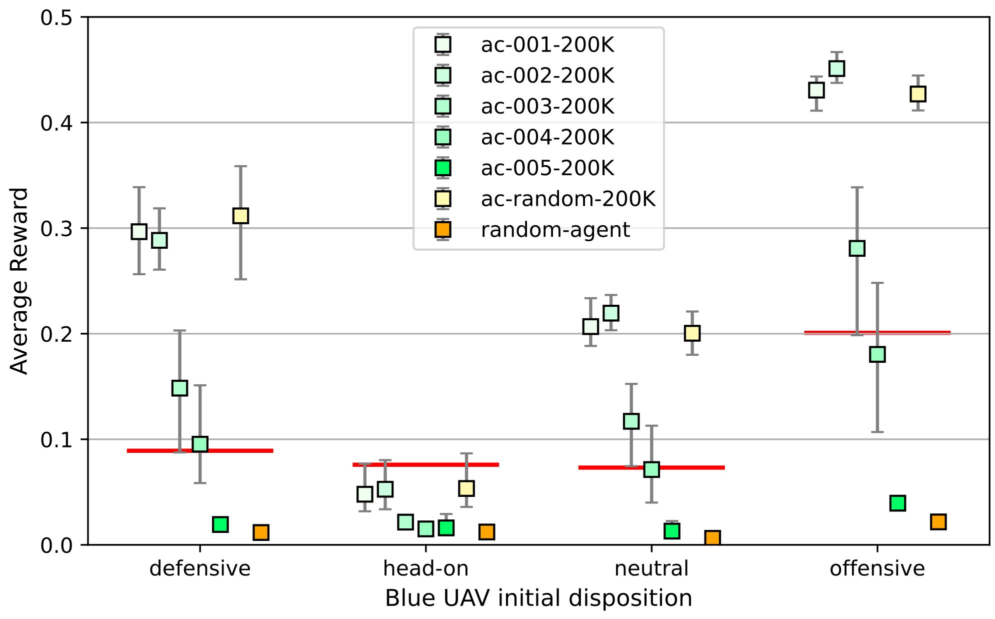


Figure 4.4: The performance of the ACET agents against the Smart Pursuit baselines, which are indicated by the red horizontal lines. Each vertical bar shows a mean and a 95% confidence interval.

4.3.3 Trajectories

All the agents in this chapter learn by facing a simple opponent, one that flies in a straight line at a constant speed. The objective of RL is that policies generated through successful learning are generalised enough that they can be used to navigate situations that have not been encountered during learning. This section compares the performance of ac-random, an agent that learned successfully, and ac-005, an agent that failed to learn, in navigating two complex scenarios, complex scenarios 1 and 2, that the agents have not seen during learning. The blue line in Figure 4.5 shows the trajectory of ac-random for complex scenario 1. The trajectory shows that the agent successfully follows its opponent throughout the 1000s run. By contrast, the blue line in Figure 4.6 shows a failed attempt of ac-005 facing its opponent. Figures 4.7 and 4.8 show the trajectories of ac-random and ac-005, respectively, for complex scenario 2. Again, ac-random successfully follows its opponent whereas ac-005 fails to do so.

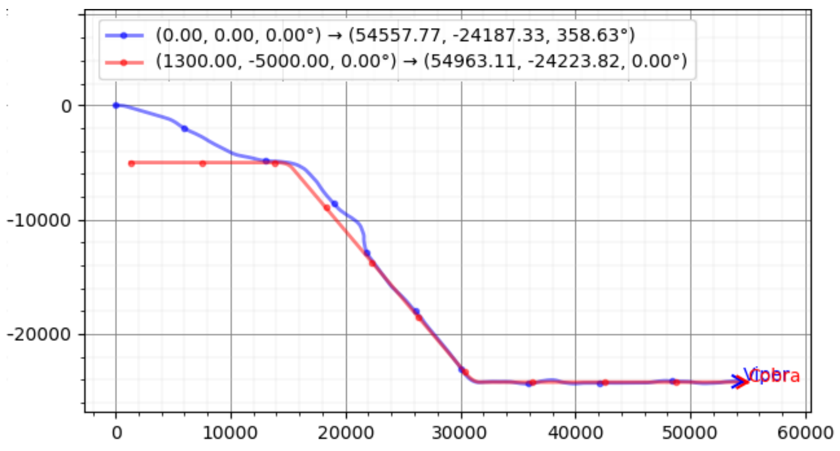


Figure 4.5: The trajectory of ac-random for complex scenario 1. Distances are in metres

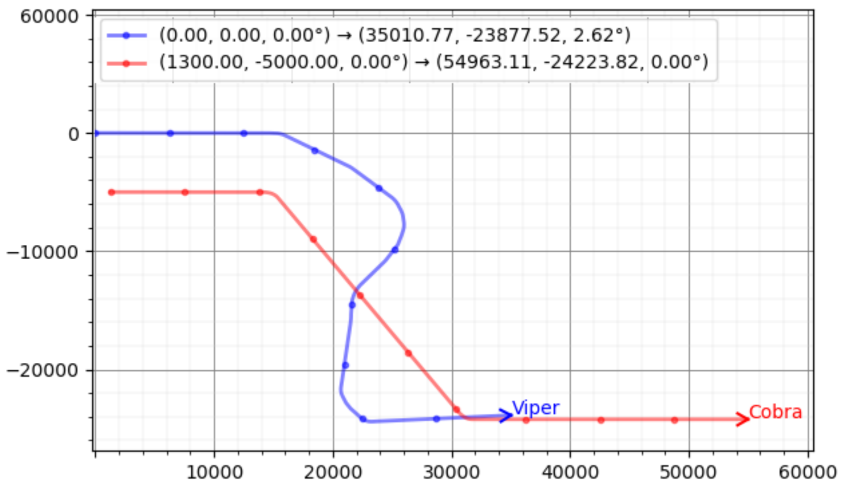


Figure 4.6: The trajectory of ac-005 for complex scenario 1. Distances are in metres

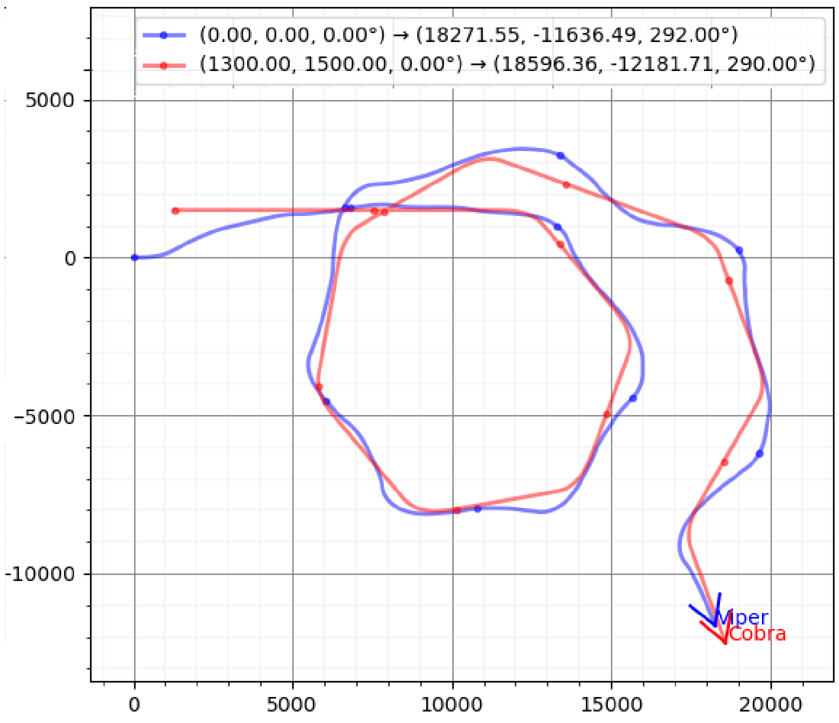


Figure 4.7: The trajectory of ac-random for complex scenario 2. Distances are in metres

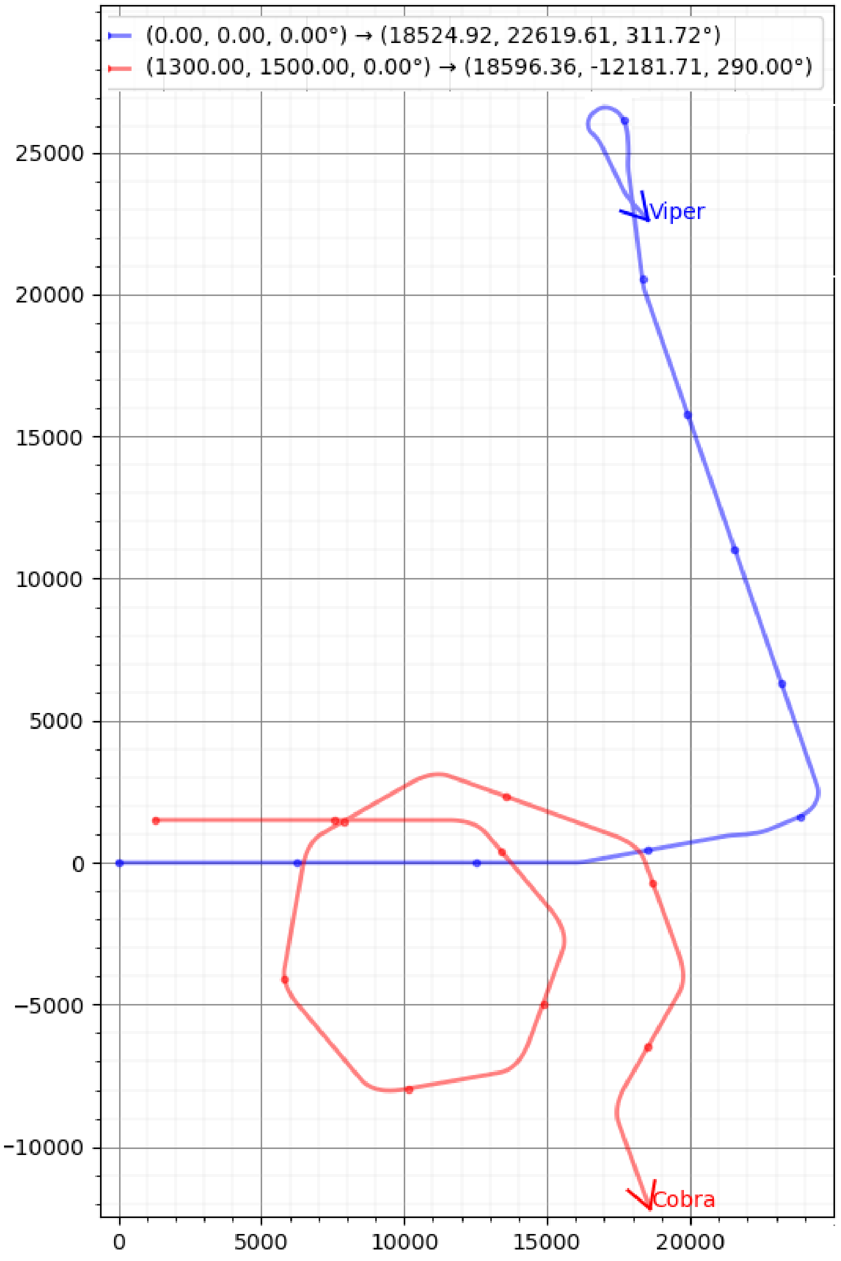


Figure 4.8: The trajectory of ac-005 for complex scenario 2. Distances are in metres

4.3.4 Learning Times

Table 4.8 shows the time taken for each agent to complete learning. There are no significant differences among the different initial positions. However, the cost per episode for ACET is almost twice as expensive as the cost for Q(λ), as shown in Table 4.9. This is because every update in Q(λ) involves changing one single Q value, whereas in ACET it updates all the elements in the policy table. Figures 4.9 and 4.10 show the 95% confidence intervals of the bootstrap resamples

Agent	Mean	Median	Standard Deviation
ql-001-200K	17.60	17.59	0.15
ql-002-200K	17.64	17.60	0.16
ql-003-200K	17.42	17.39	0.12
ql-004-200K	17.50	17.49	0.12
ql-005-200K	17.34	17.33	0.12
ql-random-200K	17.42	17.42	0.14
ac-001-200K	31.17	31.21	0.73
ac-002-200K	30.66	30.59	3.54
ac-003-200K	31.16	31.33	0.47
ac-004-200K	31.01	30.91	0.56
ac-005-200K	30.98	31.30	0.66
ac-random-200K	32.08	32.13	0.17

Table 4.8: Q(λ) and ACET learning time in hours

of the data in Tables 4.8 and 4.9, respectively.

Agent	Mean	Median	Standard Deviation
ql-001-200K	0.32	0.32	0.00
ql-002-200K	0.32	0.32	0.00
ql-003-200K	0.31	0.31	0.00
ql-004-200K	0.32	0.31	0.00
ql-005-200K	0.31	0.31	0.00
ql-random-200K	0.31	0.31	0.00
ac-001-200K	0.56	0.56	0.01
ac-002-200K	0.55	0.55	0.06
ac-003-200K	0.56	0.56	0.01
ac-004-200K	0.56	0.56	0.01
ac-005-200K	0.56	0.56	0.01
ac-random-200K	0.58	0.58	0.00

Table 4.9: Q(λ) and ACET learning time per episode in seconds

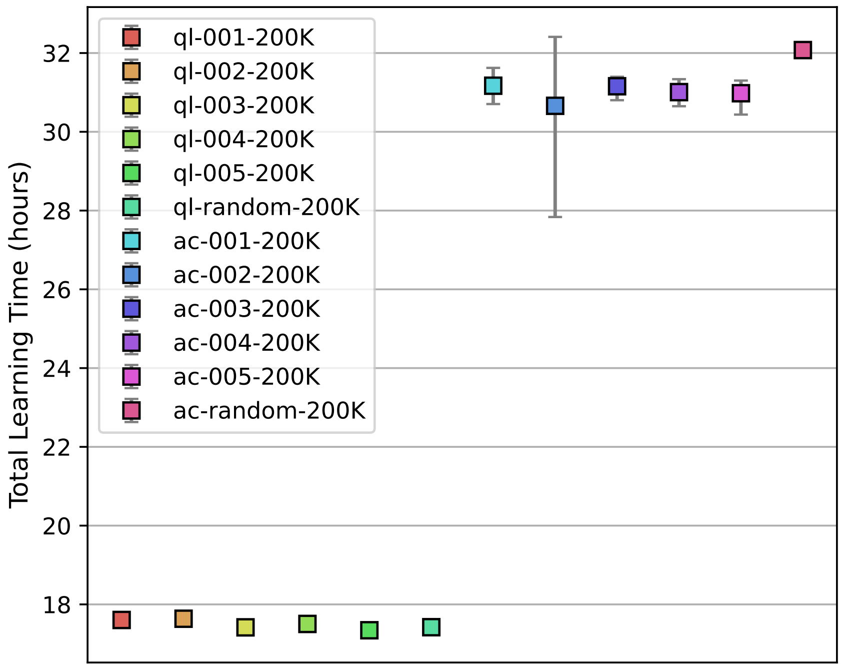


Figure 4.9: Q(λ) and ACET learning times in hours

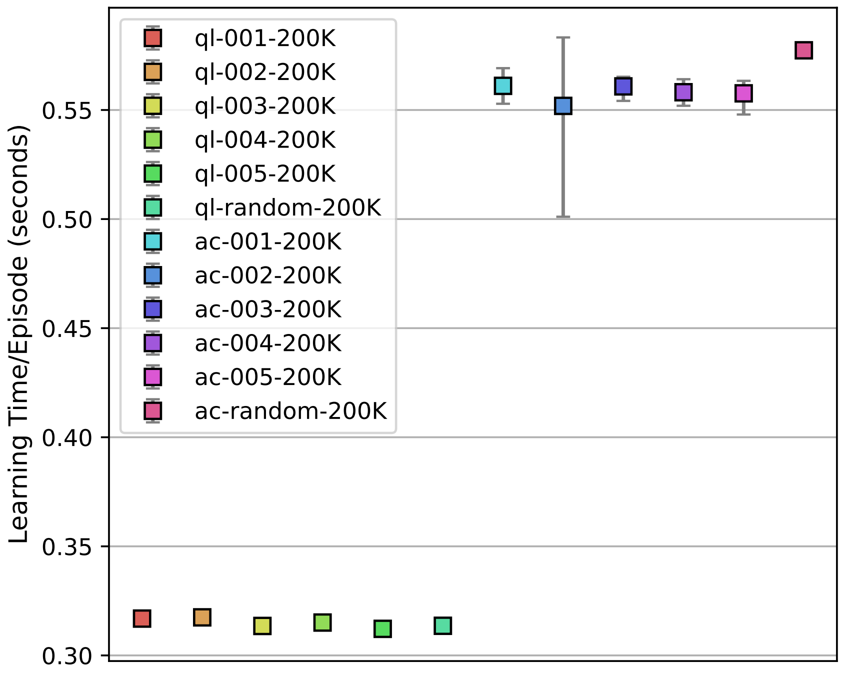


Figure 4.10: Q(λ) and ACET learning times per episode in seconds

4.4 Multipolicy ACET Agents

A complex RL task can be decomposed into simpler sub-tasks, for example by using methods like Q-decomposition, where multiple sub-agents, each trained with its own reward function and learn independently, work together to form a more complex agent that acts as an arbitrator. The arbitrator selects an action by taking into account the Q-values from each sub-agent. It selects an action by maximising the sum of Q-values from all the sub-agents (Russel and Zimdars, 2003).

For this section, a separate experiment is conducted which is unrelated to the experiment in the previous section. Five ACET agents learns a policy by starting from fixed initial positions 001, 002, 003, 004 and 005 for 100,000 episodes. Each agent is therefore expected to be most skilful in the layouts most similar to its starting position. This section investigates if it is possible to improve the performance of the agents by using some or all of the five policies within the same agent, which is called a multipolicy ACET agent. The key in this experiment is to choose, for each timestep, a policy that will be used to select an action from. Three policy selection strategies, called mp1, mp2 and mp3, are tested. For comparison purposes, a separate agent called ac-random-500K is trained from random initial positions for 500,000 episodes, i.e. for the same total number of episodes used in training the sub-agents for the multipolicy agent. Policies from ac-random-500K are not used by the multipolicy agents.

The mp1 strategy is based on the AA and ATA angles, as given in Table 4.10. For instance, if $|AA| \leq 45$ and $|ATA| \geq 135$, use policy 5 (the policy from ac-005), and so on.

Policy	Condition
5	$\|ATA\| \leq 45$ and $\|AA\| \geq 135$
4	$45 \leq \|ATA\| \leq 135$ and $\|AA\| \geq 135$
3	$45 \leq \|ATA\| \leq 135$ and $\|AA\| \leq 45$
2	$135 \leq \|ATA\|$ and $\|AA\| < 135$
1	other than the above

Table 4.10: mp1 policy selection

The policy selection strategy for mp2 is to select an action from the most 'confident' softmax-ed numerical action preferences as described in Algorithm 7. It uses all five policies from agents that learned from starting positions 001, 002, 003, 004 and 005.

The policy selection strategy for mp3 is similar to that for mp2, except that mp3 uses three policies instead of five. The three policies are from the best performing agents, namely the agents that learned from starting positions 001, 002 and 003.

Figure 4.11 shows the performance of the ac-mp1, ac-mp2 and ac-mp3 agents compared to the single-policy agents whose policies they use and ac-random-500K. Each vertical bar shows the mean and confidence interval of a bootstrap-resampled test result. ac-mp2 and ac-mp3 are of special interest since both outperform the other agents in the offensive and head-on initial dispositions. They also achieved this without any requirement for additional human expertise,

Algorithm 7: mp2 policy selection

```
   Parameter(s): discrete state $s$, all policies $p[]$
   Result: policy to use
1  $max\_values = []$;
2  for each policy $p$ in $p[]$ do
3      $Q \leftarrow$ softmax($p[s]$);
4      Add the maximum value in numerical preferences $Q$ to $max\_values$;
5  end
6  $index \leftarrow$ argmax($max\_values$);
7  Select policy pointed by $p[index]$;
```

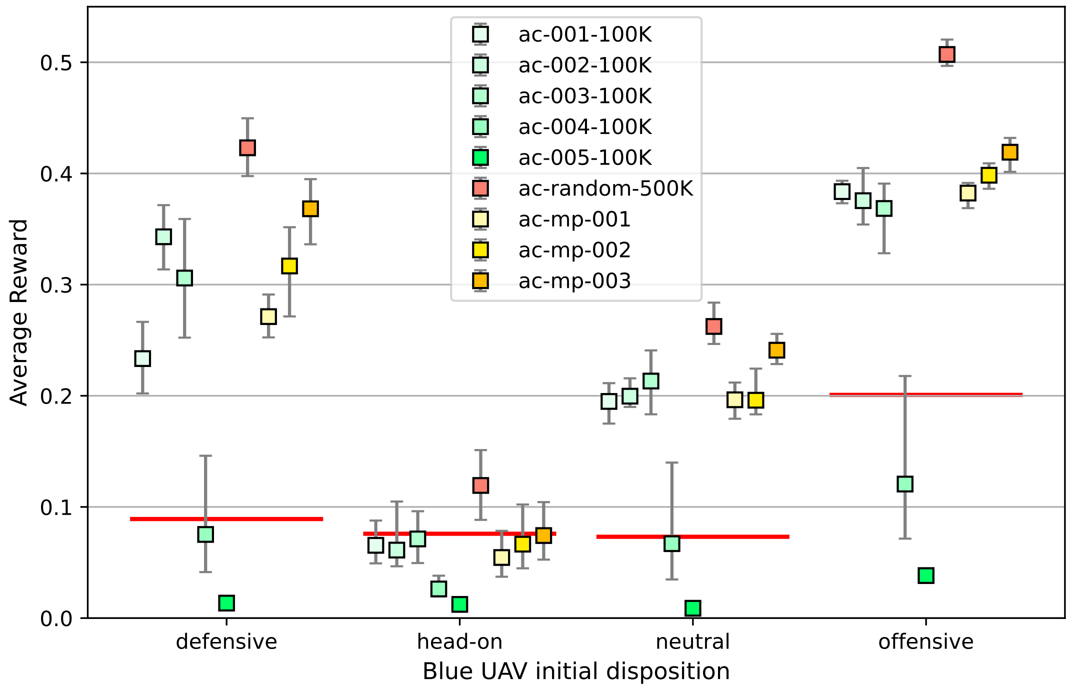


Figure 4.11: The performance of the multipolicy ACET agents against the Smart Pursuit baselines, which are indicated by the red horizontal lines. Each vertical bar shows a mean and a 95% confidence interval.

whereas mp1 required human input to design the policy-selection rules. This proves that a multipolicy agent may outperform its policy-contributing agents, depending on how a policy is selected. The advantages of using multipolicy agents is no additional learning is required, the agents simply use existing policies. It can also be argued that it may support distributed learning by having agents learning specific tasks and combining the results using this novel RL strategy. ac-mp3, which uses three policies instead of five, is better than ac-mp2 in all four initial dispositions, probably because some highly confident numerical action preferences in the excluded policies, which would have been selected should the policies had been included, provide inaccurate guidance to the agent. The ac-mp3 agent, however, underperforms ac-random-500K that went through more learning episodes.

4.5 Summary

This chapter uses Q(λ) and ACET to address the air combat manoeuvring domain, which is a continuous-state problem. Because both methods are tabular methods, observations from the air combat simulator need to be discretised before being passed to the agent. A discretisation strategy that calculates the distance between the two aircraft, AA and ATA angles and speed difference is used to convert a continuous state space into a discrete state space containing 14,000 states. The action space used in this thesis consists of five actions: 10° turn to the left, 10° turn to the right, increase speed by 10%, reduce speed by 10% and do nothing. The reward function is chosen after an experiment to find out which of three reward functions induces the fastest learning. The reward function selected is made up of the McGrew score with a negative offset:

$$\text{Reward} = \text{McGrewScore} - 0.5$$

Six Q(λ) agents and six ACET agents are used in the experiment. Five Q(λ) agents and five ACET agents start from fixed initial positions 001, 002, 003, 004 and 005, respectively. The sixth Q(λ) agent and the sixth ACET agent learn using random initial positions. All the agents learn for 200,000 episodes and each episode consists of 700 timesteps. All experiments are repeated ten times using a set of ten random seeds. Every agent uses the same set of random seeds and the same discretisation strategy, action space and reward function.

The learning curves generated from the learning sessions show that the ACET agents learn more quickly than the Q(λ) agents. Table 4.7 shows the average reward for each starting position for both Q(λ) and ACET agents. The table shows that initial positions 001, 002 and Random allow both agents to learn the fastest. The biggest difference occurs for initial position 003, where the ACET agent scores an average reward over episodes that is 23 times higher than what is achieved by the Q(λ) agent. In general, the low average scores for the Q(λ) learning curves indicate that the agents failed to learn. By contrast, all the ACET agents, except the one starting from initial position 005, managed to learn.

To measure performance, all the Q(λ) and ACET agents are compared against the Smart Pursuit baseline. Each agent is trained ten times to obtain ten policies. The ten test results from the ten policies are resampled using the bootstrap method 10,000 times. The means and 95% confidence intervals from the resamples are plotted as the performance charts in Figures 4.3 and 4.4. The former shows the poor performance of the Q(λ) agents, which underperform the baseline. The latter shows that the ACET agents beat the baseline in three of four initial dispositions. The most performant ACET agents are those that learned from initial positions 001, 002 and random, agents with the highest average rewards during learning. It is clear there exists a positive correlation between learning success and performance.

The finding that ACET is better than Q(λ) at handling a continuous-spaced problem such as the air combat manoeuvring problem is consistent with another experiment involving a continuous state space, which is presented in Appendix B. It could be an indication that tabular Q(λ) is not suitable for problems with a continuous state space.

The last section of this chapter investigates action selection strategies that combine ACET policies from different learning sessions. The results show that two of the three multipolicy strategies tested yield better performance than the individual policy-contributing agents. The main advantage of the multipolicy scheme is that performance enhancement can be achieved without further learning since the multipolicy agent uses existing policies. It may also support distributed learning by having agents learning specific tasks and combining the results using this novel RL strategy.

Chapter 5

Deep Reinforcement Learning Baseline Methods

The tabular approach discussed in Chapter 4 comprises a family of model-free reinforcement learning (RL) techniques that are known for their simplicity. The disadvantage of tabular methods is they are powerless to solve RL problems with high-dimensional state and/or action spaces, a phenomenon known as the curse of dimensionality (Bellman, 1961). To deal with high-dimensional problems requires approximate solution methods (Sutton and Barto, 2018). The general trend today is to use neural networks as function approximators, resulting in algorithms collectively known as neural RL or deep RL methods. This chapter discusses four deep RL methods—deep Q-network (DQN), DQN with a target network (DQN-wTN), double DQN and proximal policy optimisation (PPO)—and presents the results of applying them in air combat manoeuvring. The results are used as baselines for discrete-to-deep and multiobjective methods in the upcoming chapters.

5.1 Overview of the Methods Used

DQN and its variants are selected as DQN is a milestone discovery and PPO because it is considered today's modern deep RL algorithm. DQN, DQN-wTN and double DQN use Q-learning as their learning algorithm and PPO is a policy gradient algorithm. All methods are explained in the following subsections.

5.1.1 Deep Q-Network

As mentioned in Chapter 2, DQN is a deep RL technique that uses experience replay to decorrelate samples generated in RL learning and was first used to train RL agents to play ATARI games (Mnih et al., 2013).

An experience, or sometimes called a transition, is a tuple (s, a, r, s'), where s is the state the agent is in while taking action a, resulting in reward r and new state s'. In many cases, state s needs to be pre-processed through a function $\phi(s)$, like in the original ATARI game playing system by Mnih et al. (2013) where every frame is pre-processed into a smaller image. In this case, $\phi(s)$ is used instead of s. An experience is generated at each learning timestep and stored in a replay memory or buffer D, which is a first in, first out (FIFO) data structure such as a queue. When the replay memory is full, the oldest experience will be removed as a new experience is inserted. At each timestep, after an experience is generated and stored in the replay memory, a batch of experiences are randomly sampled from the replay memory to update the neural network.

DQN is presented in Algorithm 8.

Algorithm 8: DQN (Mnih et al., 2013)

```
   Parameter(s): discount-rate $\gamma \in (0,1]$, small $\epsilon > 0$
   Result: Q
1  Init. replay memory $D$ to capacity $N$;
2  Init. action-value function $Q$ with random weights $\theta$;
3  for each episode do
4      Initialise $s_1$;
5      for each step $t$ of episode or until $s_t$ is terminal do
6          With probability $\epsilon$, select a random action $a_t$, otherwise $a_t = argmax_a Q(\phi(s_t), a; \theta)$;
7          Take action $a_t$, observe reward $r_t$ and state $s_{t+1}$;
8          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$, where $\phi_t = \phi(s_t)$, in $D$;
9          Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$;
10         Set $y_j = \begin{cases} r_j, & \text{if episode terminates at step } j+1 \\ r_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \theta), & \text{otherwise} \end{cases}$;
11         Perform gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to $\theta$;
12     end
13 end
```

5.1.2 Deep Q-Network with A Target Network

The same team of researchers that invented DQN extended it by introducing a second network called the target network to better de-correlate samples. This new method is known as DQN with a target network (DQN-wTN) (Mnih et al., 2015) and was also tested within the ATARI game playing context. DQN-wTN was able to improve on some games over DQN and its success can be attributed to a target Q network that gives consistent targets during temporal difference backups.

DQN-wTN is presented in Algorithm 9. The lines printed in green highlight the differences between DQN-wTN and DQN.

Algorithm 9: DQN with A Target Network (Mnih et al., 2015). The lines in green highlight the differences between DQN-wTN and DQN

```
   Parameter(s): discount-rate $\gamma \in (0,1]$, small $\epsilon > 0$
   Result: Q
1  Init. replay memory $D$ to capacity $N$;
2  Init. action-value function $Q$ with random weights $\theta$;
3  Init. target action-value function $\hat{Q}$ with weights $\theta^{-}$=$\theta$;
4  for each episode do
5      Initialise $s_1$;
6      for each step $t$ of episode or until $s_t$ is terminal do
7          With probability $\epsilon$, select a random action $a_t$, otherwise $a_t = argmax_a Q(\phi(s_t), a; \theta)$;
8          Take action $a_t$, observe reward $r_t$ and state $s_{t+1}$;
9          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$, where $\phi_t = \phi(s_t)$, in $D$;
10         Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$;
11         Set $y_j = \begin{cases} r_j, & \text{if episode terminates at step } j+1 \\ r_j + \gamma max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^{-}), & \text{otherwise} \end{cases}$;
12         Perform gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to $\theta$;
13         Every $C$ steps, reset $\hat{Q} = Q$;
14     end
15 end
```

5.1.3 Double Deep Q-Network

Both DQN and DQN-wTN are not without flaws. In particular, they suffer from overestimations in some ATARI games (van Hasselt et al., 2016). Double DQN, which is based on double Q-learning (Van Hasselt, 2010), was introduced to reduce the observed overestimations and led to better performance on several games over DQN and DQN-wTN. In both double Q-learning and double DQN, there are two Q functions, Q^A and Q^B. Each Q function is updated using a value from the other Q function. At first glance, DQN-wTN and double DQN look similar because both use two neural networks, but there is a fundamental difference between the two methods. In DQN-wTN the second neural network is a clone of the first network and is used to update the first neural network at every timestep and a new clone is created every C steps. In double DQN both neural networks are of equal importance where the first network is used to update the second network and the second network is used to update the first network. At each timestep there is a 50% probability that a network will be chosen to select an action and get updated. One thing to note about double DQN is that while it outperformed DQN and DQN-wTN in some ATARI games, it failed to keep up with both in other games (van Hasselt et al., 2016).

Double DQN is presented in Algorithm 10.

Algorithm 10: Double DQN (van Hasselt et al., 2016)

```
   Parameter(s): discount-rate $\gamma \in (0,1]$, small $\epsilon > 0$
   Result: $Q^A$
1  Init. replay memory $D$ to capacity $N$;
2  Init. action-value functions $Q^A$ and $Q^B$ with random weights $\theta$;
3  for each episode do
4      Initialise $s_1$;
5      With 50% probability, randomly select $UPDATE(A)$ or $UPDATE(B)$;
6      for each step $t$ of episode or until $s_t$ is terminal do
7          With probability $\epsilon$, select a random action $a_t$; otherwise $a_t =$
             $\arg\max_a Q^A(\phi(s_t), a; \theta)$ if $UPDATE(A)$ or $a_t = \arg\max_a Q^B(\phi(s_t), a; \theta)$ if
             $UPDATE(B)$;
8          Take action $a_t$, observe reward $r_t$ and state $s_{t+1}$;
9          Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$, where $\phi_t = \phi(s_t)$, in $D$;
10         Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$;
11         With 50% probability, randomly select $UPDATE(A)$ or $UPDATE(B)$;
12         if UPDATE(A) then
13             Define $a^* = \arg\max_a Q^A(s', a)$;
14             $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(r + \gamma Q^B(s', a^*) - Q^A(s,a))$;
15         else
16             Define $b^* = \arg\max_b Q^B(s', a)$;
17             $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(r + \gamma Q^A(s', b^*) - Q^B(s,a))$;
18         end
19     end
20 end
```

5.1.4 Proximal Policy Optimisation

PPO is a family of policy gradient methods that are based on actor-critic (Schulman et al., 2017). The objective of PPO is to learn a parameterised policy $\pi(a|s,\boldsymbol{\theta})$ that is differentiable with respect to its parameters $\boldsymbol{\theta} \in \mathbb{R}^{d'}$. Like all other policy gradient methods, PPO uses an iterative update rule that follows this pattern.

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{5.1}$$

where $J(\boldsymbol{\theta})$ is some scalar performance measure and $\widehat{\nabla J(\boldsymbol{\theta}_t)} \in \mathbb{R}^{d'}$ a stochastic estimate whose expectation approximates the gradient of the performance measure with respect to its argument $\boldsymbol{\theta}_t$ (Sutton and Barto, 2018).

PPO utilises this widely used form of gradient estimator.

$$\hat{g} = \hat{\mathbb{E}}_t \left[\nabla \log \pi(a_t|s_t, \boldsymbol{\theta}_t) \hat{A}_t \right] \tag{5.2}$$

where $\hat{A}_t$ is an estimator of the advantage function at timestep t. Instead of differentiating the estimate, PPO uses and maximises this "surrogate" objective:

$$\hat{\mathbb{E}}_t = \left[\frac{\pi(a_t|s_t, \boldsymbol{\theta}_t)}{\pi(a_t|s_t, \boldsymbol{\theta}_{old})} \hat{A}_t \right] \tag{5.3}$$

where $\boldsymbol{\theta}_{old}$ is the vector of policy parameters before the update. The probability ratio of the new update and the old update is defined as

$$r_t(\boldsymbol{\theta}) = \frac{\pi(a_t|s_t, \boldsymbol{\theta}_t)}{\pi(a_t|s_t, \boldsymbol{\theta}_{old})} \tag{5.4}$$

The value of $r_t(\boldsymbol{\theta})$ is then clipped, resulting in this objective.

$$L^{CLIP}(\boldsymbol{\theta}) = \hat{\mathbb{E}}_t \left[\min(r_t(\boldsymbol{\theta})\hat{A}_t, \text{clip}(r_t(\boldsymbol{\theta}), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{5.5}$$

where ϵ is a hyperparameter. This means, large updates are ignored and replaced with $(1-\epsilon)$ or $(1+\epsilon)$.

Algorithm 11 presents the PPO algorithm. It follows the standard flow of the other RL algorithms in this thesis and uses a buffer to store experiences. However, unlike DQN that updates its neural network every timestep, PPO updates its neural networks only when the buffer is full using all the samples in the buffer. Dissimilar to DQN, there is no attempt to de-correlate samples by presenting the same samples repeatedly to the neural networks.

Algorithm 11: Actor-critic based PPO (Schulman et al., 2017)

```
   Parameter(s): number of actors $N \geq 1$, number of epochs $K$, buffer and minibatch size
                 $M$, number of timesteps $NT$
   Result: $\theta$
 1 Init. buffer $D$ to capacity $M$;
 2 for each episode do
 3     Take action $a_t$, observe reward $r_t$ and state $s_{t+1}$;
 4     Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$, where $\phi_t = \phi(s_t)$, in $D$;
 5     if D full then
 6         Get samples from $D$ and empty $D$;
 7         Compute advantage estimates $\hat{A}_1, ..., \hat{A}_T$;
 8         Optimise surrogate $L$ wrt $\theta$, with $K$ epochs and minibatch size $M \leq NT$;
 9         $\theta_{old} \leftarrow \theta$;
10     end
11 end
```

5.2 Experimental Setup

When using a deep RL method to solve a sequential decision problem with a continuous state space, three components are required for the setup: state normalisation, action spaces and the reward function. This section discusses these three components and the architecture of the neural networks used in this chapter.

5.2.1 State Normalisation

The state variables in this thesis project have different physical units and varying values with different ranges. This can make the neural network slow or sometimes impossible to learn. To enable or expedite the training of the neural networks, it is necessary to normalise the values so they fall within similar ranges. This is discussed in (Ioffe and Szegedy, 2015) (Lillicrap et al., 2015).

The four state variables—R (distance), AA (attack angle), ATA (antenna train angle) and $delta\_v$ (speed difference)—are normalised as follows:

$$R = R/4500.00$$

$$AA = AA/180.00$$

$$ATA = ATA/180.00$$

$$delta\_v = delta\_v/40.00$$

Early experiments conducted for this chapter revealed that learning failed without state normalisation.

5.2.2 The Action Space

The deep RL agents implemented and tested in this chapter use the same action space as the one used in Chapter 4. There are five actions in the action space:

- 10° turn to the left
- 10° turn to the right
- increase speed by 10%
- reduce speed by 10%
- do nothing

5.2.3 The Reward Function

The same reward function as the one discussed in Section 4.2.3 is used for the experiments in this chapter.

$$\text{Reward} = \text{McGrewScore} - 0.5 \tag{5.6}$$

5.2.4 Neural Network Architecture

The DQN, DQN-wTN and double DQN agents in this chapter use four-layer neural networks. The input and output layers consist of four nodes and five nodes, respectively, the same as the number of state variables and the number of actions. Each of the two hidden layers consists of 300 nodes. The values of the state variables are normalised before they are fed to the input layer. This normaliser replaces the convolutional layer used for image preprocessing in the original ATARI-playing DQN network and is suitable for the air combat manoeuvring problem whose input is not an image. The normaliser ensures the agent learns fast. Adam, which extends stochastic gradient descent and is an algorithm for first-order gradient-based optimisation of stochastic objective functions, is chosen as the optimiser. It is based on adaptive estimates of lower-order moments, is easy to implement and computationally efficient (Kingma and Ba, 2014). For the DQN-wTN agents an update frequency C=700 is used, meaning every 700 timesteps the target network is synchronised with the main network. As shown in section 5.3, this simple network architecture is suitable for problems with a low-dimensional state space.

Table 5.1 shows the hyperparameters for the neural networks used in the DQN, DQN-wTN and double DQN agents.

Hyperparameter	**Value**	**Description**
replay memory size	1,000,000	maximum number of samples that can be stored in the replay memory
minibatch size	64	number of samples for each stochastic gradient descent (SGD) update
hidden layer dimension	[300, 300]	number of nodes in the hidden layers
initial ϵ	1	value of ϵ in ϵ-greedy exploration for the first learning episode
final ϵ	0.01	value of ϵ in ϵ-greedy exploration for the last learning episode
discount factor	0.99	discount factor (γ) used in Q-learning updates
learning rate	0.001	learning rate used in the optimiser
update frequency C	700	The update frequency for DQN-wTN target network

Table 5.1: Hyperparameters for DQN, DQN-wTN and double DQN agents

Each of the PPO agents in this chapter uses two neural networks, an actor and a critic. The actor's input and output layers consist of four nodes and five nodes, respectively, the same as the number of state variables and the number of actions. It has two hidden layers consisting of 300 nodes each. The values of the state variables are normalised before they are fed to the input layer. The normaliser ensures the agent learns fast. The values of the output layer are passed to an activation function before being returned to the agent. The critic has a similar architecture to the actor. It has a four-node input layer, two hidden layers consisting of 300 nodes each and an activation function that receives its input from the last hidden layer. The output of the critic is used in calculating the advantage function. Both the actor and the critic use an Adam optimiser. Table 5.2 shows the hyperparameters for the neural networks used in the PPO agents.

Hyperparameter	**Value**	**Description**
buffer size	4000	the size of the buffer to store samples
hidden layer dimension	[300, 300]	number of nodes in the hidden layers
ϵ	0.2	a constant to determine when updates need to be clipped
learning rate (actor)	0.0003	learning rate used in the actor's optimiser
learning rate (critic)	0.001	learning rate used in the critic's optimiser
training iterations (actor)	80	the number of iterations used in training the actor's neural network
training iterations (critic)	80	the number of iterations used in training the critic's neural network

Table 5.2: Hyperparameters for PPO agents

5.3 Results

Six DQN, six DQN-wTN, six double DQN and six PPO agents learn by piloting the blue aircraft, facing a red opponent that flies in a straight line at a constant speed. The first five of the DQN, DQN-wTN, double DQN and PPO agents start from fixed initial positions 001, 002, 003, 004 and 005, respectively, and each agent learns for 20,000 episodes that each consists of 700 timesteps. As detailed in 3.2, the initial position at the beginning of every episode is slightly randomised so no two episodes have the same initial position. The DQN agents are code-named dqn-001, dqn-002, dqn-003, dqn-004 and dqn-005. The dqn-001 agent learns using the 001 initial position, the dqn-002 agent learns using the 002 initial position and so on. Similarly, the five DQN-wTN agents are code-named dqn-wtn-001, dqn-wtn-002, dqn-wtn-003, dqn-wtn-004 and dqn-wtn-005. In addition, the five double DQN agents are code-named ddqn-001, ddqn-002, ddqn-003, ddqn-004 and ddqn-005 and the five PPO agents are code-named ppo-001, ppo-002, ppo-003, ppo-004 and ppo-005.

The sixth DQN agent, the sixth DQN-wTN agent, the sixth double DQN agent and the sixth PPO agent learn using random initial positions for 20,000 episodes and are code-named dqn-random, dqn-wtn-random, ddqn-random and ppo-random, respectively.

The experiments involving the 24 agents are repeated ten times using a set of ten random seeds. Every agent uses the same set of random seeds. At the end of a learning session, a policy is saved and later used for performance testing. For each of the DQN-wTN and double DQN agents, only the primary network is saved as a policy because action selection can be made of a single network.

5.3.1 Learning Curves

Figures 5.1 show the learning curves of the DQN, DQN-wTN and double DQN agents starting from six different initial positions. Each of the learning curves is obtained by running the agents for 20,000 episodes with ten different random seeds. Every 50 consecutive data points are averaged into one new data point. The learning curves for the DQN, DQN-wTN and double DQN agents are drawn in blue, orange and green, respectively. The darker line shows the mean of the new data points over seeds. The shaded area is obtained by drawing two more curves (mean ± standard deviation) and filling the area between them. Because the Ace Zero reward has been offset by -0.5, the average rewards for the learning curves are increased by 0.5 to reflect the original McGrew scores.

Unlike the tabular method agents in Chapter 4 that could only learned in five out of six initial positions (with failure when starting from initial position 005), all the DQN, DQN-wTN and double DQN agents learned successfully in all starting positions, even though only the DQN-wTN agent did well for initial position 005. It is also apparent that all the learning curves are still trending upward around the final learning episodes, suggesting that they could have learned more had learning not been terminated. The reason for limiting the number of learning episodes to 20,000 is the high cost of learning for the deep RL agents, as will be discussed in detail in section 5.3.3.

Like the tabular method agents in Chapter 4, the DQN, DQN-wTN and double DQN agents obtained the highest average rewards when learning from fixed initial positions 001 and 002 and from random initial positions. Initial position 001 is the easiest position to get a high average reward because the Blue starts right behind Red, a layout that gives Blue a relatively high reward. Blue just needs to maintain its position throughout all the 700 timesteps in each episode.

Initial position 002 starts Blue right behind Red, giving Blue some of the lowest rewards. However, Blue seems to immediately realise this and make a left or a right to get behind Red and start earning higher rewards. The fact that Red flies in a straight line toward Blue helps Blue outmanoeuvre Red.

Initial position 003 is a bigger challenge to Blue since the average rewards for both agents when starting from this position are slightly lower than when they start from position 001 or 002. However, this layout proves not too difficult for all the agents and all of them were successful.

In general, starting from initial position 004 is harder for Blue to get behind Red. One way to do this is to accelerate and then make a 270° turn to the right. Alternatively, it could reduce its speed and make a left. Despite this difficult initial disposition, the deep RL agents managed to overcome the challenge.

Initial position 005 is the hardest start position because Blue and Red start by flying away from each other. By the time Blue realises it has to make a U-turn to earn higher rewards, the learning session is nearly terminated and the positions of the aircraft reset. Only the DQN-wTN agent is mildly successful when starting from this position. The same cannot be said of the DQN and double DQN agents, which obtained much lower average rewards for this initial position.

The random initial positions present the third or fourth highest rewards for Blue. Even though Blue can be started at the most challenging layout, it is definitely helped by the fact that most of the time Blue starts from a closer range from Red (0-1600 metres) than when it follows one of the fixed initial positions which always places it 1,500 metres away from Red.

In all initial positions except 005, the DQN agent performs better than the other two agents and the DQN-wTN agent comes second, followed by the double DQN agent in the last place. Section 5.3.2 discusses possible causes for this and Section 5.3.4 the variations between trials.

Figures 5.2 show the learning curves of the PPO agents starting from six different initial positions. Each of the learning curves is obtained by running the agents for 20,000 episodes with ten different random seeds. Every 50 consecutive data points are averaged into one new data point. For comparison, each chart also shows the learning curves of the DQN agent learning from the same initial position. The learning curves for the PPO agents are drawn in brown and those of the DQN agents in blue. The darker line shows the mean of the new data points over seeds. The shaded area is obtained by drawing two more curves (mean $\pm$ standard deviation) and filling the area between them.

Only the PPO agents that started from initial positions 001 and 002 reached average scores comparable to the DQN agents. The other PPO agents did not learn successfully.

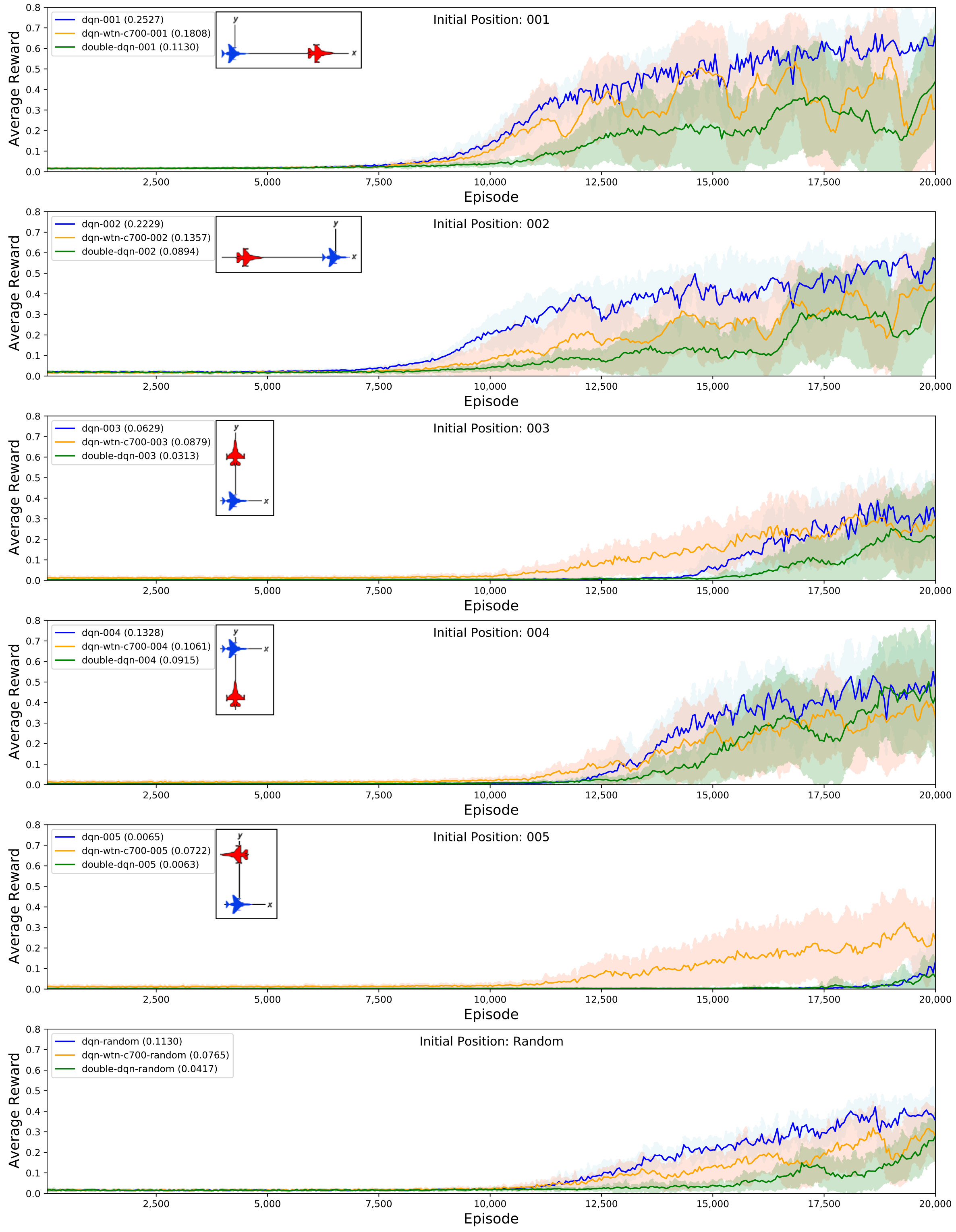


Figure 5.1: The learning curves of the DQN, DQN-wTN and double DQN agents for different initial positions, obtained by running the agents ten times with ten different seeds and averaging every 50 consecutive data points. The darker line is the mean of the new data points over seeds. The shaded area represents the mean ± the standard deviation. The values in the legend show the averages for all trials and the average rewards have been increased by 0.5.

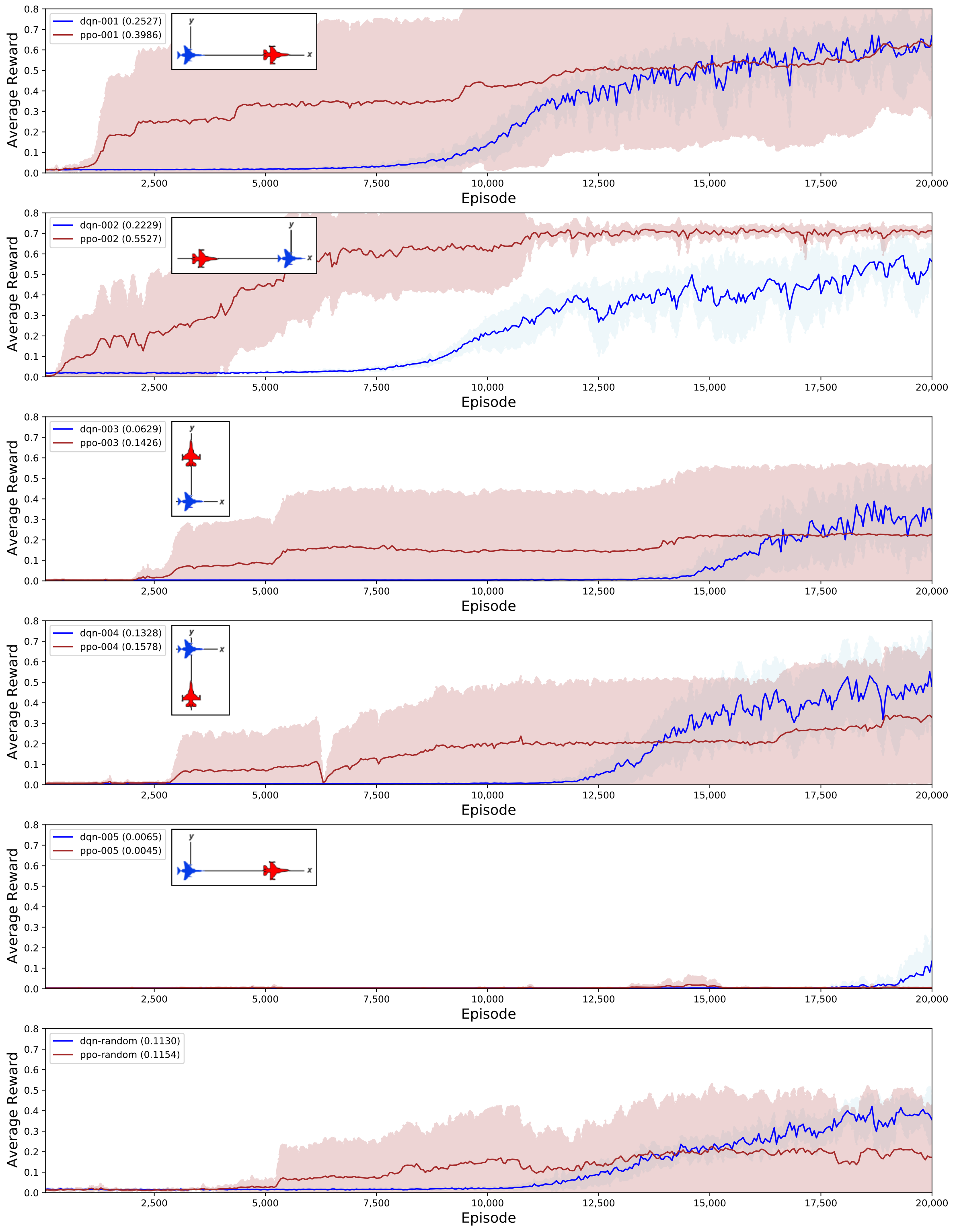


Figure 5.2: The learning curves of the PPO agents for different initial positions compared to those of the DQN agents, obtained by running the agents ten times with ten different random seeds and averaging every 50 consecutive data points. The darker line is the mean of the new data points over seeds. The shaded area represents the mean ± the standard deviation. The values in the legend show the averages for all trials

5.3.2 Performance

The test results from all the deep RL agents are resampled using the bootstrap method as explained in 3.5 and the means and confidence intervals from the resamples plotted. Figures 5.3, 5.4, 5.5 and 5.6 show the performance of the DQN, DQN-wTN, double DQN and PPO agents, respectively, all using six different policies compared to the performance of the Smart Pursuit agent against the simple agent that moves in a straight line. All tests are done using 144 different initial positions discussed in Chapter 3. The red horizontal lines show the baseline value for each initial disposition and each vertical bar shows the mean and 95% confidence interval of each agent.

In general for the DQN, DQN-wTN and double DQN, an agent that received a higher average reward during learning tends to have higher performance. The agents perform better than the tabular method agents as shown by some agents—the dqn-001, dqn-002, dqn-random, dqn-wtn-002, dqn-wtn-005, dqn-wtn-random—beating the baselines in all four initial dispositions. The dqn-wtn-005 agent, that obtained the highest average reward when starting from initial position 005, also performs better than the dqn-005 and ddqn-005 agents. It is also worth noting that in the case of DQN and double DQN, the agents that learned from initial position 001, where Blue is in the offensive position, also perform better than all the other agents that learn by starting in other fixed initial positions. Likewise, the agents that learned from initial position 002, where Blue is in defensive position, also perform better than all the other agents that learn by starting in other fixed initial positions. The DQN, DQN-wTN and double DQN agents that learned by starting from random positions generally perform well and in many cases beat the other agents implementing the same algorithm.

The PPO agents, however, perform poorly, way below the Smart Pursuit baselines, which are not shown in Figure 5.6 because the chart uses a much lower scale. The failure includes the ppo-001 and ppo-002 agents, that learned successfully from initial positions 001 and 002, respectively. This indicates that the two agents did not generalise well enough to be able to handle the 144 tests that had not been seen during learning.

There are two things that can explain the poor performance of PPO compared to DQN and its variants. First, PPO only updates its neural networks when its buffer is full. This means, for the same 20,000 learning episodes, the PPO agents actually had far less chance to learn. Second, PPO does not randomly sample its buffer to minimise correlations between samples, as is done in DQN and its variants.

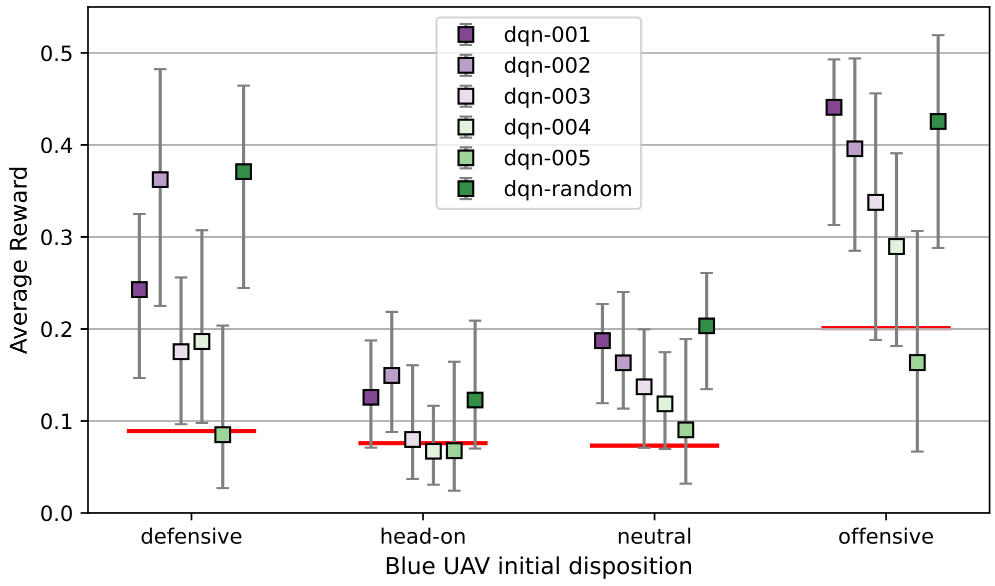


Figure 5.3: The performance of the DQN agents against the Smart Pursuit baselines where all the agents learned for 20,000 episodes. The red horizontal lines are the scores of the baseline for each initial disposition. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

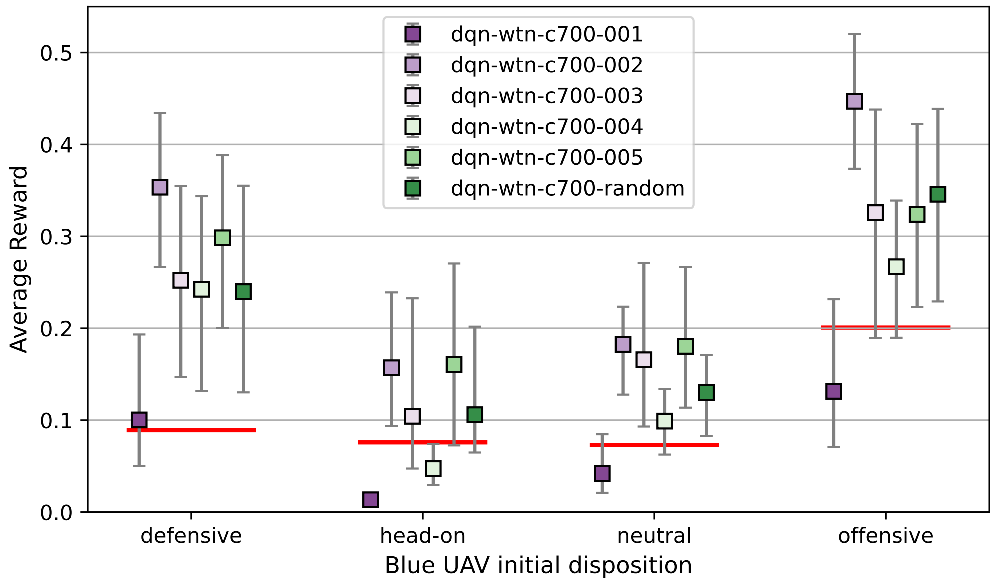


Figure 5.4: The performance of the DQN-wTN agents against the Smart Pursuit baselines where all the agents learned for 20,000 episodes. The red horizontal lines are the scores of the baseline for each initial disposition. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

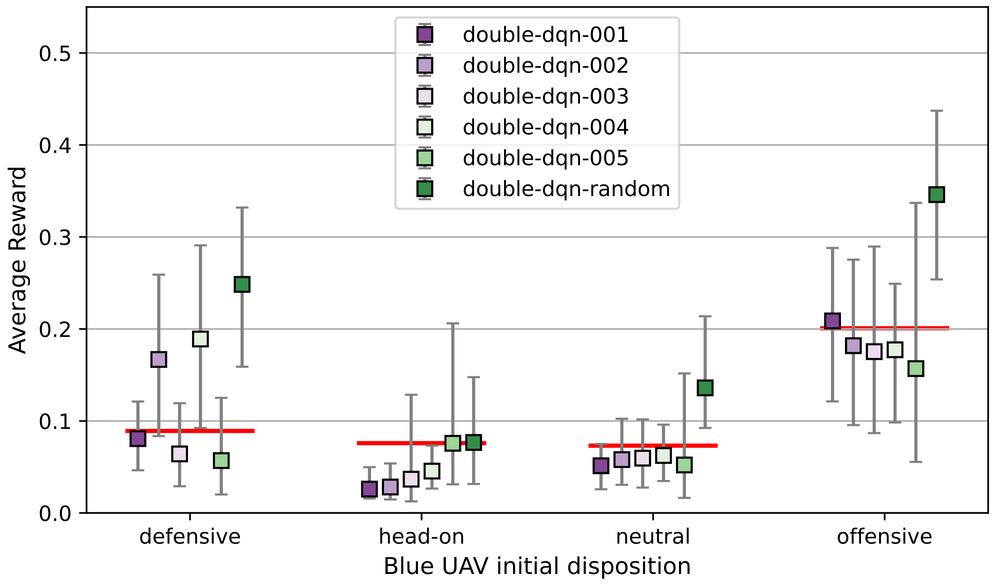


Figure 5.5: The performance of the double DQN agents against the Smart Pursuit baselines where all the agents learned for 20,000 episodes. The red horizontal lines are the scores of the baseline for each initial disposition. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

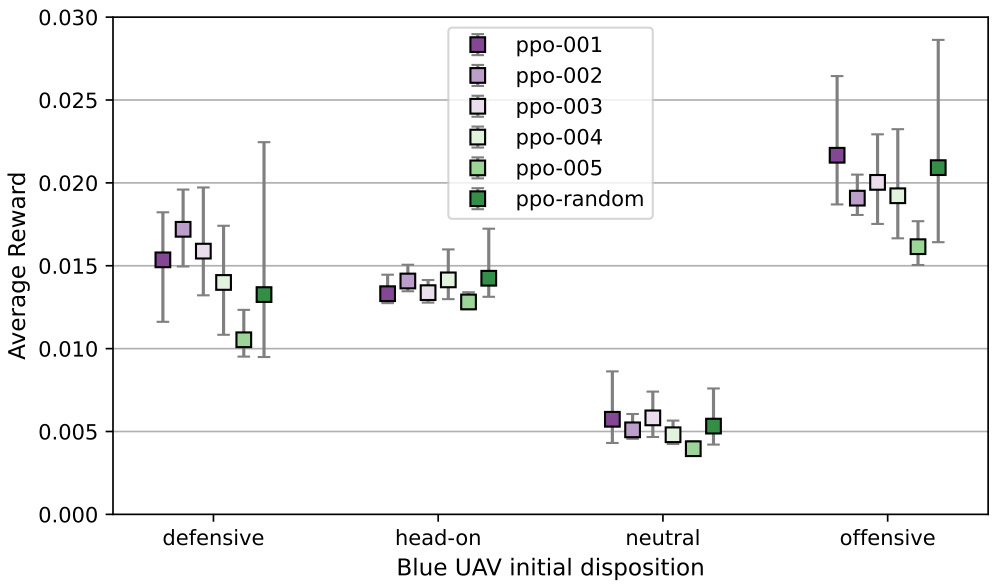


Figure 5.6: The performance of the PPO agents against the Smart Pursuit baselines where all the agents learned for 20,000 episodes. The baselines are not shown here due to the scale chosen to show the differences in average rewards among the agents. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

Agent	Mean	Median	Standard Deviation
dqn-001	36.19	36.81	4.17
dqn-002	36.88	35.96	5.45
dqn-003	37.76	37.12	4.03
dqn-004	38.15	38.12	1.96
dqn-005	38.24	38.06	1.97
dqn-random	39.46	39.46	0.09
dqn-wtn-001	34.48	29.93	8.08
dqn-wtn-002	36.47	32.84	7.05
dqn-wtn-003	32.49	33.31	1.54
dqn-wtn-004	32.88	32.54	0.52
dqn-wtn-005	33.52	33.42	0.28
dqn-wtn-random	39.42	39.39	0.19
ddqn-001	29.85	29.64	0.78
ddqn-002	30.11	30.63	0.75
ddqn-003	31.27	31.15	0.44
ddqn-004	29.83	28.82	1.73
ddqn-005	30.38	29.61	1.68
ddqn-random	39.60	39.51	0.18
ppo-001	23.11	23.14	0.65
ppo-002	20.56	20.43	0.54
ppo-003	21.95	23.60	3.88
ppo-004	23.46	23.24	1.46
ppo-005	25.40	25.29	0.85
ppo-random	21.18	21.18	0.31

Table 5.3: DQN, DQN-wTN, double DQN and PPO learning times for 20,000 episodes

5.3.3 Learning Times

Table 5.3 shows the time taken, in hours, by each agent to complete 20,000 learning episodess and Table 5.4 shows the cost per episode in seconds. Figures 5.7 and 5.8 show the means and confidence intervals of the bootstrap-resamples of the data in Tables 5.3 and 5.4, respectively. The learning times are similar within the DQN family of methods. PPO is much faster than the DQN family because the neural networks in PPO are updated relatively infrequently.

Agent	Mean	Median	Standard Deviation
dqn-001	6.52	6.63	0.75
dqn-002	6.64	6.47	0.98
dqn-003	6.80	6.68	0.73
dqn-004	6.87	6.86	0.35
dqn-005	6.88	6.85	0.35
dqn-random	7.10	7.10	0.02
dqn-wtn-001	6.21	5.39	1.45
dqn-wtn-002	6.56	5.91	1.27
dqn-wtn-003	5.85	6.00	0.28
dqn-wtn-004	5.92	5.86	0.09
dqn-wtn-005	6.03	6.02	0.05
dqn-wtn-random	7.10	7.09	0.03
ddqn-001	5.37	5.34	0.14
ddqn-002	5.42	5.51	0.13
ddqn-003	5.63	5.61	0.08
ddqn-004	5.37	5.19	0.31
ddqn-005	5.47	5.33	0.30
ddqn-random	7.13	7.11	0.03
ppo-001	4.16	4.17	0.12
ppo-002	3.70	3.68	0.10
ppo-003	3.95	4.25	0.70
ppo-004	4.22	4.18	0.26
ppo-005	4.57	4.55	0.15
ppo-random	3.81	3.81	0.05

Table 5.4: DQN, DQN-wTN, double DQN and PPO learning times per episode

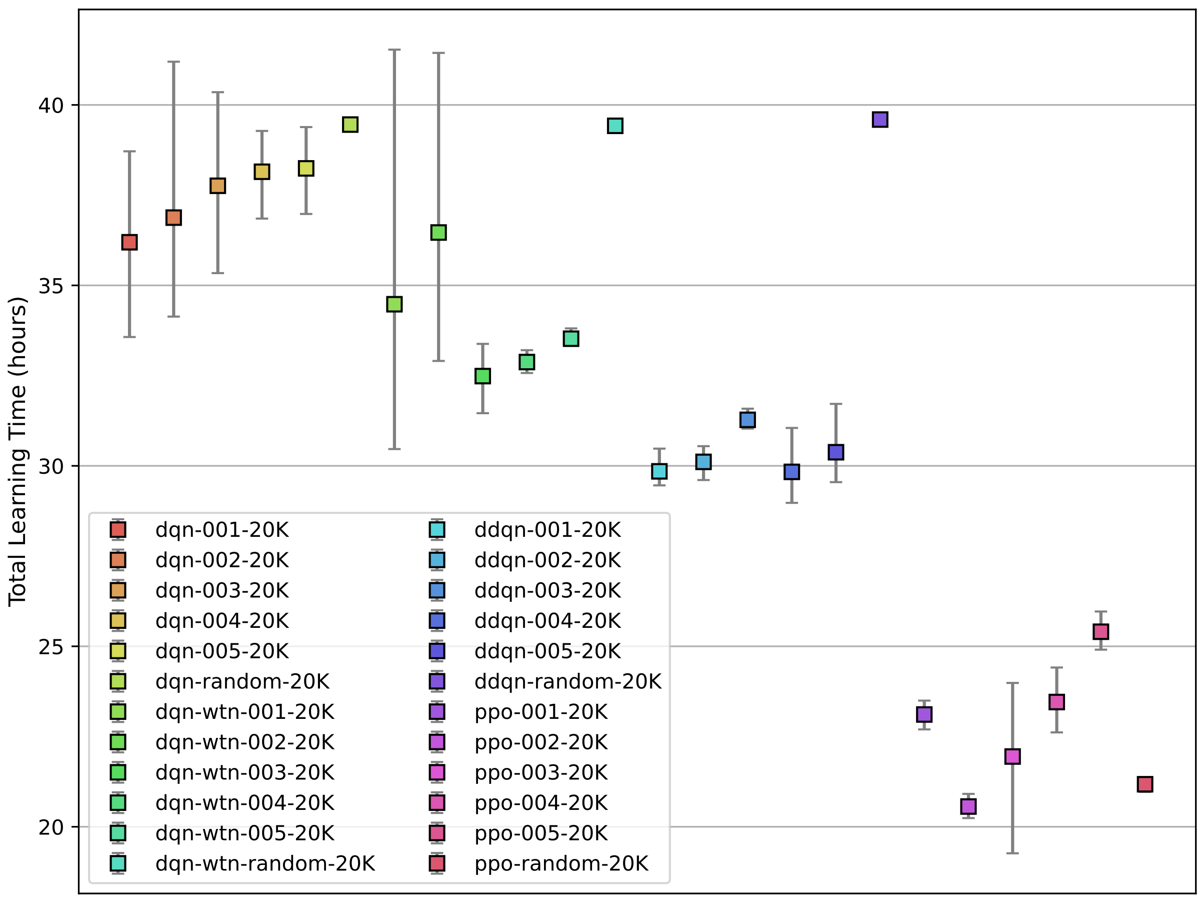


Figure 5.7: DQN, DQN-wTN, double DQN and PPO learning times in hours

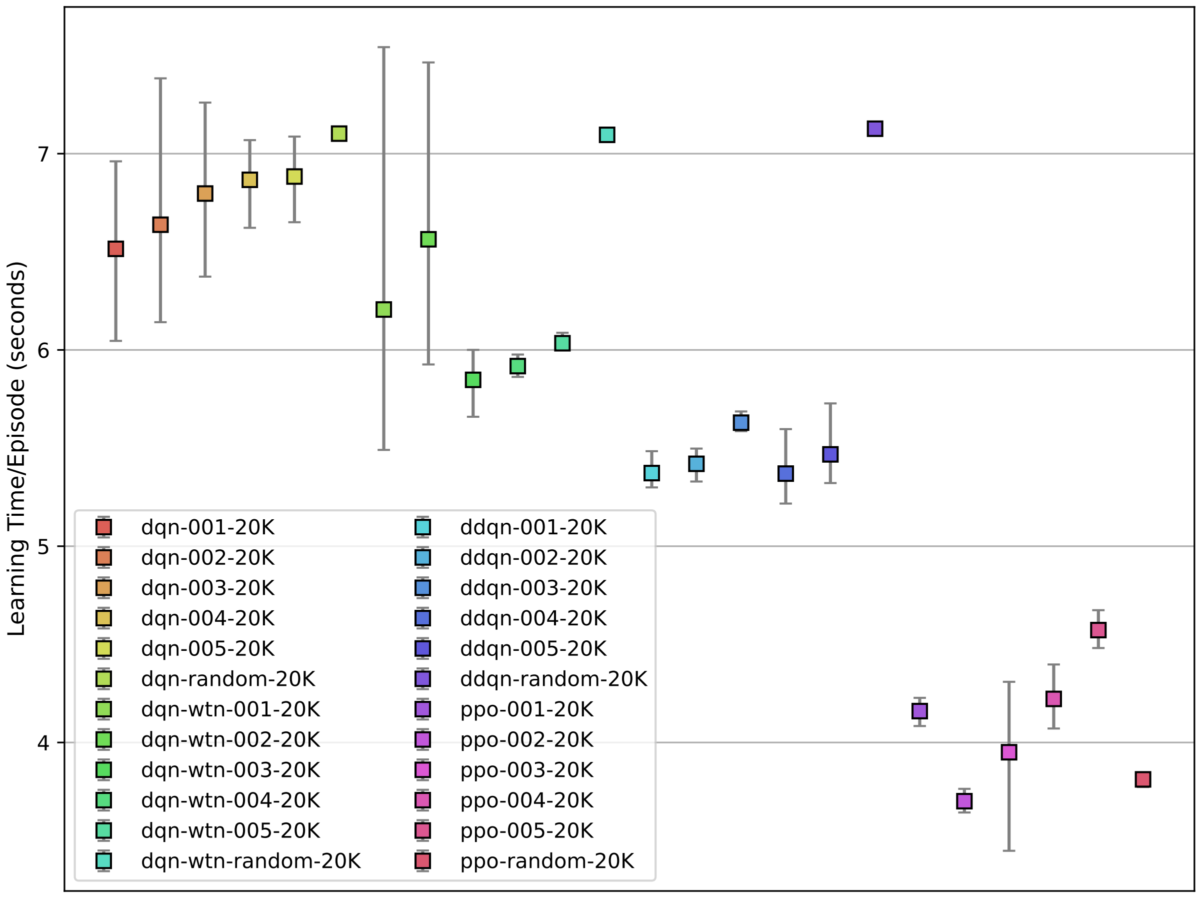


Figure 5.8: DQN, DQN-wTN, double DQN and PPO learning times per episode in seconds

5.3.4 Learning Instability

Just like the learning curves of the tabular methods in chapter 4, the learning curves for the deep RL methods in this chapter do not increase monotonically. In fact, the learning curves of deep RL agents can be very unstable and noisy, where the average reward goes up and down as learning progresses. This kind of instability was also observed in the original DQN paper. The authors, upon discovering such a phenomenon, asserted on Page 6, “The average total reward metric tends to be very noisy because small changes to the weights of a policy can lead to large changes in the distribution of states the policy visits” (Mnih et al., 2013).

DQN-wTN is not immune to this symptom either, as can be seen in (Mnih et al., 2015) and investigated by Van Hasselt et al. (van Hasselt et al., 2016). In fact, Van Hasselt et al. introduced double DQN to tackle this very same problem. It is shown that double DQN is able to improve on DQN-wTN over several ATARI games, but not all games (van Hasselt et al., 2016). This behaviour is also reported in other studies, including (Thrun and Schwartz, 1993) and (Anschel et al., 2017).

Similar instability is found in the deep RL methods in this chapter. The learning curves shown in Figure 5.1 hide the severity of the instability because all data points are averaged over ten random seeds. Such instability is more apparent in Figure 5.9, where individual learning curves of DQN, DQN-wTN and double DQN agents are shown, and in Figure 5.10, which shows an individual learning of a PPO agent. This instability only occurs in a minority of learning trials and is only apparent in individualised learning plots that have not been averaged out with reward scores from other trials.

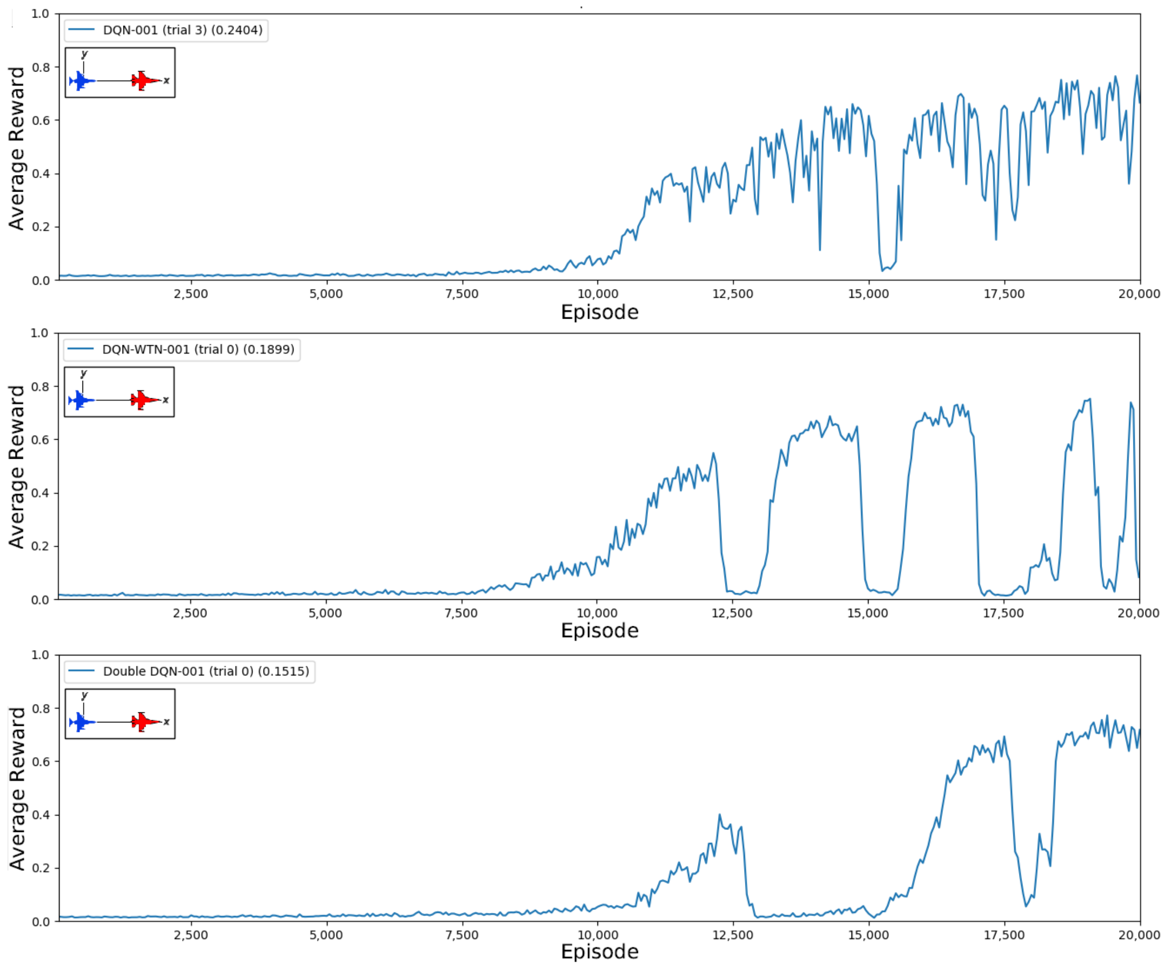


Figure 5.9: Instability of the DQN family of methods. The top learning curve comes from a trial of a DQN agent, the middle one from a DQN-wTN agent and the bottom one from a double DQN agent. The charts have been smoothed out by averaging every fifty data points into one new data point and the average rewards have been increased by 0.5.

This kind of instability has been identified by Sutton and Barto (Sutton and Barto, 2018) as being due to the influence of a combination of three factors knows as the ‘deadly triad’:

- function approximation, which is the use of neural networks,

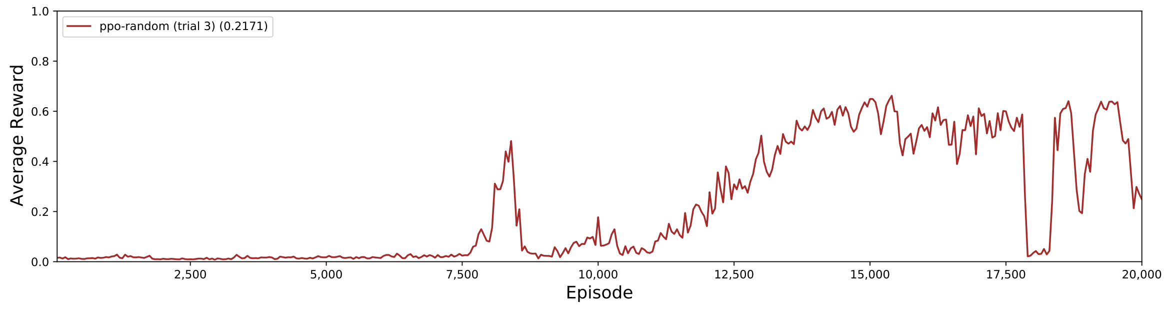


Figure 5.10: Instability in PPO. The learning curve comes from a trial of a PPO agent. The chart has been smoothed out by averaging every fifty data points into one new data point and the average rewards have been increased by 0.5.

- bootstrapping, in this case the estimates of the Q values are updated partially using existing estimates,
- off-policy learning, in which the target policy is different from the behaviour policy.

The cause of this instability is explained by Achiam et al. (Achiam et al., 2019), who examined how Q values change under a standard update in the DQN family of methods. They theorised that the update was either a contraction or an expansion in the sup norm. When it is a contraction the deep neural network behaves stably, and when it is an expansion the neural network is expected to diverge.

5.3.5 The Effect of C in DQN-wTN

C is the frequency of the target network update. Setting C to 100 means the target network is synchronised with the main network (its weights are updated using the weights of the main network) every 100 timesteps.

The four charts in Figure 5.11 show the effect of C in four different learning sessions conducted using the same random seed but with different values of C (10, 50, 500, 700). The higher the value of C, the more the target network is allowed to diverge from the main network. In this research project, the divergence of networks introduces divergence in learning which is more apparent in higher values of C.

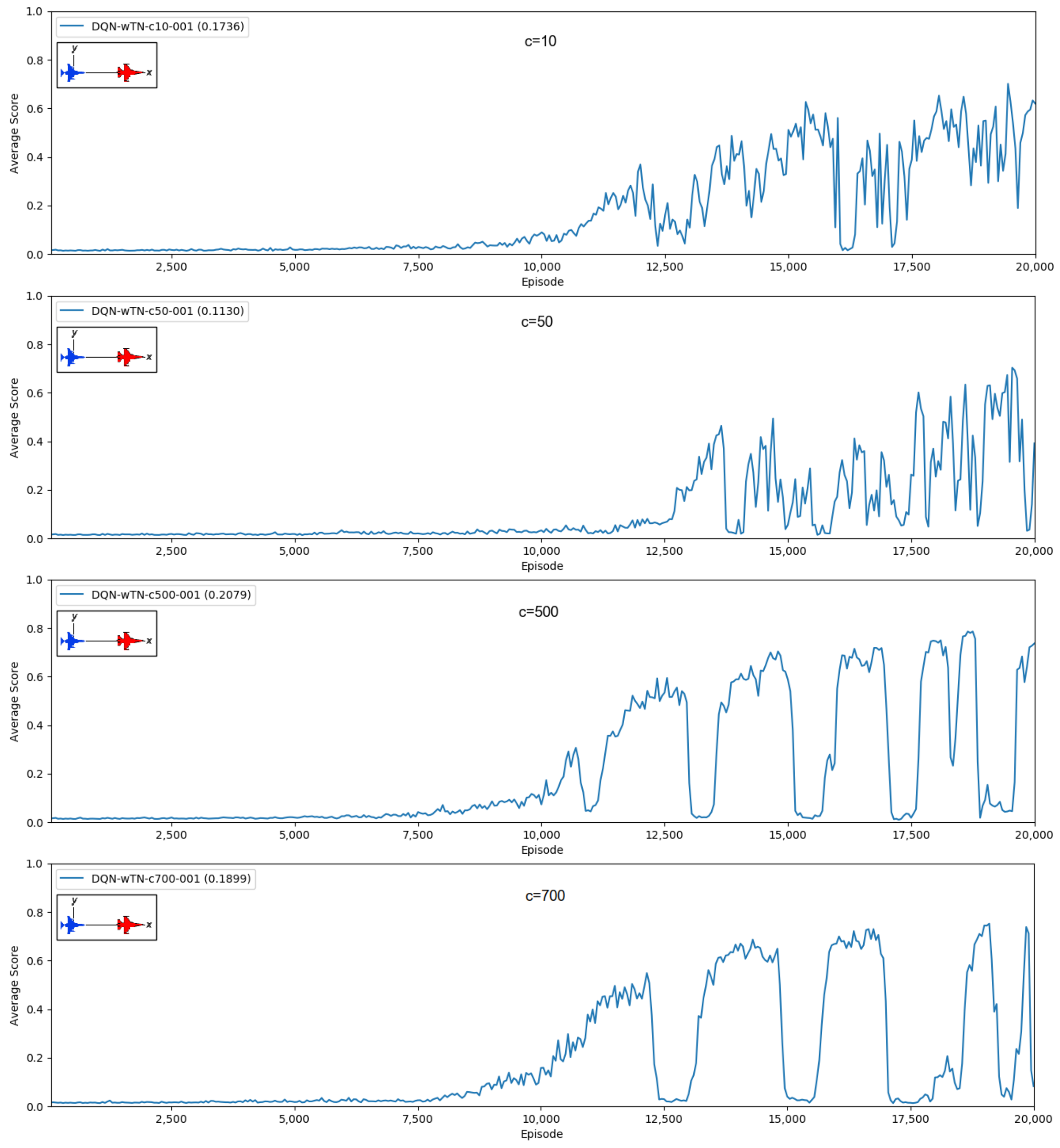


Figure 5.11: DQN-wTN learning curves with different c values (10, 50, 500 and 700). Every fifty data points are averaged into a new data point

5.4 Summary

This chapter applies four deep RL methods—DQN, DQN-wTN, double DQN and PPO—to address the air combat manoeuvring domain. DQN, DQN-wTN and double DQN use Q-learning as their learning algorithm and proved to be able to successfully learn policies to address the problem, albeit with different degrees of success. As all the deep RL methods contain elements of the deadly triad, they suffer from instability that was also observed in the original DQN paper (Mnih et al.,

2013) and its variant DQN-wTN (Mnih et al., 2015). The cause of this instability is explained in (Achiam et al., 2019) and was attempted to be fixed in (van Hasselt et al., 2016) by introducing double DQN, an effort that managed to improve the performance of some ATARI games, but not all games.

In this thesis project, the DQN agents obtained the highest average rewards in four out of six learning initial positions (001, 002, 004 and random). The DQN-wTN agents excelled in two initial positions (initial positions 003 and 005). The double DQN agents learned the most poorly among all the DQN family of agents. Agents that learned successfully generally perform well in the performance tests. In this chapter, the DQN agents are the best performers. The better performance of the DQN agents over the DQN-wTN agents can be explained by the choice of the update frequency C of the target network in the DQN-wTN agents. In all experiments C was set to a relatively high value of 700. Figure 5.11 shows that a higher value of C introduces more divergence in learning.

The double DQN agents are the worst performers among all the DQN family of agents. A possible cause of its poor performance compared to the DQN and DQN-wTN agents is under-training. This is because at every timestep only one of the two neural networks in a double DQN agent is updated. This means the network that is later saved as a policy for a double DQN agent is updated approximately half the number of updates received by the network in a DQN or the main network in a DQN-wTN agent that learned for the same number of episodes. On the other hand, the instability that was supposed to be the main problem for which Double DQN has been designed to alleviate only occurred mildly in the experiments in this chapter. The learning curves in Figure 5.1, that show that in most cases the double DQN agents reached a peak that is lower than those achieved by the DQN and DQN-wTN agents, are indications of under-training.

PPO performs worse than DQN, which can be explained by the "under-training" and the absence of effort to de-correlate samples, as is done in DQN and its variants. It is also worth noting that the authors of the PPO paper tested their techniques with continuous control problems and the PPO experiment in this chapter is conducted with a discrete action space.

With regard to learning times, all the DQN family of methods have similar learning times. PPO learns faster than DQN and its variants, thanks to its design to only update its neural networks when the buffer is full.

Chapter 6

Discrete-to-Deep Methods

Table-based reinforcement learning (RL) techniques are fast and easy to implement but suffer from the classic curse-of-dimensionality, which means they are powerless to address problems with high-dimensional spaces. This was shown in Chapter 4, where Q(λ) agents had difficulty learning. High-dimensional problems normally require approximate solution RL methods, namely algorithms that employ function approximation, a common technique in supervised learning (SL) (Sutton and Barto, 2018). A family of approximate solution RL methods that use neural networks as function approximators are called neural RL or deep RL. In Chapter 5, deep RL agents were successful in learning good policies for the air combat manoeuvring problem even though they were trained with 10% the number episodes used to train the tabular agents in Chapter 4. However, deep RL methods learn much more slowly than tabular agents.

This chapter introduces two novel methods that combine the advantages of tabular and deep RL methods: discrete-to-deep supervised policy learning (D2D-SPL) and discrete-to-deep supervised Q-value learning (D2D-SQL). In both methods, a tabular method is used to generate off-policy data to feed a neural network. Two experiments are conducted to test the new methods. Experiment 1 compares the methods with other existing methods, such as DQN (Mnih et al., 2013), double DQN (van Hasselt et al., 2016) and A3C (Mnih et al., 2016). The agents in Experiment 1 obtain their off-policy data from table-based actor-critic with eligibility traces (ACET) agents trained using initial position 001. The agents in this experiment use a small buffer containing 500 off-policy samples. Experiment 2 is designed to use a big buffer of 2,000 samples and be similar to the experiments in Chapters 5 and 6 so the results can be compared with the results from those chapters. Off-policy data for these agents come from off-policy data of table-based ACET agents that learned using random initial positions. Aside from the different initial positions used to generate off-policy data for the agents, the other difference between Experiment 1 and

Experiment 2 is the numbers of episodes the tabular ACET agents are run to generate off-policy data for D2D-SPL and D2D-SQL. For Experiment 1 it is 20,000 episodes and for Experiment 2 it is 50,000 episodes.

6.1 Overview of the Methods Used

D2D-SPL is a variant of supervised policy learning designed to train a neural network as a function approximator to solve RL problems with both continuous state spaces and low-dimensional state variables. D2D-SPL is simple and based on the standard ACET algorithm (Barto et al., 1983). ACET is chosen instead of Q(λ) due to the latter's poor performance in solving problems with a continuous state space, as discussed in Chapter 4. D2D-SPL has proven to be able to successfully solve two classic RL problems, Cartpole and Lunar Lander (Kurniawan et al., 2021).

D2D-SQL is similar to D2D-SPL, but adds a second reinforcement phase that uses a neural RL method such as DQN (Mnih et al., 2013). DQN has achieved impressive feats but is notoriously slow. The original DQN, for example, needed 38 days to learn a single game of ATARI (Mnih et al., 2013). D2D-SPL and D2D-SQL combine RL and SL and aim at acquiring the generalisability of a neural network at a cost nearer to that of a tabular method.

The differences between the two methods are summarised in Table 6.1.

	Reinforcement Phase	Supervised Phase	Second Reinforcement Phase
D2D-SPL	tabular method	classification	(none)
D2D-SQL	tabular method	regression	approximate solution method

Table 6.1: Comparison of D2D-SPL and D2D-SQL

D2D-SPL and D2D-SQL are based on the idea that a fast-learning tabular method can generate off-policy data to accelerate learning in deep RL. D2D-SPL uses the data to train a classifier which is then used as a controller for the RL problem. D2D-SQL uses the data to initialise a neural network which is then allowed to continue learning using another RL method. D2D-SPL and D2D-SQL are suitable for continuous-space environments with low-dimensional state variables.

Techniques whereby data is generated from an RL policy to train a neural network have been used in studies called guided policy search (Levine et al., 2015) and supervised policy learning (Chebotar et al., 2016). Both studies are different from D2D-SPL and D2D-SQL because they used model-based RL, whereas D2D-SPL and D2D-SQL use ACET, which is model-free. Additionally, they used data from multiple policies to train a classifier whereas D2D-SPL and D2D-SQL only require one policy. Furthermore, the method proposed in Chebotar et al. (2016) requires an optimisation strategy before data can be fed to the neural network. D2D-SPL and D2D-SQL do not require optimisation between the RL part and the classifier. A technique called supervised actor-critic (Rosenstein and Barto, 2002) and its variant (Wang et al., 2018) also combine RL and SL. In this architecture, the actor receives a signal from the critic as well as from a neural network. By contrast, D2D-SPL is a two-step process and D2D-SQL a three-step process.

6.1.1 Discrete-to-Deep Supervised Policy Learning (D2D-SPL)

Tabular methods such as Q-learning and actor-critic learn relatively fast because Q-values in these methods are updated once every timestep. On the other hand, deep RL methods are slow learners because an epoch in neural network training consists of many passes. The idea behind D2D-SPL is to reduce the learning time of the neural network by training it with data obtained more cheaply from a tabular method.

D2D-SPL is suitable for solving continuous-state RL problems with discrete actions. It uses off-policy data from an actor-critic policy to train a neural network that thereafter can be used as a controller for the RL problem. D2D-SPL works in two phases, a reinforcement phase and a supervised phase. First, it discretises the continuous state space and learns a policy using actor-critic with eligibility traces (Barto et al., 1983). This policy, which is based on coarse discretisation, should be able to be learned more quickly than a policy based on the full, continuous state-space. Second, it uses data generated during RL to train a classifier. Not all samples are used. The method selects from each discrete state an input vector and the action with the highest numerical preference as an input/target pair. The classifier learns when all input/target pairs are presented to it at the same time, thus eliminating the need for online learning.

D2D-SPL starts by discretising the state space into discrete states, whose number varies depending on the complexity of the problem. The air combat manoeuvring problem in this thesis requires 14,000 states to learn a good policy. As will be seen shortly, the number of discrete states is also the maximum number of samples for the second-stage supervised learning. The number of continuous state variables is the same as the number of input nodes to the classifier.

The discrete states are then used in ACET learning. In every episode state variables are grouped by discrete state and the total reward of the episode is stored in memory. Once learning is finished, the top 5% of episodes having the highest total rewards are selected and the values of each state variable in each discrete state are averaged. The selected samples are used as inputs to the neural network. The number of samples n is less than or equal to the number of discrete states. n can be lower than the number of discrete states because discrete states that were never visited during reinforcement learning are filtered out.

Algorithm 12 shows the reinforcement phase of D2D-SPL. It is basically the ACET algorithm (Barto et al., 1983) with a buffer for storing tuples of state variable values and the number of times a discrete state is visited. The highlighted lines are steps that are specific to D2D-SPL. The buffer and the resulting policy are then used as inputs to the supervised_phase function in Algorithm 13, which shows how samples are selected and prepared for training the classifier. To save memory, buffer sorting and trimming can be done every n episodes during the reinforcement phase.

Algorithm 12: D2D-SPL reinforcement phase. It is basically actor-critic and lines specific to D2D-SPL are highlighted.

```
    Input: a differentiable policy parameterisation $\pi(a|s,\boldsymbol{\theta})$
    Input: a differentiable state-value function parameterisation $\hat{v}(s,\mathrm{w})$
    Parameter(s): number of discrete states $N_{ds} > 0$, number of actions $N_a > 0$, number of
                  state variables $N_{sv} > 0$
    Parameter(s): step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathrm{w}} > 0$
    Parameter(s): trace decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathrm{w}} \in [0,1]$
 1  Initialise policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{N_{ds}xN_a}$ and state-value weights $\mathrm{w} \in \mathbb{R}^{N_{ds}}$ (e.g, to 0);
 2  Initialise empty buffer $B$;
 3  for each episode do
 4      Initialise $S$ (first state of episode);
 5      Initialise $M_{sv} \in \mathbb{R}^{N_{ds}xN_{sv}}$ for storing aggregate values of state variables;
 6      Initialise $M_{sc} \in \mathbb{R}^{N_{ds}}$ for keeping track of the number of times a value has been added
         to $M_{sv}$;
 7      $\mathrm{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ ($N_{ds}$x$N_a$-component eligibility trace matrix);
 8      $\mathrm{z}^{\mathrm{w}} \leftarrow \mathbf{0}$ ($N_{ds}$-component eligibility trace vector);
 9      $I \leftarrow 1$ ;
10      $R_{total} \leftarrow 0$;
11      for each step of episode or until S is terminal do
12          $S_{disc} \leftarrow$ discretise($S$);
13          $A \sim \pi(.|S_{disc},\boldsymbol{\theta})$;
14          Take action $A$, observe $S'$, $R$;
15          $M_{sv}[S_{disc}] \leftarrow M_{sv}[S_{disc}] + S$;
16          $M_{sc}[S_{disc}] \leftarrow M_{sc}[S_{disc}] + 1$;
17          $R_{total} \leftarrow R_{total} + R$;
18          $S'_{disc} \leftarrow$ discretise($S'$);
19          $\delta \leftarrow R + \gamma\hat{v}(S'_{disc},\mathrm{w}) - \hat{v}(S_{disc},\mathrm{w})$   (if $S'$ is terminal, $\hat{v}(S'_{disc},\mathrm{w}) \doteq 0$);
20          $\mathrm{z}^{\mathrm{w}} \leftarrow \gamma\lambda^{\mathrm{w}}\mathrm{z}^{\mathrm{w}} + \nabla\hat{v}(S_{disc},\mathrm{w})$;
21          $\mathrm{z}^{\boldsymbol{\theta}} \leftarrow \gamma\lambda^{\boldsymbol{\theta}}\mathrm{z}^{\boldsymbol{\theta}} + I\nabla\ln\pi(A|S_{disc},\boldsymbol{\theta})$;
22          $\mathrm{w} \leftarrow \mathrm{w} + \alpha^{\mathrm{w}}\delta\mathrm{z}^{\mathrm{w}}$;
23          $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\delta\mathrm{z}^{\boldsymbol{\theta}}$;
24          $I \leftarrow \gamma I$;
25          $S \leftarrow S'$;
26      end
27      Add ($R_{total}$, $M_{SV}$, $M_{SC}$) to $B$;
28  end
29  Call $d2dspl\_supervised\_phase(B, \boldsymbol{\theta})$;
```

Figure 6.1 shows an example of how data is consolidated. For simplicity it shows a case where there are only four discrete states (each represented by a box in the top diagram) and there are

Algorithm 13: D2D-SPL supervised phase

```
   Input: buffer containing tuples of ($R_{total}$, $M_{sv}$, $M_{cv}$)
   Input: policy parameter $\boldsymbol{\theta}$
   Parameter(s): number of discrete states $N_{ds} > 0$, number of state variables $N_{sv} > 0$
 1 Sort $B$ by $R_{total}$ in descending order and delete the bottom 95% elements;
 2 Initialise $ConsM_{sv} \in \mathbb{R}^{N_{ds} x N_{sv}}$ for storing consolidated values of $M_{sv}$;
 3 Initialise $ConsM_{sc} \in \mathbb{R}^{N_{ds}}$ for storing consolidated values of $M_{sc}$;
 4 Initialise $T \in \mathbb{R}^{N_{ds}}$ for storing targets for the classifier;
 5 for each element $E$ in $B$ do
 6     $R_{total}, M_{sv}, M_{sc} \leftarrow E$;
 7     for $i \leftarrow 0$ To $N_{ds}$ - 1 do
 8         $ConsM_{sv}$[i] $\leftarrow ConsM_{sv}[i] + M_{sv}[i]$;
 9         $ConsM_{sc}$[i] $\leftarrow ConsM_{sc}[i] + M_{sc}[i]$;
10     end
11 end
12 for $i \leftarrow N_{sd}$ - 1 To 0 do
13     if $ConsM_{sc}[i] == 0$ then
14         Delete element $i$ from $ConsM_{sv}$ and $T$;
15     else
16         $ConsM_{sv}[i] \leftarrow ConsM_{sv}[i] / ConsM_{sc}[i]$;
17         $T[i] \leftarrow$ index of the largest value of $\boldsymbol{\theta}[i]$
18     end
19 end
20 Use $ConsM_{sv}$ as inputs and $T$ as targets to train classifier;
```

two state variables in each state. At every timestep, the values of the state variables of the visited state are recorded. The data is collected after eight timesteps, in which discrete state 1 has been visited three times with state variable tuples (1,1), (2,2) and (3,1); discrete state 2 visited once with state variable tuple (6,2) and so on. Since only the average values of state variables in each discrete state are used and not the individual tuple values, the algorithm can save memory by just keeping the totals of state variable values in every discrete state in M_{sv} and the number of times that state is visited in M_{sc}. Along with the total reward for the episode, the tuple (R_{TOTAL}, M_{sv}, M_{sc}) is inserted to buffer B. At the end of Algorithm 12, the number of tuples in buffer B will be the same as the number of episodes run.

Buffer B and the actor-critic policy are passed to function *supervised_phase* in Algorithm 13. The buffer contains the fore-mentioned tuples and the policy contains numerical preferences (one for each action) for each discrete state. The objective of this function is to produce an input/target pair for every discrete state, excluding states that were never visited during training. The function starts by selecting the top 5% of tuples with the highest total rewards in the buffer and consolidating the selected tuples into at most one tuple for each discrete state. Data is consolidated by summing the values of each state variable in a discrete state and dividing them by the number of times the state was visited. For each discrete state in the policy, the action with

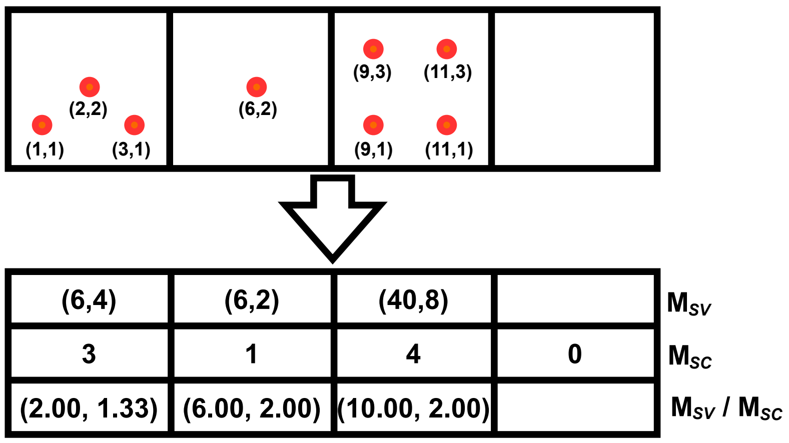


Figure 6.1: Data consolidation in D2D-SPL

the highest numerical preference is selected as a target. All states that are not present in the input set are removed from the training set and the training set is passed to the classifier.

6.1.2 Discrete-to-Deep Supervised Q-Value Learning (D2D-SQL)

D2D-SPL uses off-policy data to train a classifier which is then used as a controller to solve an RL problem. This section will investigate whether the introduction of a subsequent reinforcement phase will improve performance. DQN (Mnih et al., 2013) or a variant of it seems a good choice for the second reinforcement phase, considering DQN has been a groundbreaking deep RL method that is still popular today and worked best of the deep RL agents in Chapter 5. This extension to D2D-SPL is called discrete-to-deep supervised Q-value learning (D2D-SQL) because DQN is based on Q-learning, which stores its policy as Q-values. D2D-SQL adds another reinforcement phase to the two phases of D2D-SPL, making D2D-SQL a three-phase method. Another difference between D2D-SPL and D2D-SQL is, because the neural network in DQN acts as a regression model rather than a classifier, off-policy data in D2D-SQL are presented as pairs of states/numerical preferences instead of pairs of states/actions. This presentation of the off-policy data is needed for compatibility with Q-learning which is a value-based RL method.

The first reinforcement phase of D2D-SQL, shown in Algorithm 14, is similar to Algorithm 12, except that there are two more sets of values, the next states and rewards, that are stored in the buffer. The lines specific to D2D-SQL are highlighted. The supervised phase of D2D-SQL, shown in Algorithm 15, is similar to Algorithm 13, but uses the action numerical preferences, rather than the actions themselves, as the target to build a regression model. The highlighted lines in Algorithm 15 are specific to D2D-SQL. The second reinforcement phase of D2D-SQL is implemented using a DQN agent (Mnih et al., 2013) or one of its variants.

Algorithm 14: D2D-SQL reinforcement phase. The highlighted lines are the additional steps compared to D2D-SPL

```
   Input: a differentiable policy parameterisation $\pi(a|s,\boldsymbol{\theta})$
   Input: a differentiable state-value function parameterisation $\hat{v}(s,\mathrm{w})$
   Parameter(s): number of discrete states $N_{ds} > 0$, number of actions $N_a > 0$, number of
                 state variables $N_{sv} > 0$
   Parameter(s): step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathrm{w}} > 0$
   Parameter(s): trace decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathrm{w}} \in [0,1]$
1  Initialise policy parameter $\boldsymbol{\theta} \in \mathbb{R}^{N_{ds}xN_a}$ and state-value weights $\mathrm{w} \in \mathbb{R}^{N_{ds}}$ (e.g, to 0);
2  Initialise empty buffer $B$;
3  for each episode do
4      Initialise $S$ (first state of episode);
5      Initialise $M_{sv} \in \mathbb{R}^{N_{ds}xN_{sv}}$ for storing aggregate values of state variables;
6      Initialise $M_{sc} \in \mathbb{R}^{N_{ds}}$ for keeping track of the number of times a value has been added
         to $M_{sv}$;
7      Initialise $M_{svn} \in \mathbb{R}^{N_{ds}xN_{sv}}$ for storing aggregate values of next state variables;
8      Initialise $M_r \in \mathbb{R}^{N_{ds}}$ for storing aggregate rewards;
9      $\mathrm{z}^{\boldsymbol{\theta}} \leftarrow \mathbf{0}$ ($N_{ds}$x$N_a$-component eligibility trace matrix);
10     $\mathrm{z}^{\mathrm{w}} \leftarrow \mathbf{0}$ ($N_{ds}$-component eligibility trace vector);
11     $I \leftarrow 1$ ;
12     $R_{total} \leftarrow 0$;
13     for each step of episode or until $S$ is terminal do
14         $S_{disc} \leftarrow$ discretise($S$);
15         $A \sim \pi(.|S_{disc},\boldsymbol{\theta})$;
16         Take action $A$, observe $S'$, $R$;
17         $M_{sv}[S_{disc}] \leftarrow M_{sv}[S_{disc}] + S$;
18         $M_{sc}[S_{disc}] \leftarrow M_{sc}[S_{disc}] + 1$;
19         $M_{svn}[S_{disc}] \leftarrow M_{svn}[S_{disc}] + S'$;
20         $M_r[S_{disc}] \leftarrow M_r[S_{disc}] + R$;
21         $R_{total} \leftarrow R_{total} + R$;
22         $S'_{disc} \leftarrow$ discretise($S'$);
23         $\delta \leftarrow R + \gamma\hat{v}(S'_{disc},\mathrm{w}) - \hat{v}(S_{disc},\mathrm{w})$    (if $S'$ is terminal, $\hat{v}(S'_{disc},\mathrm{w}) \doteq 0$);
24         $\mathrm{z}^{\mathrm{w}} \leftarrow \gamma\lambda^{\mathrm{w}}\mathrm{z}^{\mathrm{w}} + \nabla\hat{v}(S_{disc},\mathrm{w})$;
25         $\mathrm{z}^{\boldsymbol{\theta}} \leftarrow \gamma\lambda^{\boldsymbol{\theta}}\mathrm{z}^{\boldsymbol{\theta}} + I\nabla\ln\pi(A|S_{disc},\boldsymbol{\theta})$;
26         $\mathrm{w} \leftarrow \mathrm{w} + \alpha^{\mathrm{w}}\delta\mathrm{z}^{\mathrm{w}}$;
27         $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha^{\boldsymbol{\theta}}\delta\mathrm{z}^{\boldsymbol{\theta}}$;
28         $I \leftarrow \gamma I$;
29         $S \leftarrow S'$;
30     end
31     Add ($R_{total}$, $M_{SV}$, $M_{SC}$) to $B$;
32 end
33 Call $d2dsql\_supervised\_phase(B, \boldsymbol{\theta})$;
```

Algorithm 15: D2D-SQL supervised phase. The highlighted lines are the additional steps compared to D2D-SPL

```
Input: buffer containing tuples of ($R_{total}$, $M_{sv}$, $M_{cv}$, $M_{svn}$, $M_r$)
Input: policy parameter $\theta$
Parameter(s): number of discrete states $N_{ds} > 0$, number of state variables $N_{sv} > 0$
1  Sort $B$ by $R_{total}$ in descending order and delete the bottom 95% elements;
2  Initialise $ConsM_{sv} \in \mathbb{R}^{N_{ds}xN_{sv}}$ for storing consolidated values of $M_{sv}$;
3  Initialise $ConsM_{sc} \in \mathbb{R}^{N_{ds}}$ for storing consolidated values of $M_{sc}$;
4  Initialise $ConsM_{svn} \in \mathbb{R}^{N_{ds}xN_{svn}}$ for storing consolidated values of $M_{svn}$;
5  Initialise $ConsM_r \in \mathbb{R}^{N_{ds}}$ for storing consolidated values of $M_r$;
6  for each element $E$ in $B$ do
7      $R_{total}$, $M_{sv}$, $M_{sc}$, $M_{svn}$, $M_r$, $\leftarrow E$;
8      for $i \leftarrow 0$ To $N_{ds}$ - 1 do
9          $ConsM_{sv}$[i] $\leftarrow ConsM_{sv}[i] + M_{sv}[i]$;
10         $ConsM_{sc}$[i] $\leftarrow ConsM_{sc}[i] + M_{sc}[i]$;
11         $ConsM_{svn}$[i] $\leftarrow ConsM_{svn}[i] + M_{svn}[i]$;
12         $ConsM_r$[i] $\leftarrow ConsM_r[i] + M_r[i]$;
13     end
14 end
15 for $i \leftarrow N_{sd}$ - 1 To 0 do
16     if $ConsM_{sc}[i] == 0$ then
17         Delete element $i$ from $ConsM_{sv}$;
18         Delete element $i$ from $ConsM_{svn}$ and $ConsM_r$;
19     else
20         $ConsM_{sv}[i] \leftarrow ConsM_{sv}[i] / ConsM_{sc}[i]$;
21         $ConsM_{svn}[i] \leftarrow ConsM_{svn}[i] / ConsM_{sc}[i]$;
22         $ConsM_r[i] \leftarrow ConsM_r[i] / ConsM_{sc}[i]$;
23     end
24 end
25 Use $ConsM_{sv}$, $ConsM_r$ as inputs and $ConsM_{svn}$ as targets to train neural network;
```

6.2 Overview of the Experiments

Two experiments, Experiment 1 and Experiment 2, are conducted for this chapter. The differences between the two are given in Table 6.2. In Experiment 1, an ACET agent is run 20,000 episodes each to generate a batch of 500 off-policy samples that are then used by D2D-SPL and D2D-SQL agents. Another ACET agent learning for double the number of episodes of the base ACET agent is also tested for comparison. In addition, separate DQN, double DQN and asynchronous advantage actor critic (A3C) tests are done for more comparison. This entire process is repeated for ten independent trials.

Experiment 2 is conducted to examine the effect of increasing the duration of the initial phase and the size of the buffer used to accumulate the training data for the supervised phase. This experiment also allows for a fairer comparison against the agents developed in earlier chapters. In Experiment 2, an ACET agent is run for 50,000 episodes to generate 2,000 off-policy samples. The policies from these ACET agent are called AC 50K. The off-policy samples generated by the ACET agent are used by D2D-SPL and D2D-SQL agents. This entire process is repeated for ten independent trials. The results are compared with ACET and deep RL agents. First, the results are compared with one of the ACET agents discussed in Chapter 4, all of which learned for 200,000 episodes or four times the number of episodes assigned to AC 50K. Next, the results are compared with the best performing DQN agents from Chapter 5.

	Experiment 1	Experiment 2
Number of episodes	20,000	50,000
Initial positions used to generate off-policy data	001	random
Number of off-policy samples	500	2,000
Deep RL method for D2D-SQL	double DQN	DQN
Result comparison	with independent deep RL tests	with results from Chapters 5 & 6

Table 6.2: Comparison of the experiments

6.3 Experiment 1: Small Buffer

In this experiment, an ACET agent, similar to the one in Chapter 4, is used to generate off-policy data for training a D2D-SPL and a D2D-SQL agents. The policies are then used to perform performance tests and the results are compared to the results of several deep RL agents that use different architectures to those discussed in Chapter 5. All the tabular and deep RL agents are trained using initial position 001 as explained in Chapter 3 and an episode consists of 700 timesteps.

6.3.1 Experimental Setup

For the D2D-SPL tests, the ACET agent is run for 20,000 episodes ten times using ten different random seeds, resulting in ten policies collectively code-named AC 20K. The same instances of the agent are then run for another 20,000 episodes, resulting in ten new policies code-named AC 40K. The base policies AC 20K are also used for D2D-SPL. Separately, for comparison purposes, DQN agents are trained for 20,000 and 40,000 episodes, double DQN agents for 20,000 and 40,000 episodes and asynchronous advantage actor-critic (A3C) agents for 20,000 and 200,000 episodes algorithms ten times, to generate policies code-named DQN 20K, DQN 40K, DDQN 20K, DDQN 40K, A3C 20K and A3C 200K, respectively. The choice for 200,000 episodes for the second A3C solution is due to the fact that A3C is data inefficient and need more episodes to achieve comparable scores (Wang et al., 2016).

For the D2D-SQL tests, a D2D-SQL agent is developed that encapsulates an ACET agent and a double DQN (DDQN) agent from the D2D-SPL experiments. For the first reinforcement phase, the ACET agent is trained for 20,000 episodes to generate off-policy data, in the same way as data for AC 20K is obtained for the D2D-SPL test. The difference is the next states and rewards are saved. The data from the tabular RL agent is then used to train the main neural network in the DDQN agent, using states as the input and action numerical preferences as the target. This is the supervised phase of D2D-SQL. The neural network is trained until a minimum loss of 0.001 is achieved. Next, for the second reinforcement phase, the off-policy data is used to populate the DDQN agent's replay memory and the main neural network of the DDQN agent is initialised with the weights from the supervised neural network from the supervised phase. The DDQN agent is then trained using the DDQN algorithm (van Hasselt et al., 2016) for 10,000 episodes.

Two tests are conducted that differ in how often the D2D-SQL agent explores. The first test uses an ϵ-greedy function that is a linear annealing function returning 0.1 for the first episode and 0.01 for the last episode. The test identifier for this test is D2D-SQL 10K (af), af standing for annealing function. The second test uses a fixed value of 0.05 as the ϵ value and this test is called D2D-SQL 10K (fe), where fe stands for fixed epsilon.

State Discretisation

Because ACET is a tabular method and the air combat problem has a continuous state space, the states need to be discretised. The discretisation strategy used in this experiment is the same as the one in Section 4.2.1 where a state is discretised into one of 14,000 discrete states.

The Action Space

The ACET agents implemented and tested in this chapter use the same action space as the one used in Chapter 4. There are five actions in the action space:

- 10° turn to the left
- 10° turn to the right
- increase speed by 10%
- reduce speed by 10%
- do nothing

The Reward Function

The same reward function as the one discussed in Section 4.2.3 is used for the experiments in this chapter.

$$\text{Reward} = \text{McGrewScore} - 0.5 \tag{6.1}$$

Hyperparameters

Tables 6.3, 6.4 and 6.5 show the hyperparameters for the ACET, DQN/DDQN and A3C agent, respectively.

Hyperparameter	Value	Description
λ^{θ}	0.9	actor trace-decay rate
λ^{w}	0.8	critic trace-decay rate
α^{θ}	0.5	actor step-size
α^{w}	0.5	critic step-size
discount factor	0.95	discount factor (γ) used in updates

Table 6.3: Hyperparameters for ACET agents in Experiment 1

Hyperparameter	Value	Description
replay memory size	1,000,000	maximum number of samples that can be stored in the replay memory
minibatch size	64	number of samples for each stochastic gradient descent (SGD) update
hidden layer dimension	[50, 50]	number of nodes in the hidden layers
initial ϵ	1	value of ϵ in ϵ-greedy exploration for the first learning episode
final ϵ	0.01	value of ϵ in ϵ-greedy exploration for the last learning episode
discount factor	0.99	discount factor (γ) used in Q-learning updates
learning rate	0.001	learning rate used in the optimiser

Table 6.4: Hyperparameters for the DQN and double DQN agents used for comparison with the D2D-SPL and D2D-SQL agents in Experiment 1

Hyperparameter	Value	Description
hidden layer dimension (actor network)	[100, 100]	number of nodes in the hidden layers of the actor neural network
hidden layer dimension (critic network)	[50, 50]	number of nodes in the hidden layers of the critic neural network
discount factor	0.90	discount factor (γ) used in updates

Table 6.5: Hyperparameters for A3C agents in Experiment 1

6.3.2 Results

This section discusses the test results of D2D-SPL and D2D-SQL and compares them with the results of the deep RL agents.

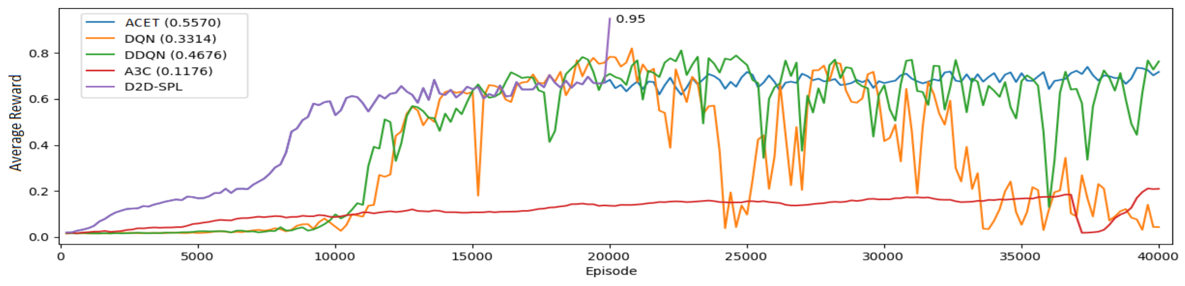


Figure 6.2: Reward/episode for aircraft manoeuvring for all solutions. Every 200 consecutive data points are averaged into one new data point. The average rewards have been increased by 0.5 to compensate for the lowering of the reward from the environment as explained in Chapter 4.

Learning Curves

Figure 6.2 shows the average reward per episode for all the solutions. The numbers in the legend show the average score for the whole learning session. Because the Ace Zero reward has been offset by -0.5, the average rewards for the learning curves are increased by 0.5 to reflect the original McGrew scores. The graphs have been smoothed-out by replacing every 200 consecutive rewards with their mean. It shows that using data from AC 20K to train the D2D-SPL network results in an average reward of 0.95 when the training set is re-applied to the resulting network. This score is much higher than the scores of the other methods.

Figure 6.2 also shows instability in the DQN learning, which may be due to the overestimation issue reported in (van Hasselt et al., 2016). DDQN also exhibits some instability, although not as marked as for DQN.

Figure 6.3 shows the learning curve from the third phase of the D2D-SQL agent learning with an epsilon annealing function and that of a DDQN agent. The DDQN agent uses the same annealing function as the D2D-SQL agent. Figure 6.4 shows the learning curve from the third phase of a D2D-SQL agent learning with a fixed epsilon of 0.05 compared with the learning curve of the previous DDQN agent. The learning curves are obtained by running the agent with ten different random seeds. Every 50 data points are averaged into one new data point. The darker line shows the mean of the new data points over seeds. The shaded area represents the mean ± the standard deviation. The values in the legend show the averages for all trials.

The initialisation of the neural networks in both D2D-SQL tests results in a much steeper curve than the comparison DDQN agent, proving the benefit of D2D-SQL. This is especially apparent in D2D-SQL 10,000 (fe), which explores much less due to a smaller value of ϵ right from the beginning, yielding an instant rise in the average reward. Exploration at the start of learning becomes less compelling when the replay memory is already initialised with off-policy data.

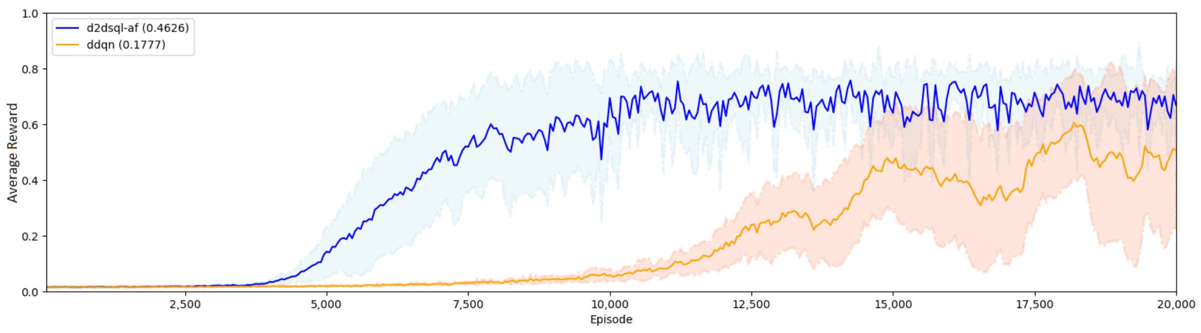


Figure 6.3: The learning curve from the third phase of the D2D-SQL agent learning with an epsilon annealing function, compared to that of DDQN

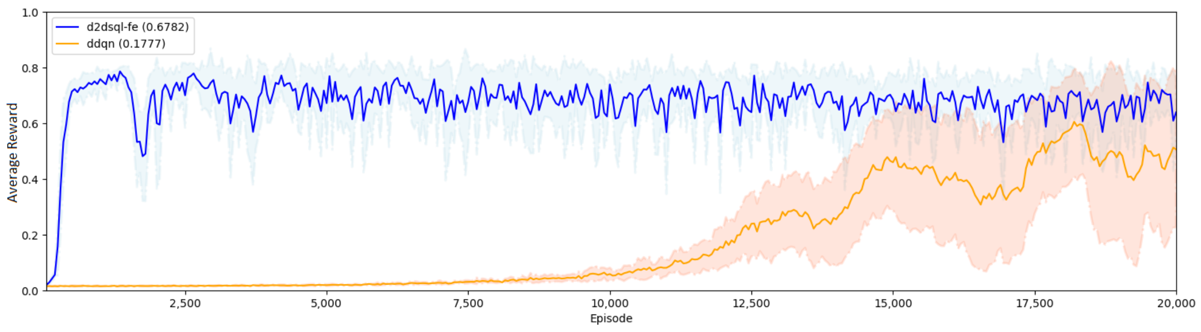


Figure 6.4: The learning curve from the third phase of the D2D-SQL agent learning with a fixed epsilon of 0.05, compared to that of DDQN

Performance

The policies from AC 20K and AC 40K as well as models from D2D-SPL, D2D-SQL, DQN, DDQN and A3C solutions are used to test agents against an opponent that flies along paths that were not seen during training. The four diagrams in Figure 6.5 show four test scenarios applied against the D2D-SPL models. The red paths represent the opponent's trajectories and the blue ones our agent's. The trajectories of Red in the four tests are of increasing complexity, with the first being the simplest and the fourth the hardest.

The test results are then resampled using the bootstrap method, as explained in 3.5, and the resulting means and confidence intervals drawn. Figures 6.6 to 6.9 show the results for the four test scenarios. Each diagram shows the means and 95% confidence intervals of the agents involved.

In the tabular RL cases, AC 40K is generally better than AC 20K because the learning curve of actor-critic is more stable, not as noisy as the DQN, DDQN or A3C. This means, with actor-critic, longer learning tends to produce a better policy. However, because of the possible overestimation in DQN and its variant algorithms, there is no guarantee that longer learning will produce a better model than shorter learning. For example, the results for DQN 40K are worse that those for DQN 20K.

In the easier tasks (test scenarios 1 and 2) the ACET agents perform better than the baseline deep RL agents, but they perform worse in the more complex tasks (test scenarios 3 and 4). In all cases, D2D-SPL performs better than its base AC 20K and even AC 40K. D2D-SPL also performs better than the deep RL methods in three of four cases and better than some of the deep RL agents in all cases.

Among all the baseline deep RL methods, DDQN 20K performs the best, a fact that motivates the choosing of DDQN for the second reinforcement phase of D2D-SQL. In this experiment, both variants of the D2D-SQL agent achieve better performance than those of the deep RL agents in a majority of cases. The D2D-SQL agent with a fixed epsilon performs better than the baseline deep RL methods in all test cases. Interestingly, D2D-SQL also outperform D2D-SPL in test scenarios 3 and 4, which are two of the more complex tasks.

Learning Times

Table 6.6 shows the learning times for all the solutions. All values are rounded to two decimal places except for AC 20K and D2D-SPL so that the slight difference between the two can be shown. All values are relative to the average of AC 20K. AC 40K takes about twice the time taken by AC 20K, DQN 40K runs in about twice the time taken by DQN 20K, and DDQN 40K completes in twice the time taken by DDQN 20K. A3C 200K runs ten times longer than A3C 20K. Because A3C uses four concurrent agents, its per episode learning is faster than its DQN and DDQN rivals. Among all the deep RL solutions being compared, D2D-SPL learns the fastest as it only takes 0.01% longer than AC 20,000.

For D2D-SQL, learning time is the total of the time spent on running the actor-critic with traces agent to obtain off-policy data, the time to train a supervised neural network on that data and the time for the neural network to be used by a DDQN agent to continue learning. Overall, because tabular methods such as actor-critic are much faster to learn than neural RL methods and the supervised phase of D2D-SQL takes only about 5% of the total learning time, the bulk of the learning time of the D2D-SQL 10,000 agent is spent on the second reinforcement phase. On average a D2D-SQL 10,000 agent spends at least 38% less in learning than a DDQN agent that does 20,000 learning episodes, despite the D2D-SQL training for 50% more episodes (20,000 for the initial tabular ACET plus another 10,000 after the supervised phase).

Trial	AC 20K	AC 40K	D2D SPL	D2D SQL(af)	D2D SQL(fe)	DQN 20K	DQN 40K	DDQN 20K	DDQN 40K	A3C 20K	A3C 200K
0	0.9515	1.91	**0.9515**	2.52	2.58	4.70	11.13	3.56	7.25	1.39	13.53
1	0.9476	1.89	**0.9476**	2.58	2.34	4.89	12.29	3.49	7.27	1.36	13.58
2	1.1920	2.39	**1.1923**	2.33	2.27	4.44	13.89	3.55	7.30	1.27	13.47
3	1.1778	2.36	**1.1779**	2.19	2.17	4.51	12.31	4.41	8.71	1.22	14.33
4	1.2071	2.44	**1.2072**	2.30	2.52	3.57	7.38	4.28	8.33	1.24	14.08
5	0.9106	1.84	**0.9106**	2.53	2.06	3.59	7.40	4.17	8.22	1.37	14.59
6	0.9104	1.84	**0.9104**	2.37	2.44	3.59	7.36	3.86	7.71	1.50	13.90
7	0.9127	1.85	**0.9127**	2.39	2.51	3.59	7.36	3.79	7.63	1.55	13.93
8	0.8952	1.81	**0.8952**	2.44	2.06	3.93	8.95	3.86	7.85	1.40	14.99
9	0.8952	1.80	**0.8952**	2.52	2.52	3.89	8.69	3.84	7.91	1.31	13.77
Mean	1.0000	2.01	**1.0001**	2.42	2.35	4.07	9.68	3.88	7.82	1.36	14.02

Table 6.6: D2D-SPL and D2D-SQL learning times relative to other methods in Experiment 1. The learning times for D2D-SPL and D2D-SQL are the total times that include both reinforcement and supervised phases.

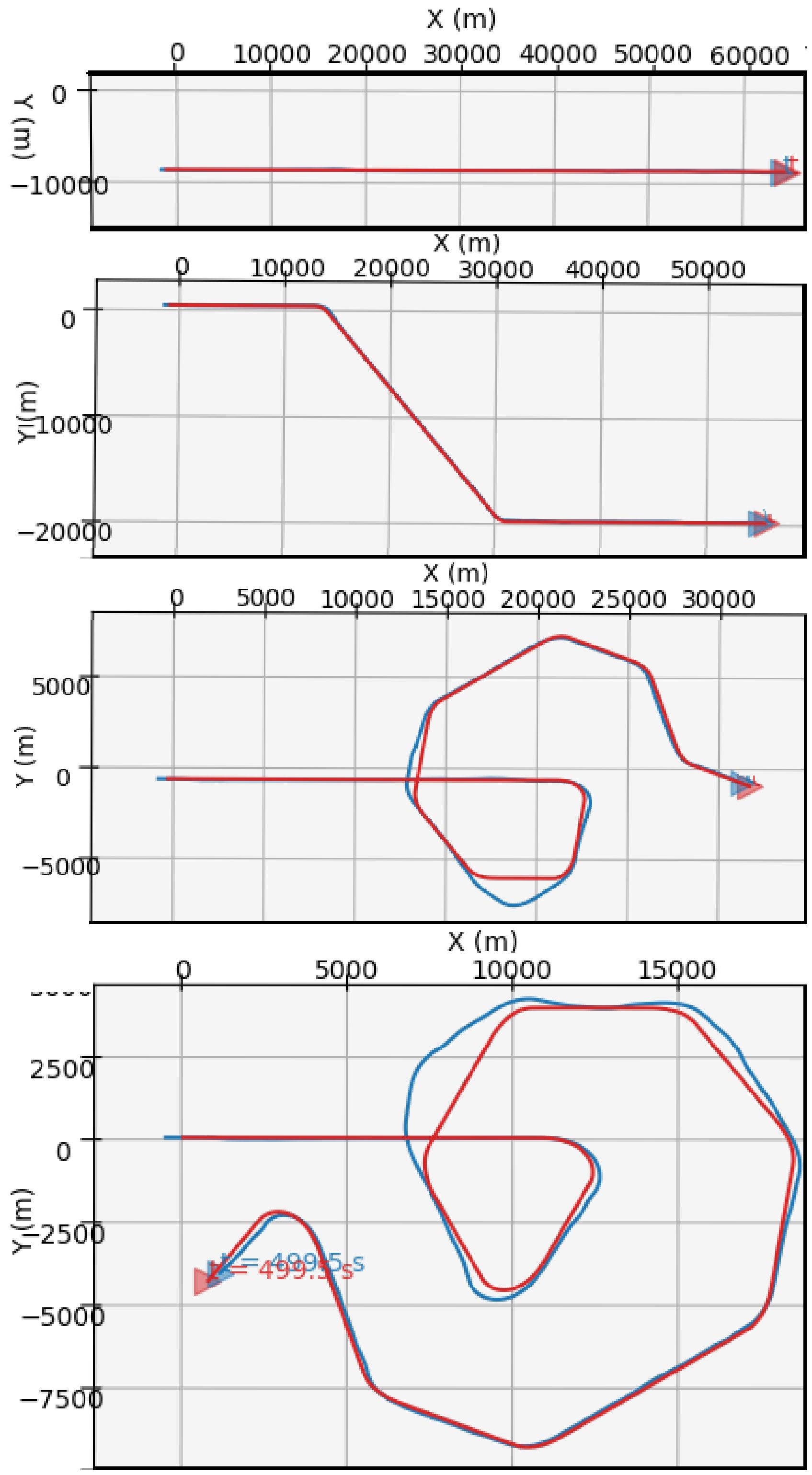


Figure 6.5: Trajectories for test scenarios 1, 2, 3 and 4

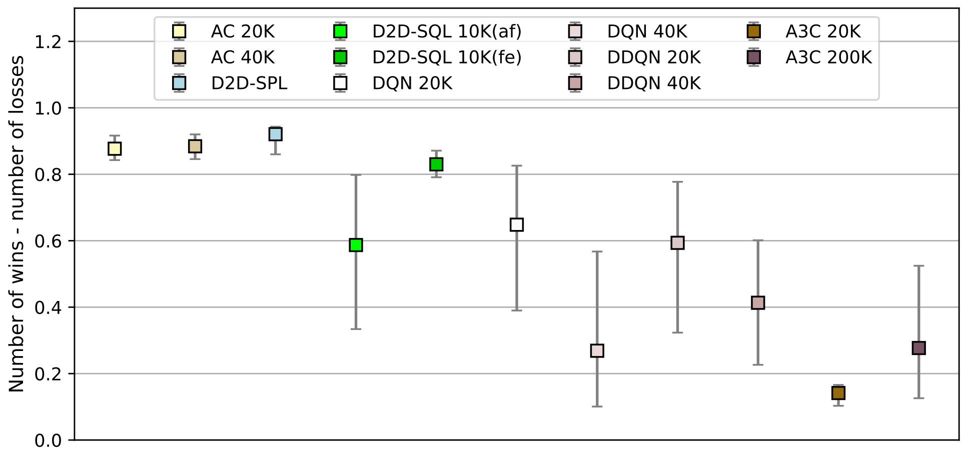


Figure 6.6: Aircraft manoeuvring test results for test scenario 1. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

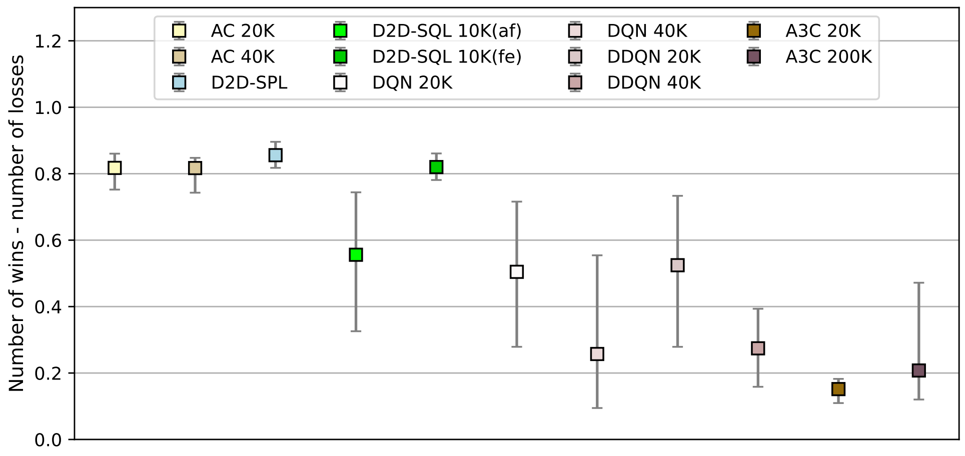


Figure 6.7: Aircraft manoeuvring test results for test scenario 2. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

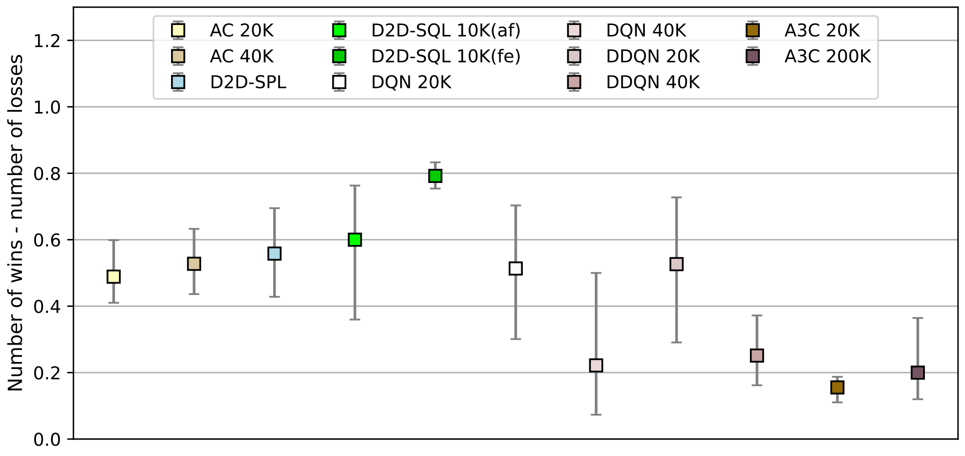


Figure 6.8: Aircraft manoeuvring test results for test scenario 3. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

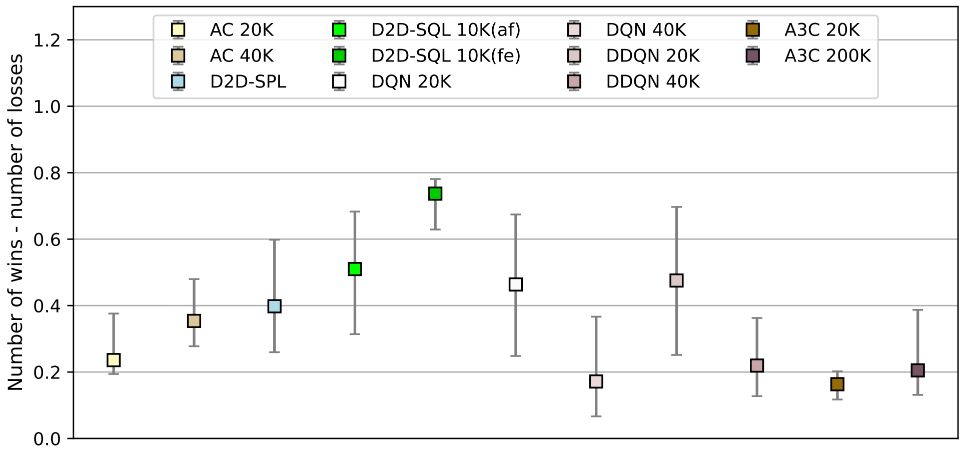


Figure 6.9: Aircraft manoeuvring test results for test scenario 4. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

6.4 Experiment 2: Larger Buffer

In this experiment, an ACET agent, similar to the one in Chapter 4, is run ten times, each time for 50,000 episodes, to generate off-policy data for training a D2D-SPL and D2D-SQL agents. All agents are trained using random initial positions as explained in Chapter 3 and an episode consists of 700 timesteps. The results are then compared with the best results of the ACET agents in Chapter 4 and the best results of the DQN agents in Chapter 5. The objective of this experiment is to show that training the ACET agent for longer improves the performance of the D2D methods while still keeping the overall run-time well below that of the deep RL algorithms.

6.4.1 Experimental Setup

For the D2D-SPL tests, the ACET agent is run for 50,000 episodes ten times using ten different random seeds, resulting in ten policies collectively code-named AC-50K. The generated off-policy samples are used to train the classifier in the D2D-SPL agent.

For the D2D-SQL tests, a D2D-SQL agent is developed that encapsulates an ACET agent and a DQN agent from the D2D-SPL experiments. For the first reinforcement phase, the ACET agent is trained for 50,000 episodes to generate off-policy data, in the same way as data for AC-50K is obtained for the D2D-SPL test. For the supervised phase, the off-policy data is used to populate the neural network in a DQN agent, using states as the input and action numerical preferences as the target. The neural network is trained until a minimum loss of 0.001 is achieved. For the second reinforcement phase, the DQN agent is cloned four times and the four DQN agents are trained using the DQN algorithm (Mnih et al., 2013) for 10,000, 11,000, 12,000 and 13,000 episodes, respectively. In Experiment 1 the D2D-SQL agents employed two ϵ-greedy functions, and the tests show the agent using a fixed value returns a better result. Because of this, the four D2D-SQL agents in Experiment 2 are only tested with a fixed ϵ-greedy value of 0.05.

State Discretisation

Because ACET is a tabular method and the air combat problem has a continuous state space, the states need to first discretised. The discretisation strategy used in this experiment is the same as the one in Section 4.2.1 where a state is discretised into one of 14,000 discrete states.

The Action Space

The ACET agents implemented and tested in this chapter use the same action space as the one used in Chapter 4. There are five actions in the action space:

- 10° turn to the left
- 10° turn to the right

- increase speed by 10%
- reduce speed by 10%
- do nothing

The Reward Function

The same reward function as the one discussed in Section 4.2.3 is used for the experiments in this chapter.

$$\text{Reward} = \text{McGrewScore} - 0.5 \tag{6.2}$$

Hyperparameters

Hyperparameter	Value	Description
λ^{θ}	0.9	actor trace-decay rate
λ^{w}	0.8	critic trace-decay rate
α^{θ}	0.5	actor step-size
α^{w}	0.5	critic step-size
discount factor	0.95	discount factor (γ) used in Q-learning updates

Table 6.7: Hyperparameters for ACET agents in Experiment 2

Hyperparameter	Value	Description
replay memory size	1,000,000	maximum number of samples that can be stored in the replay memory
minibatch size	64	number of samples for each stochastic gradient descent (SGD) update
hidden layer dimension	[300, 300]	number of nodes in the hidden layers
initial ϵ	1	value of ϵ in ϵ-greedy exploration for the first learning episode
final ϵ	0.01	value of ϵ in ϵ-greedy exploration for the last learning episode
discount factor	0.99	discount factor (γ) used in Q-learning updates
learning rate	0.001	learning rate used in the optimiser

Table 6.8: Hyperparameters for DQN agents in D2D-SQL in Experiment 2

6.4.2 Results

Learning Curves

The learning curves for the ACET agent can be found in Chapter 4 and those for the deep RL agents in Chapter 5.

Performance

Figure 6.10 shows the results of the D2D-SPL and D2D-SQL agents compared to the base ACET agent (ac-50K), the ac-001 agent in Chapter 4 and the best-performing DQN agents in Chapter 5 (dqn-001, dqn-002 and dqn-random). Each vertical bar shows the mean and 95% confidence interval of the bootstrap-resample of the test results. There are four versions of the D2D-SQL agent, which learned for 10,000, 11,000, 12,000 and 13,000 episodes, respectively, in the second reinforcement phase. Their results are marked as d2dsql-10K, d2dsql-11K, d2dsql-12K and d2dsql-13K, respectively. All tests are done using the 144 different tests as explained in Chapter 3. The horizontal red lines show the baselines from the Smart Pursuit engineered agent used in this thesis, which was explained in Chapter 3.

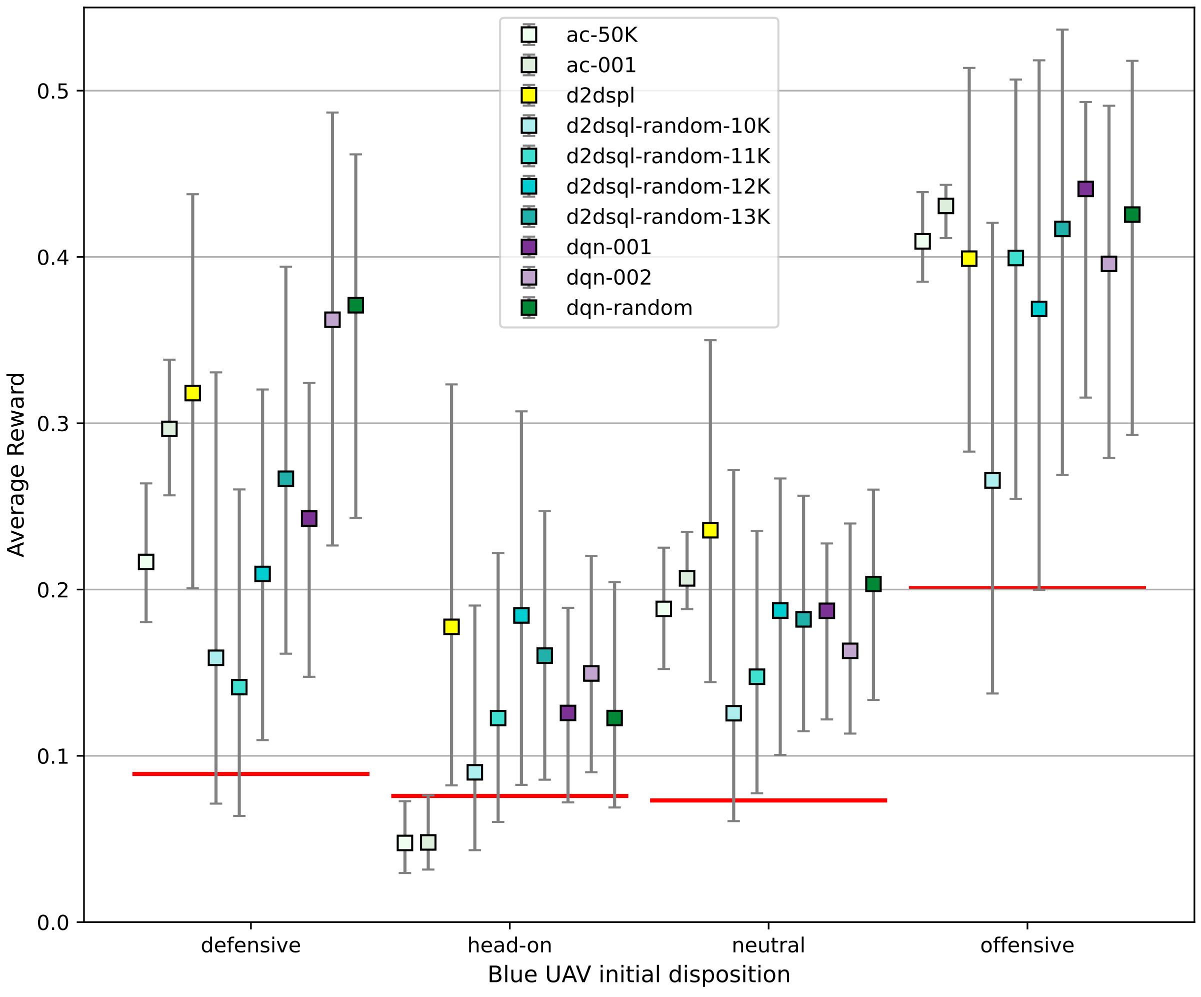


Figure 6.10: The performance of D2D-SPL and D2D-SQL agents compared to ACET and DQN agents. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

In three of four initial dispositions (defensive, head-on and neutral), the D2D-SPL agent out-

Trial	AC 50K	AC 001	D2D SPL	D2D-SQL 10K	D2D-SQL 11K	D2D-SQL 12K	D2D-SQL 13K
0	0.96	3.85	**1.01**	2.92	3.14	3.35	3.53
1	0.98	3.82	**1.03**	2.90	3.15	3.32	3.57
2	0.98	3.97	**1.03**	2.94	3.16	3.36	3.49
3	0.99	3.97	**1.05**	2.95	3.18	3.38	3.46
4	1.00	3.86	**1.08**	2.97	3.12	3.29	3.54
5	1.01	3.84	**1.13**	2.96	3.13	3.28	3.55
6	1.01	4.06	**1.16**	2.94	3.15	3.31	3.59
7	1.01	4.02	**1.21**	2.96	3.17	3.39	3.54
8	1.03	4.08	**1.23**	2.97	3.16	3.37	3.51
9	1.03	4.11	**1.29**	2.97	3.19	3.40	3.49
Mean	1.00	3.96	**1.12**	2.95	3.16	3.35	3.53

Table 6.9: D2D-SPL and D2D-SQL learning times relative to other methods in Experiment 2. The learning times for D2D-SPL and D2D-SQL are the total times that include both reinforcement and supervised phases.

performs its base agent (AC-50K), even though the difference in performance is only statistically significant in the head-on disposition. The D2S-SPL agent fares better than the ac-001 agent in three initial dispositions (defensive, head-on and neutral), even though the latter learned for four times the number of episodes than the base AC-50K agent. This proves that generalisability obtained by using a neural network may outperform agents trained using a tabular method alone.

The chart also shows that in three initial dispositions (head-on, neutral and offensive) at least one D2D-SQL agent outperforms the DQN agents that trained for 20,000 episodes. Even though the differences are not statistically significant, this proves that D2D-SQL can expedite learning of deep RL agents with no significant reduction in performance.

Learning Times

Table 6.9 shows the learning times for all solutions. All values are rounded to two decimal places and relative to the average of AC-50K, which on average took 33,276 seconds. For example, ac-001 took about four times the time taken by AC-50K, as expected.

For D2D-SPL, learning time is the total of the time spent on running the ACET agent to obtain off-policy data and train the classifier. The D2D-SPL in Experiment 2 took about 12% more than the AC-50K agent, but is still much faster than the AC-200K agent.

For D2D-SQL, learning time is the total of the time spent on running the ACET agent to generate off-policy data, the time to initialise the neural network and the time spent on the second reinforcement phase. Since the objective of D2D-SQL is to improve on the learning time of deep RL methods, it makes sense to compare the learning time of the D2D-SQL agent with the learning time of its underlying deep RL agent, in this case the dqn-random agent in Chapter 5. The dqn-random agent, which learned for 20,000 episodes, on average took 39.46 (36.19) hours

Agent	Relative Learning Time
D2D-SQL 10K	69.05%
D2D-SQL 11K	73.95%
D2D-SQL 12K	78.40%
D2D-SQL 13K	82.61%
dqn-random	100.00%

Table 6.10: D2D-SQL learning times relative to DQN-001-20K

or 142,056 (130,284) seconds to complete learning. In relative term, the dqn-random agent took 4.27 times the average learning time of AC-50K. This means, the D2D-SQL 10K, D2D-SQL 11K, D2D-SQL 12K and D2D-SQL 13K agents are faster to train than the dqn-random agent by 30.95%, 26.05%, 21.60% and 17.39%, respectively, as shown in Table 6.10.

6.5 Summary

The objective of this chapter is to investigate whether it is possible to obtain the generalisability of a neural network at a cost closer to that of a tabular method, by proposing two novel methods, D2D-SPL and D2D-SQL, and testing them against the air combat manoeuvring problem. D2D-SPL uses off-policy data generated by an ACET agent to train a classifier and use the classifier to select actions in a sequential decision making problem. D2D-SQL uses the same off-policy data to train DQN and double DQN agents to expedite their learning.

Two experiments are conducted to test the methods. Experiment 1 uses double DQN and Experiment 2 uses DQN for D2D-SQL. The results show D2D-SPL learns much faster than deep RL methods and that D2D-SQL can shorten the learning time of a DQN agent and in some cases outperform D2D-SPL. In Experiment 1, which uses 500 off-policy samples to train the D2D-SPL classifier, the total learning time is only about 0.01% of the learning time of the base ACET agent. In Experiment 2, where off-policy data consists of 2,000 samples, the learning time of the D2D-SPL agent is about 12% more than the base ACET agent.

Performance-wise, the D2D-SPL shows that it can outperform an ACET agent that spent twice or even four times its learning times. In some cases the D2D-SQL agents which trained for approximately half the time used for training a DQN agent actually outperform the DQN agent. The D2D-SQL agents in both experiments are shown to outperform the DQN or double DQN agents that trained for twice the number of episodes in some initial predispositions. D2D-SPL is demonstrated to offer generalisability of deep RL methods at a cost closer to a tabular method and D2D-SQL is demonstrated to expedite DQN and double DQN learning.

CHAPTER 7

MULTIOBJECTIVE REINFORCEMENT LEARNING METHODS

So far this thesis has focused exclusively on the single objective of outmanoeuvring the opponent, as the majority of reinforcement learning (RL) algorithms only allow for a single objective or reward. However, the air combat domain has multiple, often conflicting, objectives, which make it a multiobjective reinforcement learning (MORL) problem. For example, one of the objectives of the air combat pilot is to be able to outmanoeuvre their opponent, which requires the aircraft to accelerate fast and fly as nimbly as possible. This objective is in conflict with the need to save fuel, which necessitates it to fly at a constant speed.

Unlike Chapters 4, 5 and 6, which evaluated single-objective RL methods, this chapter investigates four MORL methods and uses seven reward functions in conjunction with the methods, with tests conducted split into two experiments. This chapter has two objectives. First, it investigates which of the four MORL algorithms used here are effective in addressing the air combat manoeuvring problem. Second, it investigates which of the seven reward functions allow the agents to learn good policies. Rather than treating the air combat manoeuvring problem as a truly multiobjective problem (i.e. by finding a trade-off between conflicting objectives), this chapter views the problem as a single-objective problem with the reward decomposed into simpler components, a technique known as multiobjectivisation (Brys et al., 2014). As reward decomposition may present richer information to the learning agent, this approach might be expected to outperform an agent with scalar reward, as shown for instance in Russel and Zimdars (2003) and Tesauro et al. (2008).

7.1 Overview of the Methods Used

Like in single-objective RL, at every timestep the MORL agent selects an action and passes it to the environment. The difference is, instead of returning the next state and a scalar reward as in single-objective RL, in MORL the environment returns the next state and a reward vector. The MORL agent then uses the reward vector to update its policy or policies. Action selection is done by aggregating values from the single policy or multiple policies. The experiments in this chapter are based on four MORL algorithms, which are all derived from single-objective RL methods.

- multiobjective Q(λ)
- multiobjective actor-critic with eligibility traces (ACET)
- multiobjective DQN with a shared network
- multiobjective DQN with multiple networks

The algorithms are single-policy MORL methods and action selection is carried out with the help of a scalarisation function f that projects the multiobjective value V^{π} to a scalar value.

$$\mathrm{V}^{\pi}_{\mathbf{w}} = f(\mathrm{V}^{\pi}(s), \mathbf{w}),$$

where $\mathbf{w}$ is a weight vector parameterising f (Roijers et al., 2013). A commonly used type of scalarisation function is the linear scalarisation function, which is a simple and intuitive way to scalarise (Roijers et al., 2013; Lizotte et al., 2010) and which the four MORL methods evaluated in this chapter use. When linear scalarisation is used, with either uniform or non-uniform weights $\mathbf{w}$, the multiobjective problem can be solved using an existing single-objective RL algorithm by letting the agent learn a vector-valued value function and scalarise the value function when selecting actions. The argument for adopting this approach is that the values of individual objectives may be easier to learn than the scalarised value, especially when function approximation is used (Tesauro et al., 2008). For problems where linear scalarisation is not suitable there exist non-linear scalarisation techniques such as Chebyshev scalarisation (Voß et al., 2008), hypervolume-based scalarisation (van Moffaert et al., 2013), fuzzy logic-based scalarisation (Lin and Chung, 1999; Zhao et al., 2010), time-discounting scalarisation (van Vaerenbergh et al., 2012) and scalarisation based on thresholded lexicographic ordering (TLO) (Gabor et al., 1998).

The following subsections explain the algorithms used in this chapter in more detail.

7.1.1 Multiobjective Q(λ)

In multiobjective Q(λ) a scalar reward is replaced by a reward vector containing the same number of elements as there are objectives. A Q-table or multiple Q-tables may be used. If there is only one Q-table and there are n rewards, the number of rows are multiplied by n. If multiple Q-tables are used, there must be one Q-table for each objective. In this thesis, multiple Q-tables are used.

The agent uses each reward in the reward vector returned by the MORL environment to update its policy. The first reward in the vector is used to update the first Q-table, the second to update the second Q-table, and so on. If the problem has a continuous state space, a discretiser is needed to discretise the state from the environment. For action selection, the discrete state is used to look up the Q-tables and the return values from all the Q-tables are summed and the action with the highest total value is selected. Multiobjective Q(λ) is presented in Algorithm 16. The algorithm is an extension to single objective Q(λ) by Watkins (Watkins, 1989) with differences between the two highlighted.

Algorithm 16: Multiobjective Q(λ)

```
Parameter(s): step size $\alpha \in (0,1]$, discount-rate $\gamma \in (0,1]$, decay rate $\lambda \in (0,1]$, small $\epsilon > 0$
Parameter(s): number of objectives $n$
Result: Q[]
Init. $Q(s,a)[]$ arbitrarily and $e(s,a)[] = 0$ for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, $Q(terminal,.)[] = 0$;
for each episode do
    Initialise $S$;
    $Visited[] = []$;
    for each step of episode or until S is terminal do
        Choose $A$ from $\sum_{k=0}^{n-1} Q(s,a)[k]$ with exploration rate determined by $\epsilon$ (e.g.,
          $\epsilon$-greedy);
        Take action $A$, observe $R[]$, $S'$;
        for $k$=0, $n$-1 do
            $\delta \leftarrow R[k] + \gamma \max_a Q(S',a)[k] - Q(S,A)[k]$;
            $Q(S,A)[k] \leftarrow Q(S,A)[k] + \alpha\delta$;
            $greedy = Q(S,A)[k] == \max_a Q(S,a)[k]$;
            if $greedy$ then
                for all $(s,a) \in Visited[k]$ do
                    $Q(s,a)[k] \leftarrow Q(s,a)[k] + \alpha\delta e(s,a)[k]$;
                    $e(s,a)[k] \leftarrow \gamma\lambda e(s,a)[k]$;
                end
            else
                for $(s,a)$ in $Visited[k]$ do
                    $e(s,a)[k] \leftarrow 0$;
                end
                Delete all elements in $Visited[k]$;
            end
            $e(S,A)[k] \leftarrow 1$ (replacing traces) or $e(S,A)[k] + 1$ (accumulating traces);
            If $(S,A) \notin Visited[k]$, add $(S,A)$ to $Visited[k]$;
        end
        $S \leftarrow S'$;
    end
end
```

7.1.2 Multiobjective Actor-Critic with Eligibility Traces (ACET)

Actor-critic algorithms are some of the earliest RL methods (Witten, 1977; Sutton, 1984). First introduced by Barto et al. (1983), the actor-critic algorithm is a policy-based method that also uses a value function to learn the policy parameter. Actor-critic learns the policy parameter based on the gradient of some scalar performance measure with respect to the policy parameter, and therefore belongs to the policy-gradient family of methods.

Multiobjective ACET is an extension to the ACET algorithm explained in Chapter 4. In multiobjective ACET, the agent creates a policy parameter $\boldsymbol{\theta}$ and a traces table $\mathbf{w}$ for each objective. Upon receiving a reward vector from the environment, the agent uses the reward vector to update all policy parameters and traces tables. For problems with a continuous state space, the state is discretised and the discrete state is used to look up the policy parameters. For action selection, the discrete state is used to look up the numerical preferences in all the policies. These numerical preferences are summed and the sums are passed to the softmax function. The probability distribution returned by the softmax function is used select the action. Multiobjective ACET is shown in Algorithm 17.

7.1.3 Multiobjective DQN with A Shared Network

Multiobjective DQN with a shared network is a multiobjective RL algorithm that is based on DQN (Mnih et al., 2013). It uses a single neural network for all its objectives and has the same number of input nodes as the number of state variables. The number of output nodes in the neural network is the same as the number of actions multiplied m by the number of objectives n. The first m output nodes return the Q-values of all the state-actions of the first objective, the second m output nodes return the Q-values of all the state-actions of the second objective and so on. To select an action, the Q-values for all the objectives are summed and the action with the highest value in the sum is selected. Algorithm 18 shows multiobjective DQN with a shared network with differences from single objective DQN highlighted.

7.1.4 Multiobjective DQN with Multiple Networks

Multiobjective DQN with multiple networks is a multiobjective RL algorithm that is based on DQN (Mnih et al., 2013). It uses multiple neural networks, one for each objectives. The algorithm is equivalent to using n single-objective DQN agents, where n is the number of objectives. To select an action, the Q-values returned by the output nodes of all the neural networks are summed and the action with the highest value is selected. Algorithm 19 shows multiobjective DQN with multiple networks. Differences between this algorithm and DQN are highlighted.

Algorithm 17: Multiobjective actor-critic with eligibility traces for episodic problems

```
1  Input: an array of differentiable policy parameterizations $\pi(a|s,\boldsymbol{\theta})[]$
2  Input: an array of differentiable state-value function parameterizations $\hat{v}(s,\mathrm{w})[]$
   Parameter(s): step sizes $\alpha^{\boldsymbol{\theta}} > 0$, $\alpha^{\mathrm{w}} > 0$
   Parameter(s): trace decay rates $\lambda^{\boldsymbol{\theta}} \in [0,1]$, $\lambda^{\mathrm{w}} \in [0,1]$
3  Parameter(s): $n$ number of objectives
4  Initialise policy parameter $\boldsymbol{\theta[]} \in \mathbb{R}^{nxd'}$ and state-value weights $\mathrm{w}[] \in \mathbb{R}^{nxd}$ (e.g, to 0) ;
5  for each episode do
6      Initialise $S$ (first state of episode);
7      $\mathrm{z}^{\boldsymbol{\theta}}[] \leftarrow \mathbf{0}$ ($d'$-component eligibility trace vector) ;
8      $\mathrm{z}^{\mathrm{w}}[] \leftarrow \mathbf{0}$ ($d$-component eligibility trace vector) ;
9      $I[] \leftarrow 1$ ;
10     for each step of episode or until $S$ is terminal do
11         $A \sim \sum_{k=0}^{n-1} \pi(.|S,\boldsymbol{\theta})[k]$ ;
12         Take action $A$, observe $R[]$, $S'$;
13         for $k$=0, $n$-1 do
14             $\delta[k] \leftarrow R[k] + \gamma\hat{v}(S',\mathrm{w})[k] - \hat{v}(S,\mathrm{w})[k]$     (if $S'$ is terminal, $\hat{v}(S',\mathrm{w})[k] \doteq 0$) ;
15             $\mathrm{z}^{\mathrm{w}}[k] \leftarrow \gamma\lambda^{\mathrm{w}}\mathrm{z}^{\mathrm{w}}[k] + \nabla\hat{v}(S,\mathrm{w})[k]$ ;
16             $\mathrm{z}^{\boldsymbol{\theta}}[k] \leftarrow \gamma\lambda^{\boldsymbol{\theta}}\mathrm{z}^{\boldsymbol{\theta}}[k] + I[k]\nabla \ln \pi(A|S,\boldsymbol{\theta})[k]$ ;
17             $\mathrm{w[k]} \leftarrow \mathrm{w[k]} + \alpha^{\mathrm{w}}\delta\mathrm{z}^{\mathrm{w}}[k]$ ;
18             $\boldsymbol{\theta[k]} \leftarrow \boldsymbol{\theta[k]} + \alpha^{\boldsymbol{\theta}}\delta[k]\mathrm{z}^{\boldsymbol{\theta}}[k]$;
19             $I[k] \leftarrow \gamma I[k]$;
20         end
21         $S \leftarrow S'$;
22     end
23 end
```

Algorithm 18: Multiobjective DQN with a shared network

```
   Parameter(s): discount-rate $\gamma \in (0,1]$, small $\epsilon > 0$
   Result: Q
1  Init. replay memory $D$ to capacity $N$;
2  Init. action-value function $Q$ with random weights $\theta$;
3  for each episode do
4      Initialise $s_1$;
5      for each step $t$ of episode or until $s_t$ is terminal do
6          With probability $\epsilon$, select a random action $a_t$, otherwise
7          $a_t$=$argmax_a sum(split(Q(\phi(s_t),a;\theta)))$;
8          Take action $a_t$, observe rewards $r[]_t$ and state $s_{t+1}$;
9          Store transition $(\phi_t, a_t, r[]_t, \phi_{t+1})$, where $\phi_t = \phi(s_t)$, in $D$;
10         Sample random minibatch of transitions $(\phi_j, a_j, r[]_j, \phi_{j+1})$ from $D$;
11         Set $y_j = \begin{cases} r[]_j, & \text{if episode terminates at step } j+1 \\ r[]_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \theta), & \text{otherwise} \end{cases}$;
12         Perform gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to $\theta$;
13     end
14 end
```

Algorithm 19: Multiobjective DQN with multiple networks

```
   Parameter(s): discount-rate $\gamma \in (0,1]$, small $\epsilon > 0$
1  Parameter(s): $n$ number of objectives
   Result: Q
2  Init. an array of replay buffers $D[]$ each to capacity $N$;
3  Init. an array of action-value functions $Q[]$ with random weights $\theta$;
4  for each episode do
5      Initialise $s_1$;
6      for each step $t$ of episode or until $s_t$ is terminal do
7          With probability $\epsilon$, select a random action $a_t$, otherwise
8          $a_t = argmax_a sum(Q(\phi(s_t),a;\theta)[])$;
9          Take action $a_t$, observe reward $r[]_t$ and state $s_{t+1}$;
10         for $k$=0, $n$-1 do
11             Store transition $(\phi_t, a_t, r[k]_t, \phi_{t+1})$, where $\phi_t = \phi(s_t)$, in $D[k]$;
12             Sample random minibatch of transitions $(\phi_j, a_j, r[k]_j, \phi_{j+1})$ from $D[k]$;
13             Set $y[k]_j = \begin{cases} r[k]_j, & \text{if episode terminates at step } j+1 \\ r[k]_j + \gamma max_{a'} Q(\phi_{j+1}, a'; \theta[k]), & \text{otherwise} \end{cases}$;
14             Perform gradient descent step on $(y_j - Q(\phi_j, a_j; \theta[k]))^2$ with respect to $\theta[k]$;
15         end
16     end
17 end
```

7.2 Overview of the Experiments

Two experiments, each with a set of reward functions (builders), are presented. Experiment 1 uses all the MORL methods explained in the previous section, but Experiment 2 only evaluates multiobjective ACET because of poor performance of the other MORL algorithms in Experiment 1.

There are seven reward functions or reward builders used in the experiments. In the first experiment, all the reward builders use a combination of the Ace Zero attributes discussed in Chapter 3—the range, AA angle and ATA angle—which together form the McGrew score. Breaking these apart may assist the agent's learning as demonstrated in the multiobjectivisation study by Brys et al. (2014). In the second experiment, the agent's McGrew score and the inverse of the opponent's McGrew score are used as rewards. Different reward builders carry out this decomposition in different ways and with different weightings to see which works best. Only the random initial positions are used in both experiments, ten tests for each agent and reward builder. Due to the high number of tests with the random initial positions, the fixed initial positions used in the previous chapters are skipped.

7.2.1 State Discretisation

The tabular methods in both experiments use the same state discretisation strategy as in the previous chapters, where a state is discretised into one of 14,000 discrete states. This discretisation strategy was described in Section 4.2.1

7.2.2 The Action Space

All the methods in all the experiments use the same action space as in the previous chapters. There are five actions in the action space:

- 10° turn to the left
- 10° turn to the right
- increase speed by 10%
- reduce speed by 10%
- do nothing

7.2.3 State Normalisation

The multiobjective DQN methods used in this chapter employ a state normaliser to normalise the state variable values in order to expedite neural network learning. This is the same state normalisation strategy used by the deep RL and discrete-to-deep agents in Chapters 6 and 7.

In this strategy, the four state variables—R (distance), AA (attack angle), ATA (antenna train angle) and $delta\_v$ (speed difference)—are normalised as follows:

$$R = R/4500.00$$

$$AA = AA/180.00$$

$$ATA = ATA/180.00$$

$$delta\_v = delta\_v/40.00$$

7.3 Experiment 1 (RB001 to RB003)

In this experiment, three multiobjective Q(λ) agents, three multiobjective ACET agents, three multiobjective DQN agents with a shared network and three multiobjective DQN agents with multiple networks are used to evaluate three reward functions.

7.3.1 Experimental Setup

All runs use the state discretisation, action space and state normalisation explained in Section 7.2. As the rewards, the experiment decomposes the McGrew score, which is used in single-objective RL algorithms in the previous chapters, into its components: the range score, AA score and ATA score. It then applies a technique similar to the multiobjectivisation study in Brys et al. (2014). Three reward functions or reward builders, code-named RB001, RB002 and RB003, are used. RB001 and RB003 return a vector containing the range score, AA score and ATA score in different weightings. RB002 uses zero range score and zero AA score to see how the agents will perform if the ATA score is the only reward component that is available.

- RB001: [0.33 range_score, 0.33 AA_score, 0.33 ATA_score]
- RB002: [0, 0, ATA_score]
- RB003: [range_score, AA_score, ATA_score]

7.3.2 Results

Learning Curves

Figures 7.1, 7.2 and 7.3 show the learning curve of the agents using reward builders RB001, RB002 and RB003, respectively. All the learning curves are obtained by running the agents ten times with ten different random seeds and averaging every 50 consecutive data points into a new data point. The darker line is the mean of the new data points over seeds. The shaded area represents the mean ± the standard deviation. The average of each reward is shown in the legend.

For RB001 the maximum value for each reward is 0.33 thanks to the weighting used. The first reward, the range score (represented by the blue lines), measures the distance between the agent and the opponent. As shown in Chapter 1, a positive value for the is achieved when the distance is between 153 and 914 metres, with a value of 1 at a distance of 380 metres. The multiobjective ACET agent does best at maximising this reward, followed by the Q(λ) agent. The two multiobjective DQN agents have no success in doing so. With regards to the AA and ATA scores, all the four agents manage to improve their AA and ATA scores, with the ACET agent outperforming the other three agents.

For RB002, the first two rewards, the range and AA scores, are zero. The third reward, the ATA score, has a value between 0 and 1. All the four agents manage to improve the ATA score with the ACET agent again outperforming the others. The tabular methods increase the average score of the third reward steadily, whereas the multiobjective DQN agents show some peaks and troughs, like the learning curves in the single-objective tests of the deep RL methods.

RB003 is similar to RB001 except each of the three rewards can have a value between 0 and 1. Like in the RB003 tests, all agents manage to improve their AA and ATA scores and the multiobjective ACET agent has the most success in increasing the range score with an average score of 0.1370.

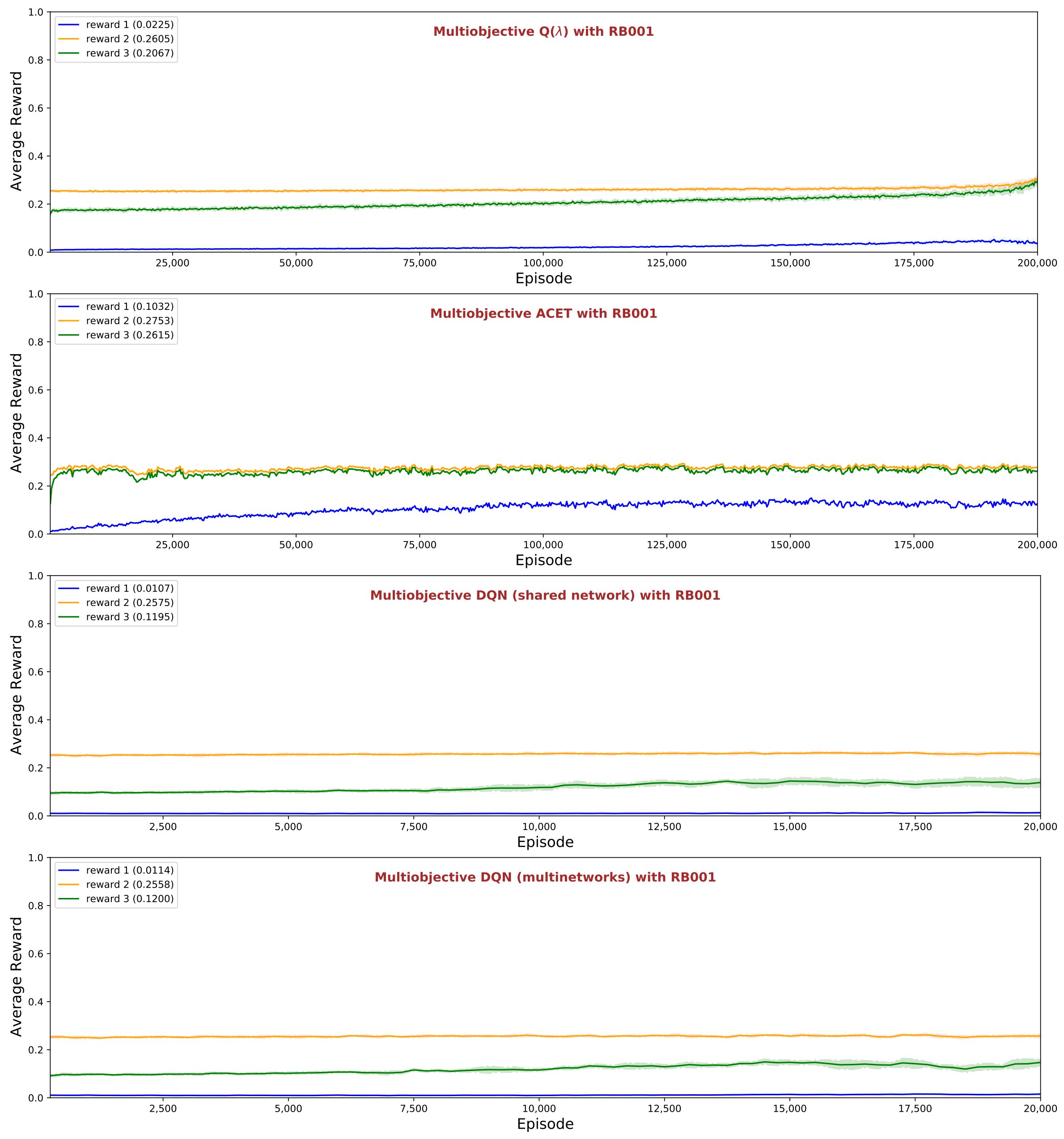


Figure 7.1: The learning curves of the multiobjective Q(λ) agent, multiobjective ACET agent, multiobjective DQN agent with a shared network and multiobjective DQN agent with multiple networks using reward builder RB001. The curves are obtained by running the agents ten times with ten different random seeds and averaging every 50 consecutive data points. The darker line is the mean of the new data points over seeds. The shaded area represents the mean ± the standard deviation. The values in the legend show the averages for all trials.

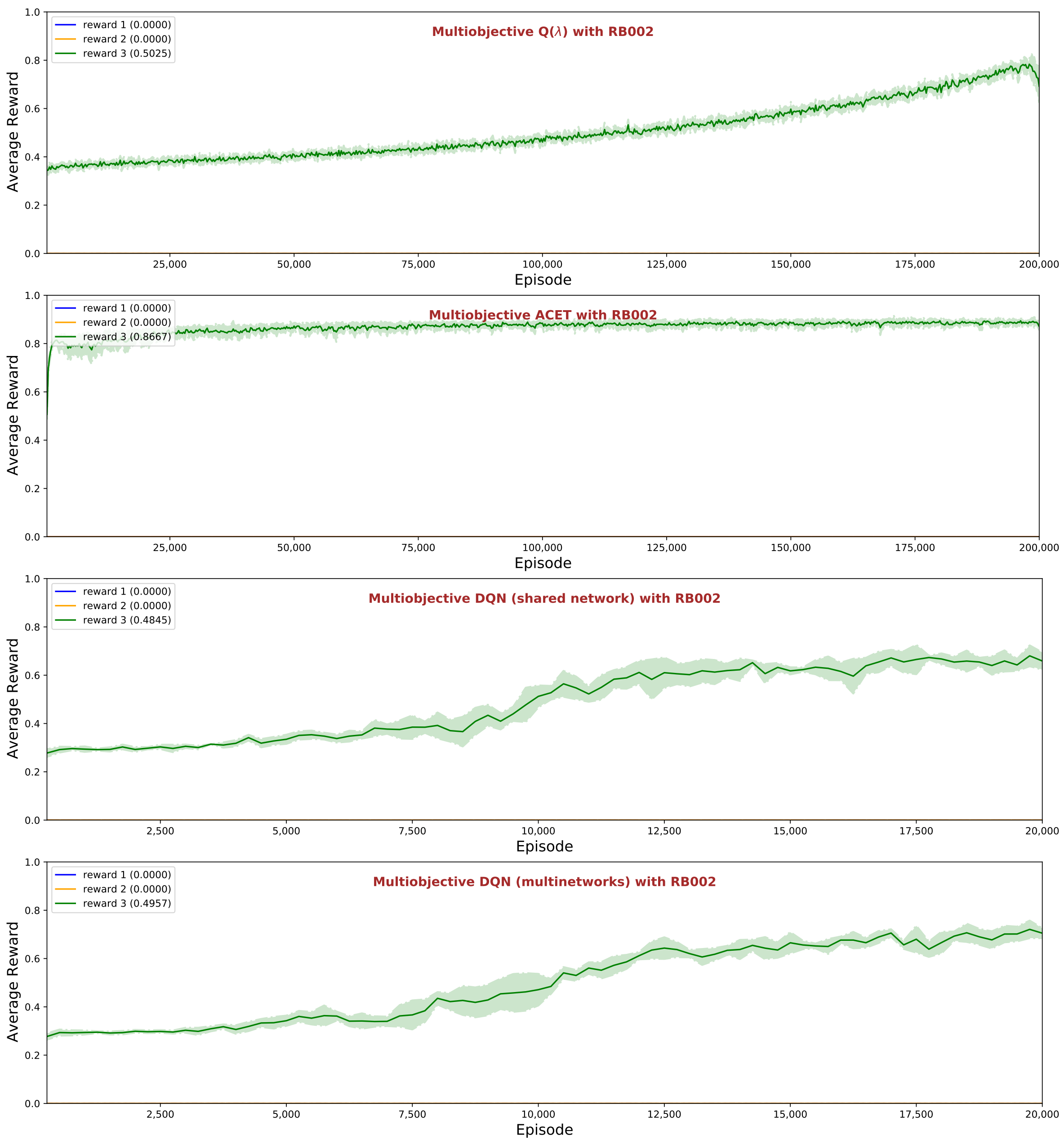


Figure 7.2: The learning curves of the multiobjective Q(λ) agent, multiobjective ACET agent, multiobjective DQN agent with a shared network and multiobjective DQN agent with multiple networks using reward builder RB002. The curves are obtained by running the agents ten times with ten different random seeds and averaging every 50 consecutive data points. The darker line is the mean of the new data points over seeds. The shaded area represents the mean ± the standard deviation. The values in the legend show the averages for all trials

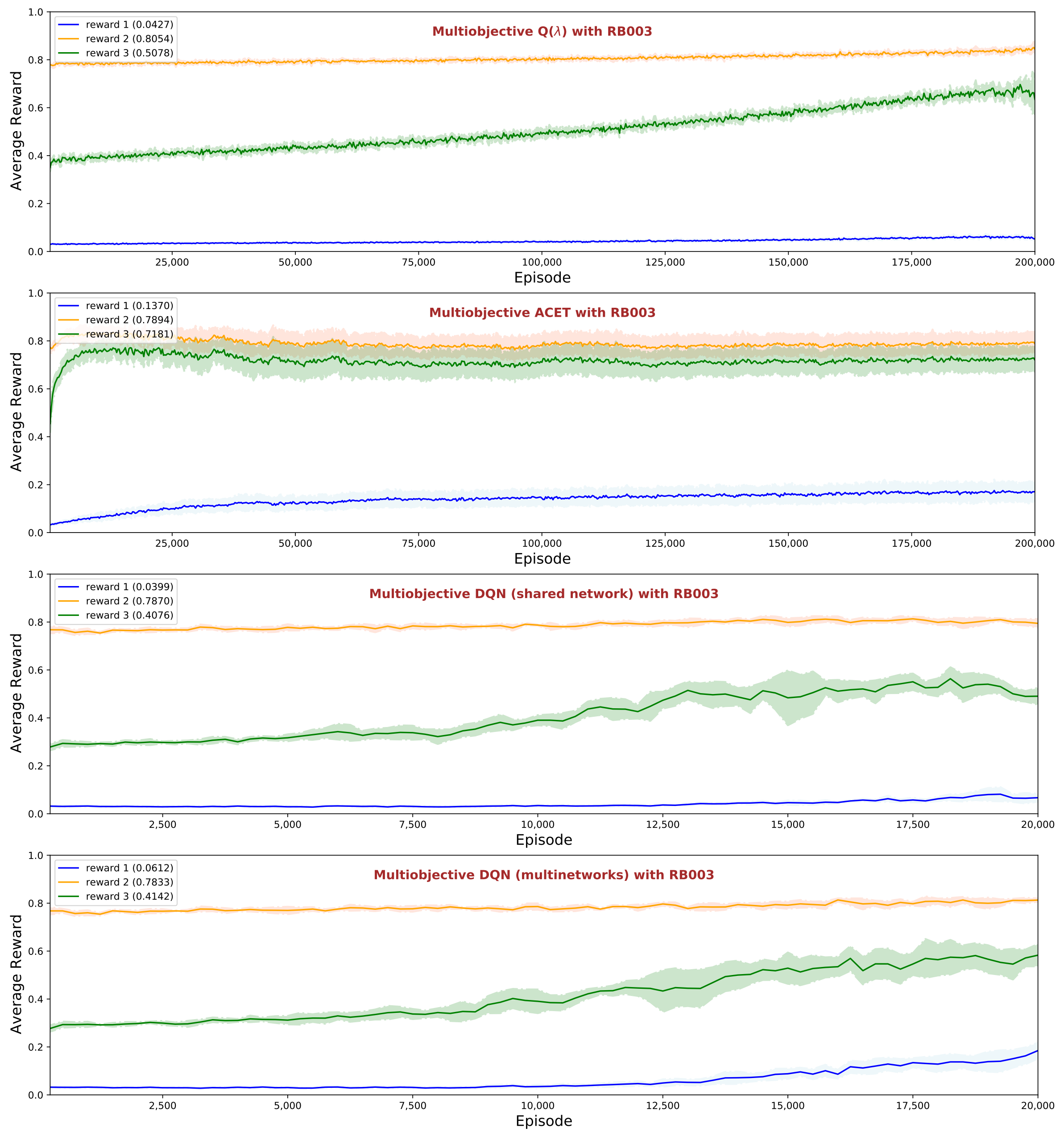


Figure 7.3: The learning curves of the multiobjective Q(λ), multiobjective ACET, multiobjective DQN with a shared network and multiobjective DQN with multiple networks agents with RB003, obtained by running the agents ten times with ten different seeds and averaging every 50 consecutive data points. The darker line is the mean of the new data points over seeds. The shaded area represents the mean $\pm$ the standard deviation. The values in the legend show the averages for all trials

Performance

Figure 7.4 shows the performance of the agents. Each vertical bar shows the mean and 95% confidence interval after resampling the original test result of the agent, as described in 3.5. The average rewards are based on the original scalar McGrew score. Despite their ability to increase the total rewards during learning, only the multiobjective ACET agents are able to beat the Smart Pursuit baselines. The multiobjective ACET agent using reward builder RB001 is the best performer in three of four initial dispositions.

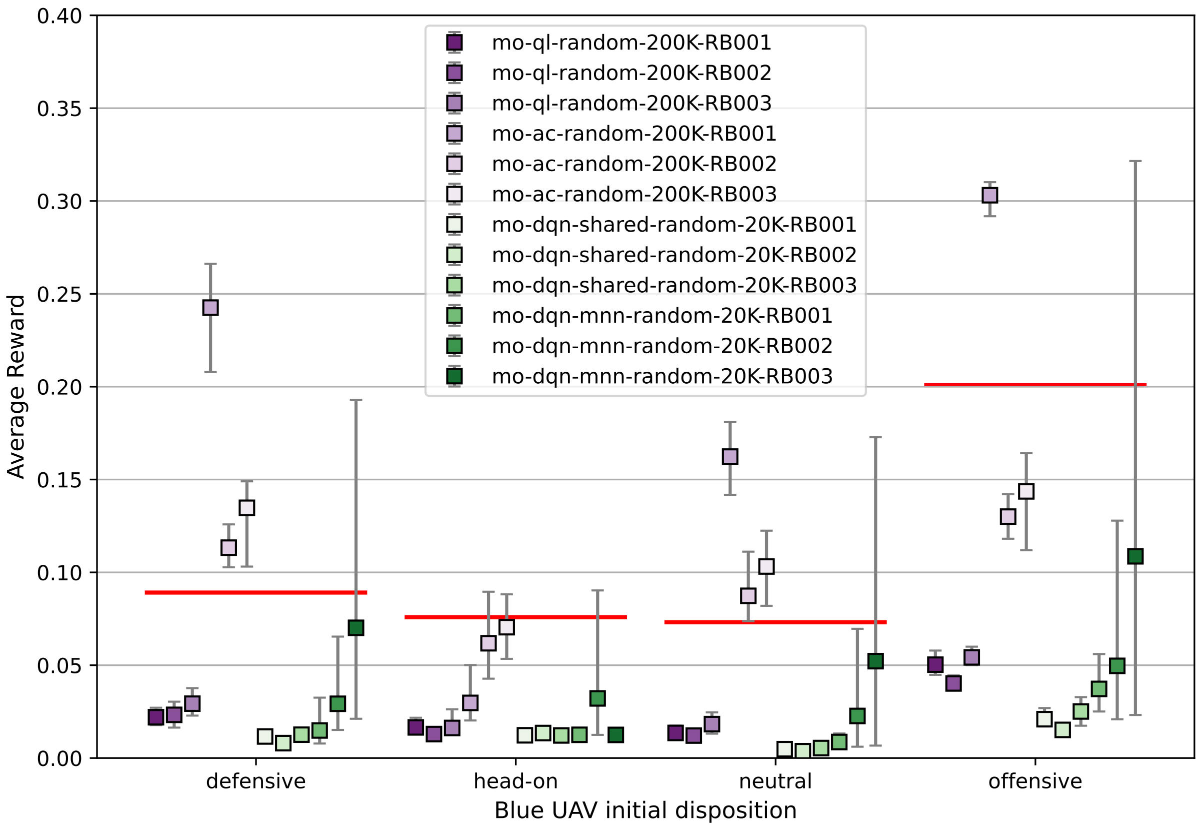


Figure 7.4: The performance of all agents using reward builders RB001, RB002 and RB003. The average rewards are based on the original scalar McGrew score

Learning Times

Table 7.1 shows the time taken, in hours, by multiobjective Q(λ) and multiobjective ACET agents to complete 200,000 episodes and by multiobjective DQN agents to complete 20,000 episodes. Table 7.2 shows the cost per episode in seconds. Figures 7.5 and 7.6 show means and 95% confidence intervals after resampling 10,000 times the data in Tables 7.1 and 7.2, respectively. The learning times for the multiobjective Q(λ) and ACET methods are higher than the single-objective versions of the methods, because the multiobjective agents employ multiple Q-tables. For the multiobjective DQN method with a shared network, the learning times are close to the

Agent	Mean	Median	Standard Deviation
ql-random-200K	17.42	17.42	0.14
mo-ql-random-200K-RB001	22.65	22.63	0.51
mo-ql-random-200K-RB002	22.72	22.71	0.27
mo-ql-random-200K-RB003	22.38	22.36	0.15
ac-random-200K	32.08	32.13	0.17
mo-ac-random-200K-RB001	61.68	61.69	3.34
mo-ac-random-200K-RB002	60.12	59.29	2.72
mo-ac-random-200K-RB003	60.42	59.14	1.90
dqn-random	39.46	39.46	0.09
mo-dqn-shared-random-20K-RB001	45.08	45.00	0.30
mo-dqn-shared-random-20K-RB002	45.82	45.47	1.79
mo-dqn-shared-random-20K-RB003	32.60	32.24	1.37
mo-dqn-mnn-random-20K-RB001	152.68	152.74	5.14
mo-dqn-mnn-random-20K-RB002	145.74	146.10	5.40
mo-dqn-mnn-random-20K-RB003	147.13	147.13	7.18

Table 7.1: MORL agent learning times in hours for RB001, RB002 and RB003. The highlighted rows are learning times for single-objective equivalents from previous chapters.

learning times of the single-objective DQN method, as the former only uses a single neural network. However, the multiobjective DQN method with multiple networks costs much more to train due to the fact the neural networks are updated sequentially.

Agent	Mean	Median	Standard Deviation
ql-random-200K	0.31	0.31	0.00
mo-ql-random-200K-RB001	0.41	0.41	0.01
mo-ql-random-200K-RB002	0.41	0.41	0.00
mo-ql-random-200K-RB003	0.40	0.40	0.00
ac-random-200K	0.58	0.58	0.00
mo-ac-random-200K-RB001	1.11	1.11	0.06
mo-ac-random-200K-RB002	1.08	1.07	0.05
mo-ac-random-200K-RB003	1.09	1.06	0.03
dqn-random	7.10	7.10	0.02
mo-dqn-shared-random-20K-RB001	8.11	8.10	0.05
mo-dqn-shared-random-20K-RB002	8.25	8.18	0.32
mo-dqn-shared-random-20K-RB003	5.87	5.80	0.25
mo-dqn-mnn-random-20K-RB001	27.48	27.49	0.93
mo-dqn-mnn-random-20K-RB002	26.23	26.30	0.97
mo-dqn-mnn-random-20K-RB003	26.48	26.48	1.29

Table 7.2: MORL learning time/episode in seconds for RB001, RB002 and RB003. The highlighted rows are learning times for single-objective equivalents from previous chapters.

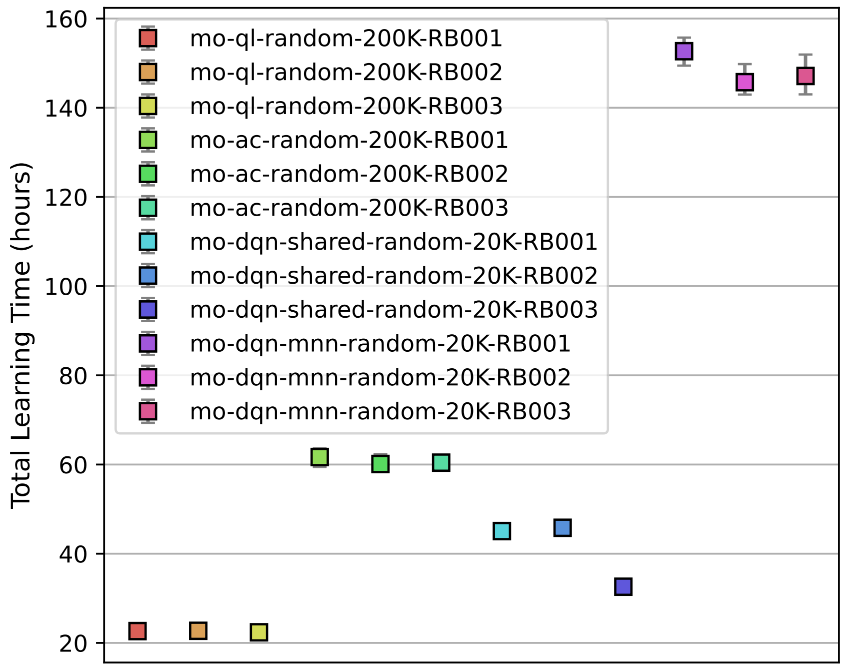


Figure 7.5: MORL learning times for RB001, RB002 and RB003 in hours. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

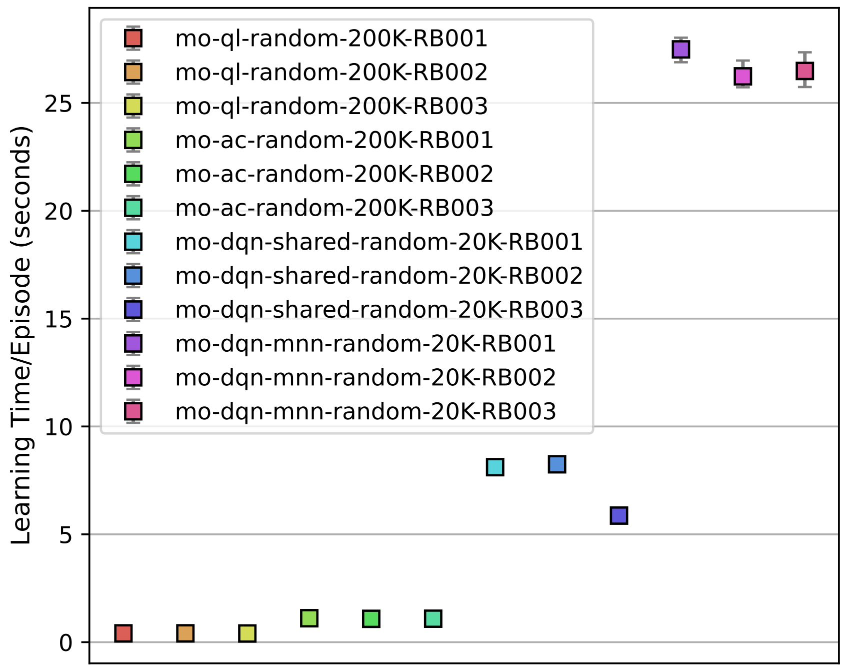


Figure 7.6: MORL learning times per episode for RB001, RB002 and RB003 in seconds. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

7.4 Experiment 2 (RB004 - RB007)

This experiment uses the most successful algorithm from Experiment 1, multiobjective ACET, to evaluate four new reward functions/builders. Unlike the reward builders in Experiment 1, that were based on the decomposition of the McGrew score, the reward builders in this experiment use the agent's McGrew score and the inverse of the opponent's McGrew score. This strategy has the potential to outperform the single-objective agents in the previous chapters as there is now more information to aid in the agent's learning.

7.4.1 Experimental Setup

All the trials in this experiment use the state discretisation and action space explained in Section 7.2. In addition, the following reward functions or reward builders, code-named RB004, RB005, RB006 and RB007, are used. Each of the reward builders returns a fraction of Blue's McGrew score and a fraction the negative of Red's McGrew score.

- RB004: [blue McGrew_score, -red McGrew_score]
- RB005: [0.50 blue McGrew_score, -0.50 red McGrew_score]
- RB006: [0.75 blue McGrew_score, -0.25 red McGrew_score]
- RB007: [0.25 blue McGrew_score, -0.75 red McGrew_score]

7.4.2 Results

Learning Curves

Figure 7.7 shows the learning curve of the multiobjective ACET agents when trained using reward builders RB004, RB005, RB006 and RB007, respectively. All the learning curves are obtained by running the agents ten times with ten different random seeds and averaging every 50 consecutive data points into a new data point. The darker line is the mean of the new data points over seeds. The shaded area represents the mean ± the standard deviation. The average of each reward is shown in the legend. Because the second reward has a potential value between 0 and -1 (due to the fact that the opponent's McGrew score ranges between 0 and 1), the range of the Y axis in each graph has been designed to be between -0.2 and 0.4. A grey line is drawn along Y = 0 to make it easier to view how the average rewards progress over episodes.

The learning curves show that different reward builders lead to different average rewards. Using the reward builders RB004, RB005 and RB006, the multiobjective ACET agents managed to increase the first reward but obtained different results with regard to the second reward, which ideally should always be maximised to zero, a position where Blue is behind Red and flying in the same direction as its opponent. RB004 was not able to keep the second reward close to zero all the time, whereas RB005 and RB006 were able to do it. RB006, the only reward builder with a larger

Agent	Mean	Median	Standard Deviation
mo-ac-random-200K-RB004	35.94	36.06	2.26
mo-ac-random-200K-RB005	37.64	36.81	3.01
mo-ac-random-200K-RB006	37.93	36.86	3.11
mo-ac-random-200K-RB007	38.12	37.10	2.98

Table 7.3: Learning times in hours for generating policies

weight for the first reward than the second reward, was especially successful in maximising both rewards. For reward builder RB007, the agents failed to increase either reward. It appears that a larger weight for the first reward is fundamental in increasing *both* rewards. This is supported by the fact that RB007, a reward builder with a higher weight for the second reward, failed to increase even the first reward.

Performance

Figure 7.8 shows the performance of the multiobjective ACET agents paired with reward builders RB004, RB005, RB05 and RB007, over ten trials. Each vertical bar shows the mean and 95% confidence interval after resampling the original data, as explained in 3.5. The average rewards are based on the original scalar McGrew score. Included in the chart for comparison is the performance of a single-objective ACET agent trained using random initial positions, just as the multiobjective ACET agents. Three of the four agents, those using RB004, RB005 and RB006, perform better than the Smart Pursuit baselines as well as the single-objective ACET agent. The performance differences of these three agents compared to the other agents are all statistically significant in all four initial dispositions. The only exception is the difference between the mo-ac-random-200K-RB004 and ac-random-200K agents for the head-on initial disposition, in which the difference is not statistically significant. One multiobjective ACET agent, the one using reward builder RB007, does not fare well. It performs worse than the baselines and the single-objective ACET agent. Considering that both the single- and multiobjective ACET agents trained for the same number of episodes, this chart shows the potential of the multiobjective methods.

Learning Times

Table 7.3 shows the learning times, in hours, of the multiobjective ACET agents with reward builders RB004, RB005, RB006 and RB007. Table 7.4 shows the per-episode costs of the same agents. The test results are resampled using the bootstrap method and the means and 95% confidence intervals are shown in Figure 7.9.

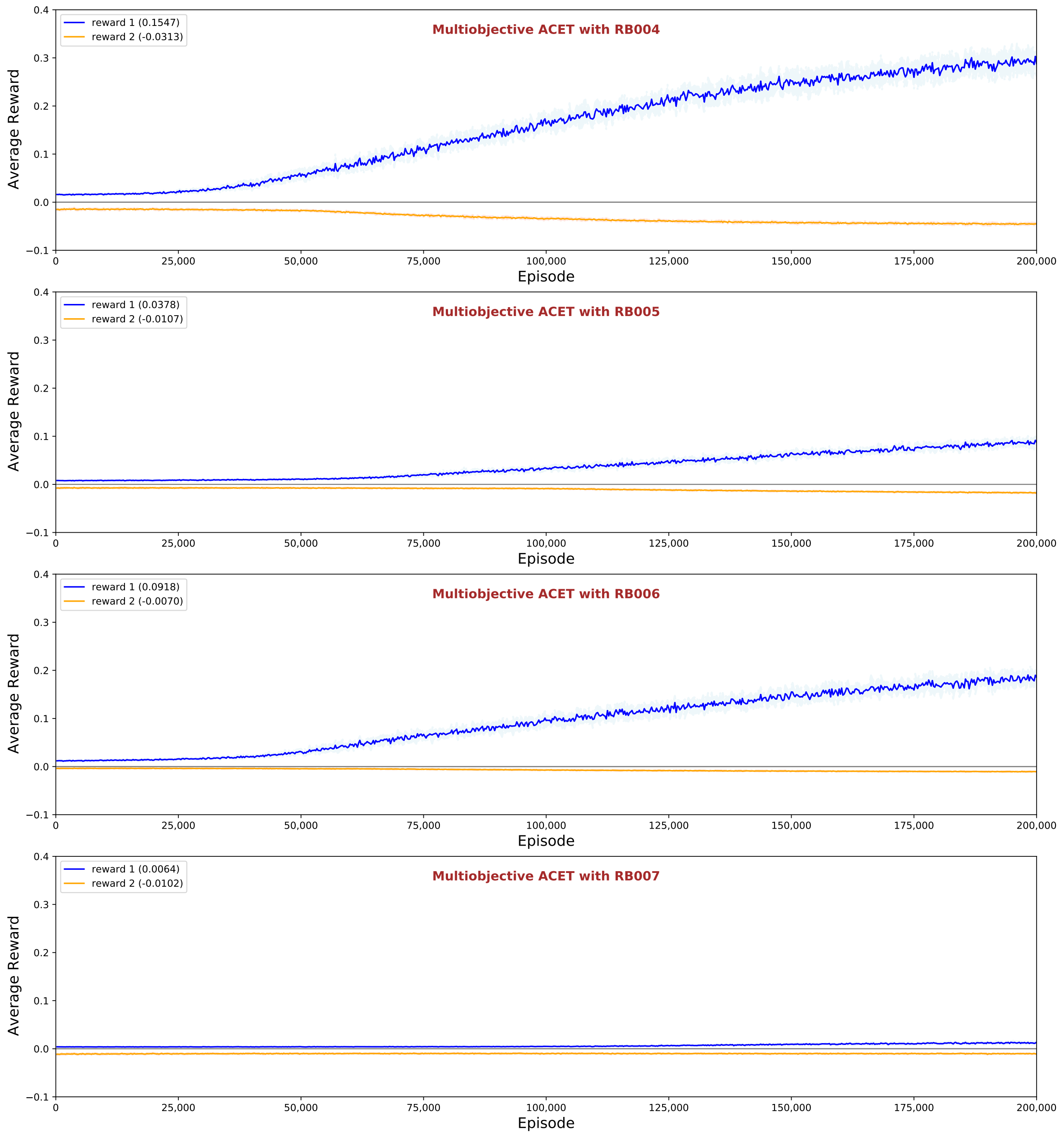


Figure 7.7: The learning curves of the multiobjective ACET agents for random initial positions trained using reward builders RB004, RB005, RB006 and RB007. The curves are obtained by running the agents ten times with ten different random seeds and averaging every 50 consecutive data points. The darker line is the mean of the new data points over seeds. The shaded area represents the mean ± the standard deviation. The values in the legend show the averages for all trials.

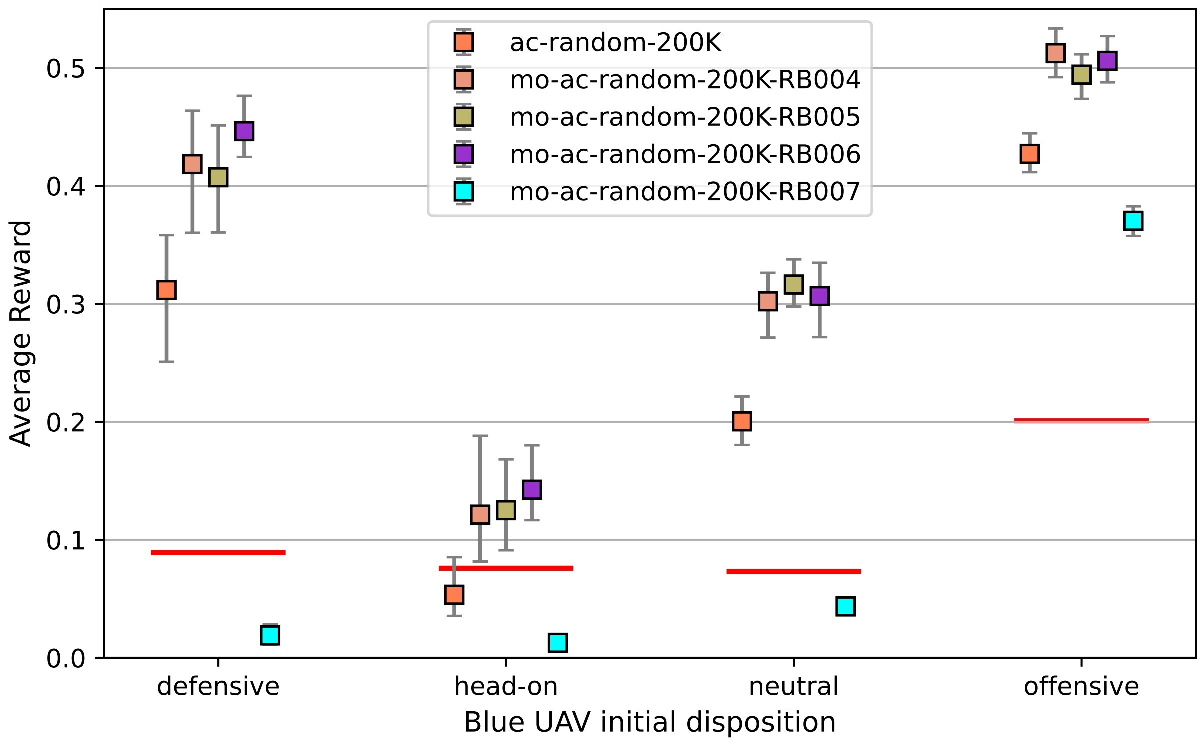


Figure 7.8: The performance of multiobjective actor-critic agents utilising reward builders RB004, RB005, RB006 and RB007, compared to a single-objective ACET agent. The average rewards are based on the original scalar McGrew score. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

Agent	Mean	Median	Standard Deviation
mo-ac-random-200K-RB004	0.65	0.65	0.04
mo-ac-random-200K-RB005	0.68	0.66	0.05
mo-ac-random-200K-RB006	0.68	0.66	0.06
mo-ac-random-200K-RB007	0.69	0.67	0.05

Table 7.4: Learning times in hours for generating policies

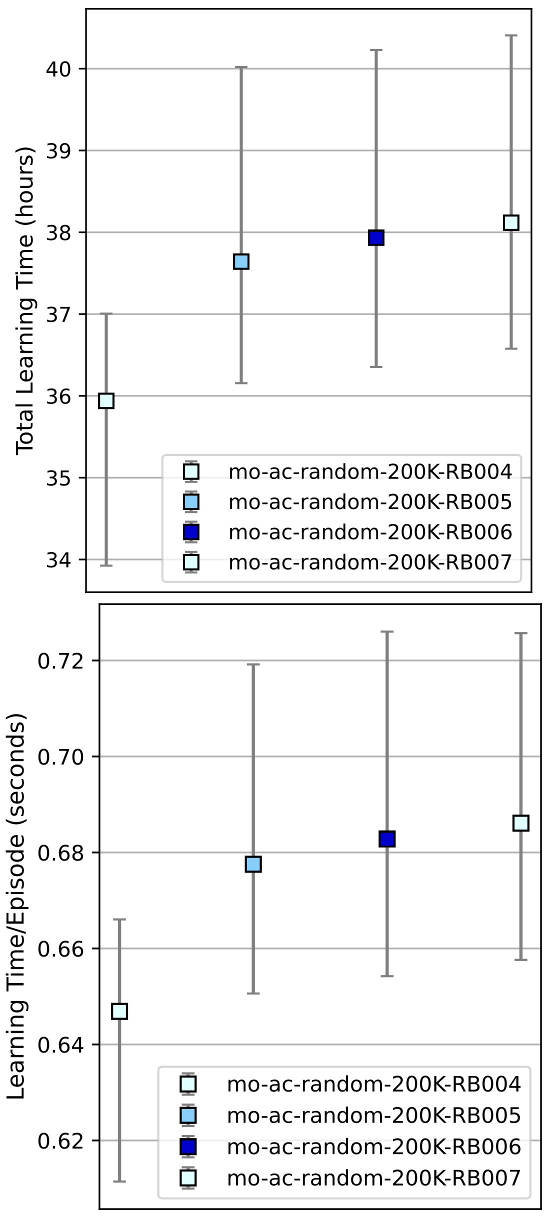


Figure 7.9: MORL learning times for RB004, RB005, RB006 and RB007 in hours (top) and learning times per episode in seconds (bottom)

7.5 Summary

The air combat manoeuvring domain has multiple, often conflicting, objectives, which makes it a MORL problem. In this chapter two experiments are conducted. Rather than handling the problem as a truly multiobjective problem (i.e. by finding a trade-off between conflicting objectives), the experiments use multiobjectivisation Brys et al. (2014) to decompose the reward into simpler components. This approach might be expected to outperform an agent with scalar reward, as shown for instance in Russel and Zimdars (2003) and Tesauro et al. (2008).

In the first experiment, four different MORL algorithms, two tabular and two deep RL, are evaluated, using three reward builders that each returns three rewards. The three rewards—the range score, AA score and ATA score—are decomposed from the McGrew score used in single-objective agents in Chapters 5, 6 and 7. The results show that the multiobjective ACET method is the only method that managed to outperform the Smart Pursuit baselines. All the other methods, despite showing success during learning, perform worse than the baselines.

The second experiment uses four multiobjective ACET agents to test four reward builders, code-named RB004, RB005, RB006 and RB007, that each returns two rewards. The first reward is the agent's McGrew score and the second the inverse of the opponent's McGrew score. The four reward builders differ in the weightings of the two reward components. The performance of three of the four agents, those using RB004, RB005 and RB006, are better than the Smart Pursuit baselines. One agent, the one using RB007, performs much worse than the baselines.

One question that may spring to mind after reviewing the MORL experiments in this chapter is, How well do the MORL agents fare compared with single-objective agents? Is there any point of doing multiobjectivisation? These questions are answered in Chapter 8.

Chapter 8

Evaluation

Four categories of reinforcement learning (RL) methods—tabular, deep, discrete-to-deep and multiobjective—are implemented and tested for this thesis. This chapter evaluates and discusses the best performing agents in each category: tabular actor-critic with eligibility traces (ACET), deep Q-network (DQN), discrete-to-deep supervised Q-value learning (D2D-SQL) and multiobjective ACET. Algorithms that did not yield successful learning, such as Q(λ) and multiobjective DQN, and algorithms that are the worst performers in their category, such as DQN with a target network, double DQN and proximal policy optimisation (PPO), are excluded. In addition, because one of the aims of this chapter is to find out which one is the best method for addressing the air combat manoeuvring problem, not all agents implementing a successful method are included in every comparison. For example, ac-004 and ac-005 are not included in indirect comparisons because their performance is much worse than those of ac-001 and ac-002 and any agent that beats ac-001 and ac-002 implicitly outperforms ac-004 and ac-005. However, they are still included in direct comparisons against other agents because in direct comparison the exact performance of an agent cannot be predicted by simply monitoring the performance of other similar agents.

The agents compared in this chapter are as follows. The number of learning episodes is appended to the agent's original name due to different numbers of learning episodes being used in the different approaches.

- Six tabular ACET agents trained for 200,000 episodes from fixed initial positions 001, 002, 003, 004 and 005 and random initial positions. The code names for these agents are ac-001-200k, ac-002-200k, ac-003-200k, ac-004-200k, ac-005-200k and ac-random-200k.
- Six DQN agents trained for 20,000 episodes using fixed initial positions 001, 002, 003, 004 and 005 and random initial positions. Their code names are dqn-001-20k, dqn-002-20k,

dqn-003-20k, dqn-004-20k, dqn-005-20k and dqn-random-20k.

- A D2D-SPL agent (code-named d2dspl) and three D2D-SQL agents (code-named d2dsql-11K, d2dsql-12K and d2dsql-13K) from Experiment 2 in Chapter 6.
- Four multiobjective ACET agents trained for 200,000 episodes using random initial positions and reward builders RB004, RB005, RB006 and RB007 as explained in Chapter 7. The four agents are code-named mo-ac-200k-RB004, mo-ac-200k-RB005, mo-ac-200k-RB006 and mo-ac-200k-RB007, respectively.

For neural-network-based agents, the hyperparameters are the same as the one used in the original experiments in Chapters 5, 6 and 7.

This chapter starts by evaluating the learning times and curves of the various methods and continues with indirect and direct performance comparisons. The indirect comparison is performed as in the previous chapters, whereby RL agents are compared against an engineered agent that provides a baseline. The direct comparison pits RL agents against each other. Judging from their learning curves, none of the selected agents has obtained an optimum policy and in theory can improve their performance given more learning time. However, to make comparison fair, they have been allowed to spend similar amounts of wall-clock learning time. The last section of this chapter analyses the learned policies of the RL methods by observing sample trajectories from the direct comparisons to discover any learned skills and behaviours developed during learning.

8.1 Learning Times

This section presents the learning times of the selected agents. The code name for each agent, as explained in Chapter 3, is a combination of the method name, initial position and number of learning episodes. For example, ac-001-200K is an ACET agent that learns from initial position 001 for 200,000 episodes. dqn-random-20K is a DQN agent learning from random initial positions for 20,000 episodes. The multiobjective agents have one more component to their code, the reward builder. For instance, mo-ac-random-200K-RB004 is a multiobjective ACET agent that learned from random initial positions for 200,000 episodes and whose reward structure is based on reward builder RB004. The reward builders for the MORL agents were discussed in Chapter 7. The numbers for the D2D-SPL and D2D-SQL agents are from Experiment 2 in Chapter 6, whereby off-policy data was generated by running an ACET agent for 50,000 episodes starting from random initial positions.

Table 8.1 shows the number of hours spent by the agents to learn a policy. Table 8.2 presents the learning time per episode in seconds. It can be seen that the tabular methods, both single-objective and multiobjective, are much faster than the deep reinforcement learning (RL) methods. The learning time of a typical deep RL method is about twelve times slower than that of a

Agent	Mean	Median	Standard Deviation
ac-001-200K	31.17	31.21	0.73
ac-002-200K	30.66	30.59	3.54
ac-003-200K	31.16	31.33	0.47
ac-004-200K	31.01	30.91	0.56
ac-005-200K	30.98	31.30	0.66
ac-random-200K	32.08	32.13	0.17
dqn-001-20K	36.19	36.81	4.17
dqn-002-20K	36.88	35.96	5.45
dqn-003-20K	37.76	37.12	4.03
dqn-004-20K	38.15	38.12	1.96
dqn-005-20K	38.24	38.06	1.97
dqn-random-20K	39.46	39.46	0.09
d2dspl	8.78	8.64	0.72
d2dsql-random-11K	24.68	24.68	0.16
d2dsql-random-12K	26.17	26.25	0.32
d2dsql-random-13K	27.59	27.65	0.29
mo-ac-random-200K-RB004	35.94	36.06	2.26
mo-ac-random-200K-RB005	37.64	36.81	3.01
mo-ac-random-200K-RB006	37.93	36.86	3.11
mo-ac-random-200K-RB007	38.12	37.10	2.98

Table 8.1: Learning times in hours

single-objective tabular method. A multiobjective tabular method is about twenty percent slower than a single-objective counterpart because the former updates its value tables sequentially due to the difficulty in implementing multithreading in Python and the fact that learning is restricted to a single CPU core, a decision that is explained in Appendix A. The per-episode learning time of the multiobjective tabular methods could be closer to the learning time of the single-objective tabular methods if all value tables are updated concurrently. Nonetheless, per episode, the multiobjective tabular methods are still ten times faster than the deep RL methods.

The learning time for the D2D-SPL agent is the sum of the reinforcement and supervised phases. The learning times for the D2D-SQL agents are the sums of both reinforcement phases and the supervised phase. The D2D-SPL and D2D-SQL agents have the lowest learning times because their off-policy data were obtained by running an ACET agent for 50,000 episodes, which is a quarter of the number of learning episodes of the tabular ACET agents.

The data in Tables 8.1 and 8.2 are resampled using the bootstrap method, as explained in 3.5, and their means and 95% confidence intervals displayed in Figures 8.1 and 8.2, respectively.

Agent	Mean	Median	Standard Deviation
ac-001-200K	0.56	0.56	0.01
ac-002-200K	0.55	0.55	0.06
ac-003-200K	0.56	0.56	0.01
ac-004-200K	0.56	0.56	0.01
ac-005-200K	0.56	0.56	0.01
ac-random-200K	0.58	0.58	0.00
dqn-001-20K	6.52	6.63	0.75
dqn-002-20K	6.64	6.47	0.98
dqn-003-20K	6.80	6.68	0.73
dqn-004-20K	6.87	6.86	0.35
dqn-005-20K	6.88	6.85	0.35
dqn-random-20K	7.10	7.10	0.02
d2dspl	0.16	0.16	0.01
d2dsql-random-11K	0.44	0.44	0.00
d2dsql-random-12K	0.47	0.47	0.01
d2dsql-random-13K	0.50	0.50	0.01
mo-ac-random-200K-RB004	0.65	0.65	0.04
mo-ac-random-200K-RB005	0.68	0.66	0.05
mo-ac-random-200K-RB006	0.68	0.66	0.06
mo-ac-random-200K-RB007	0.69	0.67	0.05

Table 8.2: Learning times per episode in seconds

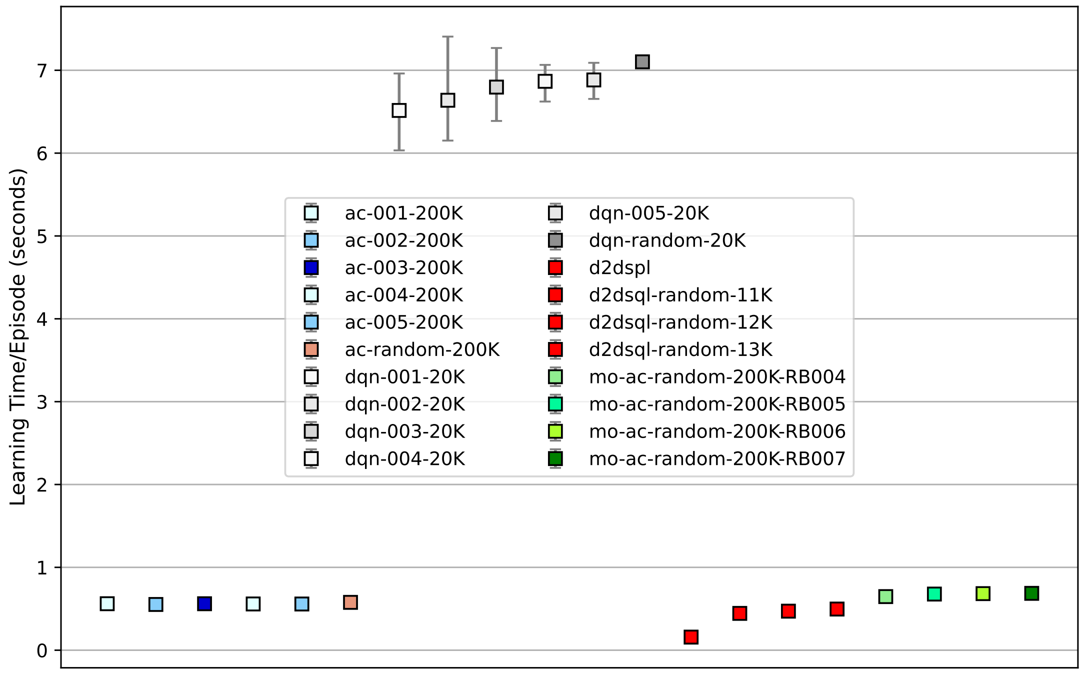


Figure 8.2: Total learning times per episode of all approaches in seconds. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

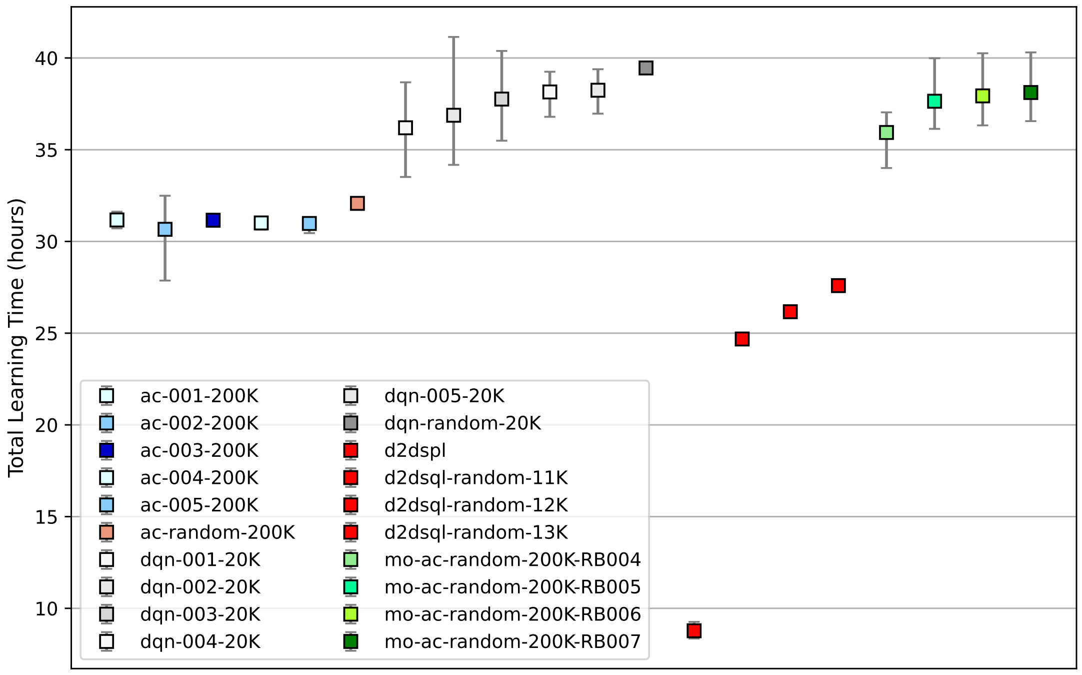


Figure 8.1: Total learning times of all approaches in hours. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

8.2 Learning Curves

An RL agent can be said to learn successfully if its average reward per episode increases with the episode, albeit not necessarily monotonically. This section compares the learning curves of the single-objective tabular and deep RL methods. Because MORL agents have multiple rewards, meaningful comparison between their learning curves and those of single-objective agents is not possible. However, their performance can still be compared by measuring their McGrew scores, as shown in the next sections in this chapter.

Figures 8.3 to 8.7 display the learning curves of the ACET and DQN agents that start from initial positions 001, 002, 003, 004 and 005, respectively. Figure 8.8 compares the learning curves of the ACET and DQN agents that start from random initial positions. All learning curves are obtained by running the agents with ten different random seeds. Every 250 consecutive data points are averaged into one new data point. The darker line shows the mean of the new data points over seeds. The shaded area represents the mean ± the standard deviation. The values in the legend show the averages for all trials.

All the ACET agents learn for 200,000 episodes and the DQN agents learn for 20,000 episodes. Those figures show that all the DQN agents achieve higher average rewards in 20,000 episodes than the ACET agents. In initial positions 003, 004 and 005, the DQN agents even learn more in 20,000 episodes than the ACET agents manage to learn in 200,000 episodes. In the random initial position cases, initially the ACET agents learn faster but then are overtaken by the DQN agents. The better performance of the deep RL method than the tabular method is consistent with the recent success stories of deep RL methods, especially those of DQN (Mnih et al., 2013, 2015), in solving high-dimensional continuous problems. Due to the limited number of states used that ACET can support, its ability to generalise is weaker than the deep neural network used in DQN, which explains the learning curves in this experiment.

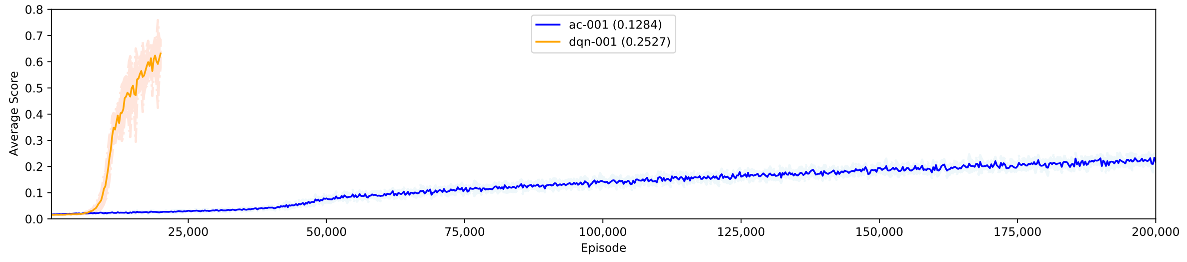


Figure 8.3: AC and DQN learning curves for initial position 001

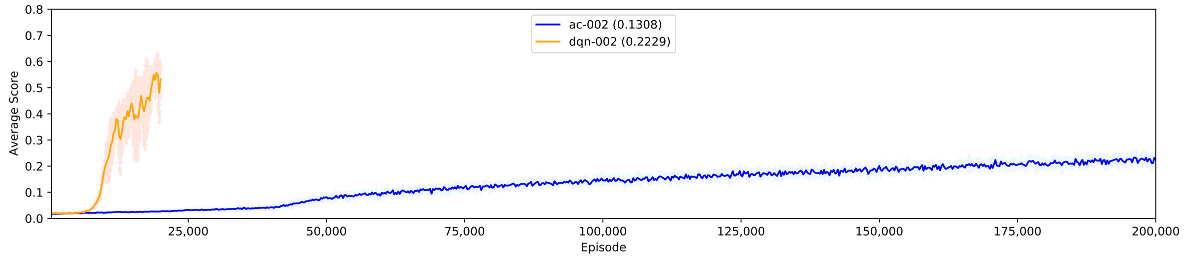


Figure 8.4: AC and DQN learning curves for initial position 002

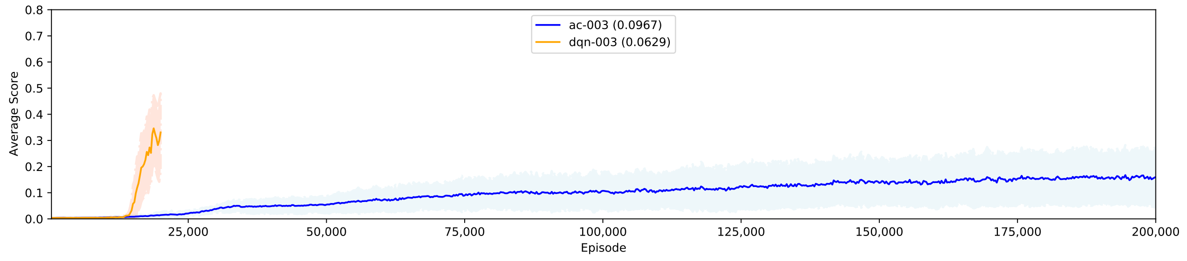


Figure 8.5: AC and DQN learning curves for initial position 003

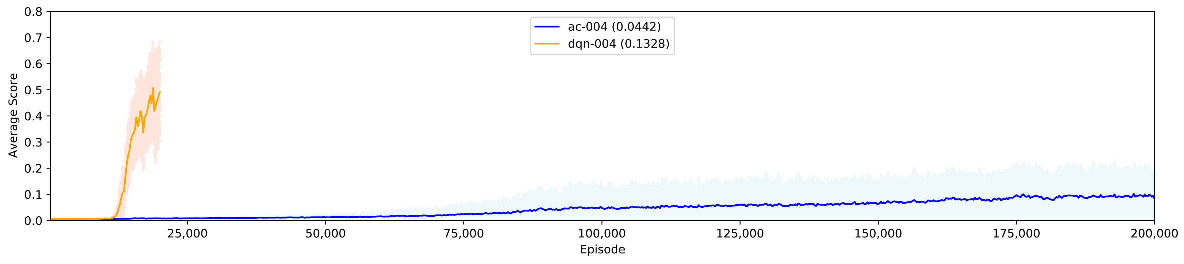


Figure 8.6: AC and DQN learning curves for initial position 004

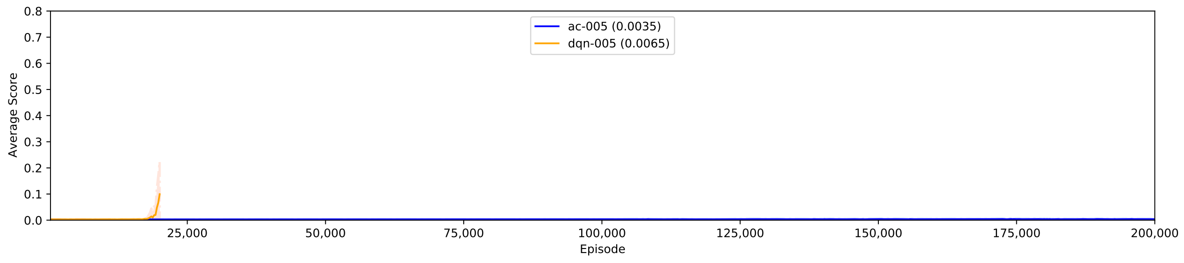


Figure 8.7: AC and DQN learning curves for initial position 005

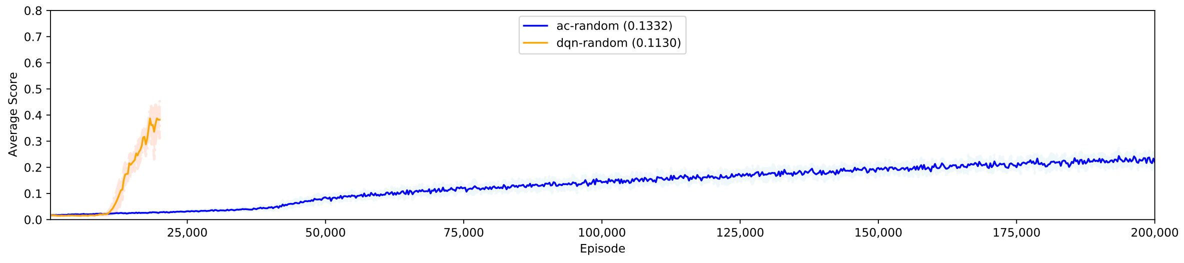


Figure 8.8: AC and DQN learning curves for initial random positions

8.3 Indirect Performance Comparison

This section presents the results of indirectly comparing the learning agents. They are indirect because each agent faces a scripted agent flying in a straight line and at a constant speed and the result is the average McGrew scores over all the timesteps during testing. This test is repeated 144 times from 144 different initial layouts between the agent and the scripted agent. The 144 initial layouts are grouped into four different dispositions and the results from each group are averaged. To measure how well the agents perform, an engineered agent, the Smart Pursuit agent, is also run against the same scripted agent to get the score of the engineered agent. The result of the engineered agent is the baseline and depicted in the plot as horizontal red lines.

Figure 8.9 shows the average score and 95% confidence interval of each agent when facing the scripted agent after the corresponding test results are bootstrap-resampled, as described in

3.5. By comparing the performance of all agents relative to the other agents, the chart exhibits how well an agent performs.

The top performers are the first three multiobjective agents that outperform the other agents in three of the four initial dispositions. The performance differences between the three multiobjective agents and the other agents are statistically significant for the neutral initial disposition and not statistically significant in the other categories. This indicates that decomposing the McGrew score into separate components of a reward vector is beneficial for training of the agent, probably because it provides less sparse rewards as the scalar McGrew reward will be near zero if any of the angular or range measures are unfavourable. Considering that the learning costs of the multiobjective ACET agents are similar to the single-objective ACET agents and much less than the deep RL agents, the results shows promising prospects of the MORL techniques.

The D2D-SPL and D2D-SQL agents are the cheapest to train and their performance is comparable to those of the single-objective ACET and deep RL agents. This is an interesting finding and probably justifies further research on the discrete-to-deep algorithms in the future.

Another aspect of the chart shows that the confidence intervals of the DQN, D2D-SPL and D2D-SQL agents are much higher than those of the single-objective and multiobjective ACET agents. The high variances of the deep RL agents could be due to DQN's instability, as discussed in Chapter 5.

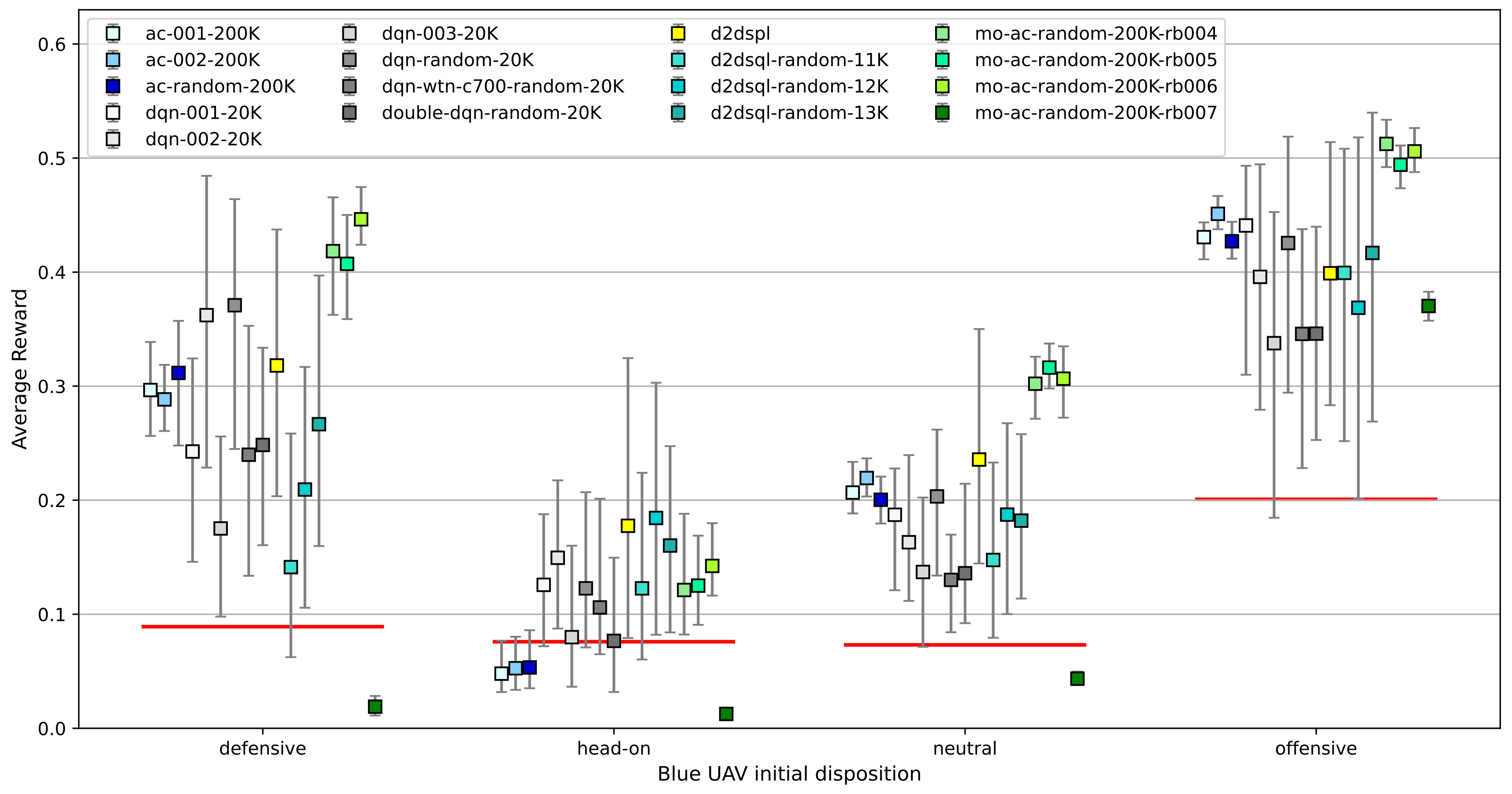


Figure 8.9: Indirect performance comparison of ACET, DQN, D2D and MORL agents

8.4 Direct Performance Comparison

The performance comparisons of the RL agents and the Smart Pursuit baseline engineered agent in the previous section are indirect because the RL agents never face the engineered agent. The Pure Pursuit and Stern Conversion engineered agents were not used because the Smart Pursuit agent outperforms them in all initial dispositions. This section presents direct comparisons of the RL agents and the Pure Pursuit, Smart Pursuit and Stern Conversion agents by placing them in a one-on-one match. Like in the previous section, a test involving an RL agent and an engineered agent is repeated 144 times from 144 different initial layouts between the agent and the engineered agent and the 144 initial layouts are grouped into four different dispositions and the results from each group are averaged. Each test is scored as either a Blue win (if Blue wins more than 72 times), a Red win (if Red wins more than 72 times), or a tie (if Blue and Red win the same number of times).

This section starts with a software robustness test by pitting an agent against itself. The results should make it easy to identify any defects in the software. It then evaluates the direct performance comparison against the engineered agents and restricted engineered agents. A restricted engineered agent is one whose physical capabilities have been restricted to be equal to those of the RL agent.

8.4.1 A Reinforcement Learning Agent against Itself

It is interesting to know what happens if two agents based on the same policy face each other. In this subsection the single-objective ACET agent (Blue) that learned a policy after training for 200,000 episodes from random initial positions (code named ac-random-200K) tries to outmanoeuvre another agent (Red) powered by the same policy. Because of this, Red can be thought of as a clone of Blue. Both aircraft start from the same 144 initial positions used to test agents in the previous chapters and each test is conducted ten times using ten different random seeds. The results are given in Table 8.3, which show in every trial Blue wins 66 times and Red wins 66 times and there are twelve ties.

Figure 8.10 shows the symmetry in the trajectories between Blue and Red. The top trajectory shows the initial position where Blue starts behind Red and the bottom trajectory shows the trajectory for when Red starts behind Blue. The symmetry is expected of agents running the same deterministic policy and provides an indication that the software used to run the test is bug-free.

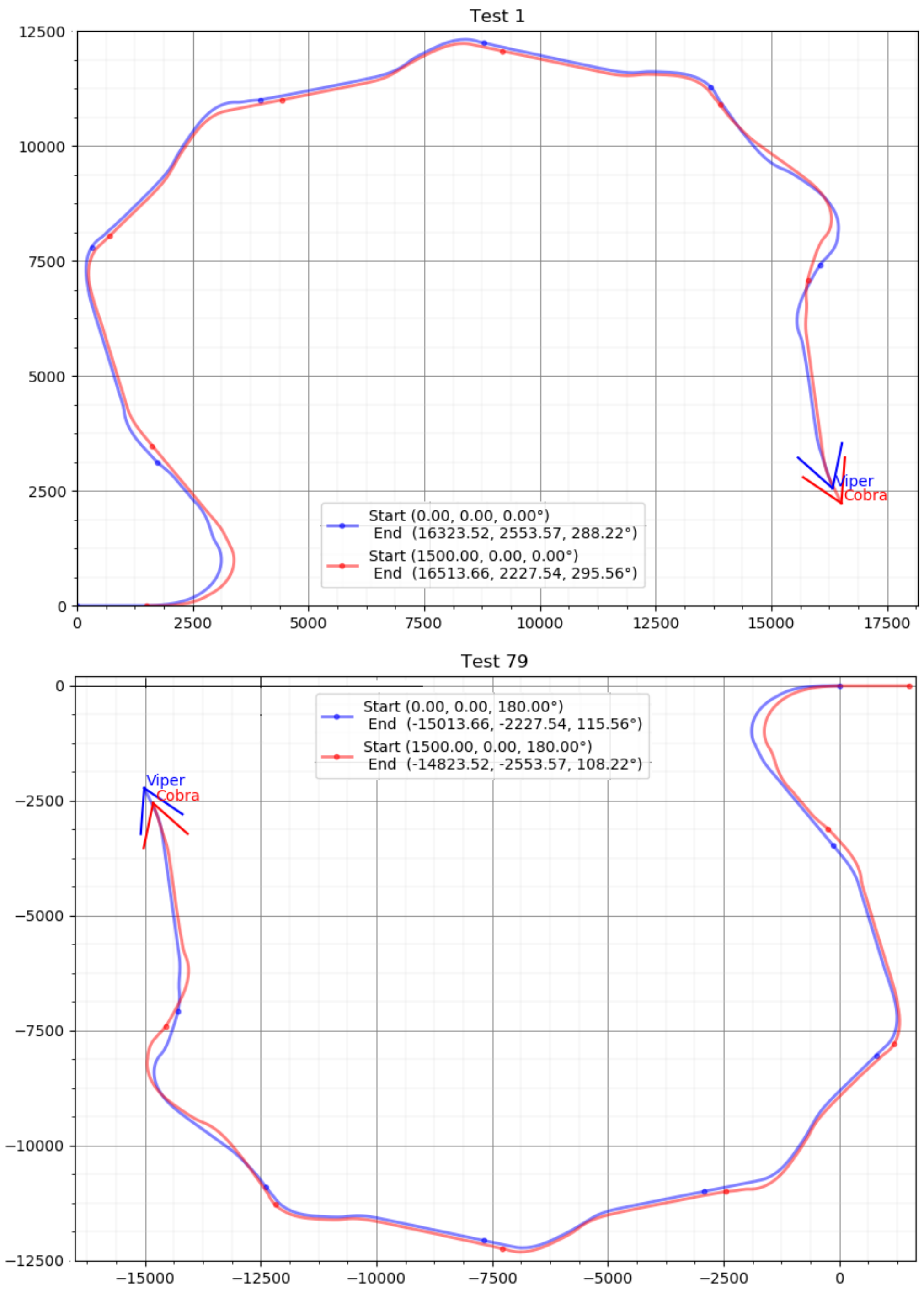


Figure 8.10: Trajectories 1 and 79 of ac-random-1m vs ac-random-1m

	Blue ac-random-200K	Red ac-random-200K
Trial 0	66	66
Trial 1	66	66
Trial 2	66	66
Trial 3	66	66
Trial 4	66	66
Trial 5	66	66
Trial 6	66	66
Trial 7	66	66
Trial 8	66	66
Trial 9	66	66

Table 8.3: Direct comparison of the ac-random-200K agent against itself

8.4.2 Direct Comparisons with Engineered Agents

This section shows direct performance comparison between the RL agents and the engineered agents. The RL agents are single- and multiobjective actor-critic agents and single-objective DQN agents and the engineered agents are Pure Pursuit, Smart Pursuit and Stern Conversion agents. Multiobjective DQN agents are not included because of their poor performance. Figures 8.11, 8.12 and 8.13 show the results for the Pure Pursuit, Smart Pursuit and Stern Conversion agents, respectively. Each graph shows the number of wins an agent has against an engineered agent. A positive value means the agent wins more than it loses. Because there are 144 matches in each trial, the maximum and minimum possible values for an agent are 144 and -144, respectively. The plots show the means and 95% confidence intervals of the resamples of the original data by using the bootstrap method. Only the best performing agents from each RL approach are included.

In all the graphs, the means of all the agents except dqn-002-20K are less than 0, indicating that on average the RL agents lose to the engineered agent. This is expected because the engineered agents are physically superior to the learning agents (which are limited to five actions). The fact that dqn-002-20K outperformed two of the three engineered agents shows that RL has the potential to effectively address the air combat manoeuvring problem.

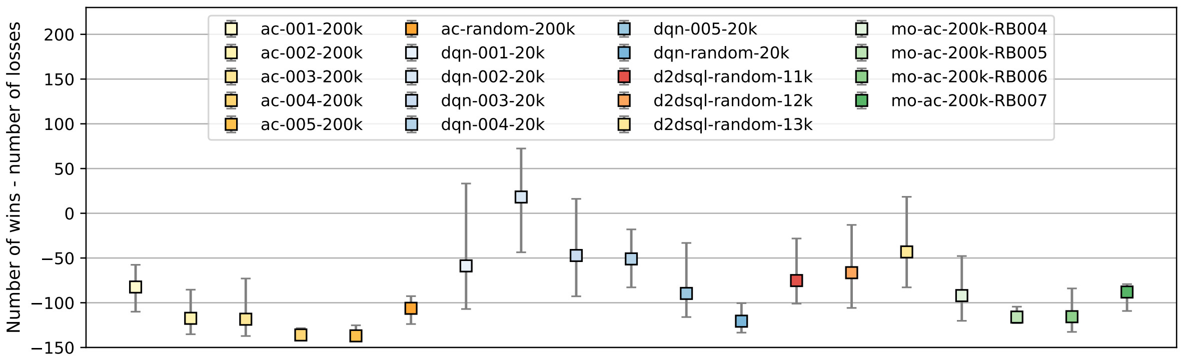


Figure 8.11: Direct comparisons with the Pure Pursuit agent. Each vertical bar show the mean and 95% confidence interval for the corresponding agent.

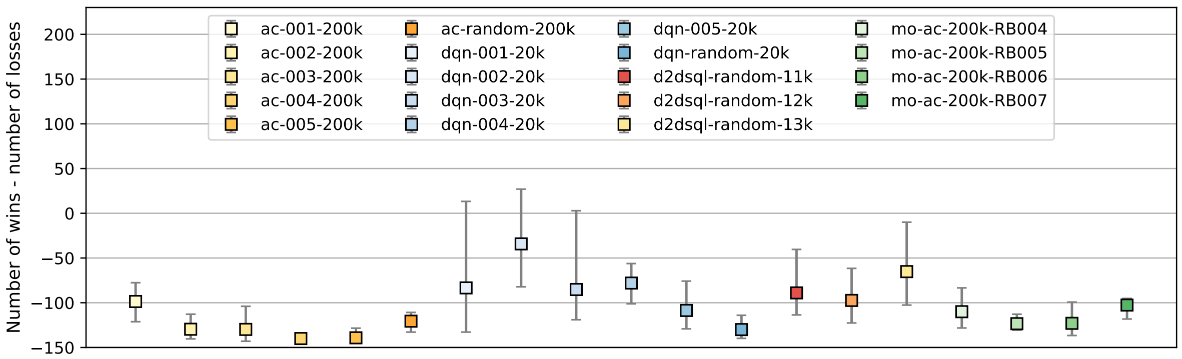


Figure 8.12: Direct comparisons with the Smart Pursuit agent. Each vertical bar show the mean and 95% confidence interval for the corresponding agent.

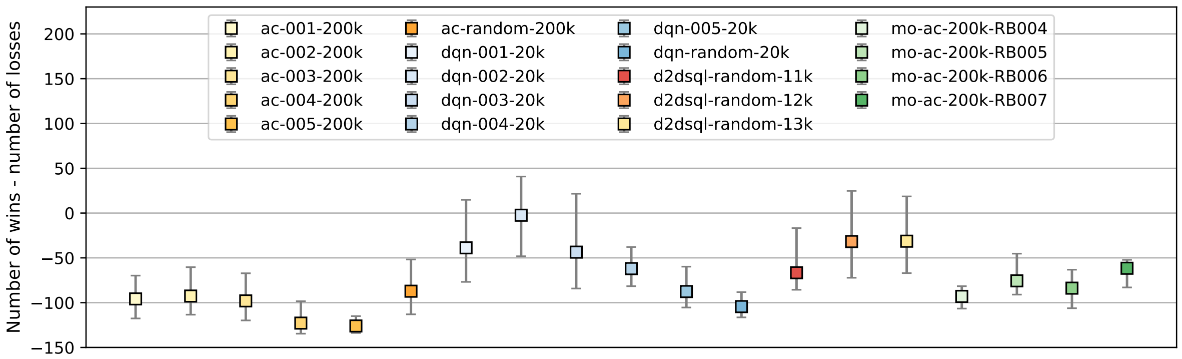


Figure 8.13: Direct comparisons with the Stern Conversion agent. Each vertical bar show the mean and 95% confidence interval for the corresponding agent.

8.4.3 Direct Comparisons with Restricted Engineered Agents

This section makes comparison fairer by making the engineered agents more physically similar to the RL agents by putting a limit on how they can move. In these tests, the Pure Pursuit agent is restricted to turning left and right by a maximum of 10°. The Smart Pursuit and Stern

Conversion agents are restricted to turning left and right by a maximum of 10° but are not restricted in how much they can adjust their speed.

Figures 8.14 to 8.16 show the performance of the ACET and DQN agents against restricted Pure Pursuit, restricted Smart Pursuit and restricted Stern Conversion agents, respectively. As expected, by making the engineered agents more similar physically to the RL agents, more learning agents are able to defeat the engineered agents. Even though the ACET and D2D-SQL agents lose to all the three restricted engineered agents, one of the DQN and three of the multiobjective MORL agents outperform two of the three engineered agents. In particular, among the DQN agents, the dqn-002-20K agent is the best performer as it beats all the three restricted engineered agents. Three of the four multiobjective ACET agents (those using reward builders RB004, RB005 and RB006) achieve similar feats, able to beat the Pure Pursuit and Smart Pursuit restricted agents. If the single- and multiobjective ACET agents are indirectly compared by looking at their numbers of wins in the three charts, it is apparent that the three multiobjective agents are much better than their single-objective competitors, which again shows the benefits of multiobjectivisation (Brys et al., 2014).

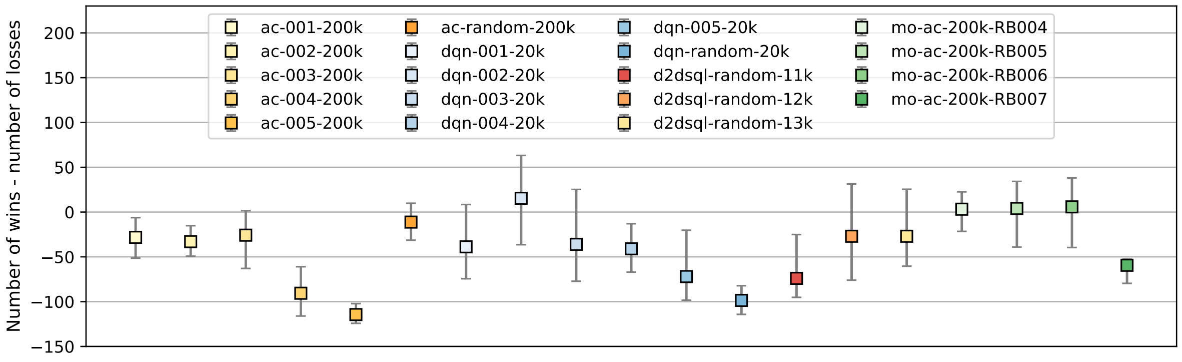


Figure 8.14: Direct comparisons with the restricted Pure Pursuit agent. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

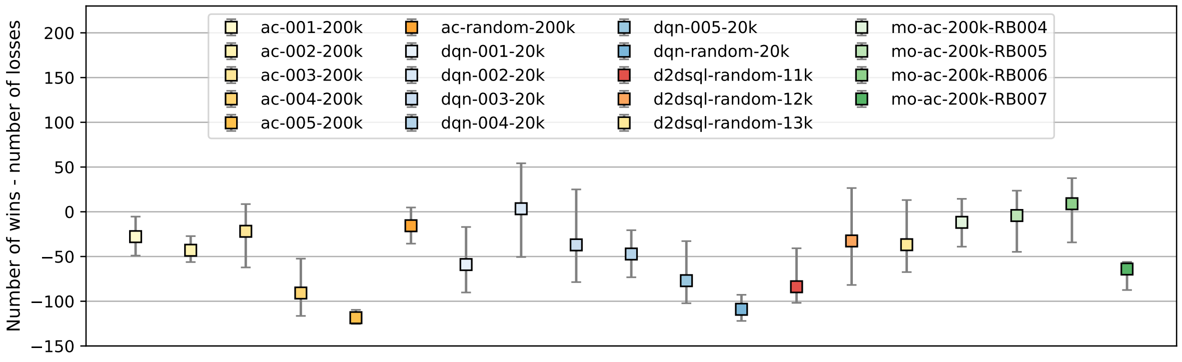


Figure 8.15: Direct comparisons with the restricted Smart Pursuit agent. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

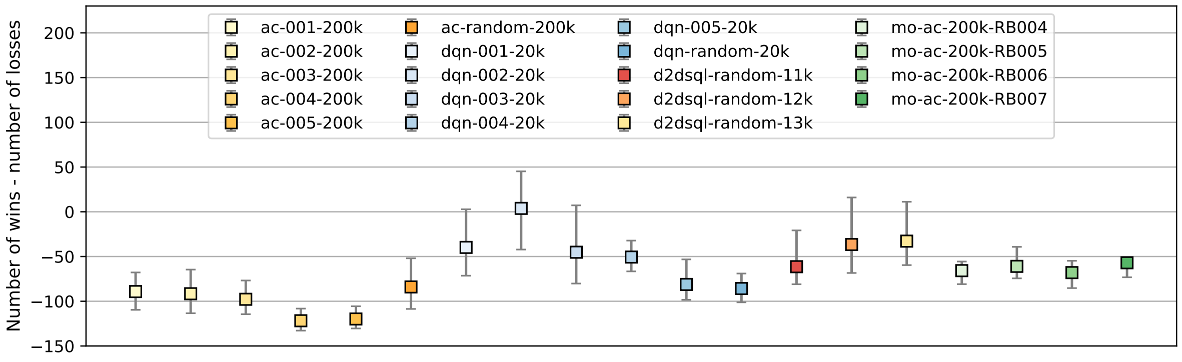


Figure 8.16: Direct comparisons with the restricted Stern Conversion agent. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

8.4.4 Direct Comparisons among RL Agents

This section compares the RL agents directly by pitting one RL agent against another.

DQN and Multiobjective ACET Agents vs. A Single-Objective ACET Agent

This section plots the performance of the DQN and multiobjective ACET agents against the best-performing single-objective ACET agent (ac-random-200K). Figure 8.17 shows the results. Three of the four multiobjective ACET agents outperform the single-objective ACET agent while all the DQN agents are worse than it. The performance of the multiobjective ACET agents is consistent with the results from the previous tests, however the fact that all the DQN agents lose is surprising as DQN agents have previously beaten the single-objective ACET agents. One explanation would be that the DQN agents are pitted against the best performer in its category, however it has to be admitted that some degree of randomness is present here.

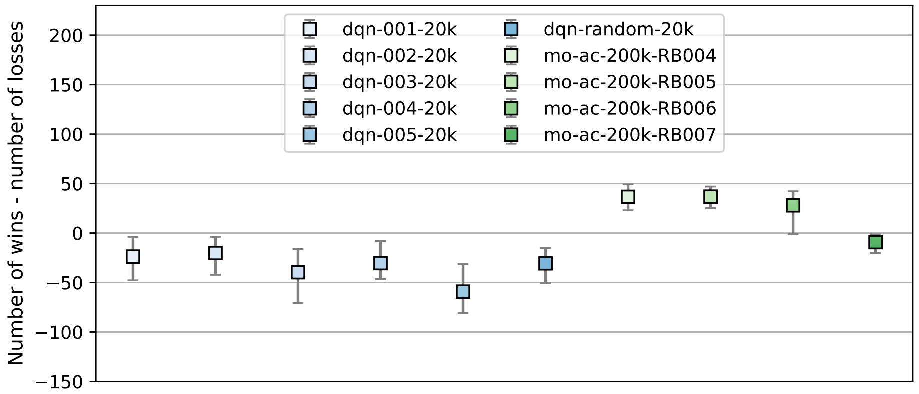


Figure 8.17: Direct comparisons with the best performing single-objective ACET agent (ac-random-200k). Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

DQN Agents vs. Multiobjective Actor-Critic Agents

This section compares the DQN agents and the four multiobjective ACET agents. The results are given in Figures 8.18, 8.19, 8.20 and 8.21. In the first three tests, the DQN agents underperform. In the last test, however, the DQN agents outperform the ac-mo-200k-RB007 agent, which is expected as the learning curve of ac-mo-200k-RB007 is the worst compared to the other multiobjective agents, as shown in Chapter 7. Again, the strong performance of the three multiobjective ACET agents can be attributed to multiobjectivisation (Brys et al., 2014).

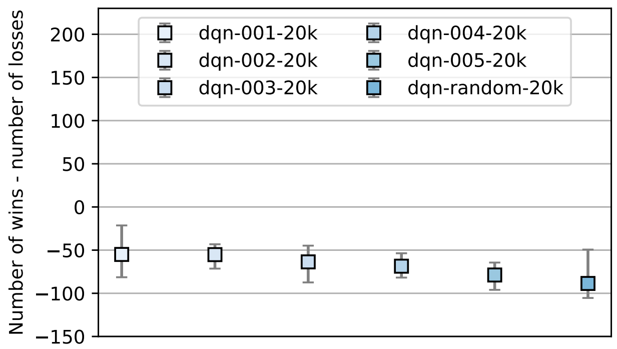


Figure 8.18: Direct comparisons with multiobjective ACET agent with RB004. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

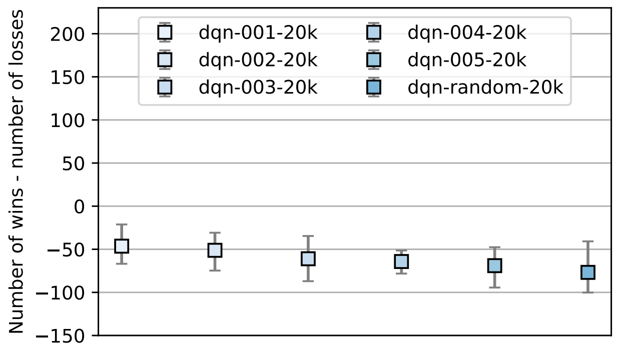


Figure 8.19: Direct comparisons with multiobjective ACET agent with RB005. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

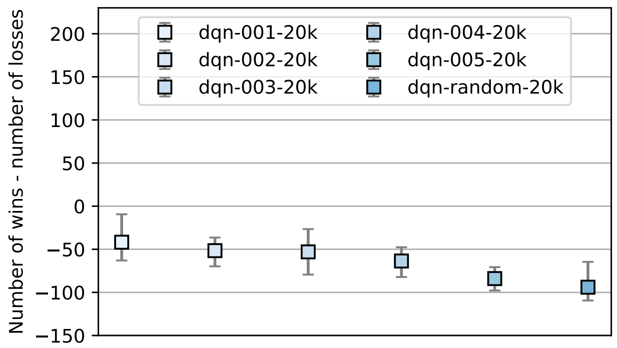


Figure 8.20: Direct comparisons with multiobjective ACET agent with RB006. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

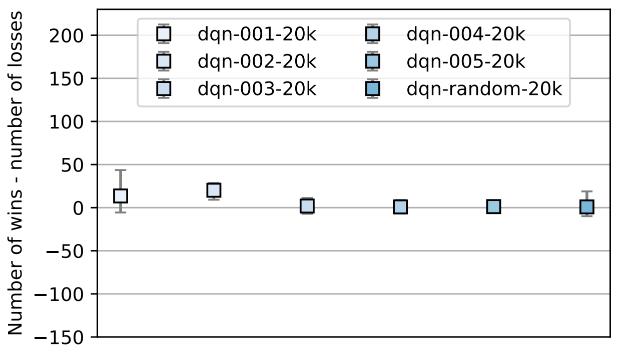


Figure 8.21: Direct comparisons with multiobjective ACET agent with RB007. Each vertical bar shows the mean and 95% confidence interval for the corresponding agent.

8.5 Learned Policies

The final product of RL is a policy that the agent can use to operate within the environment. The previous sections in this chapter measure how strong a policy is by running combat scenarios between an RL agent and an engineered agent. A policy is considered a success if the average McGrew score obtained by the agent is higher that that of the opponent. However, a McGrew score alone does not tell the whole story because it does not reveal what exactly happened during the fight or if the agents exhibited novel or emergent behaviours. An emergent behaviour emerges when an engineered agent responds to a situation by behaving in a certain way even though the behaviour itself is not programmed into the agent. For this reason, this section presents sample trajectories of the fights of some of the tabular, deep RL, discrete-to-deep and MORL agents evaluated in this thesis against the Pure Pursuit, Smart Pursuit and Stern Conversion engineered agents. A trajectory is important because sometimes one that shows a successful manoeuvring attempt by the agent does not necessarily correspond to a higher average reward for the RL agent for the particular fight. For instance, an RL agent that starts from an unfavourable position but turns from being pursued to pursuing may indicate a strong policy, yet its total McGrew score might be lower than its opponent's because it has spent more time being pursued (getting low McGrew scores) than doing the pursuing (getting high McGrew scores). By the same token, a fight that starts with the RL agent in a favourable position may give the agent a higher average McGrew score than its adversary, but the agent's policy will not be considered strong if in the end the agent gets outmanoeuvred by the opponent. In all the trajectories shown here, blue represents the track of an RL agent and red the track of an engineered one.

8.5.1 Tabular Agent

This section presents the trajectories of the ac-random-200K agent in direct fights against the engineered agents. They show the ACET agent is able to beat the engineered agents in some matches, but loses in most fights. Figure 8.22 shows a rare occasion in which the ACET agent is able to stay close to and outmanoeuvre the Pure Pursuit agent. In this event, the ACET agent starts at a 0° heading and the Pure Pursuit agent at a 60° heading, a starting layout that gives the ACET agent an advantage. Right after the start, the Pure Pursuit agent immediately tries to get behind the ACET agent by making a loop. Flying in a loop is not programmed into Pure Pursuit, so this is an emergent behaviour from the engineered agent. Soon after, the ACET agent manages to slow down to get behind its adversary and the fight ends with the ACET agent holding that position.

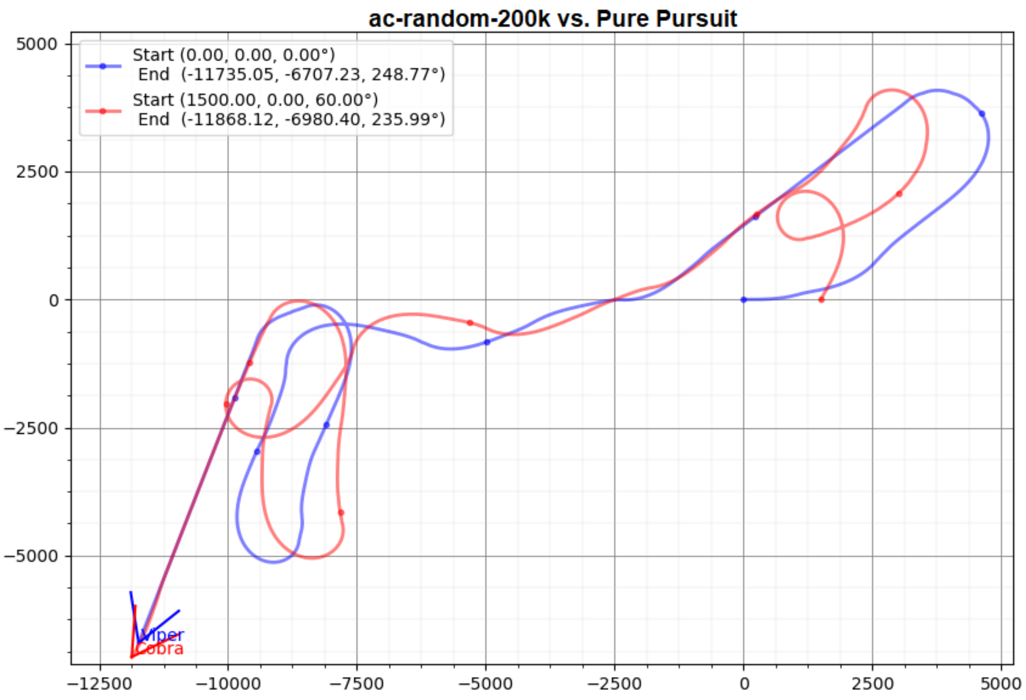


Figure 8.22: The ACET agent manages to outmanoeuvre the Pure Pursuit agent. Distances are in metres

In the majority of fights, however, the ACET agent is outmanoeuvred by the engineered agents, as shown in Figures 8.23. In this case, the Smart Pursuit agent, which at first is being pursued, makes a single loop in three different occasions in order to get behind the ACET agent. As Smart Pursuit is not programmed to have this skill, this is also an emergent behaviour from the engineered agent. Making a loop is something that the ACET agent has never been observed to be able to do.

The weakness of the ACET agent's policy is even more apparent in tests where the ACET agent starts from a favourable position of being behind the opponent. For example, in Figure 8.24, the ACET agent is easily outmanoeuvred by both the Pure Pursuit and Stern Conversion agents. In the fight against the Pure Pursuit agent, both aircraft start at a 0° heading, practically putting the RL agent right behind the engineered agent. In the match against the Stern Conversion agent, the ACET agent again starts at a 0° heading and the engineered agent at 30°, still a comfortable advantage for the ACET agent.

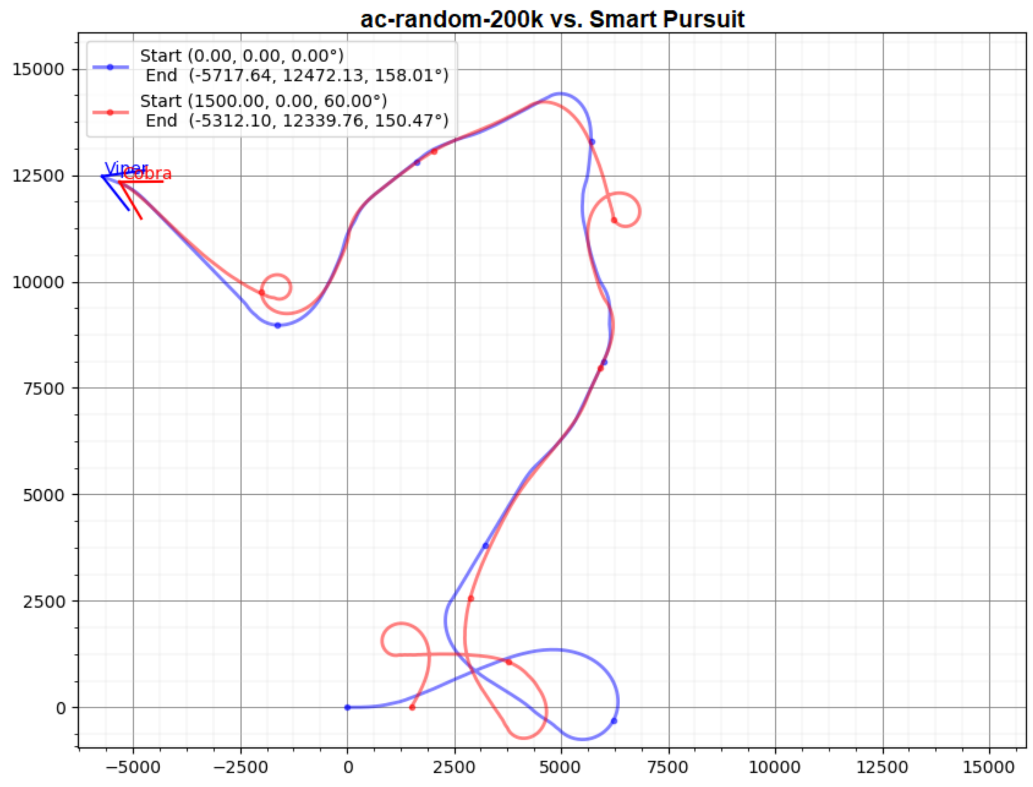
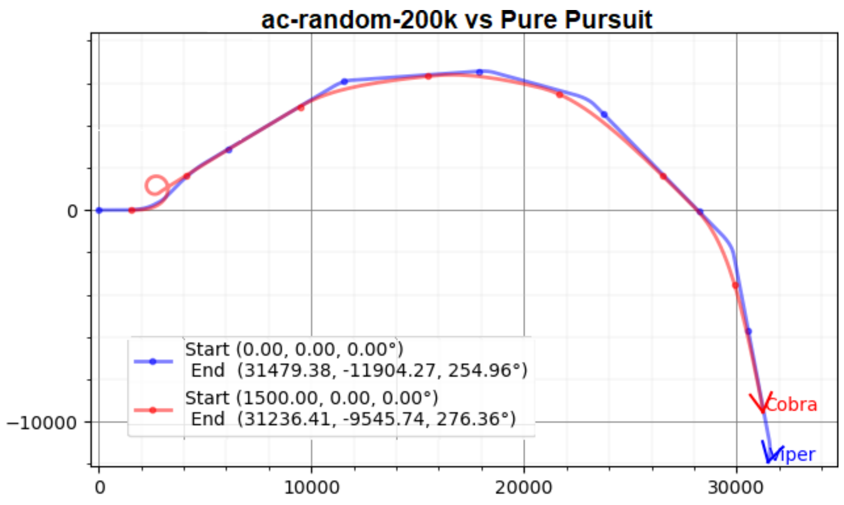


Figure 8.23: The ACET agent is outmanoeuvred by the Smart Pursuit agent. Distances are in metres



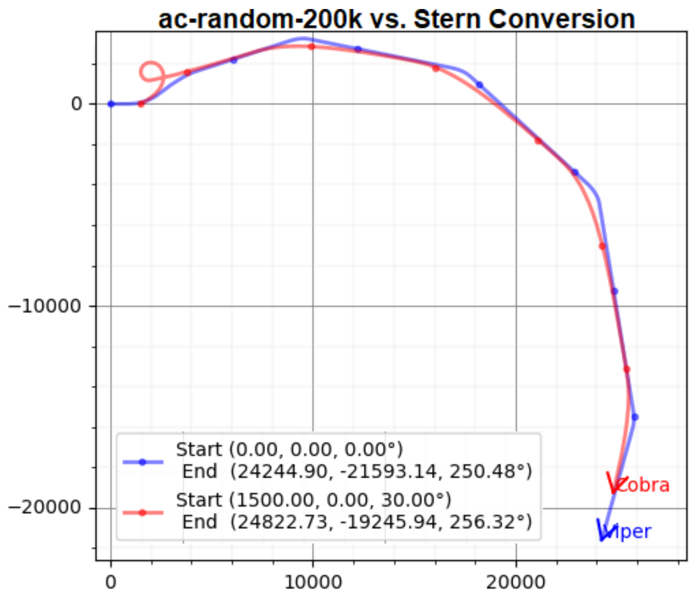


Figure 8.24: The ACET agent is outmanoeuvred by the Pure Pursuit (top) and Stern Conversion (bottom) agents despite starting from a favourable position. Distances are in metres

In another case, in a fight against the Smart Pursuit agent, the ACET agent starts from a favourable position and is immediately outmanoeuvred by its opponent. In this case, however, the ACET agent manages to turn the table in the last second of the match. The trajectory is shown in Figure 8.25.

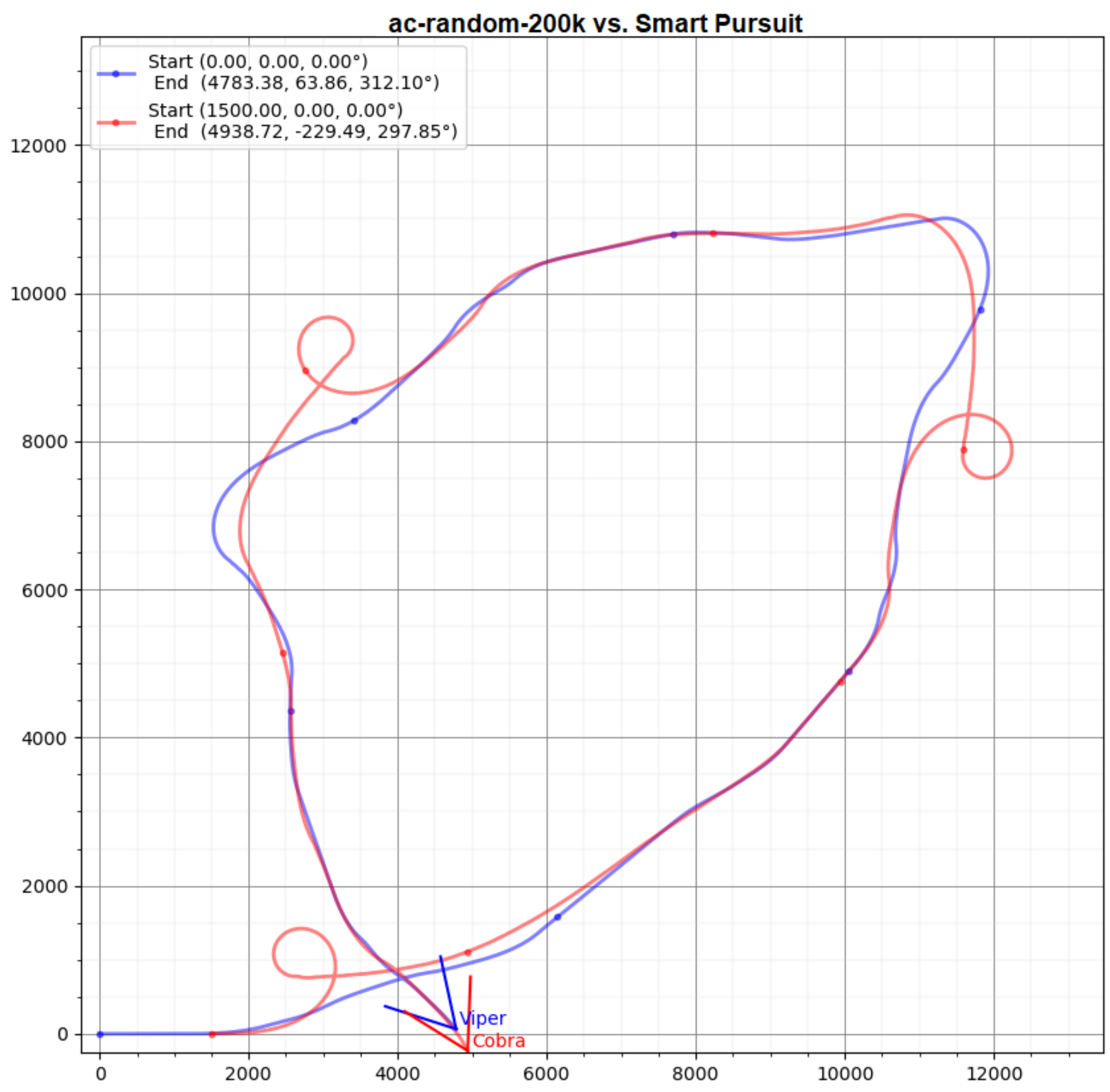


Figure 8.25: The ACET agent is initially outmanoeuvred by the Smart Pursuit agent but manages to turn the tide. Distances are in metres

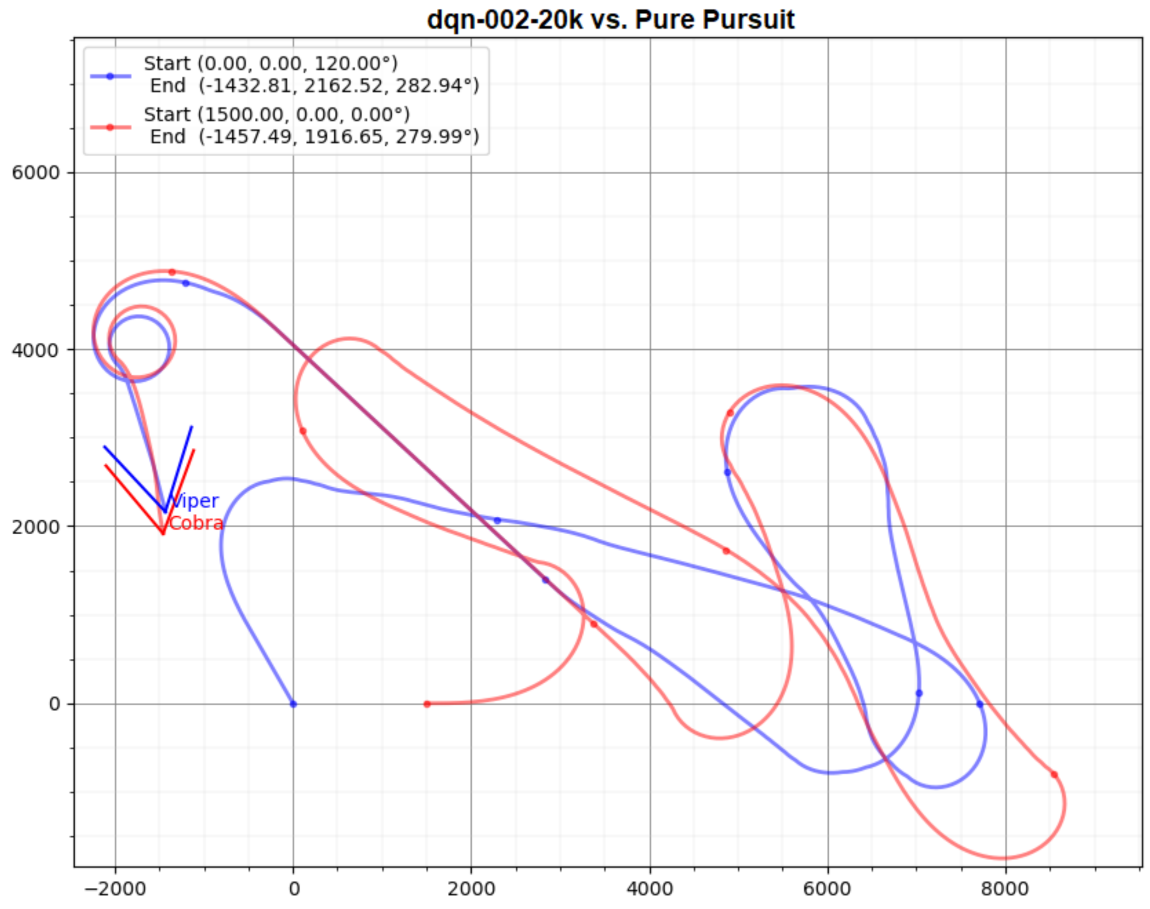


Figure 8.26: A win for the DQN agent against the Pure Pursuit agent. Distances are in metres

8.5.2 Deep RL Agent

This section discusses the policy of the strongest DQN agent, the dqn-002-20K agent. The DQN agent is shown to exhibit a wider range of behaviours than the ac-random-200K agent including developing a policy that allows the DQN agent to fly in a loop.

When the starting position is neutral or favourable to the DQN agent, it can easily beat the opponent. For example, Figures 8.26, 8.27 and 8.28 shows trajectories where the DQN agent outmanoeuvres the Pure Pursuit, Smart Pursuit and Stern Conversion agents, respectively.

Even when the starting position is unfavourable to the DQN agent, it can sometimes still defeat the engineered agent, as shown in Figures 8.29 and 8.30.

Some trajectories show the physical inferiority of the DQN agent to the engineered agents, especially the Smart Pursuit agent. For example, Figure 8.31 shows how the Smart Pursuit agent tries to outmanoeuvre the DQN agent by making two double loops. None of the deep RL agents trained in this thesis is known to be able to achieve such an aerobatics feat.

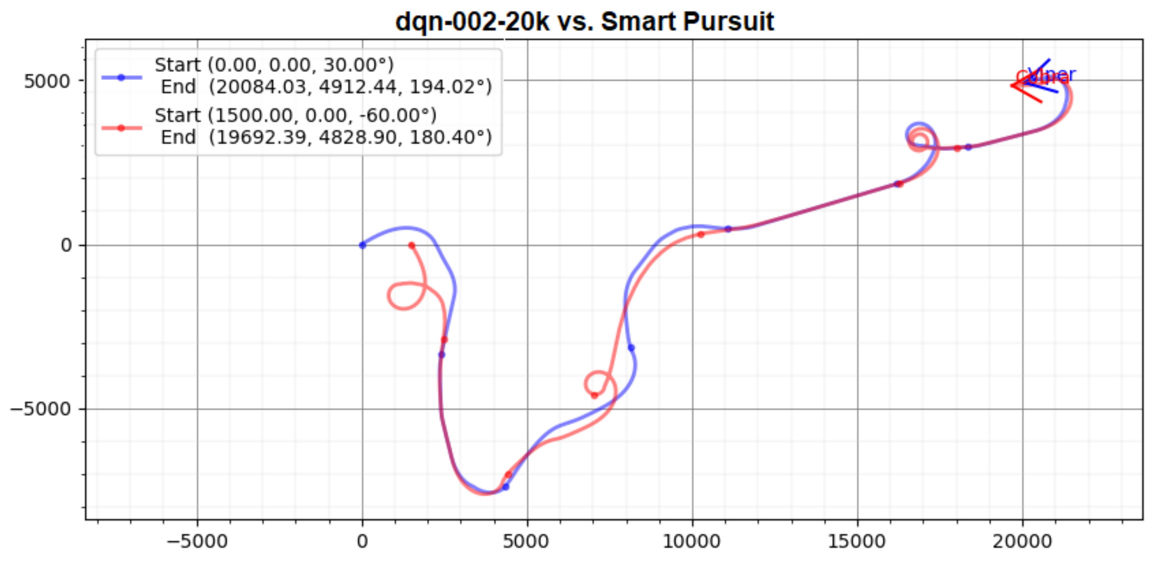


Figure 8.27: A win for the DQN agent against the Smart Pursuit agent. Distances are in metres

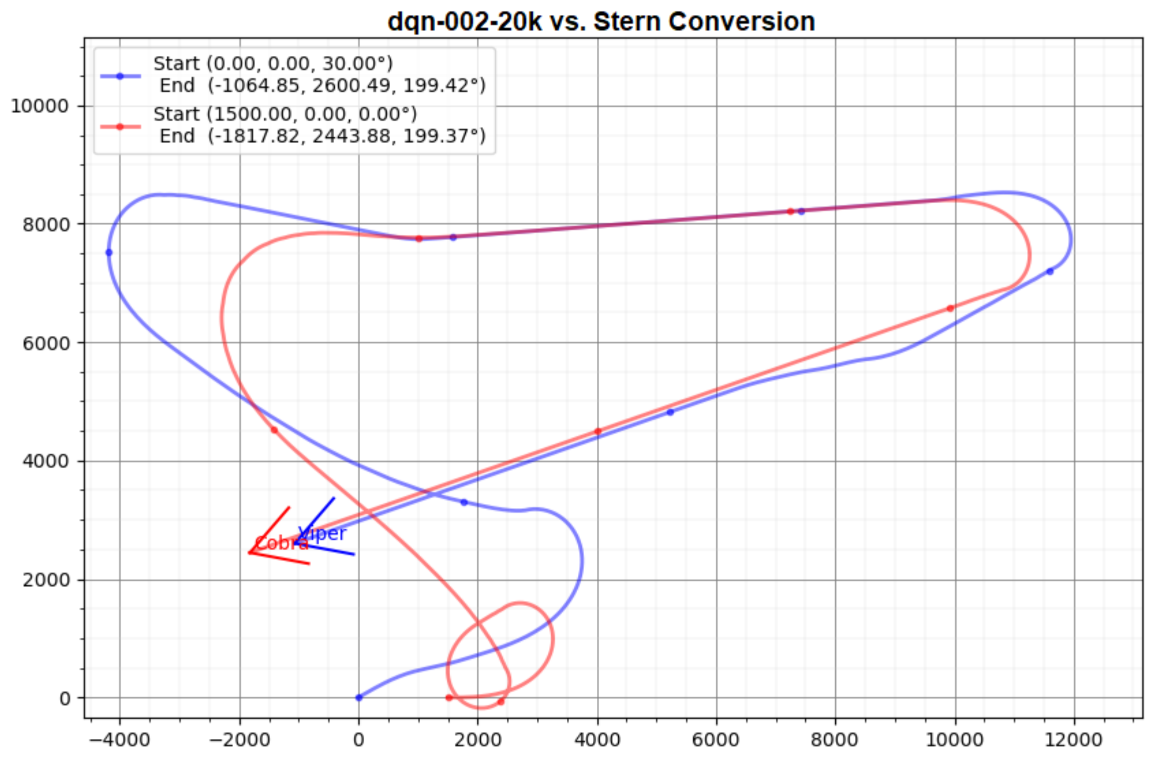


Figure 8.28: A win for the DQN agent against the Stern Conversion agent. Distances are in metres

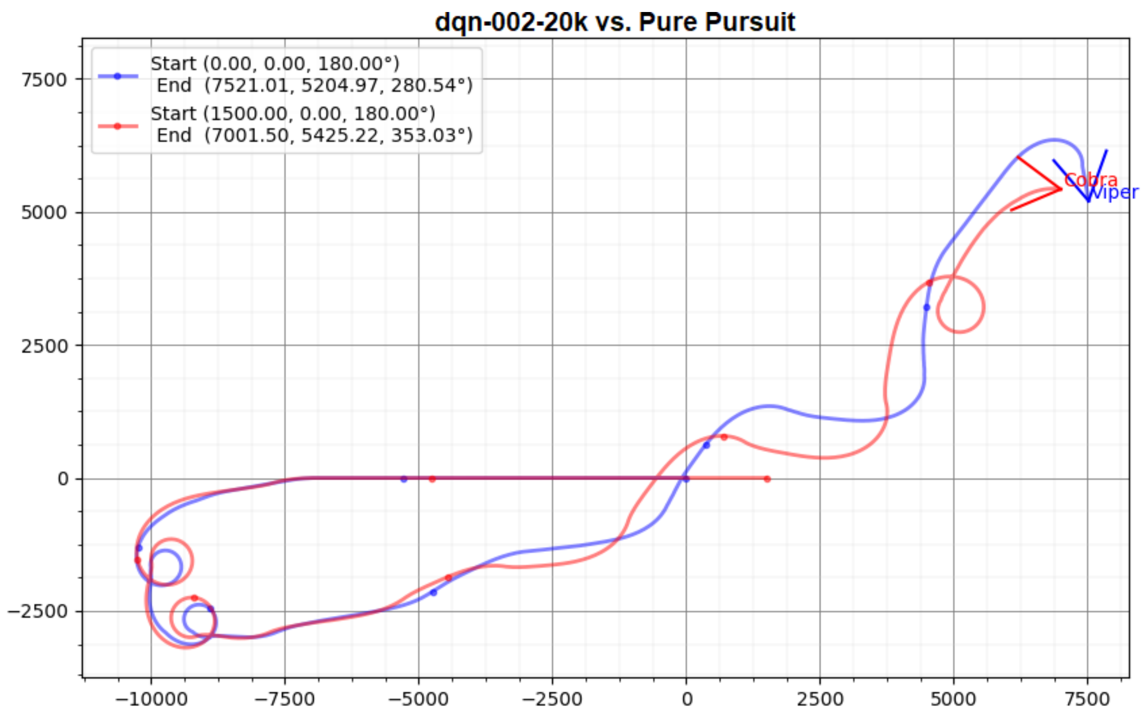


Figure 8.29: A win for the DQN agent against the Pure Pursuit agent when the RL agent starts from an unfavourable position. Distances are in metres

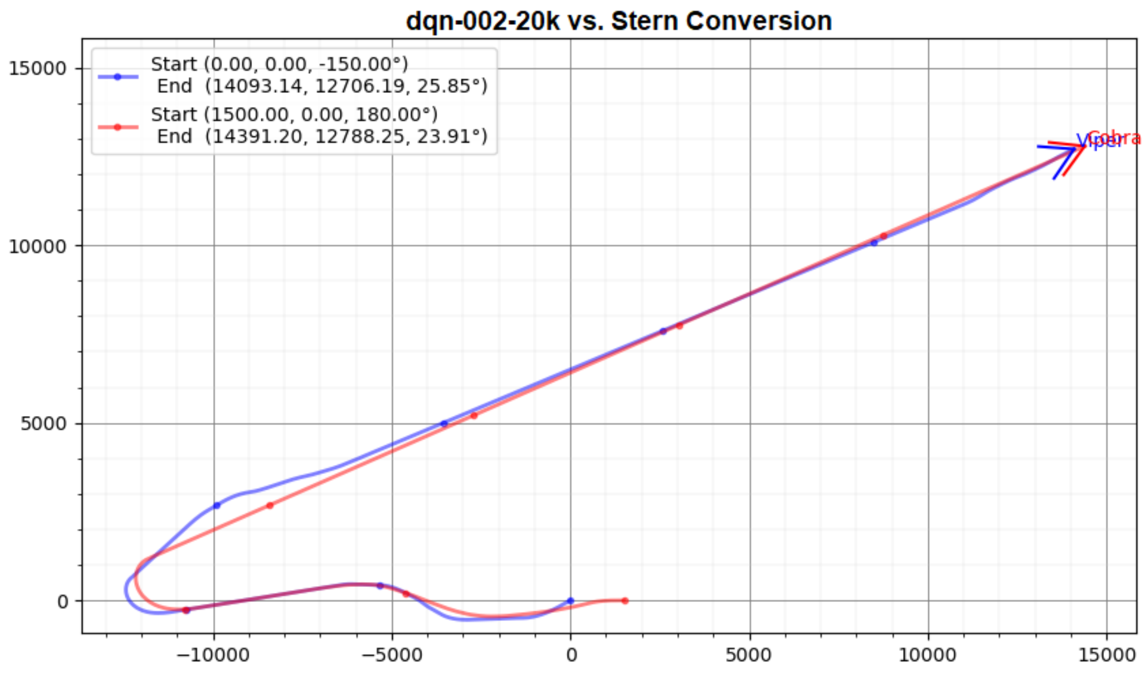


Figure 8.30: A win for the DQN agent against the Stern Conversion agent when the RL agent starts from an unfavourable position. Distances are in metres

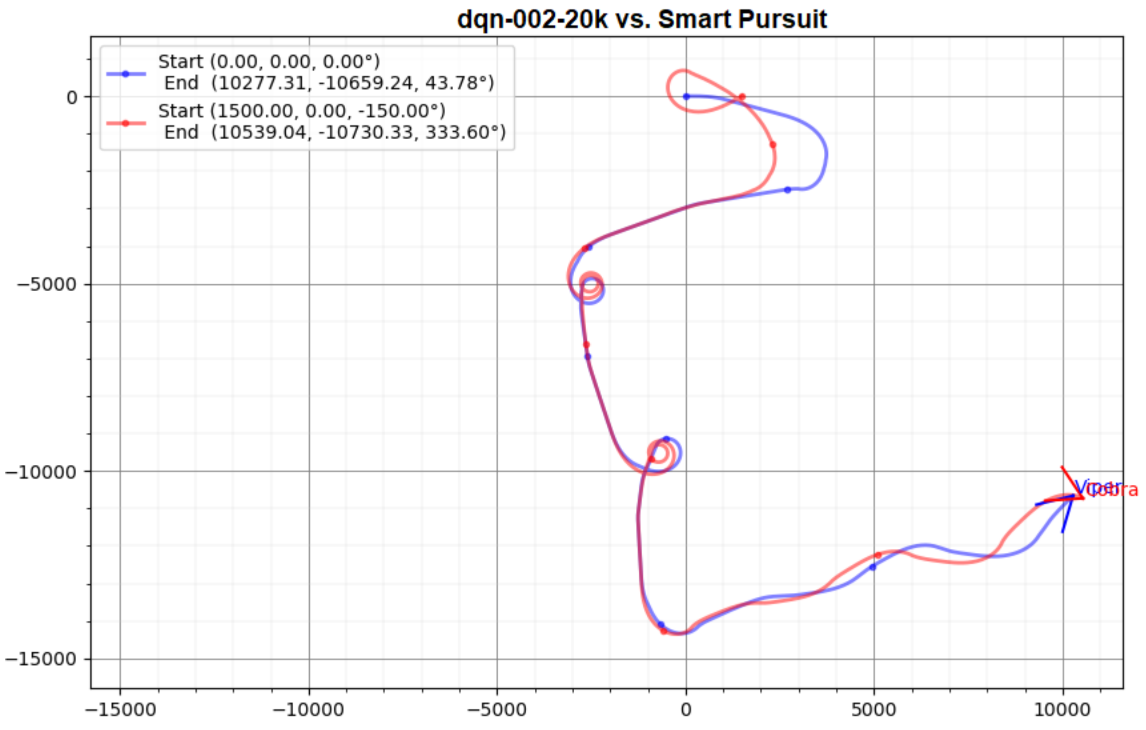


Figure 8.31: Physical constraint of the DQN agent. Distances are in metres

8.5.3 Discrete-to-Deep Agent

This section investigates the trajectories of the d2dsql-random-13K agent in its fights against the engineered agents. The trajectories are the most interesting ones as they show skills not observed in the ACET and deep RL agents in the previous subsections. Not only can the d2dsql-random-13K agent make a double loop, as shown in Figure 8.32, it is even able to create multiple loops, as shown in Figures 8.33 and 8.34.

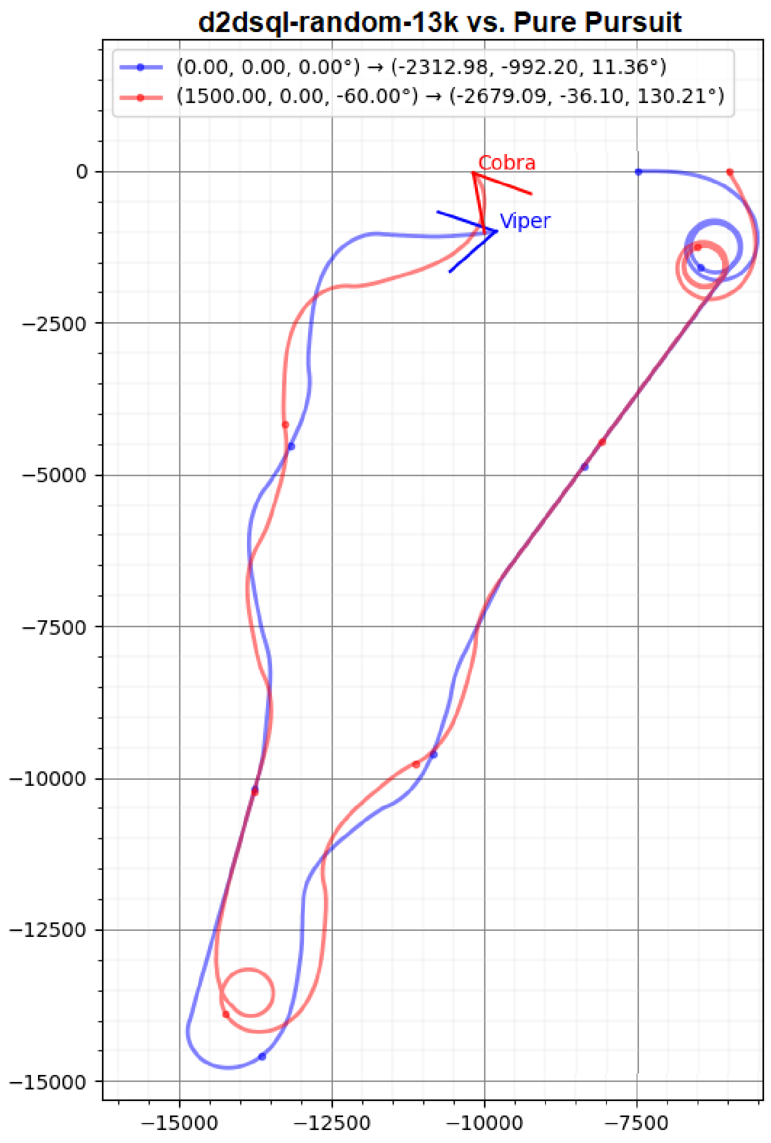


Figure 8.32: Double loop in d2dsql-random-13K. Distances are in metres

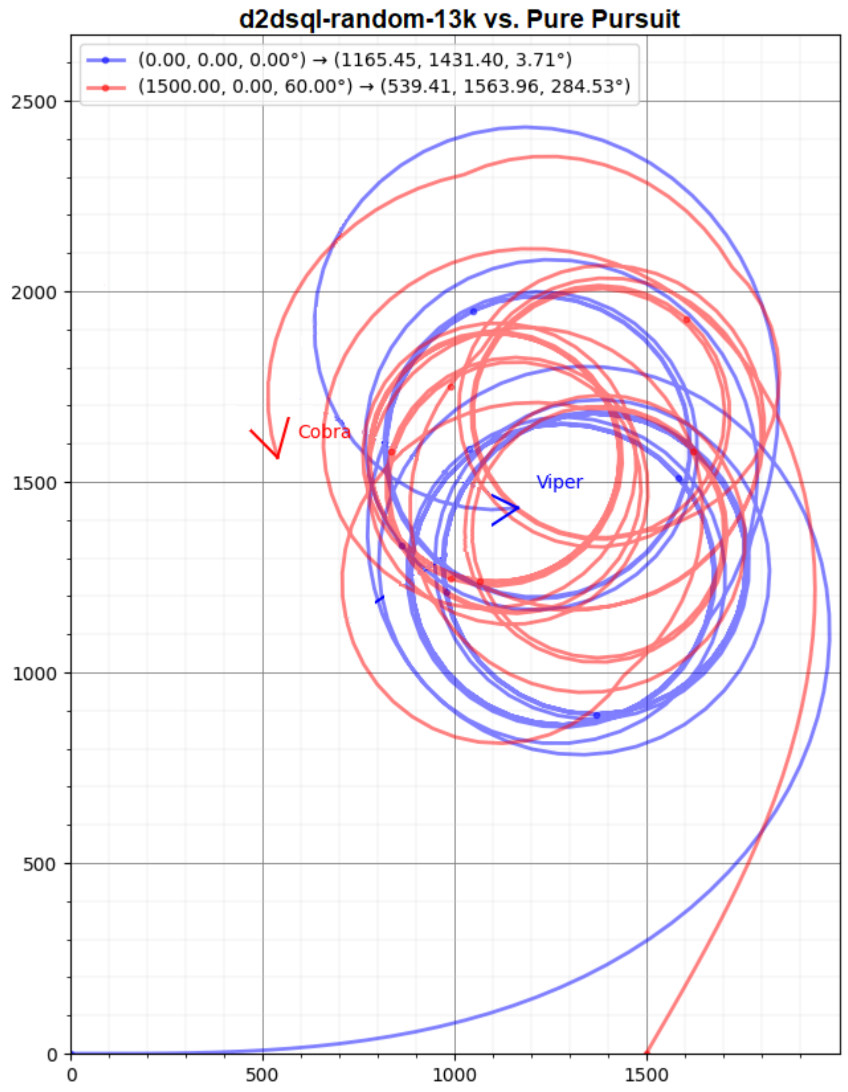


Figure 8.33: Multiple loops in d2dsql-random-13K vs. Pure Pursuit. Distances are in metres

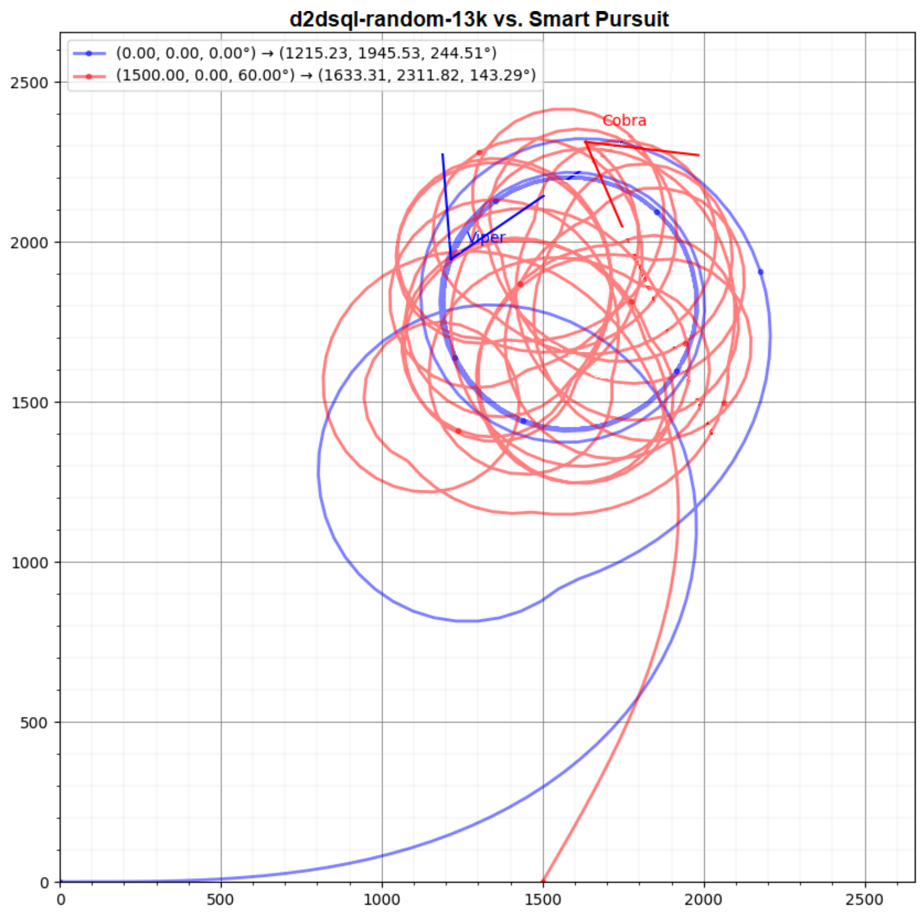


Figure 8.34: Multiple loops in d2dsql-random-13K vs. Smart Pursuit. Distances are in metres

Making loops is not the only skill the agent possesses. As shown in Figures 8.35, 8.36 and 8.37, the d2dsql-random-13K agent can slow down to outmanoeuvre its opponent.

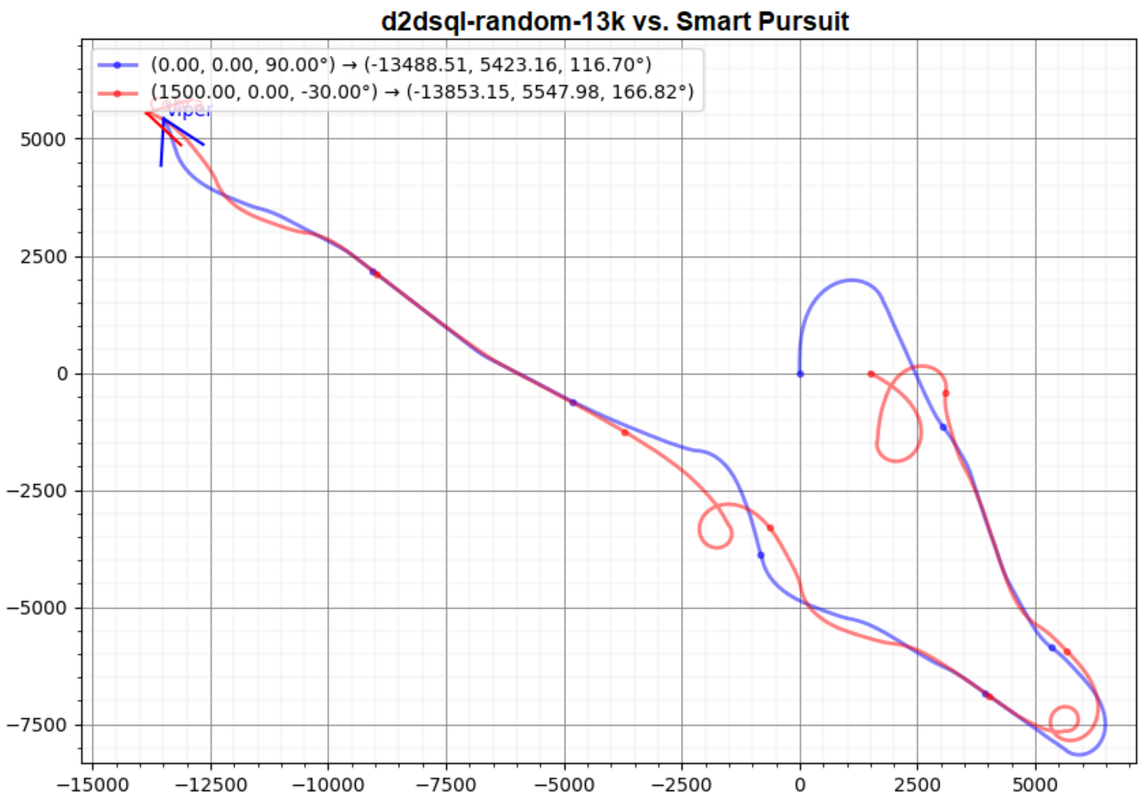


Figure 8.35: d2dsql-random-13k vs. Smart Pursuit. Distances are in metres

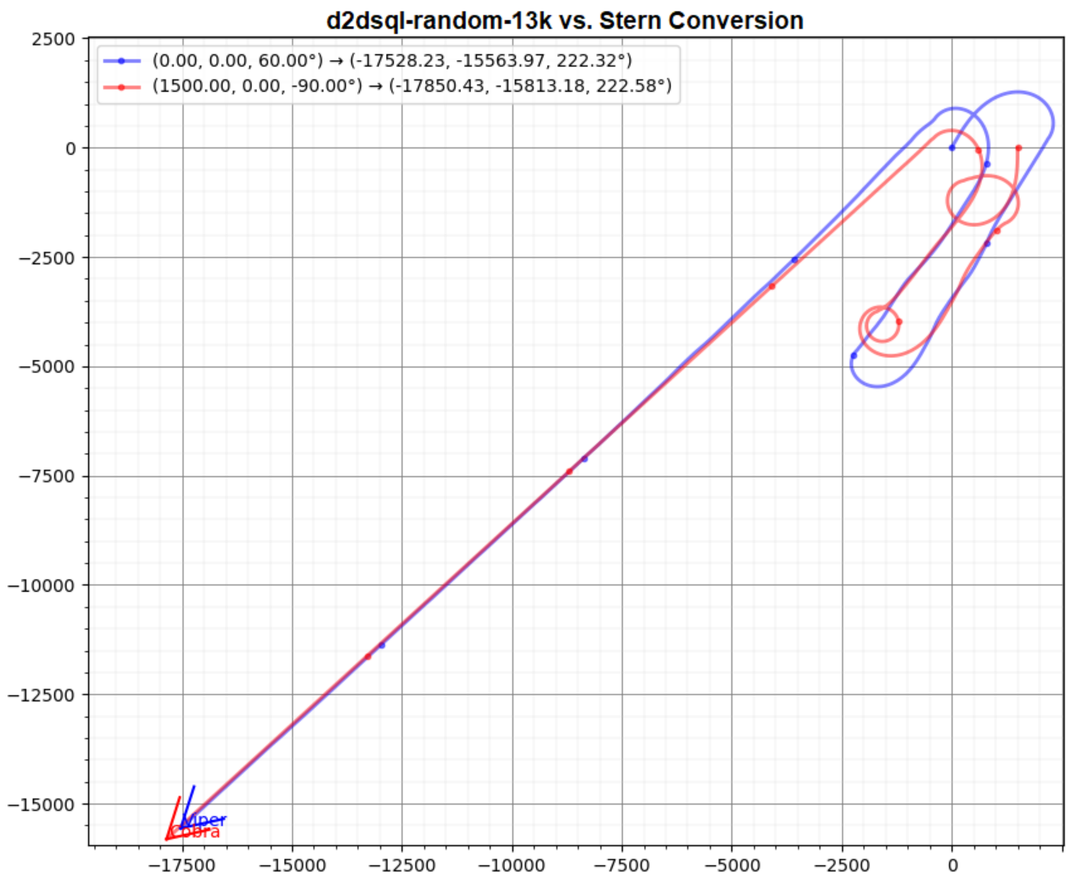


Figure 8.36: d2dsql-random-13k vs. Stern Conversion. Distances are in metres

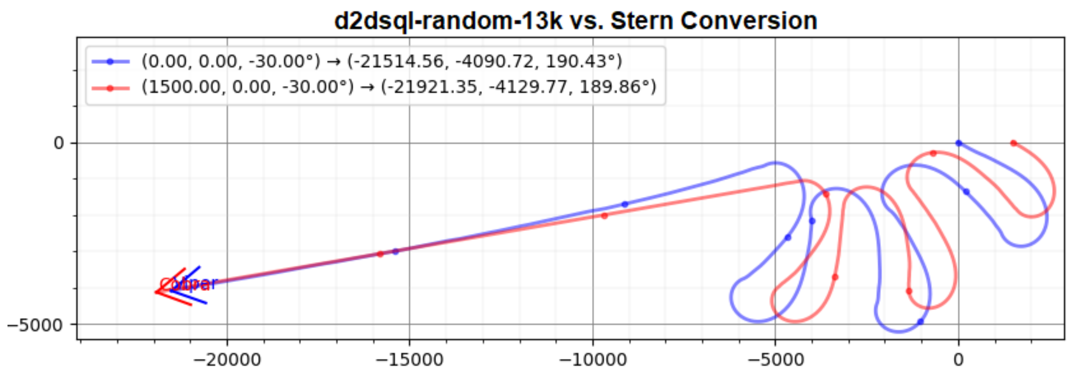


Figure 8.37: d2dsql-random-13k vs. Stern Conversion (2). Distances are in metres

8.5.4 MORL Agent

This section looks at the trajectories of the mo-ac-200k-RB004 agent in its fights against the engineered agents. The trajectories do not show advanced skills found in the deep RL and discrete-to-deep agents. In few rare cases the mo-ac-200k-RB004 agent is able to put up a good fight against the engineered agents, as shown in Figure 8.38. However, in the majority of cases, the MORL agent loses, as shown in Figures 8.39, 8.40 and 8.41. In all the three trajectories, it is apparent that the engineered agents easily outmanoeuvre the MORL agent by making a loop, a move the MORL agent seems unable to do. The only tool the MORL agent has at its disposal seems to be to slow down to get behind its adversary.

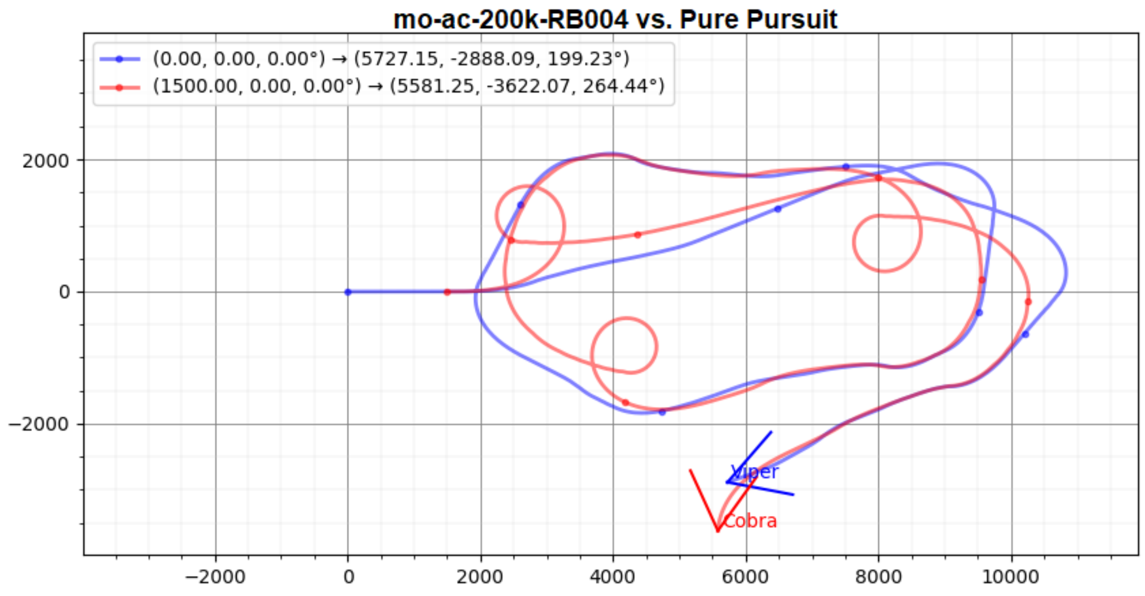


Figure 8.38: A win of mo-ac-200k-RB004 over the Pure Pursuit agent. Distances are in metres

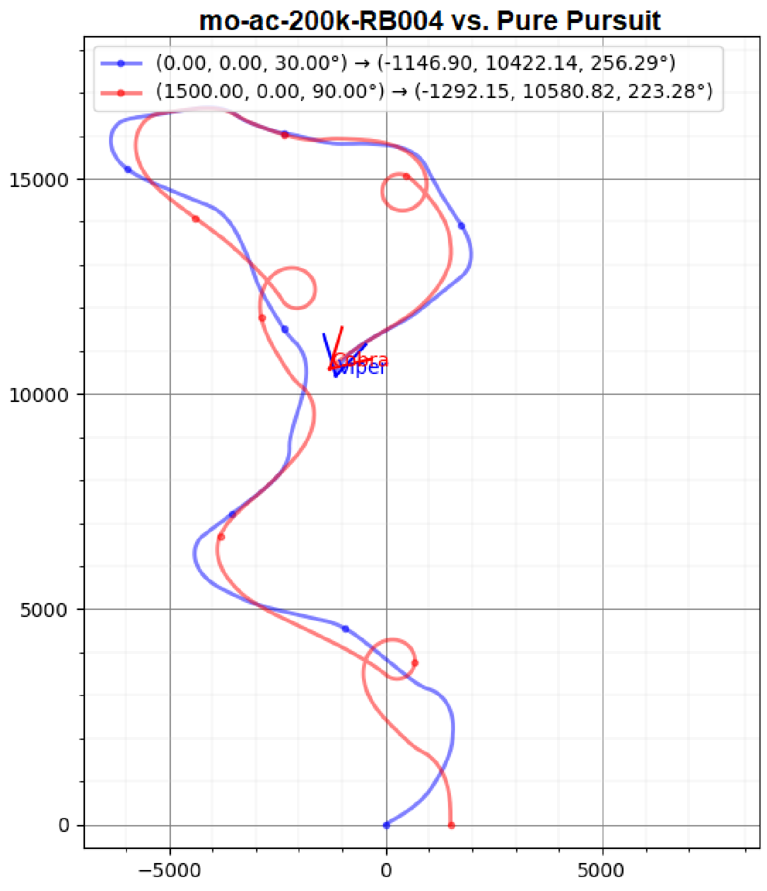


Figure 8.39: mo-ac-200k-RB004 loses to Pure Pursuit. Distances are in metres

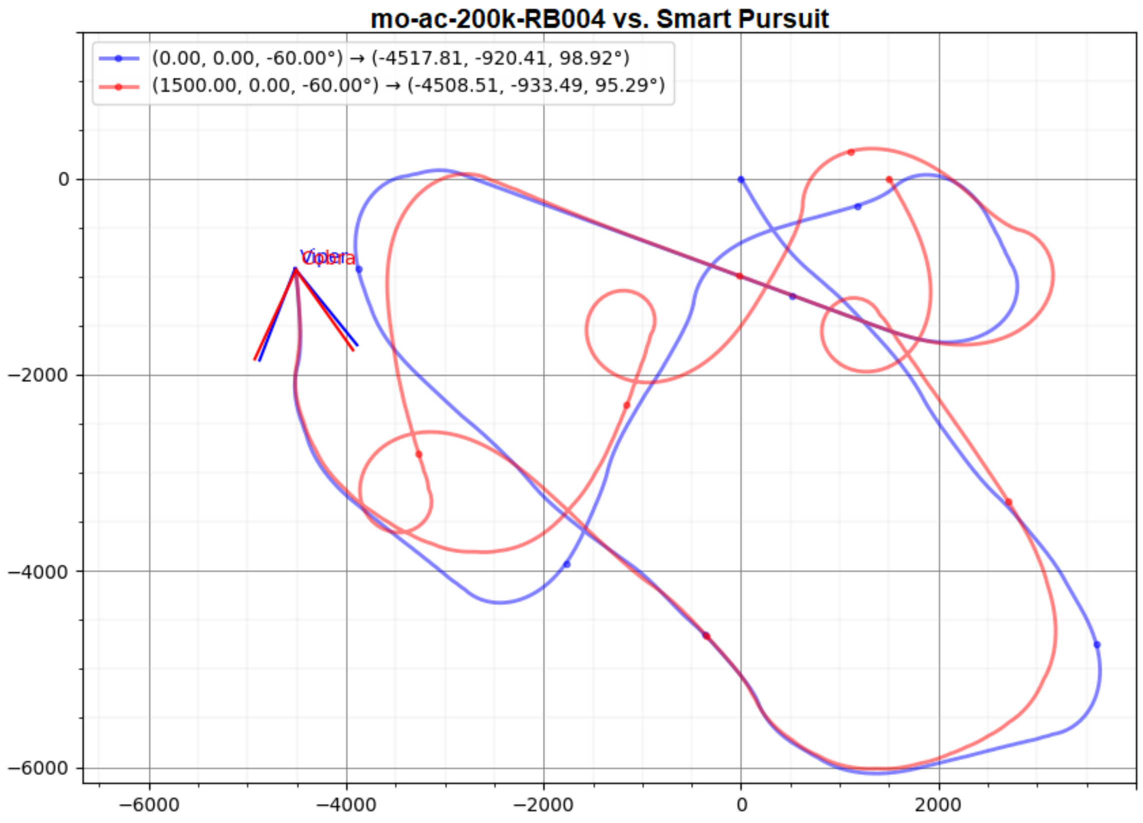


Figure 8.40: mo-ac-200k-RB004 vs. Smart Pursuit. Distances are in metres

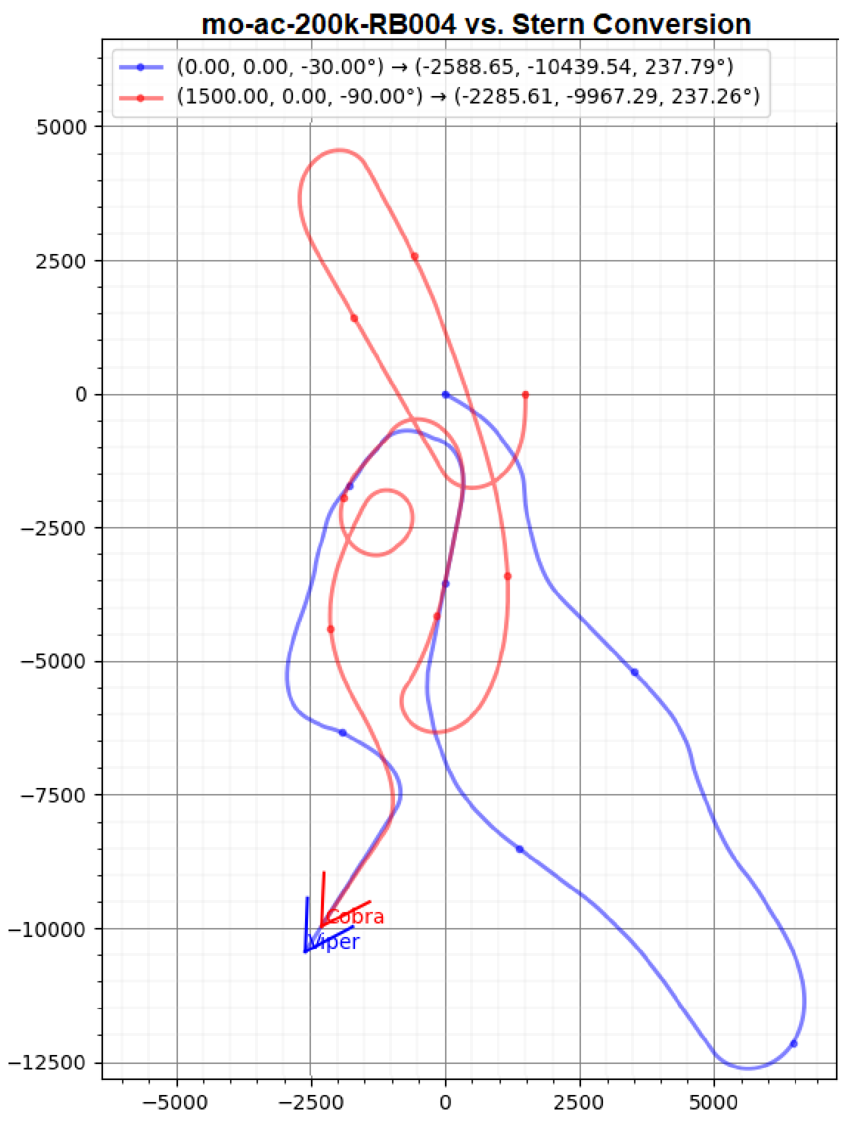


Figure 8.41: mo-ac-200k-RB004 vs. Stern Conversion. Distances are in metres

8.6 Summary

This chapter compares, indirectly and directly, the best performers from the previous chapters. The agents compared are the tabular ACET agents, the DQN agents, the D2D-SPL and D2D-SQL agents and the multiobjective ACET agents.

The tabular ACET method is the cheapest to train but suffers from the curse of dimensionality, making it powerless to solve high-dimensional problems such as the air-combat manoeuvring problem. It also does not learn as fast as the deep RL methods, whose learning curves climb more rapidly in far fewer learning episodes than the tabular methods. Deep RL methods such as DQN, however, are more expensive to train. In absolute time, 20,000 episodes of DQN learning takes much more than 200,000 episodes of tabular ACET learning. Lying between the two approaches are D2D-SPL and D2D-SQL, which are based on the idea of having the generalisability of deep RL methods at the costs closer to that of the tabular methods. The multiobjective ACET method is also cheap to train, even though the average rewards of its individual components are not directly comparable to the McGrew score of a single-objective method.

Performance-wise, the dqn-002-20K agent is the winner in many tests, even though in some tests it is outperformed by some of the multiobjective ACET agents, which in this thesis rely on multiobjectivisation (Brys et al., 2014). The dqn-002-20K agent is able to beat two of the three engineered agents that are physically superior to it. When the engineered agents are restricted to have the same physical capabilities as the RL agents, more RL agents outperform them. When different types of RL agents are compared against each other, the best multiobjective agents outperform the best single-objective agents. This shows the potential of multiobjective RL.

In terms of learned policies, the d2dsql-random-13K agent generates trajectories indicating it possesses the most advanced skills among the RL agents as it is able to make multiple loops and slow down to get behind the opponent. The dqn-002-20K agent is able to make single loops but never shows it is able to make double or multiple loops. Agents following policies generated from single- and multiple objective RL exhibit rudimentary skills and have never been observed to be able to make a loop.

Chapter 9

Conclusions

This thesis implements and evaluates reinforcement learning (RL) techniques to address dynamic adversarial games in the context of air combat simulation. The air combat domain is perhaps one of the most complex applications of constructive simulations due to it being highly dynamic, adversarial, continuous, partially observable and possessing multiple objectives. This chapter summarises the experiments conducted in this thesis, answers the thesis questions and presents the conclusions of the research. It then discusses its contributions, impact and limitations and potential future work.

9.1 Thesis Overview

This thesis uses an air combat simulator called Ace Zero from Defence Science and Technology Group, Australia. Written in Python, Ace Zero is essentially a game engine that executes in a loop, moving virtual aircraft in units of time called ticks. The RL algorithms used in this thesis can be grouped into four approaches.

- tabular approach with Watkins' Q(λ) and actor-critic with eligibility traces (ACET) methods
- deep RL approach using Deep Q-Network (DQN), DQN with a target network (DQN-wTN), double DQN and proximal policy optimisation (PPO)
- discrete-to-deep approach that introduces two novel methods, discrete-to-deep supervised policy learning (D2D-SPL) and discrete-to-deep supervised Q-value learning (D2D-SQL)
- multiobjective approach with multiobjective Q(λ), multiobjective ACET, multiobjective DQN with a shared network and multiobjective DQN with multiple networks

In this thesis learning is done by pitting an RL agent piloting an aircraft (code-named Blue) against a script-guided aircraft flying in a straight line at a constant speed (code-named Red). There are two types of initial positions used during learning:

- Five fixed initial positions where the distance and starting angle are slightly randomised. They are code-named 001, 002, 003, 004 and 005.
- Random positions where Blue starts from coordinate (0, 0) and Red from any point within a 1,500 x 1,500 square metre area. The starting angle of Blue is always zero and Red's starting angle is randomised.

All the tabular RL agents, both single- and multiobjective, learn for 200,000 episodes. All the deep RL agents, both single- and multiobjective, learn for 20,000 episodes. The D2D-SPL and D2D-SQL agents get their off-policy data from ACET agents that run for 50,000 episodes. For all learning trials, an episode is terminated after 700 timesteps. All the experiments are repeated ten times with the pseudorandom number generator within the agent seeded with the same set of numbers to ensure results are reproducible. The different numbers of learning episodes for the tabular and deep RL methods are due to the fact that the per-episode cost of a tabular method is much lower than that of a deep RL method.

For each trial, the learning time is recorded and a learning curve generated. The learning curve plots the average rewards for all episodes. A learning curve that increases steadily indicates a successful learning agent. The average reward over all the episodes can be used to measure how fast the agent learns. For agents trained using fixed initial positions, a steadily increasing learning curve means the agent manages to learn a policy that is effective for previously seen states. As will be discussed in the next section, this does not mean the policy generalises to states that have not been visited during learning.

9.2 Conclusions

9.2.1 Learning Times and Curves

Tabular Methods

For the tabular methods, there are no significant differences in learning times among the different initial positions. However, the cost per episode for ACET is almost twice as expensive as the cost for Q(λ) because every update in Q(λ) involves changing only the Q-values of the visited states, whereas in ACET it includes updating all the policy parameters, which in this case total 14,000.

The low average scores for the Q(λ) learning curves indicate that the agents failed to learn. By contrast, the ACET agents managed to learn successfully in five out of six initial positions. The only exception that an ACET agent failed was when learning started from initial position 005, which is the most challenging position as it places the agent at a significant disadvantage compared to its opponent.

Deep RL Methods

Unlike the tabular ACET agents, that learned successfully in five of six initial positions, one or more deep RL agents from the DQN family learned successfully in each of the initial positions. For initial position 005, however, only the DQN-wTN agent did well. It is also apparent that all the learning curves are still trending upward around the final learning episodes, suggesting that they could have reached a higher score had learning not been terminated. In all initial positions except 005, the DQN agents perform better than the DQN-wTN and double DQN agents.

The PPO agents were not as successful and only did well for initial positions 001 and 002 in which they reached average scores comparable to the DQN agents.

Discrete-to-Deep Methods

There are two experiments conducted for the D2D-SPL and D2D-SQL agents, both of which were discussed in Chapter 6. Unlike in Chapters 4 and 5 where six initial positions were used, the D2D-SPL and D2D-SQL agents in Experiment 1 are only tested with initial position 001. In Experiment 2 of the same chapter, the D2D-SPL and D2D-SQL agents obtained their off-policy data from ACET agents trained using random initial positions.

Experiments 1 and 2 show that the goal of these two algorithms, which is to learn a policy that can generalise as well as deep RL agents at a cost closer to tabular agents, has been achieved. In Experiment 1, the supervised phase of the D2D-SPL agents add a mere 0.01% to the learning time of the ACET agents generating the off-policy data (the base agents). The total learning times of the two D2D-SQL agents (using two different epsilon annealing functions) are 2.42 and 2.35 times the base agents' learning times but are still lower than the comparison DQN agents, that took 4.07 times the learning times of the base agents.

In Experiment 2, the supervised phase of the D2D-SPL agents adds 12% of the learning times of the base agents. The D2D-SQL agents needed between 69.05% to 82.61% of the learning times of the comparison DQN agents.

In terms of learning curves, the D2D-SPL agents in both experiments increased the total rewards after they completed the supervised phase, indicating that the supervised training of a deep neural network allowed the creation of a policy that generalises better than the base ACET agent used to gather the training data. The D2D-SQL agents in both experiments also increased the total rewards after they finished the supervised phase and the final reinforcement phase.

Multiobjective Methods

The multiobjective methods tested in this thesis address the air-combat manoeuvring problem as a single-objective problem with the reward decomposed into simpler components, a technique known as multiobjectivisation (Brys et al., 2014). The methods are tested with these reward functions/builders:

- RB001: [0.33 range_score, 0.33 AA_score, 0.33 ATA_score]
- RB002: [0, 0, ATA_score]
- RB003: [1.0 range_score, 1.0 AA_score, 1.00 ATA_score]
- RB004: [blue mcgrew_score, -red mcgrew_score]
- RB005: [0.50 blue mcgrew_score, -0.50 red mcgrew_score]
- RB006: [0.75 blue mcgrew_score, -0.25 red mcgrew_score]
- RB007: [0.25 blue mcgrew_score, -0.75 red mcgrew_score]

Reward builders RB001, RB002 and RB003 return a vector containing three rewards decomposed from the McGrew score and were tested with multiobjective Q(λ), multiobjective ACET, multiobjective DQN with a shared network and multiobjective DQN with multiple networks. Reward builders RB004, RB005, RB006 and RB007 return a vector containing two rewards, the agent's McGrew score and the inverse of the opponent's McGrew score, and were tested with multiobjective ACET. With regard to learning times, the multiobjective tabular agents spent more time than their single-objective equivalents because there are multiple Q tables in the multiobjective versions. The multiobjective DQN agents with a shared network spent learning times that are comparable to the single objective DQN agents, as the former employ a single network. On the other hand, the multiobjective DQN agents with multiple networks spent considerably more learning times due to the fact the networks were updated sequentially.

As for learning curves, for RB001, the multiobjective ACET agent does best at maximising the first reward, followed by the Q(λ) agent. The two multiobjective DQN agents have no success in doing so. With regards to the second and third rewards, all the four agents manage to improve their AA and ATA scores, with the ACET agent outperforming the other three agents.

For RB002, the first two rewards, the range and AA scores, are zero. The third reward, the ATA score, has a value between 0 and 1. All the four agents manage to improve the ATA score with the ACET agent again outperforming the others. The tabular methods increase the average score of the third reward steadily, whereas the multiobjective DQN agents show some peaks and troughs, like the learning curves in the single-objective tests of the deep RL methods.

For RB003, all agents manage to improve their range, AA and ATA scores with the multiobjective ACET agent has the most success in increasing the range score.

For reward builders RB004, RB005 and RB006, the multiobjective ACET agents managed to increase the first reward but not the second reward. For reward builder RB007, the agents failed to increase both rewards.

9.2.2 Performance

This thesis compares the RL agents' performance indirectly, through benchmarking against engineered and restricted engineered agents, and directly. Within the four categories of methods evaluated, there was at least one variant that outperforms the engineered agents that come with Ace Zero in indirect comparisons. One DQN agent outperforms two of the three engineered agents in direct comparisons. This proves that RL agents with no prior knowledge of the air-combat manoeuvring domain are able to beat engineered agents that are physically superior and specifically programmed for that.

The details of agent performance are given in the following subsections.

Tabular Methods

The Q(λ) agents perform poorly against the Smart Pursuit baselines. In contrast, some of the ACET agents beat the baselines in three of the four initial dispositions. The most performant ACET agents, those that learned from 001, 002 and random initial positions, are agents with the highest average rewards during learning. The finding that ACET is better than Q(λ) at handling a continuous-spaced problem such as the air combat manoeuvring problem is consistent with another experiment involving a continuous state space, which is presented in Appendix B. It could be an indication that tabular Q(λ) is not suitable for problems with a continuous state space.

Deep RL Methods

In general for the DQN, DQN-wTN and double DQN agents, the performance of an agent mirrors its learning curve, that is an agent that received a higher average reward during learning tends to have higher performance. The agents perform better than the tabular method agents as shown by some agents—the dqn-001, dqn-002, dqn-random, dqn-wtn-002, dqn-wtn-005, dqn-wtn-random—beating the baselines in all four initial dispositions. The dqn-wtn-005 agent, that obtained the highest average reward when starting from initial position 005, also performs better than the dqn-005 and ddqn-005 agents. It is also worth noting that in the case of DQN and double DQN, the agents that learned from initial position 001, where Blue is in the offensive position, also perform better than all the other agents that learn by starting in other fixed initial positions. Likewise, the agents that learned from initial position 002, where Blue is in defensive position, also perform better than all the other agents that learn by starting in other fixed initial positions. The DQN, DQN-wTN and double DQN agents that learned by starting from random positions generally perform well and in many cases beat the other agents implementing the same algorithm.

The PPO agents, however, perform poorly, way below the Smart Pursuit baselines. Two of the PPO agents, ppo-001 and ppo-002, learned successfully from initial positions 001 and 002 but do not generalise well and are unable to handle tests that were not seen during learning. There are two things that can explain the poor performance of PPO. First, PPO only updates its neural networks when its buffer is full. Second, PPO does not randomly sample its buffer to minimise correlations between samples, as is done in DQN and its variants.

Discrete-to-Deep Methods

While deep RL methods have achieved impressive results, the commonly used learning algorithms such as DQN, double DQN, A3C and their variants can be slow and unstable. D2D-SPL and D2D-SQL aspire to obtain the generalisability of deep RL at a cost closer to tabular methods. The two experiments in Chapter 6 show that the D2D-SPL results are consistently better than the base policy used to train it. They also show that D2D-SQL can be used to initialise a DQN or a double DQN agent, resulting in faster learning and better performance than a DQN or double DQN agent that learns from scratch. D2D-SQL takes longer to learn than D2D-SPL, however D2D-SQL performs better than D2D-SPL in the more complex tasks (test scenarios 3 and 4 of Experiment 1).

Multiobjective Methods

Three of the seven multiobjective ACET agents, those paired with reward builders RB004, RB005 or RB006, outperform the Smart Pursuit baselines and a single-objective ACET agent trained for the same number of episodes. The three multiobjective ACET agents also outperform the deep RL agents, showing the benefits of multiobjectivisation (Brys et al., 2014; Russel and Zimdars, 2003; Tesauro et al., 2008).

9.2.3 Learned Policies

Trajectories generated from fights between RL agents and engineered agents may reveal novel and emergent behaviours during the fights. The Pure Pursuit and Smart Pursuit engineered agents exhibit the ability to fly in a loop even though they are not programmed to do so. This is an emergent behaviour. Of the four RL agents whose trajectories were observed, the d2dsql-random-13K agent has the most skills. It is able to outmanoeuvre its opponent by making single, double and multiple loops or slowing down. Here a multiple loop is defined as triple or more loops. The dqn-002-20K agent comes next: It is able to make single loops but is never observed to make double or multiple loops. The policies generated by tabular learning, both single- and multiobjective ACET, exhibit simple behaviour. Agents following the policies can slow down to outmanoeuvre the opponent but are never observed to fly in a loop.

9.2.4 Comparison of Different Methods

Table 9.1 compares the different approaches to the air combat manoeuvring problem in this thesis. The points of comparison are explained below.

	Tabular	Deep RL	D2D	MORL
Requiring Domain Knowledge	Good	Good	Good	Good
Implementation Difficulty	Good	Bad	Bad	OK
Learning Time	Good	Bad	OK	OK
Indirect Comparison Performance	OK	Good	Good	Good
Direct Comparison Performance	OK	Good	Good	Good
Learned Policies	OK	Good	Good	OK

Table 9.1: Comparison of families of RL methods

- **Requiring domain knowledge**. In terms of whether or not domain knowledge needs to be encoded, all the four approaches are good because they require little domain knowledge. Requiring little or no domain knowledge is one of the attributes of RL. Domain knowledge might still be required if a discretiser or normaliser (discussed in Chapters 4 and 5, respectively) is needed. If the state variables of a problem are normalised and this fact is known, then no domain knowledge is required. For example, most if not all the environments in OpenAI Gym meet this criteria, and an RL agent can learn to solve them without domain knowledge. Even in the event domain knowledge is required, it is still much less than would be required by other AI systems.

- **Implementation difficulty**. Of the four approaches, the tabular approach is the easiest to implement as it uses a table to store Q-values or parameters for each discrete state. The deep RL and D2D methods are technically challenging because they need one or more neural networks. The MORL methods depend on the whether the base algorithm is tabular or deep. It is considered OK (somewhere in the middle between Good and Bad) because the MORL methods in this thesis are a mix of tabular and deep RL methods.

- **Learning time**. In terms of learning time per episode, the tabular approach is the cheapest to train and the deep RL methods the most expensive. D2D methods lie somewhere between the tabular and deep methods, and therefore are considered OK. The MORL algorithms in this thesis consist of both tabular and deep methods and are also regarded as OK.

- **Indirect comparison performance**. When compared to the baseline engineered agents, all the approaches show some degree of success in outmanoeuvring the opponent, with the tabular approach having moderate success and the other approaches greater success.

- **Direct comparison performance**. When agents from each of the four approaches are compared against each other, the tabular agents are the worst performers. The other agents perform better, with one of the DQN agents being the best.

- **Learned policies**. The trajectories of the air combat scenarios against the engineered agents reveal what tactics the agents developed. The single-objective and multiobjective tabular agents developed a skill set that make them prone to being outmanoeuvred by the engineered agents. The deep and D2D agents developed more advanced skills that enable them to defeat the engineered agents in more scenarios.

9.3 Answers to Thesis Questions

The main question this thesis tries to answer is *"How can dynamic adversarial games problems, in the context of air combat manoeuvring simulation, be addressed using single- and multi-objective RL algorithms?"*

The question is answered by conducting experiments using four RL approaches: tabular, deep, discrete-to-deep and multiobjective, as presented in Chapters 4, 5, 6 and 7, respectively. The test results show that at least one of the methods in the four approaches is able to provide satisfactory results that outperform the baselines that are based on engineered agents that come with the air combat simulator. Direct performance comparisons that pit the RL agents with engineered agents that are physically superior also reveal that some of the RL agents can beat the engineered agents.

The following are answers to the more specific thesis questions.

1. *To what extent are the speed of reinforcement learning and the quality of policies affected by any of the following factors?*

   - *the structure of the reward function*
   - *the state-space representation*
   - *the choice of the learning algorithm*

   All these factors affect the speed of learning and/or the quality of policies. As shown in Chapter 4, the structure of the reward function affects the speed of learning and the state-space representation affects the quality of the policies. Tabular methods are faster to learn than deep RL methods, but the latter require fewer episodes to achieve similar results. D2D-SPL and D2D-SQL combine the benefits of the tabular and deep RL methods by being able to obtain the generalisability of deep RL at a cost closer to tabular RL.

2. *Is it possible to combine RL and supervised learning techniques to improve learning time and agent experience?* It is, and this thesis proposes two novel methods, D2D-SPL and D2D-SQL, presented in Chapter 6, that combines RL and supervised learning. The main objective of these discrete-to-deep methods is to obtain the generalisability of deep RL methods at a cost that is closer to those of tabular methods.

3. *To what extent does including multiple objectives result in the discovery of better policies?* As explained in Chapter 7, multiobjective RL methods can outperform single-objective ones because MORL allows more information to be fed to the agent. The success of multiobjective RL methods relies heavily on how the rewards are structured.

9.4 Contributions

The main contributions of this thesis are as follows.

- Two novel methods that combine RL and supervised learning and have the generalisability of deep RL at a cost closer to the tabular method. The methods, D2D-SPL and D2D-SQL, are able to outperform engineered agents.
- Demonstration of how to take the air combat manoeuvring problem and turning it into a multiobjective RL problem and the application of multiobjectivisation (Brys et al., 2014) to learn policies that outperform single-objective RL methods. The use of multiobjectivisation to address air combat manoeuvring problem is a new idea.

In addition, this thesis also makes these contributions.

- Demonstration of how to take the air combat manoeuvring problem and turn it into a single-objective RL problem.
- Development of a state discretisation strategy and demonstration of how to use such a strategy to aid in the learning of RL policies using tabular RL methods.
- Development of a state normalisation strategy and demonstration of how to use such a strategy to help in the learning of RL policies using deep RL methods.
- Comparison of tabular, deep, discrete-to-deep and multiobjective RL methods.
- An open-source framework for training and testing single- and multiobjective RL agents called Spy RL.

9.5 Impacts

The following are possible impacts that can come out of this research.

- This thesis shows that multiobjectivisation (Brys et al., 2014) may outperform single-objective methods in a single-objective high-dimensional problem like the air combat manoeuvring domain. For the MORL community this means MORL techniques may now be used to address single-objective problems if reward-decomposition can be done.
- One of the contributions of this thesis is demonstrate that RL agents can defeat engineered agents that have been specifically programmed to outmanoeuvre opponents, in both direct and indirect performance comparisons. For the operations research/simulation community this means another proven tool RL for solving similar agent learning problems. It is a justification to choose RL in future research projects.
- One of the most popular uses of RL has been to play dynamic adversarial games. RL has even proved to be better than human players in Tesauro (1995), Mnih et al. (2013) and

Silver et al. (2016). This thesis is another piece of evidence that shows the effectiveness of RL in addressing dynamic adversarial games.

9.6 Limitations

The following are limitations of the thesis experiments.

1. A limitation of the experiments involving deep RL methods is, only ten results are available for each method. This makes it very hard to determine if the results are uniformly distributed, making performing a T-test impossible. The scarcity of samples in this study and other research involving neural RL are due to the fact that neural RL is very time-consuming. To put it in perspective, Mnih et al. (2015) from Google's DeepMind, presumably having had access to some of the most powerful computers in the world, spent a total 38 days to complete their tests. Irpan (2018) sums it up well when he says, "When your training algorithm is both sample inefficient and unstable, it heavily slows down your rate of productive research. Maybe it only takes 1 million steps. But when you multiply that by 5 random seeds, and then multiply that with hyperparam tuning, you need an exploding amount of compute to test hypotheses effectively."
2. Only one action space was tested, again due to limitations on computing resources. Different sets of actions or different numbers of actions might have returned different results. In addition, only discrete actions were tested and no methods capable of handling continuous actions were evaluated.
3. For the deep RL methods, very little hyperparameter tuning was carried out. Similarly, little was conducted with regard to network architecture.
4. More extensive research on the state space and reward function could have been done if more computing resource had been available.
5. No experiments with more sophisticated flight dynamics models were conducted.
6. Only one scripted agent, which flew in a straight line at a constant speed, was used in learning policies.

9.7 Future Work

Future research may be conducted as an extension to this research and to evaluate topics not covered by this thesis. First, this research uses a two dimensional state space and it should be straightforward to extend it to a three-dimensional environment.

Second, some of the models are based on neural-networks, but work has not been done to tune the hyperparameters of those neural networks. Future work may involve tuning the hyperparameters of the neural-network agents.

Third, RL agents used in this study might learn faster in a reduced state space. One way to reduce the state space without adversely affecting performance is probably by taking advantage of symmetrical conditions. In addition, training with engineered agents and self-play might also be explored.

Next, the novel algorithms proposed and implemented in this thesis, the D2D-SPL and D2D-SQL methods, can be investigated further, for example whether they would help in problems with higher dimensional state variables. Future studies may take a closer look at other tabular RL algorithms, such as Q-learning and SARSA, in the reinforcement phase of D2D-SPL and D2D-SQL and further train the resulting network of D2D-SPL using a method other than D2D-SQL. As the new algorithms have only been applied in single-objective environments, it is also interesting to see how multiobjective D2D-SPL and multiobjective D2D-SQL would perform.

Finally, this work can be extended to apply to more complex domains involving multiple agents or domains that are inherently multiobjective.

Appendix A

Hardware and Software Used

This appendix explains the hardware and software used for this thesis.

A.1 Hardware

All experiments in this thesis are run on a Ubuntu workstation that has an Intel i9-7900X CPU with ten cores and 20 threads and is equipped with two Nvidia GeForce GTX 1080 graphics cards. For the reason given below, all tests are run on the CPU instead of the GPU's.

A.2 Software

All programs are written in Python 3.8 and use these libraries:

- PyTorch 1.4 for deep RL.
- Matplotlib and Seaborn for data visualisation
- scikit-learn for D2D-SPL's supervised phase

The software modules include these components:

- The AceZeroEnvironment, as explained in Chapter 4,
- Reinforcement learning agents,
- Spy RL framework, as discussed in Appendix C.

A.3 Deep RL Learning Strategy

Deep learning is notoriously slow. For example, Mnih et al. (Mnih et al., 2015) mention that each ATARI game took 38 days to complete learning. In some experiments for this thesis, running ten trials of a deep RL agent on a single CPU core would have taken about 20 days. In theory, those ten trials could have run concurrently on ten cores and finish in approximately two days. However, this is not that straightforward when PyTorch is involved. PyTorch that runs on a CPU (as opposed to a GPU) will try to acquire as many cores as possible to parallelise calculations. As a result, running multiple instances of the same program would make the multiple processes involved compete for the same resources and cause all the instances to stall!

Because of this, PyTorch was configured to use only one thread by setting the environment variable OMP_NUM_THREADS to 1. In Linux this can be done using this command.

```
export OMP_NUM_THREADS=1
```

Therefore, the choice is between running a single program instance and letting PyTorch acquire as many threads as it pleases and running multiple instances of the same program and restricting PyTorch to one thread. Measuring the learning time of a task involving a regression model and 20,000 episodes of PyTorch-based Ace Zero agent learning returns the following:

- Running a single program instance and letting PyTorch use as many threads as it wants takes about 61,000 seconds.
- Running two instances of the same task and letting PyTorch use as many threads as it wants takes 835,410 seconds for both instances to finish, which means a unit cost of 417,705 seconds. This is not surprising considering each PyTorch instance will try to acquire as many threads as possible and both instances compete for the same resources.
- Running ten instances of the same task where each instance is restricted to one thread takes 160,900 seconds.

What this means is, letting PyTorch parallelise its load helps as it is able to complete the task in 61,000 seconds. If there is only one task to complete, then this is the obvious choice. For this thesis, however, hundreds of millions of learning episodes need to be run, so the best strategy is to run multiple instances and restrict PyTorch to one thread. In this example, the cost per unit of task is 61,000 versus 16,090, so running multiple instances costs 3.7 times less. This is detailed in Table A.1.

	10 instances (1 thread limit)	1 instance (no limit)	2 instances (no limit)
Cost per task (hours)	4.47	16.94	116.03

Table A.1: Cost of completing a PyTorch task

Appendix B

Baselines

This thesis compares the performance of reinforcement learning (RL) agents both directly, as explained in Chapter 9, and indirectly. For indirect performance comparison, it establishes baselines, which are points of reference to compare RL agents with. Ace Zero comes with three finite state machine (FSM) agents (Pure Pursuit, Smart Pursuit and Stern Conversion) that have been used in other studies (Masek et al., 2018; Lam et al., 2019; Ramirez et al., 2018, 2017). These FSM agents are referred to as engineered agents, as opposed to RL agents, that are learning agents. The FSM agents are programmatically controlled using these strategies.

- Pure Pursuit. At every timestep, this FSM agent issues a command to change direction so that it flies towards the opponent.
- Smart Pursuit. In addition to changing direction the way the Pure Pursuit agent does, at every timestep the Smart Pursuit agent also changes its speed to match the speed of its adversary.
- Stern Conversion. This agent performs a stern conversion manoeuvre against the opposing aircraft. This manoeuvre is depicted in Figure B.1

In indirect performance comparison, an RL agent is run against a scripted agent that flies in a straight line at a constant speed and is compared with the performance of the FSM agents running against the same scripted agent. In total 144 tests in Basic Test Suite, explained in Chapter 4, are conducted.

Figure B.2 shows the performance of the engineered agents against a scripted agent starting from 144 initial positions following Basic Test Suite. Because the Smart Pursuit agent out-

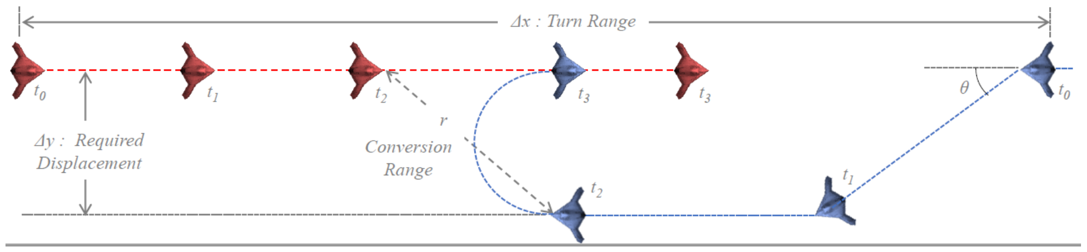


Figure B.1: Stern conversion manoeuvre. The figure is reproduced here with permission from the authors of (Ramirez et al., 2018)

performs the other two FSM agents in all tests, only the Smart Pursuit agent is used in this performance comparison test.

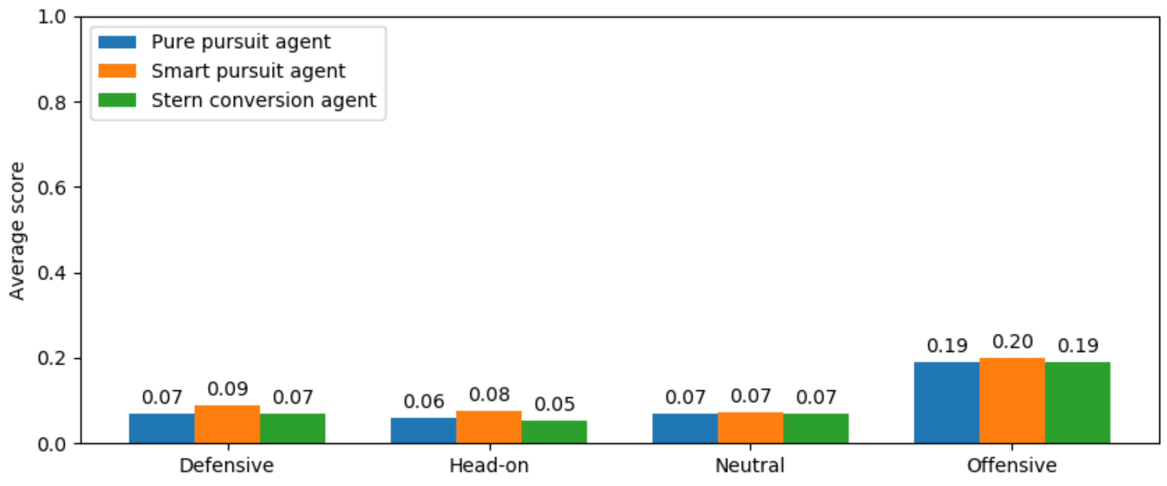


Figure B.2: Baselines for Basic Test Suite

Appendix C

Spy RL Framework

For this thesis, a modular and extensible framework was designed and developed to easily run and test reinforcement learning (RL) agents. The decision to write a new RL framework was taken after reviewing existing RL frameworks such as OpenAI Baselines (Dhariwal et al., 2017), Stable Baselines from ENSTA Paris (Hill et al., 2018), RLLib (Liang et al., 2018) and Acme from DeepMind (Hoffman et al., 2020). Even though these frameworks are easy to use and allow users to choose from a variety of algorithms, they lack these qualities:

- No support for running multiple trials of the same learning session. The user must run the same test repeatedly.
- Not extensible. For example, to add certain functionality, such as writing rewards to a text file for later use, the user has to modify the current code.
- Do not support dependency injection. Dependency injection has long been the standard for modular software that allows programmers to change the implementation of an interface easily.
- Some frameworks are not cross-platform. For instance, RLLib does not run on Windows.
- They only create policies for the last episodes, even though policies from the last episodes are not necessarily the best performers.
- They do no allow different discretisers to be used. This way, they force state discretisation to occur in the environment itself.
- No support for multiobjective RL.

To meet the requirement above, a single- and multiobjective RL framework called Spy RL was developed for this thesis. It is open-source, object-oriented, extensible and reusable. It is easy to run an experiments for many trials and generate a learning curve for all the trials as well as create best policies from a learning session.

C.1 General Implementation of Reinforcement Learning

Generally, an RL learning algorithm implements the basic diagram and workflow in Figure C.1. The algorithm starts by telling the agent the initial state it is in. For each timestep, the agent selects an action that it passes to the environment and the environment responds by returning a reward and the next state to the agent. The back and forth continues until the agent reaches a terminal state or some other condition is met. The whole cycle is called an episode. Typically, a learning session consists of many trials that each is made up of episodes. A Python code for general RL is given in Listing C.1.

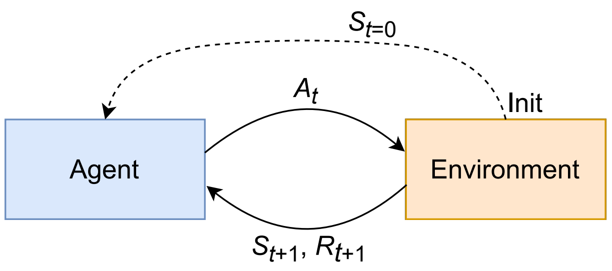


Figure C.1: A generic RL algorithm

```python
for trial in range(num_trials):
    seed = random()
    env = create_environment(seed)
    agent = create_agent(seed)
    for episode in range(num_episodes):
        state = env.reset()
        agent.init(episode)
        for step in range(num_steps):
            action = agent.select_action(state)
            next_state, reward, terminal, env_data = env.step(action)
            agent.update(state, action, reward, next_state, terminal, env_data)
            state = next_state
            if terminal:
                break
```

Listing C.1: Learning with Spy RL

C.2 Spy RL Framework

This section explains the architecture of Spy RL, how to install it and where to download the source. The end of the section presents an example of how to use it.

C.2.1 Architecture

Spy RL achieves its extendibility by using the observer pattern, a classic design pattern that has been used in software engineering for decades and first catalogued by Gamma et al. (Gamma et al., 1994). This pattern allows an object, called the subject, to maintain a list of observers (listeners) that get notified of any state changes in the subject. One of the advantages of the observer pattern is that the subject and its observers are decoupled, meaning the subject does not need to know about its future observers.

The observer pattern is commonly used in graphical user interface (GUI) systems, such as Swing, JavaFX and HTML controls. When the creators of such a GUI system write the code for a button, for example, they do not need to know what the programmers of their GUI system will do with the button. The creators just need to make sure that other elements can register interest with the button for some state change in the button and that the button will notify such a change to interested observers.

Figure C.2 shows the UML class diagram of the observer pattern and Figure C.3 the observer pattern as used in Spy RL.

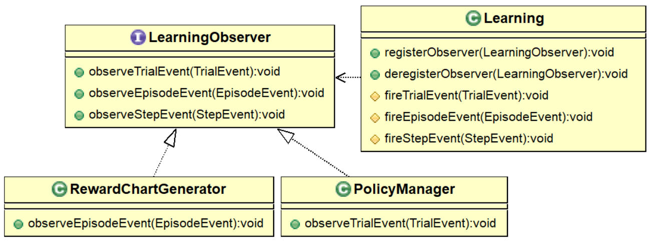

LearningObserver
observeTrialEvent(TrialEvent):void
observeEpisodeEvent(EpisodeEvent):void
observeStepEvent(StepEvent):void
Learning
registerObserver(LearningObserver):void
deregisterObserver(LearningObserver):void
fireTrialEvent(TrialEvent):void
fireEpisodeEvent(EpisodeEvent):void
fireStepEvent(StepEvent):void
RewardChartGenerator
observeEpisodeEvent(EpisodeEvent):void
PolicyManager
observeTrialEvent(TrialEvent):void

Figure C.2: The observer pattern

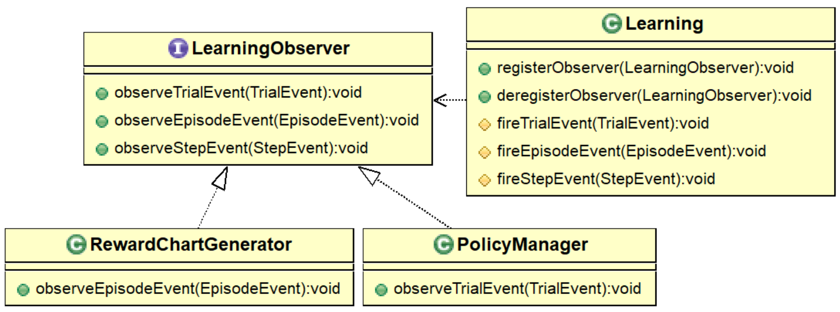

LearningObserver
observeTrialEvent(TrialEvent):void
observeEpisodeEvent(EpisodeEvent):void
observeStepEvent(StepEvent):void
Learning
registerObserver(LearningObserver):void
deregisterObserver(LearningObserver):void
fireTrialEvent(TrialEvent):void
fireEpisodeEvent(EpisodeEvent):void
fireStepEvent(StepEvent):void
RewardChartGenerator
observeEpisodeEvent(EpisodeEvent):void
PolicyManager
observeTrialEvent(TrialEvent):void

Figure C.3: The application of the observer pattern in Spy RL

Listing C.2 shows the inner working of a learning session in Spy RL. The introduction of events allow the user to write reusable event handlers that target specific tasks.

```
for trial in range(num_trials):
    seed = random()
    fire_before_trial_event(TrialEvent(trial, seed)
    env = create_environment(seed)
    agent = create_agent(seed)
    for episode in range(num_episodes):
        fire_before_episode_event(EpisodeEvent(trial, seed)
        state = env.reset()
        agent.init(episode)
        for step in range(num_steps):
            fire_before_step_event(StepEvent(trial, seed)
            action = agent.select_action(state)
            next_state, reward, terminal, env_data = env.step(action)
            agent.update(state, action, reward, next_state, terminal, env_data)
            state = next_state
            fire_after_step_event(StepEvent(trial, seed)
            if terminal:
                break
        fire_after_episode_event(EpisodeEvent(trial, seed)
    fire_after_trial_event(TrialEvent(trial, seed)
```

Listing C.2: Spy RL learning that includes event firing

C.2.2 Installation

Spy RL can be installed using pip:

```
pip install spyrl
```

C.2.3 Source Code

The source can be downloaded from https://github.com/budi-kurniawan/spyrl.

C.2.4 Usage

The idea of using an RL framework is to save researchers writing similar boilerplate code for different learning sessions or algorithms. Spy RL provides the most commonly used code that makes it easy to train and test RL agents. The steps for learning a policy are as follows:

- Create an environment
- Create an agent builder

- Create an ActivityConfig to specify the number of episodes, where the results should be save and so on.
- Create a Learning object, passing as many observers (listeners) as required. Each observer provides different functionality
- Call the learn method of the Learning object

As an example, the code in Listing C.3 learns an actor-critic policy for 1,000 episodes to solve the Gym Cartpole problem. The BasicFunctions listener used creates a learning graph, saves a policy, renders the graphical user interface and prints the average score for each episode.

```
env = gym.make('CartPole-v0')
num_actions = env.action_space.n
config = ActivityConfig(num_episodes=1000, out_path='result/')
agent_builder = ActorCriticTracesAgentBuilder(num_actions, discretiser=
    CartpoleDiscretiser())
learning = Learning(listener=BasicFunctions(render=False))
learning.learn(env, agent_builder, config)
```

Listing C.3: Using Spy RL

Appendix D

Solving Gridworld and Cartpole

This appendix compares three algorithms–Q-learning, Q(λ) and actor-critic with eligibility traces (ACET)–used to solve two classic reinforcement learning (RL) problems, the Gridworld and Cartpole. The objective of this appendix is to show that there is probably no panacea that can solve all sorts of RL problems. Specifically, it is shown in this appendix that value-based methods such as Q-learning and Q(λ) perform well on discrete problems like the Gridworld but not on continuous problems like Cartpole. Conversely, ACET, a policy gradient algorithm, performs badly on Gridworld but addresses Cartpole well. This may also explain why the Q(λ) method in Chapter 5 had difficulty learning when faced with a continuous problem like the air combat manoeuvring problem while ACET showed success.

D.1 Gridworld

Figure D.1 shows the performance of three methods trying to solve a 30x30 Gridworld problem. For each timestep, the agent receives a reward of -1 unless it reaches a goal state for which it receives 100. A random agent, not included in the graph, has an average score of -8,010.50. The numbers in the legend show the average scores for the algorithms involved over 1,000 episodes.

The learning curves shows Q-Learning and Q(λ) are very good at solving the Gridworld problem. The ACET method is not that successful.

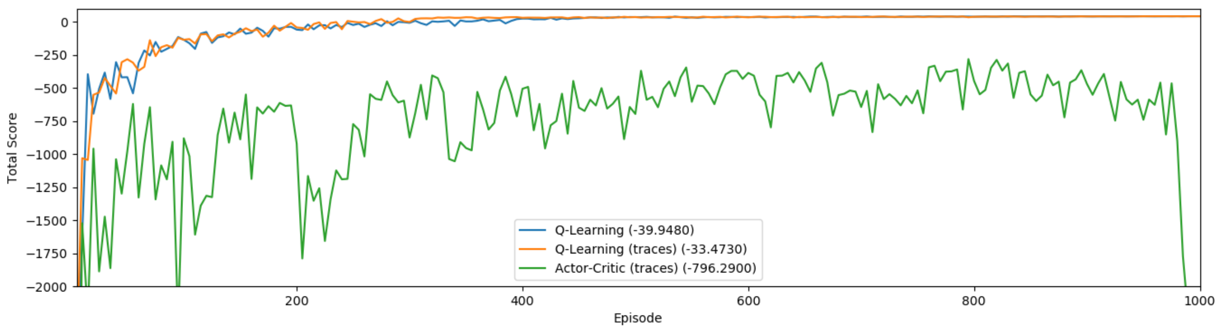


Figure D.1: Comparing methods for the Gridworld problem

D.2 Cartpole

Cartpole is a pole-balancing control problem posed in (Michie and Chambers, 1968). A solution aims to control the free-moving cart to which the pole is attached by exerting a force to the left or to the right of the cart to keep the pole standing. The OpenAI Gym Cartpole environment (Brockman et al., 2016) is used. The environment is based on an implementation by Barto et. al. (Barto et al., 1983), but is different from the original system in that the four state variables in Gym are randomised at the beginning of every episode, whereas in the original system the variables are always set to zero.

This section shows the results of applying the three methods to solve the Cartpole problem. Table D.2 shows that ACET is the most effective among the agents in solving this problem and Q(λ) comes second. Q-learning is the worst as its result is lower than the random agent.

Agent	Score
ACET	3,264
Q(λ)	202
Q-Learning	153
Random	157

REFERENCES

Achiam, J., Knight, E., and Abbeel, P. (2019). Towards characterizing divergence in deep Q-learning. *arXiv:1903.08894*.

Anschel, O., Baram, N., and Shimkin, N. (2017). Averaged-DQN: Variance reduction and stabilization for deep reinforcement learning. *Proceedings of International Conference on Machine Learning*, page 176–185.

Azak, M. and Bayrak, A. (2008). A new approach for threat evaluation and weapon assignment problem, hybrid learning with multi-agent coordination. *Proceedings of the 23rd International Symposium on Computer and Information Sciences (ISCIS) in Istanbul*, pages 1–6.

Baird, L. (1995). Residual algorithms: Reinforcement learning with function approximation. In *Machine Learning Proceedings 1995*, pages 30–37. Elsevier.

Barnard, E. (1993). Temporal-difference methods and Markov models. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(2):357–365.

Barto, A. and Mahadevan, S. (2003). Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems, vol. 13, no. 1–2, p. 41–77*.

Barto, A., Sutton, R., and Anderson, C. (1983). Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE Transactions on System, Man, and Cybernetics, Vol. SMC-13, No. 5*, pages 833–836.

Bellman, R. (1952). On the theory of dynamic programming. *Tech. rep., Proceedings National Academia of Science*.

Bellman, R. (1961). *Adaptive Control Processes*. Princeton University Press, Princeton, NJ.

Berndt, J. (2004). Jsbsim: An open source flight dynamics model in C++. *Modeling and Simulation Technologies Conference and Exhibit. American Institute of Aeronautics and Astronautics*.

Bertsekas, D. and Tsitsiklis, J. (1996). Neuro-dynamic programming.

Bilgin, A. and Kadioglu-Urtis, E. (2015). An approach to multi-agent pursuit evasion games using reinforcement learning. *2015 International Conference on Advanced Robotics (ICAR)*.

Boyan, J. and Moore, A. (1995). Generalization in reinforcement learning: Safely approximating the value function. *NIPS-7. San Mateo, CA: Morgan Kaufmann*.

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). OpenAI Gym.

Brys, T., Harutyunyan, A., Vrancx, P., Taylor, M., Kudenko, D., and Nowe, A. (2014). Multi-objectivization of reinforcement learning problems by reward shaping. *2014 International Joint Conference on Neural Networks (IJCNN)*, pages 2315–2322.

Burgin, G. and Fogel, L. (1975). An adaptive maneuvering logic computer program for the simulation of one-on-one air-to-air combat. vol. i: General description. *NASA CR 2582*.

Burgin, G. and Sidor, L. (1988). Rule-based air combat simulation. *NASA CR 4160*.

Chebotar, Y., Hausman, K., Kroemer, O., Sukhatme, G., and Schaal, S. (2016). Generalizing re-grasping with supervised policy learning. *International Symposium on Experimental Robotics*.

Coggan, M. (2001). Exploration and exploitation in reinforcement learning. *Proc. 4th Int. Conf. Comput. Intell. Multimedia Appl.*, pages 1–44.

Comanici, G. and Precup, D. (2010). Optimal policy switching algorithms for reinforcement learning. *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2010)*, pages 709–714.

Daley, B. and Amato, C. (2019). Reconciling λ-returns with experience replay. *33rd Conference on Neural Information Processing Systems*.

Dayan, P. and Hinton, G. (1993). Feudal reinforcement learning. *Advances in Neural Information Processing Systems, vol. 5. Morgan-Kaufmann*.

Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., Wu, Y., and Zhokhov, P. (2017). OpenAI Baselines. `https://github.com/openai/baselines`.

Doya, K. (2000). Reinforcement learning in continuous time and space. *Neural Computation, Vol. 12 , Issue 1*, page 219–245.

Efron, B. (1979). Bootstrap methods: another look at the jackknife. *Ann. Statist. 7, 101-118*.

Fang, J., Yan, W., and Fang, W. (2017a). Air combat strategies of CGF based on Q-learning and behavior tree. *DEStech Transactions on Engineering and Technology Research*.

Fang, M., Li, Y., and Cohn, T. (2017b). Learning how to active learn: A deep reinforcement learning approach. *arXiv:1708.02383*.

Floyd, M., Karneeb, J., Moore, P., and Aha, D. (2017). A goal reasoning agent for controlling UAVs in beyond-visual-range air combat. *Proc. 26th International Joint Conference on Artificial Intelligence*.

Frans, K., Ho, J., Chen, X., Abbeel, P., and Schulman, J. (2018). Meta learning shared hierarchies. *6th International Conference on Learning Representations, (ICLR 2018)*.

Fruit, R., Pirotta, M., Lazaric, A., and Ortner, R. (2018). Efficient bias-span-constrained exploration-exploitation in reinforcement learning. *Proceedings of the 35th International Conference on Machine Learning*.

Gabor, Z., Kalmar, Z., and Szepesvari, C. (1998). Multi-criteria reinforcement learning. *The 15th International Conference on Machine Learning*.

Gai, K. and Qiu, M. (2018). Optimal resource allocation using reinforcement learning for IoT content-centric services. *Applied Soft Computing, Volume 70*, pages 12–21.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design patterns: Elements of reusable object-oriented software*. Addison Wesley.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press.

Grześ, M. and Kudenko, D. (2009). Improving optimistic exploration in model-free reinforcement learning. *Adaptive and Natural Computing Algorithms*, pages 360–369.

Handl, J., Lovell, S., and Knowles, J. (2008). Multiobjectivization by decomposition of scalar cost functions. *Parallel Problem Solving from Nature–PPSN X. Springer*.

Harb, J. and Precup, D. (2017). Investigating recurrence and eligibility traces in deep Q-networks. *arXiv:1704.05495*.

Hessel, M., Modayil, J., Van Hasselt, H., Schaul, T., Ostrovski, G., Dabney, W., Horgan, D., Piot, B., Azar, M., and Silver, D. (2018). Rainbow: Combining improvements in deep reinforcement learning. *AAAI Conference on Artificial Intelligence*.

Hill, A., Raffin, A., Ernestus, M., Gleave, A., Kanervisto, A., Traore, R., Dhariwal, P., Hesse, C., Klimov, O., Nichol, A., Plappert, M., Radford, A., Schulman, J., Sidor, S., and Wu, Y. (2018). Stable Baselines.

Hoffman, M., Shahriari, B., Aslanides, J., Barth-Maron, G., Behbahani, F., Norman, T., Abdolmaleki, A., Cassirer, A., Yang, F., Baumli, K., Henderson, S., Novikov, A., Colmenarejo, S., Cabi, S., Gulcehre, C., Le Paine, T., Cowie, A., Wang, Z., Piot, B., and de Freitas, N. (2020). Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*.

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., and Silver, D. (2018). Distributed prioritized experience replay. *ICLR*.

Hossam, M., Le, T., Huynh, V., Papasimeon, M., and Phung, D. (2020). OptiGAN: Generative adversarial networks for goal optimized sequence generation. *arXiv:2004.07534*.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

Irpan, A. (2018). Deep reinforcement learning doesn't work yet. `https://www.alexirpan.com/2018/02/14/rl-hard.html`.

Ishii, S., Yoshida, W., and Yoshimoto, J. (2002). Control of exploitation–exploration meta-parameter in reinforcement learning. *Neural Networks, Volume 15, Issues 4–6*, pages 665–687.

Kaelbling, L. and Littman, M. (1996). Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, pages 237–285.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv:1412.6980*.

Kokolakis, N.-M. T., Kanellopoulos, A., and Vamvoudakis, K. G. (2020). Bounded rational unmanned aerial vehicle coordination for adversarial target tracking. In *2020 American Control Conference (ACC)*, pages 2508–2513.

Kong, W., Zhou, D., Yang, Z., Zhao, Y., and Zhang, K. (2020). UAV autonomous aerial combat maneuver strategy generation with observation error based on state-adversarial deep deterministic policy gradient and inverse reinforcement learning. *Electronics 2020, 9, 1121*.

Kurniawan, B., Vamplew, P., Papasimeon, M., Dazeley, R., and Foale, C. (2019). An empirical study of reward structures for actor-critic reinforcement learning in air combat manoeuvring simulation. *32nd Australasian Joint Conference on Artificial Intelligence*.

Kurniawan, B., Vamplew, P., Papasimeon, M., Dazeley, R., and Foale, C. (2020). Discrete-to-deep supervised policy learning. *arXiv:2005.02057*.

Kurniawan, B., Vamplew, P., Papasimeon, M., Dazeley, R., and Foale, C. (2021). Discrete-to-deep reinforcement learning methods. *Neural Computing and Applications*.

Källström, J. and Heintz, F. (2019). Multi-agent multi-objective deep reinforcement learning for efficient and effective pilot training. *Proceedings of the 10th Aerospace Technology Congress*, pages 101–111.

Källström, J. and Heintz, F. (2020). Agent coordination in air combat simulation using multi-agent deep reinforcement learning. *2020 IEEE International Conference on Systems, Man, and Cybernetics*.

Lam, C., Masek, M., Kelly, L., Papasimeon, M., and Benke, L. (2019). A simheuristic approach for evolving agent behaviour in the exploration for novel combat tactics. *Operations Research Perspectives 6*.

Lee, D. and Bang, H. (2012). Planar evasive aircrafts maneuvers using reinforcement learning. *Intelligent Autonomous Systems*.

Levine, S., Wagener, N., and Abbeel, P. (2015). Learning contact-rich manipulation skills with guided policy search. *IEEE International Conference on Robotics and Automation*, pages 156–163.

Li, K., Shi, H., Zhao, Q., Liu, P., Niu, C., and Zhang, K. (2019). An autonomous maneuvering decision algorithm of UAV based on deep deterministic policy gradient under endpoint constraints. *Eighth European Conference for Aeronautics and Space Sciences*.

Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Goldberg, K., Gonzalez, J., Jordan, M., and Stoica, I. (2018). RLlib: Abstractions for distributed reinforcement learning. *International Conference on Machine Learning (ICML)*.

Lillicrap, T., Hunt, J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Lin, C. and Chung, I. (1999). A reinforcement neuro-fuzzy combiner for multiobjective control. *IEEE Transactions on Systems, Man and Cybernetics*.

Lin, L. (1993). Reinforcement learning for robots using neural networks. *Phd thesis, Carnegie Mellon University*.

Liu, P. and Ma, Y. (2017). A deep reinforcement learning based intelligent decision method for UCAV air combat. *Modeling, Design and Simulation of Systems*, pages 274–286.

Liu, Y. and Zhang, J. (2018). Deep learning in machine translation. *Deep Learning in Natural Language Processing, 147-183*.

Lizotte, D., Bowling, M., and Murphy, S. (2010). Efficient reinforcement learning with multiple reward functions for randomized controlled trial analysis. *Proceedings of the 27th International Conference on Machine Learning*.

Ma, X., Xia, L., and Zhao, Q. (2018). Air combat strategy using deep Q-learning. *Automat. Congr. (CAC)*, pages 3952–3957.

Masek, M., Lam, C., Benke, L., Kelly, L., and Papasimeon, M. (2018). Discovering emergent agent behaviour with evolutionary finite state machines. *Int. Conf. on Principles and Practice of Multi-Agent Systems*.

McGrew, J., How, J., Williams, B., and Roy, N. (2010). Air-combat strategy using approximate dynamic programming. *Journal of Guidance, Control, and Dynamics*, 33(5).

Michie, D. and Chambers, R. (1968). Boxes: An experiment in adaptive control. *Machine Intelligence*, 2(2):137–152.

Min, S., Lee, B., and Yoon, S. (2017). Deep learning in bioinformatics. *Briefings in bioinformatics 18 (5), 851-869*.

Mizokami, K. (2017). This chart explains how crazy-expensive fighter jets have gotten. *Popular Mechanics*.

Mnih, V., Badia, A., Mirza, M., Graves, A., Harley, T., Lillicrap, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. *Proc. 33rd Int. Conf. Mach. Learn.*, 48.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing Atari with deep reinforcement learning. *NIPS Deep Learning Workshop*.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wierstra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, pages 29–33.

Mouton, H., Roodt, J., and le Roux, H. (2011). Applying reinforcement learning to the weapon assignment problem in air defence. *Scientia Militaria: The South African Journal for Military Studies*, pages 123–140.

Nair, A., Srinivasan, P., Blackwell, S., Alcicek, C., Fearon, R., DeMaria, A., Panneershelvam, V., Suleyman, M., Beattie, C., Petersen, S., Legg, S., Mnih, V., Kavukcuoglu, K., and Silver, D. (2015). Massively parallel methods for deep reinforcement learning. *ICML Deep Learning Workshop*.

Noda, K., Yamaguchi, Y., Nakadai, K., Okuno, H., and Ogata, T. (2015). Audio-visual speech recognition using deep learning. *Applied Intelligence 42 (4)*.

Papasimeon, M. and Benke, L. (2021). Multi-agent simulation for AI behaviour discovery in operations research. In *22nd International Workshop on Multi-Agent-Based Simulation (MABS 2021) at International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, London, UK.

Park, H., Lee, B., and Takh, M. (2016). Differential game based air combat maneuver generation using scoring function matrix. *International Journal of Aeronautical and Space Science*, 17(2):204–213.

Peng, J. and Williams, R. (1994). Incremental multi-step qlearning. *Proceedings of the 11th International Conference on Machine Learning*, pages 226–232.

Peters, J. and Neumann, G. (2015). Policy search: Methods and applications. *Tutorial at the 32nd International Conference on Machine Learning*.

Pollack, J. and Blair, A. (1997). Why did TD-Gammon work? *Advances in Neural Information Processing Systems*, page 10–16.

Pope, A., Ide, J., Micovic, D., Diaz, H., Rosenbluth, D., Ritholtz, R., Twedt, J., Walker, T., Alcedo, K., and Javorsek, D. (2021). Hierarchical reinforcement learning for air-to-air combat. *arXiv:2105.00990*.

Pritzel, A., Uria, B., Srinivasan, S., Badia, A. P., Vinyals, O., Hassabis, D., Wierstra, D., and Blundell, C. (2017). Neural episodic control. *Proc. 34th International Conference on Machine Learning*.

Ramirez, M., Papasimeon, M., Benke, L., Lipovetzky, N., Miller, T., and Pearce, A. (2017). Real–time UAV maneuvering via automated planning in simulations. *Proc. 26th International Joint Conference on Artificial Intelligence (IJCAI-17)*.

Ramirez, M., Papasimeon, M., Lipovetzky, N., Benke, L., Miller, T., Pearce, A., Scala, E., and Zamani, M. (2018). Integrated hybrid planning and programmed control for real time UAV maneuvering. *Proc. 17th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1318–1326.

Riedmiller, M. (2005). Neural fitted Q iteration – first experiences with a data efficient neural reinforcement learning method. *Machine Learning: ECML 2005*.

Roijers, D., Vamplew, P., Whiteson, S., and Dazeley, R. (2013). A survey of multi-objective sequential decision-making. *Journal of Artificial Intelligence Research*, 48.

Rosenstein, M. and Barto, A. G. (2002). Supervised learning combined with an actor critic architecture. *Technical report, Amherst, MA, USA*.

Rummery, G. A. and Niranjan, M. (1994). On-line Q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166. Engineering Department, Cambridge University*.

Russel, S. and Zimdars, A. (2003). Q-decomposition for reinforcement learning agents. *Proc. of the Twentieth International Conference on Machine Learning*, pages 656–661.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2015). Prioritized experience replay. *arXiv preprint arXiv:1511.05952*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimove, O. (2017). Proximal policy optimization algorithms. *arXiv preprintarXiv:1707.06347*.

Shalev, S., Shammah, S., and Shashua, A. (2016). Safe, multi-agent, reinforcement learning for autonomous driving. *CoRR*, abs/1610.03295.

Shaw, R. (1985). *Fighter combat: Tactics and maneuvering*. Naval Institute Press.

Silver, D., Huang, A., Maddison, C., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., and Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., and Riedmiller, M. (2014). Deterministic policy gradient algorithms. *ICML*.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., and Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nature*.

Singh, S. and Sutton, R. (1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*.

Socher, R., Bengio, Y., and Manning, C. (2012). Deep learning for NLP (without magic). *Tutorial Abstracts of ACL*.

Soyluoğlu, B. (2021). Modelling aircraft fighting maneuver dynamics using artificial intelligence algorithms. *Graduation project, Department of Aeronautical Engineering, Istanbul Technical University*.

Sun, Z., Piao, H., Yang, Z., Zhao, Y., Zhan, G., Zhou, D., Meng, G., Chen, H., Chen, X., Qu, B., and Lu, Y. (2021). Multi-agent hierarchical policy gradient for air combat tactics emergence via self-play. *Engineering Applications of Artificial Intelligence Volume 98*.

Sutton, R. and Barto, A. (1998). *Reinforcement Learning: An Introduction*. MIT Press.

Sutton, R. and Barto, A. (2018). *Reinforcement Learning: An Introduction (Second Edition)*. MIT Press.

Sutton, R., McAllester, D., Singh, S., and Mansour, Y. (2000). Policy gradient methods for reinforcement learning with function approximation. *Advances in Neural Information Processing Systems*, page 1057–1063.

Sutton, R. S. (1984). Temporal credit assignment in reinforcement learning. *Ph.D. thesis, University of Massachusetts, Amherst*.

Teng, T., Tan, A., Tan, Y., and Yeo, A. (2012). Self-organizing neural networks for learning air combat maneuvers. *IEEE World Congress on Computational Intelligence, Brisbane, Australia*.

Tesauro, G. (1995). TD-Gammon: A self-teaching backgammon program. *Applications of Neural Networks*.

Tesauro, G., Das, R., Chan, H., Kephart, J., Levine, D., Rawson, F., and Lefurgy, C. (2008). Managing power consumption and performance of computing systems using reinforcement learning. *NIPS*.

Thrun, S. (1992). Efficient exploration in reinforcement learning. *Technical Report CS-CMU92-102, School of Computer Science, Carnegie Mellon University*.

Thrun, S. and Schwartz, A. (1993). Issues in using function approximation for reinforcement learning. *Proceedings of the 1993 Connectionist Models Summer School, Hillsdale, NJ*.

Tsitsiklis, J. and Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE transactions on automatic control*, 42(5):674–690.

Vamplew, P., Dazeley, R., Berry, A., Issabekov, R., and Dekker, E. (2011). Empirical evaluation methods for multiobjective reinforcement learning algorithms. *Machine Learning*, 84(51).

Van Hasselt, H. (2010). Double Q-learning. *Advances in Neural Information Processing Systems, 23:2613-2621*.

van Hasselt, H., Guez, A., and Siver, D. (2016). Deep reinforcement learning with double Q-learning. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*.

van Moffaert, K., Drugan, M., and Nowé, A. (2013). Hypervolume-based multi-objective reinforcement learning. *International Conference on Evolutionary Multi-Criterion Optimization*.

van Seijen, H. and Sutton, R. (2014). True online TD(λ). *Proceedings of The 31st International Conference on Machine Learning*, page 692–700.

van Vaerenbergh, K., Rodriguez, A., Gagliolo, M., Vracx, P., Nowé, A., Stoev, J., Goossens, S., Pinte, G., and Symens, W. (2012). Improving wet clutch engagement with reinforcement learning. *The 2012 International Joint Conference on. IEEE*.

Vinberg, D. (2009). Guided reinforcement learning applied to air-combat simulation. *Master's thesis, Royal Institute of Technology, Sweden*.

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero,

E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.

Vlahov, B., Squires, E., Strickland, L., and Pippin, C. (2018). On developing a uav pursuit-evasion policy using reinforcement learning. *17th IEEE International Conference on Machine Learning and Applications (ICMLA)*.

Voß, T., Beume, N., Rudolph, G., and Igel, C. (2008). Scalarization versus indicator-based selection in multi-objective cma evolution strategies. *IEEE Congress on Evolutionary Computation*.

Voulodimos, A., Doulamis, N., Doulamis, A., and Protopapadakis, E. (2018). Deep learning for computer vision: A brief review. *Computational Intelligence and Neuroscience*.

Wang, L., Zhang, W., X., H., and Zha, H. (2018). Supervised reinforcement learning with recurrent neural network for dynamic treatment recommendation. *International Conference on Knowledge Discovery and Data Mining*, pages 2447–2456.

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and de Freitas, N. (2016). Sample efficient actor-critic with experience replay. *arXiv preprint arXiv:1611.01224*.

Wang, Z., de Freitas, N., and Lanctot, M. (2015). Dueling network architectures for deep reinforcement learning. *ArXiv e-prints*.

Wang, Z., Li, H., Wu, H., and Wu, Z. (2020). Improving maneuver strategy in air combat by alternate freeze games with a deep reinforcement learning algorithm. *Mathematical Problems in Engineering, Volume 2020, Article ID 7180639*.

Watkins, C. (1989). Learning from delayed rewards. *Phd thesis, King's College, University of London*.

Witten, I. H. (1977). An adaptive optimal controller for discrete-time Markov environments. *Information and Control, 34(4)*, pages 286–295.

Wu, X., Chen, H., Wang, J., Troiano, L., Loia, V., and Fujita, H. (2020). Adaptive stock trading strategies with deep reinforcement learning methods. *Information Sciences, Volume 538*, pages 142–158.

Yang, Q., Zhang, J., Shi, G., Hu, J., and Wu, Y. (2019a). Maneuver decision of UAV in short-range air combat based on deep reinforcement learning. *IEEE Access*.

Yang, Q., Zhu, Y., Zhang, J., Qiao, S., and Liu, J. (2019b). UAV air combat autonomous maneuver decision based on DDPG algorithm. *IEEE 15th International Conference on Control and Automation*.

Yehoshua, R., Heredia-Juesas, J., Wu, Y., Amato, C., and Martinez-Lorenzo, J. (2021). Decentralized reinforcement learning for multi-target search and detection by a team of drones. *arXiv preprint arXiv:2103.09520*.

Zhang, L., Xu, J., Gold, D., Hagen, J., Kochhar, A., Lohn, A., and Osoba, O. (2020). Air dominance through machine learning: A preliminary exploration of artificial intelligence–assisted mission planning. *Santa Monica, CA: RAND Corporation*.

Zhang, R., Zong, Q., Zhang, X., Dou, L., and Tian, B. (2022). Game of drones: multi-uav pursuit-evasion game with online motion planning by deep reinforcement learning. *IEEE Transactions on Neural Networks and Learning Systems*.

Zhang, S. and Sutton, R. S. (2017). A deeper look at experience replay. *CoRR, abs/1712.01275*.

Zhang, X., Liu, G., Yang, C., and Wu, J. (2018). Research on air combat maneuver decision-making method based on reinforcement learning. *Electronics, vol. 7, no. 11*.

Zhao, Y., Chen, Q., and Hu, W. (2010). Multi-objective reinforcement learning algorithm for MOSDMP in unknown environment. *Proceedings of the 8th World Congress on Intelligent Control and Automation*.