Strategically Training and Evaluating Agents in Procedurally Generated Environments

Richard Everett
St Catherine’s College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy

Michaelmas 2019
Abstract

Typically in reinforcement learning, agents are trained and evaluated on the same environment. Consequently, fine-tuning to the specific nuances of the environment is encouraged for both the researchers and the agents in order to maximise evaluation performance. However, while this train/test methodology can produce impressive results, it can also lead to the risk of overfitting. In such occurrences, even minor variations in the environment can cause dramatic reductions in performance or lead to unintended behaviours emerging. Indeed, for many practical applications such as self-driving cars, healthcare, and finance, the consequences of overfitting can be catastrophic. It is therefore crucial to develop training methods which can produce robust agents that can generalise to unseen settings, as well as evaluation techniques to quantitatively analyse their capabilities.

A promising approach to help alleviate the concern of overfitting is the use of procedurally generated environments. As such environments are created automatically using algorithms rather than by hand, a single procedurally generated environment is able to provide significant variation for agents during training. Therefore, agents are encouraged to learn general strategies for solving the problem rather than overfit to a specific instance. However, despite the increasing prevalence of procedurally generated environments in the literature, their use remains relatively understudied.

This thesis helps to address this gap in the literature by presenting an investigation into the use of procedurally generated environments for training and evaluating deep reinforcement learning agents. To begin, the generalisation benefits of training agents on procedurally generated environment are demonstrated with improved average-case performance at test time. Next, the failures that can emerge from training using these environments are highlighted through a worst-case analysis of state-of-the-art agents. This is achieved through the introduction of various methods for finding adversarial examples which lead agents to catastrophic failure, with such examples shown to be simple, robust, and transferable across agents. To help alleviate these failures, an adversarial training approach is then presented which improves the performance of agents by incorporating adversarial examples into training. The thesis is concluded with the proposal of a general method for evaluating agents and influencing their emergent behaviour through modifying the environment.
Statement of Originality

I declare that no part of this thesis has been, or is being, submitted for any qualification other than the degree of Doctor of Philosophy at the University of Oxford. This thesis is entirely my own work and, except where otherwise stated, describes my own research.

Richard Everett, St Catherine’s College
For those who were lost along the way.
# Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>v</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>vii</td>
</tr>
</tbody>
</table>

## 1 Introduction

1.1 Motivation .................................................. 1
1.2 Challenges .................................................. 3
1.3 Research Questions ......................................... 4
1.4 Thesis Structure ........................................... 5

## 2 Literature Review

2.1 Procedural Content Generation ............................. 7
2.2 Procedurally Generated Environments for Reinforcement Learning ...... 8
2.3 Overfitting and Generalisation in Deep Reinforcement Learning ...... 12
2.4 Adversarial Examples in Machine Learning .................. 14
   2.4.1 Supervised Learning .................................... 14
   2.4.2 Reinforcement Learning ................................. 15
   2.4.3 Safe Reinforcement Learning ........................... 16
2.5 Mechanism Design ........................................... 17
2.6 Summary ...................................................... 18

## 3 Background

3.1 Reinforcement Learning ...................................... 19
3.2 Generative Modelling ........................................ 20
3.3 Optimisation Methods ........................................ 21
3.4 Summary ...................................................... 21

## 4 Procedurally Generated Environments

4.1 Environment Settings ......................................... 22
4.2 Navigation .................................................... 23
   4.2.1 Gridworld Maze Navigation ............................ 23
5 Quantifying Generalisation in Reinforcement Learning

5.1 Introduction ................................................. 33
5.2 Generalisation Across Navigation Tasks .................. 33
  5.2.1 Setting ................................................ 33
  5.2.2 Experimental Setup .................................. 34
  5.2.3 Generalisation Results ............................... 35
5.3 Generalisation Across Video Game Levels ................ 37
  5.3.1 Setting ................................................ 37
  5.3.2 Experimental Setup .................................. 38
  5.3.3 Generalisation Results ............................... 38
  5.3.4 Comparison with Curriculum-Based Training ......... 39
5.4 Summary .................................................... 41

6 Worst-Case Analysis of State-of-the-Art Agents .......... 42

6.1 Introduction ................................................ 43
6.2 Setting ...................................................... 44
  6.2.1 Environment ........................................ 44
  6.2.2 Agents ............................................... 45
6.3 Adversarial Environment Settings ......................... 46
  6.3.1 Definition .......................................... 47
  6.3.2 Search Algorithm .................................... 47
6.4 Worst-Case Analysis ....................................... 49
  6.4.1 Mazes Leading to Catastrophic Failure ............... 49
  6.4.2 Simple Mazes Leading to Catastrophic Failure ...... 52
  6.4.3 Transfer of Failures Across Agents ................ 59
6.5 Summary .................................................... 62
9 Conclusion and Future Work

9.1 Summary of Contributions ........................................ 105

9.2 Future Work ......................................................... 108
  9.2.1 Training Robust Agents ....................................... 108
  9.2.2 Open-Ended Environments ................................... 109
  9.2.3 Generating Effective Learning Environments ................. 110

References ............................................................. 111
## List of Figures

4.1  Gridworld Navigation environment ........................................... 23
4.2  3D World Navigation environment ............................................ 24
4.3  Example mazes for the Maze Navigation environments .................... 25
4.4  Frogs environment ............................................................... 26
4.5  Example levels for the Frogs environment ................................... 27
4.6  Particle Racing environment ..................................................... 28
4.7  Resource Harvest environment .................................................. 30
4.8  Resource Harvest environment observations ................................... 30
4.9  Example procedural generation pipeline for Resource Harvest ............ 31
5.1  Navigation tasks in the Maze Navigation environments (summarised) .... 34
5.2  Video game levels in the Frogs environment (summarised) ................. 37
6.1  Navigation task in the 3D World Navigation environment (summarised) .. 44
6.2  Example of the adversarial search algorithm .................................. 48
6.3  Adversarial optimisation curves .................................................. 51
6.4  Example mazes and agent trajectories leading to low scores ............... 52
6.5  Example simplified mazes .......................................................... 54
6.6  Example simplified maze with agent and human trajectories ............... 55
6.7  Human trajectories on simplified adversarial mazes ......................... 56
6.8  Example adversarial mazes whose robustness is analysed .................. 57
6.9  Robustness to variations in the agent’s starting location ................... 58
6.10 Robustness to variations in the goal’s location ............................... 58
6.11 Transfer of catastrophic failures across agents and architectures ........ 61
6.12 Full pairwise transfer results ..................................................... 61
7.1  Navigation task in the Gridworld Navigation environment (summarised) .. 65
7.2  Distribution of maze features ..................................................... 70
7.3  Predicted mazes from the handcrafted discriminator .......................... 70
7.4  Predicted mazes from the trained discriminator ................................ 71
7.5  Example of the modified local search procedure ................................ 73
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.6 Robustness results against local search</td>
<td>81</td>
</tr>
<tr>
<td>8.1 Driving task from the Particle Racing environment (summarised)</td>
<td>86</td>
</tr>
<tr>
<td>8.2 Social dilemma from the Resource Harvest environment (summarised)</td>
<td>87</td>
</tr>
<tr>
<td>8.3 Overview of the World Agent</td>
<td>89</td>
</tr>
<tr>
<td>8.4 Example training and use of the VAE</td>
<td>91</td>
</tr>
<tr>
<td>8.5 Agent performance on randomly sampled worlds</td>
<td>94</td>
</tr>
<tr>
<td>8.6 Evaluation results on Particle Racing</td>
<td>96</td>
</tr>
<tr>
<td>8.7 Evaluation results on Resource Harvest</td>
<td>97</td>
</tr>
<tr>
<td>8.8 Optimisation Results</td>
<td>98</td>
</tr>
<tr>
<td>8.9 Influenced training results on Particle Racing</td>
<td>101</td>
</tr>
<tr>
<td>8.10 Influenced training results on Resource Harvest</td>
<td>103</td>
</tr>
</tbody>
</table>
## List of Tables

5.1 Generalisation results on the Gridworld Navigation environment . . . 35  
5.2 Generalisation results on the 3D World Navigation environment . . . 36  
5.3 Generalisation results on the Frogs environment . . . . . . . . . . . . 39  
5.4 Results of varying training methods on the Frogs environment . . . . 40  

6.1 Human seconds-to-first-goal on agent adversarial mazes . . . . . . . 55  

7.1 Robustness results against random sampling . . . . . . . . . . . . . . 78  
7.2 Robustness results against rejection sampling . . . . . . . . . . . . . 79
Chapter 1

Introduction

1.1 Motivation

In the last few years, deep reinforcement learning has had a string of notable successes across a variety of domains. By achieving human-level performance on both single-agent tasks such as Atari [1, 2] and Labyrinth [3, 4], as well as multi-agent tasks such as Go [5, 6], Starcraft [7], and social dilemmas [8, 9], it has demonstrated itself as a powerful technique for tackling complex problems.

For many of these success stories, and more generally in reinforcement learning, agents are typically trained and evaluated on the same environment. Consequently, fine-tuning to the specific nuances of the training environment is encouraged for both the researchers and the agents in order to maximise evaluation performance [10, 11]. However, while this train/test methodology can produce impressive results, it can also lead to the risk of overfitting. In such occurrences, even minor variations in the environment from what the agent saw in training can cause a dramatic reduction in performance or lead to unintended behaviours emerging [12–15].

As deep reinforcement learning is applied to complex, safety-critical domains (such as self-driving cars, healthcare, and finance), the consequences of overfitting can be catastrophic. It is therefore crucial to develop training methods which can produce agents that are robust and able to generalise to unseen settings, as well as evaluation techniques to quantitatively analyse and understand their capabilities.
To help alleviate the concern of overfitting, a number of methods have been proposed which depart from training and evaluating agents on the same environment. For example, some success has been achieved through domain randomisation during training [16–18], injecting stochasticity into the environment during the evaluation process [13, 19], and using a train/test split of environments [12, 15].

One promising approach which is gaining traction in the literature is the use of procedurally generated environments. Here, rather than handcrafting a specific instance of the environment that is held fixed every episode, the environment is instead parametrised and then procedurally generated using algorithms with little to no human input required. Consequently, a single procedurally generated environment is able to provide significant variation for the agent during training, and therefore the agent is encouraged to learn a general strategy for solving the problem rather than overfitting to a specific instance [20, 21].

However, despite the increasing prevalence of training and evaluating deep reinforcement learning agents on procedurally generated environments, the approach is still relatively understudied. Notably, a majority of the existing work has been limited to training agents on randomly generated environments and then evaluating their average performance on hold-outs from the same distribution.

To help address this gap in the literature, this thesis presents an investigation into the use of procedurally generated environments for training and evaluating deep reinforcement learning agents. It is the hope that through this work, the understanding of the benefits, challenges, and opportunities of procedurally generated environments is improved and their use for training agents is encouraged.

The remainder of this chapter is structured as follows. First, Section 1.2 highlights the challenges and opportunities of using procedurally generated environments. Next, the research questions around which the thesis is structured are introduced in Section 1.3. To conclude, Section 1.4 outlines the structure of the remaining chapters.
1.2 Challenges

There are two notable challenges which arise when training and evaluating agents on procedurally generated environments. In this section, these challenges are highlighted alongside the research opportunities that they present which are subsequently investigated throughout this thesis.

Challenge 1: Increased environment variation. The first challenge of using procedurally generated environments is that the significantly increased environment variation makes evaluating and understanding agent performance more difficult. For example, a simple $9 \times 9$ maze generator can produce $10^7$ unique variations [22], while a more sophisticated video game level generator can produce over $10^{11}$ variations [14]. As a result, it quickly becomes infeasible to train and evaluate on every variation.

In practice, agents are trained and evaluated using a sampled subset of variations. However, while sampling randomly can achieve strong average-case performance, there are many use-cases where performing well on average is insufficient. Notably, for safety-critical applications of reinforcement learning where failures can have catastrophic consequences (such as autonomous cars crashing), an understanding of the worst-case performance of agents is necessary. Furthermore, as such failures are typically extremely rare, they can be missed by random sampling.

To address this challenge, new analysis techniques are required that can efficiently find and generate environment variations which produce the desired agent behaviours.

Challenge 2: Unknown distribution of environment variation. The second challenge is that while overfitting to specific environment instances can be alleviated, agents will still fit to the distribution of variation generated by the procedural generator. Consequently, that distribution (and any biases it may contain) will have significant implications for the training and evaluation of agents.

As an example, consider a procedural generator that is heavily weighted towards producing similar variations of an environment with simple solutions (such as a maze
where the goal is always close to the agent). In this scenario, the agent’s learned strategy would be overly simplistic due to the redundant variation, however during evaluation the agent would perform strongly (even on held-out variations). Without investigating further, this can lead to overconfidence in the performance of the agents.

In other situations, the distribution’s influence on emergent agent behaviour may be more nuanced. For example, agents collecting resources can learn significantly different strategies depending on the amount typically available during training – from peacefully cooperating to aggressively competing depending on whether the resources are abundant or scarce respectively.

To address this challenge, new methods are required that can modify the distribution of environment variation which agents see during training and evaluation.

1.3 Research Questions

The objective of this thesis is to investigate the use of procedurally generated environments for training and evaluating deep reinforcement learning agents. To structure this, the following four research questions are investigated:

1. How does training on a procedurally generated environment affect an agent’s performance on unseen settings at test time?

2. Where do agents trained on procedurally generated environments fail to generalise?

3. How can adversarial examples be incorporated into training to improve the performance of agents?

4. Which emergent agent behaviours can be evaluated and influenced through variations in the environment?

By addressing these questions, this thesis will demonstrate the benefits of procedural generation, highlight and alleviate the failures that can emerge from their use, as well as introduce future research directions. Additionally, the challenges and motivations from the previous section will be studied and discussed throughout.
Concretely, the thesis of this work can be summarised as follows: while the use of procedural generation can significantly improve the generalisation of agents, such methods have important implications for the emergent behaviours which agents learn.

1.4 Thesis Structure

The remaining chapters of this thesis are structured as follows:

- **Chapter 2** reviews the related literature on procedural generation, deep reinforcement learning, and adversarial examples which this thesis builds upon.

- **Chapter 3** summarises the relevant background material and notation used throughout the thesis.

- **Chapter 4** introduces the definition of *environment settings* as well as each of the procedurally generated environments which are used throughout the thesis.

- **Chapter 5** quantifies how agent performance on unseen environment settings at test-time is affected when trained on procedurally generated environments. This addresses Research Question 1, namely how training agents on procedurally generated environments affects their ability to generalise.

  The technical content of this chapter is based on foundational work to Chapters 6 and 7, as well as unpublished results.

- **Chapter 6** introduces the problem of *adversarial environment settings* which lead agents to catastrophic failure, and presents a worst-case analysis of state-of-the-art reinforcement learning agents. This addresses Research Question 2, understanding where agents trained on procedurally generated environments fail to generalise.

  The technical content of this chapter is based on the following work co-led by the thesis author: “Uncovering Surprising Behaviors in Reinforcement Learning via Worst-case Analysis”, Avraham Ruderman, Richard Everett, Bristy Sikder, Hubert Soyer, Jonathan Uesato, Ananya Kumar, Charlie Beattie, Pushmeet

- **Chapter 7** presents an adversarial training approach for improving the robustness of agents to adversarial environment settings, reducing their failure rate on in-distribution and out-of-distribution settings. This addresses Research Question 3 on how adversarial examples can be incorporated into training to improve the performance of agents.

The technical content of this chapter is based on the following work by the thesis author: “Towards Improved Agent Robustness Against Adversarial Environments”, Richard Everett, in the *Safe Machine Learning: Specification, Robustness and Assurance Workshop at the International Conference on Learning Representations (ICLR)*, 2019 [22].

- **Chapter 8** proposes a general method for encoding and searching for environment settings based on agent behaviour, and demonstrates how it can be used to train and evaluate agents. This addresses Research Question 4 on evaluating and influencing emergent agent behaviour through varying the environment.

The technical content of this chapter is based on the following work led by the thesis author: “Optimising Worlds to Evaluate and Influence Reinforcement Learning Agents”, Richard Everett, Adam Cobb, Andrew Markham, Stephen Roberts, in the *Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2019 [24].

- **Chapter 9** summarises the thesis contributions and outlines several promising directions for future work.
Chapter 2

Literature Review

This chapter summarises the existing literature related to the application and study of deep reinforcement learning in procedurally generated environments, contrasting the approaches introduced in the thesis with related work.

2.1 Procedural Content Generation

Throughout this thesis, agents are trained and evaluated in procedurally generated environments. Importantly, such environments are created automatically through the use of algorithms, leading to a wide range of environment variation with little to no human input required. Known as procedural content generation (PCG), this technique has a rich history of applications across domains.

The most well-known and related application of PCG is that of generating content for games [25], with the intention of keeping the player interested through automatically creating variation. For example, it has been used to create storylines, levels, weapons [26], and even game rules [27]. In addition, such methods have also been extended to adapt based on the player’s behaviour and experiences [28].

A classic example of procedural content generation in games, and specifically in regards to level generation and AI research, is the work done on the game Super Mario Bros [29]: a classic 2D platform game which has also become a popular testbed for procedural content generation research over the last two decades [30, 31]. More
generally, both Rogue-like [32, 33] and text adventure games [34] also make extensive
use of procedural generation with the explicit target audience of human players.

Importantly, the focus of this thesis is not on developing new methods for pro-
cedurally generating content for video games played by humans. Instead, the target
audience is the reinforcement learning agents which are being trained and evaluated.
As a result, this narrows down the scope and applicability of the existing literature –
namely, to works which explore the creation of environments which are interesting
and useful to learning agents with limited capabilities and long training times.

One such line of related work is that based in the growing field of procedural
content generation via machine learning (PCGML): “the generation of game content
using machine learning models trained on existing content” [35]. For example, [36]
trained a generative adversarial network (GAN [37]) on handcrafted levels for the
video game Super Mario Bros and then used this model to generate new levels using
latent variable evolution [38]. Notably, the work behind Chapter 8 (i.e. [24]) was
conducted concurrently to this and similarly uses a deep generative model to encode
the environment, however there are three key differences that are worth highlighting.
In particular, here the training levels are procedurally generated rather than hand-
crafted, the agents are trained from visual inputs using reinforcement learning rather
than search based, and also the environment is populated by multiple agents.

Another example is that of [39] whereby reinforcement learning was used to gener-
ate $5 \times 5$ gridworld mazes in tandem with rule-based and learning navigation agents.
Related to this, Chapters 6 and 7 have a similar objective of generating challenging
mazes, however they achieve this through search-based methods at a larger scale.

2.2 Procedurally Generated Environments for
Reinforcement Learning

There are many examples of procedurally generated environments being used in re-
forcement learning, with this number growing each year. In this section, the more
well known environments are summarised in chronological order.
Mazes (2015-2019). One of the most common types of procedurally generated environments with numerous implementations available [13, 15, 40, 41], mazes have been a staple in reinforcement learning research for years. Typically represented in a gridworld with first-person observations, extensions have included adding rooms and various objects (such as keys and traps) to the environment in order to increase its complexity. Due to their prevalence in the literature and intuitiveness, Chapters 5 through 7 all investigate procedurally generated mazes.

OpenAI Gym (2016). Introduced in 2016, OpenAI Gym [42] provides a common interface to a range of diverse tasks, with a notable couple using procedural generation. In particular, the car racing environment utilises a procedurally generated track for the agent to drive around, while the bipedal walking environment tasks the agent with manoeuvring along procedurally generated terrain. Building on the bipedal walking environment, recent work has explicitly parametrised the terrain generator in order to simultaneously create and solve increasingly sophisticated instances [43].

DeepMind Lab (2016). Also introduced in 2016, DeepMind Lab [44] provides a suite of challenging 3D navigation and puzzle-solving tasks for learning agents. In many of these tasks, the environment is procedurally generated, and has consequently been used to investigate a variety of topics. For example, memory [4], navigation [45], and multi-task learning [46]. Similar to the Maze environments discussed before, the navigation tasks from DMLab are used in Chapters 5 and 6 to investigate generalisation and worst-case performance respectively due to their prevalence in the literature.

Sokoban (2017). The puzzle game Sokoban is a planning problem where an agent has to push boxes onto target locations without making mistakes that can render the puzzle unsolvable. To demonstrate an agent’s ability to plan using this environment, procedural generation has been used generate new puzzles every episode and therefore ensure that “the agent cannot memorize [the solution to] specific puzzles” [20].
GvG-AI (2018). Building on the General Video Game AI framework (GVG-AI), [14] designed parameterised level generators for four games: Boulderdash, Frogs, SolarFox, and Zelda. Notably, the generators were designed after an analysis of the core components of human-designed levels for each game and included a controllable difficulty parameter. The level generators were then used to investigate generalisation by training and testing agents on various difficulty-generated and human-designed distributions.

CTF (2018). Based in the same engine as the DeepMind Lab tasks, the Capture the Flag (CTF) environment [21] features procedurally generated arenas where multiple agents compete. By co-training two versus two agents within a procedurally generated environment, the trained agents were able to generalise to previously unseen arenas and also outperform humans in direct competition. Notably, this is one of the few examples of related work which feature multiple agents in the environment.

TORCS (2018). The Open Racing Car Simulator (TORCS) [47] is an open-source 3D car racing simulator which has been used in AI research for several years. Recently, [48] used the environment to investigate adversarial evaluation of trained agents. To do this, they extended the environment to support procedurally generated tracks, with each track parameterised by a 12-dimensional vector encoding the locations and curvatures of its waypoints. Importantly, as with the GVG-AI level generators mentioned previously, this is one of the few environments where the procedural generator is explicitly parametrised and therefore controllable.

CoinRun (2018). Inspired by platform games such as Sonic, the CoinRun environment [15] has been proposed to evaluate the generalisation performance of trained agents. In this environment, the agent is tasked with collecting a coin which is placed at the end of a procedurally generated level. The level itself is generated using a difficulty parameter which controls various aspects of the level such as the frequency of obstacles and the number of sections (similar to the GVG-AI procedural generators).
Interestingly, the environment was designed to be tractable for existing algorithms, with its simplicity being a highlighted feature.

**Obstacle Tower (2019).** Similar to CoinRun, the Obstacle Tower environment [49] has also been proposed as a challenging benchmark for evaluating generalisation. In addition to being procedurally generated, it features a 3D world, high fidelity visuals, and physics-driven interactions. The agent is tasked with solving a number of floors, with the layout of each floor procedurally generated. Specifically, a mission graph is generated which describes how to solve the level, and a corresponding layout map is generated which physically represents the mission graph in a 2D space.

**Neural MMO (2019).** Designed to support a large number of agents competing in a persistent world, the Neural MMO environment [50] tasks agents with staying alive as long as possible. The layout of the world itself is based on a procedurally generated tile-based map, with each tile corresponding to an entity such as water or food which agents must interact with in order to survive. Of note here is that this is one of the few procedurally generated environments which contain multiple agents, with the other exceptions being the Capture the Flag variant of DeepMind Lab [21] and the resource harvest environment introduced for Chapter 8.

**ProcGen Benchmark (2019).** Introduced by OpenAI at the end of the year, [51] released a suite of 16 procedurally generated environments and associated single-agent performance results. Importantly, this was the first public release of a large number of procedurally generated environments which were designed for deep reinforcement learning agents. To aid their adoption in the research community, results were also reported on a limited subset of training settings and were therefore easily reproducible by researchers without requiring unreasonable resources.

To summarise, while a diverse range of procedurally generated environments have been used in the literature, the extent of their use has been limited. Specifically, beyond a couple of notable exceptions which adjust the parameters of the generation
2.3 Overfitting and Generalisation in Deep Reinforcement Learning

Throughout 2018, there were a number of concurrent works which investigated overfitting and generalisation in deep reinforcement learning. In this section, these works and their methodologies are summarised, and the related findings of Chapter 5 are put into context.

To begin, [12] proposed measuring the generalisation performance of agents by training and testing them on distinct sets of video game levels. Specifically, a limited number of hand-designed levels from the Sonic the Hedgehog series of games were used to form a training set, with agents then evaluated on a test set of custom levels which were not seen during training. In addition, agents were allowed only 1 million steps at test time to adapt, further addressing the issues typically associated with testing on the training set.

Next, [13] examined how agents can overfit on procedurally generated mazes in a gridworld. Notably, they found that agents have a high capacity to memorise a large number of random mazes seen during training, and therefore recommended isolating the training and test sets to help avoid overfitting. They then further showed that increasing the number of unique mazes in the training set can lead to greater generalisation on the same gridworld environment.

In a similar vein, [14] followed this by exploring how training agents on procedurally generated levels can increase generality. By building on the General Video Game AI framework (GVG-AI) [52] to procedurally generate video game levels, they demonstrated that agents trained on few levels often fail to generalise to new levels.
Interestingly, they then highlight the challenges of ensuring that the training distribution resembles the test distribution, evaluating procedurally trained agents on both generated levels of varying difficulties as well as human-designed levels.

Following this, [15] investigated the problem of overfitting by introducing the procedurally generated CoinRun environment as a benchmark for generalisation in deep reinforcement learning. Through the use of a similar methodology to [13], they quantified the generalisation performance of an agent as the size of its training set was increased and investigated various approaches for improving generalisation. For example, deeper convolutional architectures [3] and methods from supervised learning such as data augmentation [53] and batch normalisation [54].

In the context of these concurrent works, Chapter 5 follows the same methodology of training and testing agents on procedurally generated environments, with agent performance then quantified as the size of the training set is varied. The results presented in the chapter further support the mutual finding that procedural generation can help improve generalisation. Notably, Chapter 5 achieves this by uniquely considering navigation tasks in two contrastingly complex environments (i.e. a simple two-dimensional gridworld and a high-fidelity three-dimensional world).

Recently, [49] proposed a new benchmark known as Obstacle Tower – a 3D third-person procedurally generated environment designed to test generalisation and benchmarked against human performance. Of note here is that while they initially recommended evaluating both weak and strong generalisation (referred to as in-distribution and out-of-distribution examples in this thesis), they subsequently found that “agents performed catastrophically in these [strong] cases” and opted to only test for weak generalisation in the challenge. Related to this, Chapter 5 investigates in-distribution examples, while Chapter 6 and 7 also investigate the more challenging out-of-distribution examples.
2.4 Adversarial Examples in Machine Learning

2.4.1 Supervised Learning

It has recently been shown that deep neural networks are vulnerable to well-designed input samples which fool them into producing incorrect outputs [55]. Known as adversarial examples, they highlight the fact that machine learning systems which perform very well on average may nonetheless perform extremely poorly on particular adversarial inputs, and often do so in surprising ways [56, 57].

While the setting considered throughout this thesis is similar on a conceptual level to adversarial examples in supervised learning, there are two key differences worth highlighting from this existing line of work. First, the attack vector used in this thesis consists of changing the structure of the environment (for example, the walls of a maze), rather than changing individual pixels in an input image in an unconstrained manner. Second, the resulting failure is realised over multiple steps of the agent and environment interacting with each other rather than simply being errant output from a single forward pass through a neural network.

Together, these differences raise several unique challenges which are not typically seen in the supervised learning literature. For example, finding and generating adversarial examples becomes significantly more computationally challenging due to the evaluation of the agent/environment, and therefore efficient methods are required to do so. Furthermore, any generated examples need to conform to the environment’s specification, being both valid and still solvable.

In addition to these differences, there are also a couple of parallels which are worth noting. First, similar approaches to the simplification method introduced in Chapter 6 have been explored in the context of interpretable adversarial examples in image classification. Specifically, searching for adversarial perturbations with group-sparse structure or other minimal structure [58, 59]. Second, it has been shown that image perturbations which are adversarial for one network often transfer across other networks [56, 60]. This is consistent with the findings presented in Chapter 6 where it is shown that adversarial examples can transfer between agents.
2.4.2 Reinforcement Learning

Following the success of adversarial examples in supervised learning, there is an increasingly rich literature on adversarial examples in reinforcement learning which cause agents to perform unexpectedly, with various attack vectors considered.

**Agent Inputs** The first attack vector is through perturbing the agents’ observations and naturally follows much of the literature on adversarial examples in supervised learning [61–65]. However, while this attack vector can be interesting from a security perspective, and also allows crossover with the adversarial example literature in supervised learning, it assumes direct access to the agent’s input channel. Importantly, this is a strong assumption and is often infeasible in many scenarios (such as most real-world applications). In comparison to the work presented throughout this thesis, no assumptions are made about the agent, with adversarial examples instead being considered from the structure of the environment itself.

**System Dynamics** The second attack vector is through perturbing the system dynamics of the environment (for example the wind conditions in a simulator [10]), and is a common method in the training of agents for physical settings.

In particular, because of the cost and risk of running experiments in the real-world, agents are typically trained in simulators. Due to this, it is possible to manipulate the dynamics of the system which are randomly determined by the environment in a physical setting but controllable in a simulator.

Taking advantage of this capability, it has been shown that training agents in the presence of adversarial perturbations leads to more reliable agents. Specifically, by applying adversarially constructed noise to the system dynamics of the environment, the failure rate of agents can be reduced [66, 67]. This line of work is complementary to that presented in Chapter 7 which seeks to improve the robustness of agents by training on adversarial environments. However, there are several notable challenges in the setting considered in this thesis. For example, higher dimensionality, continuous
and discrete variables, the cost of finding and generating adversarial examples, as well as both in- and out-of distribution adversarial examples.

**Environment Structure** The third attack vector is through modifying the spatial structure of the environment (for example the layout of a maze so that the agent fails to find the goal), and is the focus of this thesis.

The issue of adversarial (procedurally generated) environments which lead agents to failure was first highlighted collectively by [23] and [48] (with the former being the work behind Chapter 6), however the topic remains relatively underexplored. In the case of the latter, they went on the demonstrate the limitations of random sampling as a means of evaluating agents in two procedurally generated environments (humanoid control [68] and simulated driving [47]). To address these limitations, an adversarial evaluation approach was proposed which uncovered catastrophic failures more efficiently by reusing agent training data, with a demonstrated order of magnitude improvement in sample efficiency.

In comparison to the work presented in this thesis, there are three key differences. First, they only considered adversarial examples from within the distribution that agents are trained on. In contrast, Chapter 7 explicitly investigates adversarial examples both from inside and outside of the distribution that the agents are trained on. Next, agents were only evaluated in procedurally generated environments whereas both Chapters 7 and 8 utilise evaluation data to inform training and improve the performance of agents. Finally, only failure cases were investigated while Chapter 8 goes on to evaluate and influence other emergent behaviours such as equality and conflict in a multi-agent setting.

### 2.4.3 Safe Reinforcement Learning

Safe reinforcement learning is concerned with how agents can be trained and deployed while avoiding risky states [69]. In contrast, this thesis is concerned with sampling these states more during training and evaluation in order to improve agent performance or change agent behaviour at test (deployment) time.
There are two safety problems of interest in the literature which are related to this thesis. First, there is robustness to adversaries which examines how an agent adapts to friendly and adversarial intentions present in the environment [70]. With this thesis, the environment itself can be considered an agent which has intentions for the emergent behaviour of the agent.

Second, there is the problem of distributional shift [70]. In particular, how well an agent behaves when the test distribution (of environment variation) differs from the training environment. This is explored in Chapters 6 and 7 where the distribution of the training and testing environment variations are different.

2.5 Mechanism Design

Throughout this thesis, the environments which agents are trained and evaluated in are strategically modified with the objective of eliciting a desired behaviour from the agents despite their own reward functions. For example, modifying the structure of mazes so that the agent fails to find the goal and modifying the placement of resources in a gathering environment to reduce competition between agents. Due to this, a connection can be made with the literature on mechanism design [71] which looks at how the rules of a game (i.e. mechanisms) can be constructed such that desirable agent behaviours emerge despite the agents’ self-interests [72]. Specifically, the connection here is that the mechanism being modified is the spatial structure of the environment itself.

While reinforcement learning and deep learning have recently been used to scale and automate mechanism design, a majority of this work has focused on auctions [73, 74] and is therefore not directly applicable here. An exception to this is the literature on designing incentives to maximise social welfare in settings where selfish individuals are tempted to increase their own reward at the expense of others, despite mutual cooperation being better for all participants. Known as social dilemmas [75], they have traditionally been studied using human participants or constrained to matrix games [76]. As a result, scaling and automating their analysis and interaction...
with mechanism design has been challenging. For example, [77] use adaptive mechanism design for automatically learning incentives which promote cooperation between reinforcement learning agents in matrix games, leading to better social outcomes.

Recently, sequential social dilemmas [8] have been proposed which spatially and temporally extend matrix games, using multi-agent reinforcement learning to study emergent social outcomes. Building on this work, [78] examined how the spatial structure of the environment (such as the placement of resources and walls) can influence the emergent behaviour and social outcomes of a multi-agent system comprising independent reinforcement learning agents. Importantly, the environment’s spatial structure was modified by hand, and it is here where Chapter 8 makes its key contribution. Specifically, the chapter builds on this line of work by introducing a method for automatically adapting the environment in response to the behaviour of agents (as opposed to handcrafting each environment variation).

2.6 Summary

This chapter reviewed the related literature on the application and study of deep reinforcement learning in procedurally generated environments, contrasting it with the approaches introduced in the thesis. Specifically, a summary of the more well known procedurally generated environments was presented. This was followed by a discussion of the work on overfitting and generalisation in deep reinforcement learning, adversarial examples in machine learning, and mechanism design.
Chapter 3

Background

This chapter summarises the relevant background material and notation used throughout the thesis.

3.1 Reinforcement Learning

In reinforcement learning (RL) [79], an agent exists inside of an environment which it interacts with for episodes of length $T$ timesteps. At each timestep $t$, the agent with state $s_t \in S$ executes action $a_t \in A$ using policy $\pi(a|s)$. The agent then receives reward $r_t$ and transitions to state $s_{t+1} \in S$ with probability $P(s_{t+1}|s_t, a_t) = T(s_t, a_t, s_{t+1})$. Given this, the agent’s optimal policy $\pi^*$ maximises its expected discounted return $\sum_{t=1}^{T} \gamma^{t-1}r_t$, where $\gamma \in [0, 1)$ is the discount factor.

To learn the optimal policy $\pi^*$, methods such as Q-learning [80] can be used which are guaranteed to converge in the tabular (no approximation) case provided sufficient exploration. However, for applications where the state-space is continuous or high-dimensional (such as the visual observations used in this thesis), it quickly becomes infeasible to learn a separate value for every state.

With deep reinforcement learning [81], this limitation is overcome through the use of function approximators (such as neural networks) which capitalise on the assumption that similar actions should be executed in similar states. Throughout this thesis, the underlying learning algorithm used is Advantage Actor-Critic (A2C) [2] which learns a separate policy and state-action value function.
In multi-agent reinforcement learning (MARL), a set of agents share an environment and must each learn to maximise their own returns [82]. Such a setting is considered in Chapter 8 with each agent learning independently (i.e. each agent has its own separate network). Notationally, each agent $i$ has their own state $s_i^t$, executes their own action $a_i^t$ (which has a joint impact on the environment), and receives their own reward $r_i^t$ each timestep $t$, with their total episodic reward denoted as $R_i^t = \sum_{t=1}^{T} r_i^t$.

### 3.2 Generative Modelling

The aim of probabilistic generative modelling is to recover the true distribution of the input data $X$. By sampling this distribution, new data points can be generated which resemble the observed data. Here, the focus is on the class of models that build approximate data distributions which are conditioned on the latent space $z$ and a set of parameters $\theta_g$ (where the subscript ‘$g$’ is used to denote the parameters of the generative model).

With deep generative models, the parameters are the weights of deep neural networks. In Chapter 8, a Variational Auto-Encoder (VAE) [83] is used which learns an approximate distribution to the true data by maximising the evidence lower bound of the log likelihood of the data. The VAE is composed of an encoder-decoder architecture, where the encoder compresses data, $X$, into the latent space $z$, and a corresponding decoder is learnt to recover this data by transforming from the latent space back to the original data. This encoder-decoder structure is shown by the two distributions:

$$X \sim p(X \mid \theta_g, z), \quad z \sim q(z \mid \theta_e),$$

where the encoder $q(z \mid \theta_e)$ (approximate inference network) is introduced with its corresponding parameters $\theta_e$. Once the VAE has been learnt, the separate components can be used independently in order to either compress data into the latent space (using the encoder) or to generate new data samples (using the decoder).
3.3 Optimisation Methods

If an optimisation function needs to be optimised, but only the value $f(x)$ for a point $x \in \mathcal{X}$ is able to be queried, then the problem setup is known as black-box optimisation. The key challenge is that $f(x)$ is not available in a simple closed form, leading to an optimisation task:

$$x^* = \arg\max_{x \in \mathcal{X}} f(x), \quad (3.2)$$

where gradient information about the black-box objective function $f(x)$ is unavailable.

In Chapter 8, such tasks are solved using the following two methods:

**Evolution Strategies** (ES) [84] approach the problem by evaluating the fitness of a batch of solutions, after which the best solutions are kept while the others are discarded. Survivors then procreate (by slightly mutating all of their genes) in order to produce the next generation of solutions. Here, the Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [85] method is used which adaptively changes the size of the search space each generation.

**Bayesian Optimisation** (BO) [86] is another common optimisation method which approaches the problem by defining a prior distribution over the objective function (for example, a Gaussian process) and then selects new samples according to an acquisition function (where the acquisition function defines a trade-off between exploration and exploitation when querying the objective function).

3.4 Summary

This chapter presented the preliminaries and notation used throughout the thesis. In particular, reinforcement learning was introduced alongside its extensions to both deep and multi-agent reinforcement learning. This was followed by details on generative modelling and optimisation methods which are used in Chapter 8.
Chapter 4

Procedurally Generated Environments

Throughout this thesis, agents are trained and evaluated on a number of procedurally generated environments. In this chapter, the scope of the procedural generation is defined through the introduced definition of environment settings. This is followed by a detailed description of the environments which are categorised into the following domains: navigation, video games, driving, and social dilemmas.

4.1 Environment Settings

Rather than training and testing agents on the same fixed environment every episode, the environment can instead be procedurally generated with its initial state parametrised and then modified for each episode. Examples of such parameters include the environment’s spatial layout [23], physical properties [87], and agent structure [88], with a range of terms existing to refer to the resulting environment instantiation (such as instance, level, world, arena, map, and configuration).

In the context of this thesis, the procedurally generated parameters of the environment are known as environment settings and specifically refer to the spatial layout of the environment. As will be described in the rest of this chapter, this covers the structure of mazes for navigation tasks in gridworlds and 3D worlds (Section 4.2), the design of video game levels (Section 4.3), the structure of tracks for driving tasks (Section 4.4), and the arrangement of entities for social dilemmas (Section 4.5).
4.2 Navigation

The motivation behind investigating navigation in procedurally generated environments is two-fold. First, navigation is of central importance in reinforcement learning as it both captures the challenges posed by partial observability and requires the agent to “effectively perceive, interpret, and learn the world in order to make tactical and strategic decisions where to go and how to act” [89]. Second, recent advances have led to impressive human-level performance on navigation tasks in large procedurally generated environments [4], making them a natural target for further analysis.

In this section, two environments are introduced which focus on the task of navigation: Maze Navigation in (1) a Gridworld, and (2) a 3D World. Importantly, while these environments have contrasting levels of complexity, both utilise the same procedural generation process for their environment settings (i.e. maze structure).

4.2.1 Gridworld Maze Navigation

For this environment, navigation takes place in a gridworld which is based on the open-source MiniGrid environment [41]. Here, the agent is tasked with reaching the goal in a $9 \times 9$ procedurally generated maze where each position in the maze may contain a wall, an agent starting location, or a goal location. Presented in Figure 4.1a is an example of such a task with an illustrative agent trajectory which successfully reaches the goal. Alongside this, presented in Figure 4.1b, are the agent’s corresponding first-person observations while navigating along this trajectory.

![Example maze and agent trajectory.](a) Example maze and agent trajectory. ![Partially observable agent observations.](b) Partially observable agent observations.

Figure 4.1: Gridworld navigation. (a) Example procedurally generated maze along with an agent’s trajectory from its starting location (cyan triangle) to the goal (purple circle). (b) Frames from top left to bottom right corresponding to the agent’s visual observations as it moves along its trajectory towards the goal.
Within each time-limited episode, the task of the agent is to reach the goal by navigating through the maze. If the agent successfully moves onto the goal, the episode is terminated and the agent is rewarded proportional to the time taken: 

\[ \text{Reward} = 1 - 0.9 \times \frac{t_G - t^*_G}{T}, \]

where \( t_G \) is the timestep when the agent reached the goal, \( t^*_G \) is the minimal number of timesteps required to reach the goal, and \( T \) is the maximum episode length which is set to 100 timesteps. If the goal is not reached within 100 timesteps, the episode is terminated and the agent receives 0 reward.

At each timestep, the agent receives a 5 × 5 size observation in their rotatable facing direction (examples of which are provided in Figure 4.1b), along with their current orientation and \((x,y)\) coordinates. Additionally, the agent can perform one of the following actions: movement in any direction, rotate left, rotate right.

### 4.2.2 3D World Maze Navigation

For this environment, navigation takes place in a 3D world which uses DeepMind Lab (DMLab) [44]. Here, the agent is tasked with reaching the goal as many times as possible in a 15 × 15 procedurally generated maze where each position in the maze may contain a wall, an agent starting location, or a goal location. Presented in Figure 4.2a is a high-level example of such a task with an illustrative agent trajectory which successfully reaches the goal. Alongside this, presented in Figure 4.2b, are the agent’s corresponding first-person observations while navigating along this trajectory.

![Example maze and agent observations](image)

**Figure 4.2: 3D world navigation.** (a) Example procedurally generated maze along with an agent’s trajectory from its starting location (cyan, bottom left) to the goal (magenta, middle). (b) Frames from top left to bottom right corresponding to the agent’s visual observations as it moves along its trajectory towards the goal. Note that while navigation may appear simple given a top-down view, the agent only receives partial information about the maze at every step which makes navigation challenging.
Within each episode, the task of the agent is to reach the goal as many times as possible by navigating through the maze. If the agent successfully reaches the goal, it receives positive reward of 1 and its location is reset to its starting location. After 1800 steps (120 seconds at 15 frames per second), the episode is terminated with the agent’s score being the number of times it reached the goal.

At each timestep, the agent receives an RGB observation of size 96 \times 72 pixels (examples of which are provided in Figure 4.2b), and is able to perform one of the following actions: move forwards, move backwards, strafe left, strafe right, look left, look right, move forward + look left, move forward + look right.

**Environment Settings** For both environments, mazes are generated using the open-source procedural generator provided in DMLab [44]. Every maze is represented as a grid of either size 9 \times 9 (Gridworld) or 15 \times 15 (3D World), with each cell in the grid corresponding to one of the following entities which are procedurally assigned: agent starting location, goal location, wall, empty (visualised as cyan, magenta, black, and white respectively). The generation process itself has no constraints other than being solvable, and the mazes are generated by randomly placing rooms, connecting them via path, and then creating alternative paths. Presented in Figure 4.3 below are nine example procedurally generated mazes for both environments.

![Example mazes](image)

(a) Example 9x9 mazes.  
(b) Example 15x15 mazes.

Figure 4.3: **Example mazes for the Maze Navigation environments.** Nine procedurally generated mazes for the (a) 9x9 Gridworld environment, and (b) 15x15 3D World environment. Notably, both environments use the same process for procedurally generating mazes but at two different sizes.
4.3 Video Games

For years, video games have been a staple in artificial intelligence and machine learning, naturally providing a diverse range of rich and complex environments to evaluate agents. One well known example is that of the Atari games in the Arcade Learning Environment (ALE) [90] which have helped accelerate the development of reinforcement learning algorithms. In this section, the Frogs environment is described which uses procedurally generated video game levels and combines a navigation task with additional mechanics (therefore requiring fast reactions and dynamic planning).

4.3.1 Frogs

The Frogs environment is based on the video game ‘Frogger’ (Konami, 1981) and was introduced in [14]. Here, episodes take place in $28 \times 11$ procedurally generated levels where the agent is tasked with reaching the goal without getting hit by a car or drowning in the water. Presented in Figure 4.4 is an example of such a level which the agent fully observes.

![Figure 4.4: Video game level. Example procedurally generated level which the agent fully observes as an RGB observation of size 280 × 110 pixels. The agent itself is in the second last row next to three rows of road containing cars. This is followed by three rows of water containing logs, and then the target goal in the second row.](image)

Importantly, this environment was chosen due to the high performance achieved by standard reinforcement learning agents. As a result, this allows the investigation presented in this thesis to focus on the environment and training methods rather than the out-of-scope development of more sophisticated agent architectures.
Within each episode, the task of the agent is to reach the goal by moving through the level, avoiding the side-scrolling cars and using the side-scrolling logs to avoid drowning in the water. If the agent successfully reaches the goal, it receives a positive reward of 1 and the episode is terminated. However, if the the agent is hit by a car or drowns in the water, the episode is terminated early and the agent receives 0 reward.

At each timestep, the agent fully observes the environment, receiving an RGB observation of size 280 × 110 pixels. Additionally, the agent is able to move in one of the following directions: up, down, left, right.

**Environment Settings** Levels for the Frogs environment are generated using the open-source hand-designed procedural generator introduced in [14]. Every level is represented as a 28 × 11 grid where each position corresponds to one of the following entities which are procedurally assigned: agent, goal, tree, grass, road, car, water, log. The process for generating a level involves placing the goal at the top, agent at the bottom, filling intermediate rows with roads and water, and then adding cars and logs to both respectively. Notably, the generator can take a difficulty value as input which scales the level’s complexity from 0 (a small easy level with few entities) to 10 (a large hard level with many entities). Presented in Figure 4.5 are two example levels at these difficulty values.

![Example levels for the Frogs environment](image)

(a) Difficulty 0.  
(b) Difficulty 10.

Figure 4.5: **Example levels for the Frogs environment.** (a) An easy procedurally generated level at difficulty 0 containing no obstacles and easily solvable in four actions (up-up-up-up). (b) A hard procedurally generated level at difficulty 10 containing many obstacles in a large space, therefore requiring a long series of actions to solve.
4.4 Driving

A promising application for machine learning is that of self-driving cars. However, for such an application to become a reality, agents need to be robust to the vast amounts of variation and adversarial settings that are present in the real-world. To help support this research direction, this section introduces the Particle Racing environment which uses procedurally generated tracks.

4.4.1 Particle Racing

Within the Particle Racing environment, the agent is tasked with completing loops of a procedurally generated track as quickly as possible without crashing off of the track. Presented in Figure 4.6a is an example of such a task along with the agent’s visual observation in Figure 4.6b.

(a) Example track.  (b) Partially observable agent observation.

Figure 4.6: Driving task. (a) Example procedurally generated track with the blue circle representing the agent and the white dashed box indicating the agent’s visual observation window. (b) Raw and processed frames corresponding to the agent’s visual observation from (a).

Within each episode, the task of the agent is to complete as many loops of the track as possible without crashing, receiving $-0.2$ reward per step and a positive reward proportional to their speed along the track. In addition, if the agent leaves the track (i.e. crashes), it receives $-300$ reward and the episode is terminated. If no crash occurs within 300 timesteps, the episode is terminated. The agent itself occupies approximately a fifth of the track’s width, and therefore it has to carefully manage its speed in order to not crash off of the track.
At each timestep, the agent observes its position, directional speed, and surrounding area (an example of which is given in Figure 4.6b). Importantly, the agent’s view of its surrounding area is always centred on the agent and does not rotate. In addition, the front of the agent is based on its direction of movement (in contrast to a car for example which has a physically defined front). Furthermore, the agent is able to apply a unit of force in one of four directions, increasing its velocity in that direction which is then gradually decreased over time due to friction from the track. Together, these design decisions helped to create a task which is solvable by standard deep reinforcement learning agents, and therefore keep the focus of the investigation on manipulating the environment rather than changing the agent’s architecture.

**Environment Settings**  Tracks for the Particle Racing environment are generated using the open-source procedural generator introduced as part of the OpenAI Gym Car Racing environment [42].

### 4.5 Social Dilemmas

Inspired by laboratory experiments from behavioural game theory, sequential social dilemmas [8] make it possible to study the emergent behaviour of groups in a spatial and temporal setting using reinforcement learning. However, despite their increased study in the literature, most work has been restricted to a single fixed environment [9, 91, 92]. As a result, little has been done to investigate how the environment itself can be changed to influence the emergent behaviour of groups. To address this, this section introduces the Resource Harvest environment.

#### 4.5.1 Resource Harvest

Social dilemmas take place in the multi-agent Resource Harvest environment which is based on related work [78]. Here, agents are tasked with harvesting resources in the presence of other self-interested agents in a $15 \times 15$ procedurally generated level where each position in the level may contain a resource or a wall. Presented in Figure 4.7a
is an example of such a level. Alongside this, presented in Figure 4.7b, are each of the agents’ first-person observations.

Figure 4.7: Social dilemma. (a) Example procedurally generated level containing walls (grey), resources (green), and agents (red & blue). (b) Frames corresponding to each of the agents’ forward-facing visual observations from (a).

Within an episode, the task of each self-interested agent is to collect as many resources as possible, receiving +1 reward for each one collected. Consequently, agents are motivated to harvest the resources as fast as possible before the other agents are able to do so. However, resources recover based on the amount of nearby resources, and therefore leaving several uncollected leads to faster recovery and more to collect in the long run. Therefore, there exists a social dilemma for the agents – collect all of the resources quickly before the other agents do so, or leave the resources to recover in order to sustainably collect them over the rest of the episode.

At each timestep, every agent receives a $7 \times 7$ size forward-facing observation based on their orientation (examples of which are provided in Figure 4.7b above and Figure 4.8 below), and can perform one of the following actions: stand still, move forwards, move backwards, strafe left, strafe right, rotate left, and rotate right.

Figure 4.8: Rotatable agent observations. Example forward-facing observations received by agents when oriented (a) north, (b) east, and (c) south.
**Environment Settings** Levels for the Resource Harvest environment are generated using the procedural generator introduced in [24]. Every level is represented as $15 \times 15$ grid where each position corresponds to one of the following entities which are procedurally assigned: wall, resource, empty. The process for generating a level is visualised and described in Figure 4.9.

![Procedural Generation Pipeline](image)

**Figure 4.9:** Example procedural generation pipeline for Resource Harvest. Levels are procedurally generated for the Resource Harvest environment using the following four-step pipeline: (a) create 0 to 4 rooms, (b) merge overlapping rooms, (c) create entrances, (d) add resources.

### 4.6 Summary

In this chapter, the scope of procedural generation for the thesis was defined through the introduced definition of *environment settings*. Each procedurally generated environment was then presented with an introductory motivation, description of the environment’s mechanics, and details on the environment settings which were generated. In the following chapters, these environments will be used to investigate generalisation, worst-case performance, robustness, and emergent behaviours.
Chapter 5

Quantifying Generalisation in Reinforcement Learning

In this chapter, the following question is addressed:

How does training on a procedurally generated environment affect an agent’s performance on unseen settings at test time?

Before looking at the core technical contributions of this thesis, it is first important to demonstrate the benefits that training agents on procedurally generated environments can bring. To this end, this chapter investigates how the test-time performance of agents is affected by the number of unique examples seen during training.

Contributions To summarise, the contributions of this chapter are as follows:

1. Training on a procedurally generated environment is empirically shown to improve an agent’s performance at test time, with only a fraction of the possible environment settings required to achieve strong performance (Section 5.2).

2. This finding is demonstrated to hold across varying levels of environmental complexity including longer episode lengths, higher resolution observations, and an increased number of possible actions (Section 5.2.3.2 and 5.3).

3. Random sampling is shown to be a simple yet effective method for achieving strong average-case performance on unseen settings (Section 5.3.4).
5.1 Introduction

As described in Chapter 1, a desirable property of agents is that they are able to generalise and therefore perform well in previously unseen situations. A common methodology for evaluating such generalisation is to train and test on distinct sets. In the context of this thesis, this means training on one set of randomly sampled environment settings and then evaluating on a non-overlapping held-out set.

In this chapter, the generalisation performance of agents is quantified on three environments. By using the methodology described above, it is shown that agents trained on procedurally generated environments are able to generalise well. This is first demonstrated on navigation tasks in a gridworld and then followed by similar tasks in a more complex 3D world, therefore showing that this finding holds across varying levels of environmental complexity. The same methodology is then applied to levels of a video game with similar results. To conclude, training on randomly sampled settings is compared to an existing curriculum-based training approach.

5.2 Generalisation Across Navigation Tasks

When training agents to navigate, a desirable outcome is that the agents learn a general navigation strategy which can be used to solve unseen situations (as opposed to simply memorising exact solutions). To quantify this notion of generalisation across navigation tasks, this section investigates how agent test time performance is affected by the size of its training set using two contrastingly complex environments.

5.2.1 Setting

In this section, both of the environments are summarised alongside the corresponding agents which are trained on each. Notably, while the navigation tasks are structured the same for both environments, the complexity of the environments is markedly different. As a result, it is possible to investigate whether any resulting generalisation findings are affected by changes in the length of an episode, the resolution of first-person observations, and the size of the agent’s action set.
5.2.1.1 Gridworld Maze Navigation

Visualised in Figure 5.1a, the Gridworld Maze Navigation environment tasks agents with reaching the goal in a procedurally generated maze. The environment itself is a 9 × 9 gridworld with observations of size 5 × 5 where episodes last at most 100 steps. The agent uses the Advantage Actor-Critic (A2C) [2] algorithm with full details available in Section 7.2.2.

5.2.1.2 3D World Maze Navigation

Visualised in Figure 5.1b, the 3D World Maze Navigation environment also tasks agents with reaching the goal in a procedurally generated maze. Notably, while the mazes are 15 × 15 in size, the environment itself is three-dimensional with observations of size 96 × 72 and episodes of length 1800 steps (2 minutes). The agent is based on the Importance Weighted Actor-Learner Architecture [3] with full details available in Section 6.2.2.1

5.2.2 Experimental Setup

To investigate the ability for agents to generalise across navigation tasks, an agent is independently trained on sets of different sizes and then evaluated on a separate set of test mazes (where all sets are unique and non-overlapping). Specifically, the training set is varied in size from containing 1 to 100,000 mazes which are all randomly sampled from the procedural generator. A new agent is then trained on each until
convergence, and the resulting trained agents are then evaluated on the same set of 50,000 test mazes to quantify their performance.

5.2.3 Generalisation Results

In this section, the train and test performance results are presented for both the Gridworld and 3D World Maze Navigation environments.

5.2.3.1 Gridworld Maze Navigation

Presented in Table 5.1 are the results of this experiment on the Gridworld environment. In particular, the average reward achieved by the agent (Reward), the percentage of mazes solved (% Solved), and the average time it took to solve each maze (Steps) are quantified on both the train and test set as the size of the training set (Number of Training Mazes) is increased from 1 to 100,000 unique samples.

Table 5.1: Gridworld generalisation results. Quantifying the navigation agent’s ability to generalise to unseen mazes in the Gridworld environment as the size of the training set is increased from 1 to 100,000 mazes.

<table>
<thead>
<tr>
<th>Number of Training Mazes</th>
<th>Train Performance</th>
<th>Test Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reward</td>
<td>% Solved</td>
</tr>
<tr>
<td>1</td>
<td>0.98</td>
<td>100.0</td>
</tr>
<tr>
<td>10</td>
<td>0.97</td>
<td>100.0</td>
</tr>
<tr>
<td>100</td>
<td>0.96</td>
<td>100.0</td>
</tr>
<tr>
<td>1,000</td>
<td>0.93</td>
<td>99.5</td>
</tr>
<tr>
<td>10,000</td>
<td>0.92</td>
<td>99.4</td>
</tr>
<tr>
<td>100,000</td>
<td>0.92</td>
<td>99.3</td>
</tr>
</tbody>
</table>

As can be seen, the agent’s ability to generalise improves drastically as the number of unique training mazes (i.e. samples) is increased. Specifically, the agent’s test performance increases from 0.31 reward and 37.2% solved with 1 sample to 0.91 reward and 98.5% solved with 1,000 samples. Marginal gains are then observed with each order of magnitude increase in the number of training samples, eventually reaching 99.2% solved with 100,000 samples. Notably, this means that the agent only needs to train on a small fraction of samples to achieve a near optimal level of
performance. However, this also means that orders of magnitude more samples are required to achieve the last 1% improvement in test performance.

Interestingly, the agent becomes less effective on the training set as the number of mazes increases as the agent is unable to memorise the exact solution to every maze (with reward dropping from 0.98 to 0.92). This coincides with an increased efficiency on the test set as the number of steps to solve the mazes decreases from 71.6 to 16.1.

5.2.3.2 3D World Maze Navigation

Presented in Table 5.2 are the results of this experiment on the 3D World environment. In particular, the average reward achieved by the agent (Reward), the percentage of mazes solved (% Solved), and the average time to reach the goal (Steps) are quantified on both the train and test set as the size of the training set (Number of Training Mazes) is increased from 10 to 100,000 unique samples.

Table 5.2: 3D World generalisation results. Quantifying the navigation agent’s ability to generalise to unseen mazes in the 3D World environment as the size of the training set is increased from 10 to 100,000 mazes.

<table>
<thead>
<tr>
<th>Number of Training Mazes</th>
<th>Train Performance</th>
<th>Test Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reward</td>
<td>% Solved</td>
</tr>
<tr>
<td>10</td>
<td>44.4</td>
<td>99.6</td>
</tr>
<tr>
<td>100</td>
<td>39.8</td>
<td>99.4</td>
</tr>
<tr>
<td>1,000</td>
<td>38.4</td>
<td>99.1</td>
</tr>
<tr>
<td>10,000</td>
<td>37.5</td>
<td>98.7</td>
</tr>
<tr>
<td>100,000</td>
<td>37.8</td>
<td>98.3</td>
</tr>
</tbody>
</table>

Similar to the previous results on the Gridworld environment, the agent’s ability to generalise again improves drastically as the number of training mazes is increased. In particular, test performance quickly improves across the first 1,000 samples from 7.5 reward, 74.4% solved, and 241.6 steps to 34.1 reward (+355%), 98.0% solved (+32%) and 52.8 steps (-78%). Notably, this finding holds despite the significantly more complex environment with longer episodes, higher resolution observations, and a larger action set. This therefore demonstrates that procedural generation can be a broadly applicable method for improving generalisation performance.
5.3 Generalisation Across Video Game Levels

Following the same methodology as the previous section, this section quantifies generalisation across video game levels with agent test time performance investigated as the size of the training set is varied. Additionally, results are compared between training on randomly sampled levels and using a curriculum-based training approach.

5.3.1 Setting

In this section, both the environment and agent are summarised.

5.3.1.1 Environment

The video game levels take place in the Frogs environment (see Section 4.3.1 for full details). Each level is procedurally generated and the agent is tasked with reaching the goal without drowning in the water or getting hit by a car. Notably, the agent is only rewarded for reaching the goal and fully observes the environment. In Figure 5.2 below, two examples of such a level are visualised at different values of difficulty.

![Example level at difficulty 0.](image1)

![Example level at difficulty 10.](image2)

Figure 5.2: **Video game level.** Example procedurally generated levels for the Frogs environment at (a) difficulty 0 (i.e. easy) and (b) difficulty 10 (i.e. hard).

The motivation behind using the Frogs environment for investigating generalisation in this chapter is two-fold. First, the generated levels are fully observable as opposed to the partially observable navigation tasks seen previously. This makes it possible to investigate whether the generalisation finding also holds in this setting. Second, the Frogs environment has been used before in related work to investigate procedural generation through a curriculum-based training approach which progressively increases level difficulty to match the agent’s performance [14]. As a result, it
is possible to directly compare their findings with those here as will be seen in the latter half of the results section (Section 5.3.4).

5.3.1.2 Agent

The agent uses the reinforcement learning algorithm Advantage Actor-Critic (A2C) [2] and follows the architecture and settings used by the related work on the same environment [14]. Specifically, the neural network has three convolutional layers followed by a single fully-connected layer, and the agent is trained using 12 parallel workers, a discount factor of 0.99, and a learning rate of 0.007 with the RMSprop optimiser [93], thereby allowing a direct comparison with the results obtained in [14].

5.3.2 Experimental Setup

To investigate the agent’s ability to generalise across video game levels, the same methodology is used as in Section 5.2. Specifically, the training set is varied from containing 10 to 100,000 levels with a new agent trained on each. The resulting trained agents are then evaluated on the same set of 10,000 test levels with their performance quantified. Of note is that the levels are sampled uniformly across the available range of difficulties rather than sampling the underlying parameters directly.

5.3.3 Generalisation Results

Presented in Table 5.3 are the results of this experiment on the Frogs environment. In particular, the percentage of levels solved (% Solved) is quantified on both the train and test set as the size of the training set (Number of Training Levels) is increased from 10 to 100,000 unique samples.

As can be seen, the results on the Frogs environment follow a similar pattern to the previous section, with more training levels leading to an increase in test performance (from 0% with 1 sample to 46% with 100 and 81% with 100,000). However, one notable difference here is that the agent’s performance during training is initially low when trained on few levels (23% solved with 10 samples), rather than the near optimal performance as seen in the navigation environments (99.4-100% when trained on 10
Table 5.3: **Frogs generalisation results.** Quantifying the agent’s ability to generalise to unseen levels in the Frogs environment as the size of the training set is increased from 1 to 100,000 levels.

<table>
<thead>
<tr>
<th>Number of Training Levels</th>
<th>Train Performance % Solved</th>
<th>Test Performance % Solved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>23</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>59</td>
<td>12</td>
</tr>
<tr>
<td>100</td>
<td>79</td>
<td>46</td>
</tr>
<tr>
<td>1,000</td>
<td>86</td>
<td>73</td>
</tr>
<tr>
<td>10,000</td>
<td>85</td>
<td>78</td>
</tr>
<tr>
<td>100,000</td>
<td>84</td>
<td>81</td>
</tr>
</tbody>
</table>

samples). One explanation for this finding is that there exists an implicit curriculum in the procedurally generated distribution of levels. As a result, training on easier levels helps the agent learn to solve harder levels which it cannot learn directly, and therefore an increase in level diversity leads to a performance improvement in training.

Interestingly, while agents were able to achieve near perfect performance on the maze navigation tasks (> 98% solve rate), their performance on the Frogs levels was not near 100% (instead reaching at most 86% solve rate, even on the training set). There are three possible reasons for this. First, agents observe a top-down view of the environment rather than first-person observations which therefore makes it harder to generalise [94]. Second, the environment’s procedural generator may be flawed and yield unsolvable levels. Finally, the agent architecture used may not have had the capacity or capabilities to learn to solve the most challenging levels.

Ultimately, these findings highlight the challenges of designing procedurally generated environments for deep reinforcement learning agents, and also the necessity of parallel work on developing more capable agent architectures.

### 5.3.4 Comparison with Curriculum-Based Training

Often in reinforcement learning, training an agent directly on a task is ineffective as the task is too difficult for the agent to learn to solve it from scratch (typically due to
the reward being too sparse). For such situations, curriculum-based training methods have been successful both in supervised learning [95] and for non-procedurally generated reinforcement learning problems [96]. Given this and the explicit difficulty parameter available in the procedural level generator for the Frogs environment, a natural question to investigate is whether a training curriculum can help agents learn to solve the most difficult levels.

To this end, related work introduced a method called Progressive Procedural Content Generation (PPCG) which adaptively samples training levels based on the performance of the agent [14]. As this approach performed well on the Frogs environment, leading to an agent which could solve a majority of the most difficult levels, a further question worth investigating is how well do agents perform on such levels when trained on a random sample across all difficulties. To answer this question here, the following three training methods are compared:

1. Diff-10: Train on randomly generated levels of difficulty 10.
2. PPCG: Train on randomly generated levels based on the agent’s performance.
3. Random Sampling: Train on randomly generated levels of a random difficulty.

Presented in Table 5.4 are the results of training using each of these methods. In particular, the percentage of levels solved (% Solved) at difficulty 5 (Diff-5) and difficulty 10 (Diff-10) is quantified for each of the training methods.

Table 5.4: Frogs training method results. Quantifying the agent’s ability to solve procedurally generated levels at difficulty 5 and difficulty 10 in the Frogs environment using each of the training methods.

<table>
<thead>
<tr>
<th>Training Method</th>
<th>% Solved</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Diff-5</td>
</tr>
<tr>
<td>Diff-10</td>
<td>1</td>
</tr>
<tr>
<td>PPCG [14]</td>
<td>81</td>
</tr>
<tr>
<td>Random Sampling</td>
<td>86</td>
</tr>
</tbody>
</table>

There are three notable observations here. First, agents trained directly on the most difficult levels (i.e. training method ‘Diff-10’) never learn to solve levels at that
difficulty (0% solved). Next, strategically adapting the difficulty of training levels based upon agent performance (i.e. PPCG [14]) leads to agents that are not only able to solve the easier levels (81% solved on difficulty 5), but also solve more than half of the hardest levels generated (57% solved on difficulty 10). Finally, and most importantly, training on randomly sampled levels leads to the best performance on both difficulties (86% and 61% solved on difficulty 5 and 10 respectively).

Interestingly, these results suggest that while it may be possible to craft a training curriculum, it is can sometimes be sufficient to train on random samples from the procedurally defined space of environment settings. There are a number of possible reasons for this. For example, designing an effective curriculum can be challenging as human-defined notions of what is difficult do not necessarily correspond to what reinforcement learning agents find difficult. Additionally, the diversity and stationarity provided by random sampling can lead to experiences on harder levels helping agents solve easier levels (and vice-versa).

5.4 Summary

This chapter demonstrated the benefit of training agents on three procedurally generated environments. In particular, that supplying a diverse range of environment settings to agents during training can encourage them to learn more general strategies rather than overfitting to specific settings (therefore improving their generalisation performance). Consequently, their test-time performance improves on novel settings irrespective of changes in their train-time performance.

The takeaway contribution was that agents can achieve strong average-case performance by only training on a fraction of the full space of possible environment settings. This finding was then shown to hold across varying levels of environment complexity such as longer episodes, higher resolution observations, and a larger action set. Furthermore, training on randomly sampled settings was shown to be a simple yet effective method for achieving strong performance on unseen settings, performing comparably with an existing curriculum-based training approach.
Chapter 6

Worst-Case Analysis of State-of-the-Art Agents

In this chapter, the following question is addressed:

\[
\text{Where do agents trained on procedurally generated environments fail to generalise?}
\]

Previously, in Chapter 5, it was shown that agents are able generalise well when trained on procedurally generated environments, learning to perform across settings not seen during training. However, as the space of possible settings is now significantly larger than what an agent typically sees during training, it is important to understand if, where, and how agents fail to perform as expected. To this end, this chapter investigates a worst-case analysis of state-of-the-art reinforcement learning agents.

Contributions  To summarise, the contributions of this chapter are as follows:

1. The definition of adversarial environment settings is introduced (Section 6.3.1).
2. An effective and intuitive approach for finding simple adversarial environment settings which lead to failure is presented (Section 6.3.2).
3. State-of-the-art agents carrying out navigation tasks are shown to suffer from drastic and simple failure cases (Sections 6.4.1 and 6.4.2).
4. Adversarial environment settings leading to failure are demonstrated to transfer across agents with different hyperparameters and architectures (Section 6.4.3).
6.1 Introduction

Through the use of deep reinforcement learning, human-level performance has been achieved on a number of tasks in recent years such as Atari [1], Labyrinth [4], and Capture the Flag [21]. On these tasks, and more generally in reinforcement learning, agents are typically trained and evaluated using their average reward over environment settings as the measure of performance, i.e.

$$E_{P(e)} [R(\pi(\theta), e)] ,$$

(6.1)

where $\pi(\theta)$ denotes a policy with parameters $\theta$, $R$ denotes the total reward that the policy receives over the course of an episode, and $P(e)$ denotes the distribution over environment settings $e$ as defined by the environment’s procedural generator. But what biases does the distribution $P(e)$ contain, and what biases, or failures to generalise, do these induce in the strategies that agents learn?

To help uncover biases in the training distribution and in the strategies that agents learn, this chapter focuses on empirically evaluating the worst-case performance of agents over a sampled set of environment settings, i.e.

$$\min_{e \in E} E [R(\pi(\theta), e)] ,$$

(6.2)

where $E$ is some set of possible environment settings.

Specifically, this chapter investigates an empirically approximate worst-case analysis into the performance of two state-of-the-art agents in solving procedurally generated first-person navigation tasks in a 3D world – a task on which agents have recently achieved human-level average-case performance [4]. By optimising over the structure of mazes to minimise the performance of agents, a number of mazes are discovered where agents consistently fail to reach the goal. Referred to as a catastrophic failure, it is further shown that agents suffer from such failures even on surprisingly simple mazes despite the agents’ impressively high average-case performance. To conclude, it is demonstrated that failures transfer between different agents and even significantly different architectures.
6.2 Setting

Throughout this chapter, the worst-case performance of two different agents is analysed on first-person navigation tasks in a 3D environment. In this section, both the environment and the agents are described.

6.2.1 Environment

The navigation tasks are performed in the 3D World Maze Navigation environment (see Section 4.2.2 for full details). Each task takes place in a procedurally generated maze where the agent must find and reach the goal as many times as possible within the timespan of a single episode. Notably, the agent’s score is the number of times it reaches the goal in an episode on average, with the agent restarting in its initial starting position each time it reaches the goal within the episode.

Presented in Figure 6.1a is a high-level example of such a task with an illustrative agent path which successfully reaches the goal. Alongside this, presented in Figure 6.1b, are the agent’s first-person observations while navigating along this path.

![Example navigation task.](image1)

![Partially observable agent observations.](image2)

Figure 6.1: **Navigation task.** (a) Example procedurally generated maze together with the path taken by the agent from its starting location (cyan, bottom left) to the goal (magenta, middle). (b) Frames from top left to bottom right corresponding to the agent’s observations as it takes the path from its starting location to the goal.

There are two motivating reasons for using this specific environment to analyse the worst-case performance of agents. First, it has recently been used to train agents that have achieved human-level average-case performance [3, 4]. Therefore, both the environment and the agents are ideal candidates for analysis as they represent the current state-of-the-art for deep reinforcement learning agents trained to navigate.
Second, while the navigation tasks take place in a complex partially observable 3D world, the procedurally generated environment settings are simply the 2D layout of walls that represent the maze. The simplicity of this setting is an advantage because if agents are not robust to environment perturbations in such a simple setting, then it highlights the extreme difficulty of reaching robust agents in more complex settings (such as self-driving car simulators).

6.2.2 Agents

In this section, the two types of agents analysed throughout this chapter are described.

6.2.2.1 A2C Agent

The first of the two agents, and the main one analysed throughout this chapter, is based on the Importance Weighted Actor-Learner Architecture [3]. Henceforth referred to as ‘A2C agents’, these agents have recently been trained to achieve human-level average-case performance on the navigation tasks studied here.

**Architecture** The network architecture is as follows. First, the $96 \times 72$ visual observation is processed through two convolutional layers with 16 and 32 channels, $8 \times 8$ and $4 \times 4$ kernel sizes, and strides of 4 and 2 respectively, followed by a rectified linear unit (ReLU) activation function. The output is then passed through a single fully-connected layer and then concatenated with the last reward and action. This is then passed through an LSTM layer to produce policy logits, from which an action is sampled, and value output.

**Training** The agents were trained following [3] with three notable modifications which led to improved performance over the previously published results. Specifically, the training procedure was modified as follows: (1) agents were trained for 10 billion steps instead of 333 million steps, (2) rewards were clipped to $[-1, 1]$ rather than being asymmetrically clipped, and (3) a simplified action set was used based on [97].
In total, five separate A2C agents were trained with different entropy costs and learning rates. Each agent was trained for four days on approximately 5.5 million episodes after which they reached the goal on average 31.5 times per episode (i.e. the agents achieved a score of 31.5 on average).

6.2.2.2 MERLIN Agent

The second agent analysed in this chapter is based on the Memory, Reinforcement Learning, and Inference Network (MERLIN) model [4]. Henceforth referred to as ‘MERLIN agents’, these agents use a sophisticated memory structure to store and recall information, improving their performance on partially observable environments.

**Architecture** The network architecture is a simplified variant of the model presented in [4] and was provided by the authors (see [23] for details). Specifically, the architecture was modified as follows: (1) the stochastic latent variable model was removed, (2) the state transformation was changed to a deterministic transformation rather than sampling from a Gaussian distribution, and (3) the policy network was changed to a feedforward multi-layer perceptron.

**Training** In total, five MERLIN agents were trained for seven days on approximately 850,000 episodes each. After training, these agents reached the goal on average 35 times per episode (i.e. the agents achieved a score of 35 on average, an 11% improvement over the A2C agents).

6.3 Adversarial Environment Settings

In this section, the definition of adversarial environment settings is introduced and a search algorithm for finding such settings which lead to failure is presented.
6.3.1 Definition

An environment setting can be defined as adversarial if it causes a significant negative deviation in agent performance from their expected behaviour (often also resulting in a substantial reduction in reward). One such deviation is when an environment setting leads to a severe and undesirable event occurring - known as a catastrophic failure. Examples include an autonomous car crashing into a wall, conflict emerging between cooperative agents, and a navigation agent failing to traverse a maze.

Importantly, an environment setting may not be consistently adversarial as it depends on both the policies of the agents interacting within the environment and the environment’s stochastic behaviour. As a result, finding adversarial environment settings can be challenging as catastrophic failures typically require a specific series of events in order to occur.

In the context of the navigation tasks considered through this chapter, a catastrophic failure is defined as the agent failing to reach the goal within a single two minute episode. Therefore, an adversarial environment setting is a maze which leads to this catastrophic failure (also referred to as an adversarial maze, or more generally as an adversarial example).

6.3.2 Search Algorithm

In order to analyse the worst-case performance of agents, it is first necessary to have a method which can find environment settings which lead to agents performing poorly. However, as the environment is not differentiable, only black-box methods can be used (which rely on querying agent performance given environment settings). This therefore eliminates the use of gradient-based methods from supervised learning which are typically used to find adversarial inputs [56, 57, 98].

As a result, this section introduces a local search algorithm for finding environment settings which lead to catastrophic failure by adversarially optimising over the structure of mazes. Presented in Algorithm 1 and visualised in Figure 6.2, the search
method works by iteratively mutating candidate mazes using an agent’s performance
as the fitness measure.

Algorithm 1 Method for finding environment settings which lead to failure

1: candidates ← GENERATECANDIDATES(num_candidates)
2: for iteration i = 1 to num_iterations do
3:    best_candidate ← EVALUATE(candidates, num_evaluations, trained_agent)
4:    candidates ← MODIFY(best_candidate, num_candidates)
5: end for
6: best_candidate ← EVALUATE(candidates, num_evaluations, trained_agent)
7: return best_candidate

Figure 6.2: Example of the adversarial search algorithm. First, a set of 10 initial
candidate mazes are sampled from the procedurally defined distribution. Next, each
candidate is evaluated 30 times using the trained agent. From these, the best maze
is selected (i.e. the maze where the agent performed worst), and the MODIFY function
is applied to it by randomly moving two walls to form the next set of candidates
(Iteration 1). This process is repeated for 20 iterations, leading to a maze where the
agent gets a score of 0.09 (i.e. the agent reaches the goal once in 11 episodes).

Concretely, the algorithm starts by generating a set of initial candidates by sam-
ping mazes from the procedurally defined distribution (line 1). Next, a trained agent
is repeatedly evaluated on each candidate to find the best one (i.e. the candidate
where the agent performs worst) (line 3). This best candidate is then modified to
produce a new set of candidates (line 4) with the MODIFY function randomly moving
two walls within the maze (rejecting movements that lead to unsolvable mazes in the
mutator). Finally, the evaluate/modify optimisation loop is either repeated or the
algorithm returns the best candidate maze (depending on the number of iterations).
Notably, while a catastrophic failure is defined as the agent failing to reach the
goal, it is not the optimisation objective used by the search algorithm to select the
best candidate at each iteration. Instead, the number of times the agent reaches the
goal (i.e. score) is used as the objective. The reason for this is that the agent’s score
provides a stronger signal at the start of the optimisation process when the agent
typically reaches the goal in all the mazes.

The advantage of this algorithm is threefold. First, it is intuitive to understand
and implement while still being effective at finding mazes which lead to catastrophic
failure (as will be demonstrated in Section 6.4.1). Second, it can be easily adapted
by changing the MODIFY function (for example, to only remove walls as done in Sec-
tion 6.4.2). Third, the approach is generalisable to other procedurally generated
environments which are also parametrised to allow small local changes to the envi-
ronment while maintaining its solvability.

6.4 Worst-Case Analysis

In this section, the worst-case performance of the navigation agents from Section 6.2.2
is analysed. Specifically, the search algorithm from Section 6.3.2 is used to find
mazes which cause the agents to perform significantly worse than their average-case
performance suggests. The results of this analysis are ordered into the following
subsections: (1) mazes leading to catastrophic failure, (2) simple mazes leading to
catastrophic failure (including human experiments and a robustness analysis), and
(3) transfer of catastrophic failures across agents and architectures.

6.4.1 Mazes Leading to Catastrophic Failure

While the navigation agents studied here have achieved human-level average-case per-
cformance, an important question to ask is whether there exists environment settings
which lead to catastrophic failure with high probability?

To answer this question, this section investigates how agent performance is affected
by the mazes produced using the adversarial search algorithm.
6.4.1.1 Setup

The setup for finding and evaluating mazes which lead to catastrophic failure involves using the previously introduced search algorithm (Algorithm 1) and comparing agent performance to the average-case baseline. The details for both of these are as follows.

**Search Algorithm** In total, the search algorithm was independently ran 250 times for five separate A2C agents. The algorithm was configured to use 10 initial candidates, 20 iterations, and 30 evaluations per maze, resulting in a total of 6600 evaluations per agent per run and taking 30 minutes on average. Importantly, the results were consistent across the five agents, and therefore this section presents the results from 50 runs using a single agent (referred to as ‘the agent’).

**Baseline Performance** A key characteristic of the search algorithm is that the mazes it finds are unlikely to be in the training distribution. As a result, comparing agent performance between the two is unfair as any decrease in performance may be simply due to the adversarial mazes being out-of-distribution. Therefore, to make a fairer comparison, agent performance on the adversarial mazes is compared to the average-case agent performance on randomly perturbed mazes (i.e. baseline) which are also unlikely to be in the training distribution. Specifically, evaluation mazes are sampled from the procedurally defined distribution and randomly perturbed using the same `MODIFY` function as used by the search algorithm (i.e. move two walls).

6.4.1.2 Results

Presented in Figure 6.3 are the optimisation curves for the search algorithm showing (a) the number of goals reached by the agent, and (b) the agent’s probability of reaching the goal across 20 iterations. As can be seen, by adversarially optimising over the structure of mazes, it is possible to find examples which cause a significant negative deviation in agent performance from their average-case baseline.
Figure 6.3: **Adversarial optimisation curves.** By optimising mazes to minimise agent score (i.e. the number of goals reached), the search algorithm is able to rapidly find mazes where the agent fails to reach the goal. (a) The number of goals reached is decreased from 45 per episode to 0.33 per episode within 20 iterations. (b) The probability of reaching the goal is reduced from 98% to 20% across 20 iterations. The blue lines are computed by averaging across 50 independent runs, and the dashed lines correspond to the agent’s average-case baseline performance.

Specifically, as shown in Figure 6.3a, the number of goals reached by the agent is reduced from the average-case baseline of 45 per episode to 0.33 per episode at iteration 20 (more than a $100\times$ decrease in performance). Interestingly, a greater than $10\times$ decrease in performance is seen from iteration 3 onwards, demonstrating that few modifications are necessary for a substantial reduction in agent performance.

Similarly, the probability of catastrophic failure is also impacted. As shown in Figure 6.3b, mazes are found where the agent fails to reach the goal in a vast majority of episodes. In particular, agent performance is reduced from the average-case baseline of reaching the goal in 98% of episodes to reaching the goal in fewer than 20% of episodes at iteration 20. Furthermore, from iteration 7 onwards the agent is more likely to not reach the goal than to reach it.

Out of the 50 independent runs, 44 (88%) led to an adversarial maze where the probability of reaching the goal was less than 50%. Additionally, the 25th / 50th / 75th percentiles were as follows: 0.031 / 0.136 / 0.279 for the probability of reaching the goal, and 0.042 / 0.136 / 0.368 for number of goals reached. Therefore, it can be concluded that the search algorithm is also robust to different initialisations.
In Figure 6.4, example agent trajectories on both a randomly perturbed maze and an adversarially optimised maze are visualised. As can be seen, the trajectory on the optimised maze appears to demonstrate a failure by the agent in using its memory to explore the maze efficiently, with the agent instead repeatedly visiting the same locations multiple times. The next section investigates whether this failure is due to the increased maze complexity from moving walls, or whether there also exists simple mazes which lead to catastrophic failure.

(a) Randomly Perturbed.  
(b) Adversarially Optimised.

Figure 6.4: Example mazes and agent trajectories leading to low scores. (a) Maze with randomly perturbed walls. Despite being outside of the training distribution, agents reach the goal in 98% of episodes on such mazes and are able to reach the goal on average 45 times per episode. (b) Maze that has been adversarially optimised for 20 iterations (where two walls were moved at each iteration to minimise score). All agents find the goal on such mazes in fewer than 20% of episodes.

6.4.2 Simple Mazes Leading to Catastrophic Failure

While the existence of catastrophic failures is intriguing and perhaps even concerning, a possible explanation is that the failures are simply due to the increased complexity of the mazes in comparison to those seen by the agent during training. Furthermore, understanding the cause of failure in such mazes is challenging due to the large number of possible wall structures that may be causing the agent to fail.

To address these concerns, this section investigates whether simple mazes exist which lead to catastrophic failure. This is complemented by a study of human performance on such mazes, as well as an analysis into the robustness of their failures.
6.4.2.1 Setup

To find simple mazes which lead to catastrophic failure, a measure of maze complexity is first needed. To this end, maze complexity here is measured using the number of walls the maze contains. Specifically, if a maze has few walls then it is a simple maze, and conversely if the maze has many walls then it is a complicated maze.

In this section, the adversarial search algorithm is adapted to find simple mazes which lead to catastrophic failure. To do this, the procedure’s MODIFY function is changed to remove walls from known adversarial mazes, and the simpler mazes are then evaluated by five A2C agents.

The procedure for finding and evaluating simple adversarial mazes is as follows. First, the same procedure from the previous section (i.e. Algorithm 1) is used to produce a set of mazes which all lead to catastrophic failure. Next, the mazes from this set are independently used as the initial candidate to the adversarial search algorithm, however the MODIFY function is changed to remove a single wall each iteration instead of moving two walls. This process is then repeated for 70 iterations, yielding a set of mazes with as few walls as possible while trying to maintain the agent’s poor performance. Finally, the performance of each agent is then evaluated on each simple maze to check whether it still leads to catastrophic failure.

6.4.2.2 Results

Interestingly, a majority of the walls can be removed from an adversarial maze and it will still lead to catastrophic failure. An example of such mazes is presented in Figure 6.5, with all agents reaching the goal in fewer than 40% of episodes and the agent the maze was optimised for reaching the goal in fewer than 10% of episodes. Of note is that a number of these mazes are strikingly simple, suggesting that there exists structure in the environment that the agent has not generalised to. For example, it can be seen that placing the goal in a small room in an otherwise open maze can significantly reduce the agent’s ability to reach the goal.
6.4.2.3 Human Evaluation

While these simple mazes may lead to catastrophic failure, it is unclear whether this is because of the agent or whether the mazes are difficult in a non-obvious way. Specifically, while the mazes appear obvious from above (as they are all visualised), the world itself is a complex 3D environment that is only viewed from a restricted first-person perspective, and therefore it may actually be too difficult for the agent to reach the goal. As a result, this section investigates how humans perform on the same set of simplified mazes under the same conditions (i.e. the mazes from Figure 6.5).

**Setup** The human experiments were conducted with three different participants who had each previously played on the environment (and were therefore familiar with the controls), but who had not seen the test mazes before. In particular, each human played a single episode lasting 120 seconds on each of the ten mazes, with both their trajectory and time taken to reach the goal recorded. Importantly, to rule out visual acuity as a confounding factor, the humans also played at the same resolution as the agents (i.e. 96 × 72 pixels).
Results  Presented in Table 6.1 is the time taken by each human to reach the goal in each maze. Notably, all of the human players were able to successfully reach the goal in every maze, and were typically able to do so efficiently (requiring less than a third of the time of a full episode). This demonstrates that the mazes are comfortably solvable within the course of an episode using a general navigation strategy, even with the restricted first-person observations.

Table 6.1: Human seconds-to-first-goal on agent adversarial mazes. For each maze, the agent that performed best found the goal less than 50% of the time. In contrast, humans always found the goal, usually within less than a third of the episode.

<table>
<thead>
<tr>
<th>Maze</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human 1</td>
<td>13</td>
<td>14</td>
<td>27</td>
<td>15</td>
<td>18</td>
<td>47</td>
<td>41</td>
<td>37</td>
<td>24</td>
<td>16</td>
</tr>
<tr>
<td>Human 2</td>
<td>25</td>
<td>22</td>
<td>25</td>
<td>16</td>
<td>24</td>
<td>53</td>
<td>86</td>
<td>33</td>
<td>25</td>
<td>22</td>
</tr>
<tr>
<td>Human 3</td>
<td>64</td>
<td>14</td>
<td>16</td>
<td>18</td>
<td>17</td>
<td>52</td>
<td>17</td>
<td>33</td>
<td>22</td>
<td>42</td>
</tr>
</tbody>
</table>

While both agents and humans achieve comparable average-case performance on non-adversarial mazes, these results suggest that their performance on simple mazes is markedly different. To investigate this difference, Figure 6.6 presents a side-by-side comparison of an agent’s trajectory and Human 3’s trajectory on the same maze. Interestingly, the agent’s trajectory is repetitive in the large open space, repeatedly visiting the same locations while failing to explore the enclosed space in the bottom right. This contrasts with Human 3’s trajectory which briefly looks around the environment from the starting location before navigating into the enclosed space where the goal is located.

Figure 6.6: Example simplified maze with agent and human trajectories. (a) Maze obtained through additional iterations of removing walls. (b) Example agent trajectory which fails to reach the goal. (c) Example human trajectory which successfully reaches the goal.
As shown in Figure 6.7, the navigation strategy employed by Human 3 was not only efficient but also consistent across all ten of the simplified adversarial mazes.

![Mazes](image)

Figure 6.7: Trajectories taken by Human 3 on the simplified adversarial mazes. Even on Maze 1, which took the longest, the human was able to reach the goal within just over half the time of a full episode (64/120 seconds).

### 6.4.2.4 Robustness Analysis

A natural question that arises concerns the extent to which these adversarially optimised mazes are isolated points in the space of mazes. That is, if an adversarial maze was changed slightly, would it no longer lead to catastrophic failure, or would the catastrophic failure remain?

Importantly, if the adversarial mazes are robust to changes, then it suggests that there is a systemic problem which can be addressed (for example, in the agent architecture or in the training method). However, if the mazes are not robust, then it suggests that they are merely over-engineered edge cases, and, while interesting, are not informative. As a result, this section investigates how sensitive the discovered adversarial mazes are with respect to changes in the agent’s and goal’s location.

**Setup** To measure the robustness of adversarial mazes, the agent’s performance in reaching the goal is evaluated when the goal and player locations are modified. For example, if the maze still leads to catastrophic failure under a number of different
variations, then it is robust. Conversely, if the maze no longer leads to catastrophic failure even with minor changes, then the adversarial maze is not robust.

In this section, two different types of robustness are investigated: (1) robustness to changes in the agent’s starting location, and (2) robustness to changes in the goal’s location. To do this, simplified adversarial mazes are evaluated using five A2C agents (with three maze examples and sample agent trajectories presented in Figure 6.8).

Figure 6.8: **Example adversarial mazes whose robustness is analysed.** Three simplified mazes which lead to catastrophic failure with example trajectories.

The process for evaluating an adversarial maze’s robustness involves fixing the location of one entity while exhaustively evaluating all possible locations of the other entity. For example, the maze’s robustness to changes in the agent’s start location is evaluated by fixing the goal’s location and iteratively evaluating the agent’s performance at each valid starting location in the maze (repeating each 30 times). Similarly, the maze’s robustness to changes in the goal’s location is evaluated by fixing the player’s starting location and iterating over all valid goal locations. Once completed, this process produces a heatmap representing the agent’s probability of reaching the goal under each variation which can overlay the maze.

**Results** To begin, robustness to changes in the agent’s starting location is investigated. As can be seen in Figure 6.9, for a large range of starting locations, the mazes still lead to catastrophic failure. Specifically, starting the agent in any location other than the enclosed space where the goal is located leads to catastrophic failure with the agent reaching the goal in fewer than 10% of episodes.
Next, robustness to changes in the goal’s location is investigated. As can be seen in Figure 6.10, a majority of the valid goal locations do not lead to catastrophic failure, and instead have a 100% chance of being reached by the agent. However, placing the goal at any location in the enclosed space in the bottom left drastically drops the agent’s probability of reaching the goal.

Together, these results suggest that the adversarial mazes are not merely isolated points in the space of mazes, but in fact represent flaws in the learned navigation strategy of the agent. Specifically, while the agent has learned a general strategy that achieves high average-case performance, it is flawed in that it repeats itself in large spaces and also avoids entering enclosed areas. Therefore, there exists a subset of mazes with this structure which will typically lead to catastrophic failure.
6.4.3 Transfer of Failures Across Agents

So far, adversarial mazes which lead to catastrophic failure (i.e. failure cases) have been discovered for individual agents. However, to what extent do these failure cases highlight a specific peculiarity of the individual agent versus a more general failure of a certain class of agents, or even a shortcoming of the distribution used for training?

To answer these questions, this section investigates whether mazes which have been optimised to lead one trained agent to failure also lead other agents to failure. Specifically, failure transfer is considered between agents with the same model architecture and also between agents with different model architectures.

6.4.3.1 Setup

To measure the transfer of failure cases, failures independently obtained for a number of different agents are evaluated using other agents. For example, if Maze 1 (which was adversarially optimised to lead Agent 1 to failure) also consistently leads Agent 2 to failure, then there is strong transfer. Conversely, if Agent 2 typically reaches the goal, then there is weak transfer. Additionally, if Agent 2 has a medium probability of reaching the goal, then there is moderate transfer.

In this section, two different types of transfer are investigated: (1) transfer between different hyperparameters of the same model architecture (for example, A2C to A2C), and (2) transfer between different model architectures (for example, A2C to MERLIN). To do this, adversarial mazes are found and cross-evaluated between 10 trained agents: a set of five A2C agents and a set of five MERLIN agents.

The process for evaluating transfer is as follows. First, the adversarial search algorithm (Section 6.4.1) is used to produce a collection of 50 unique adversarial mazes for each agent (i.e. 10 collections of 50 mazes each). Next, every agent is evaluated 100 times on each maze in each collection. Finally, the average performance (i.e. probability of reaching the goal) of each agent on each collection is reported.
6.4.3.2 Results

Result 1: Failure cases moderately transfer across all agents

First, there exists some level of transfer across all agents (both within the same architecture and across different architectures). In particular, as can be seen in Figure 6.11, the probability of one agent reaching the goal on mazes adversarially optimised for another agent is significantly and consistently below the average-case performance of 0.98. However, the probability of reaching the goal is not as low as for the original agent that the maze was optimised for (i.e. the $x = y$ diagonal in Figure 6.11a), indicating that there is not complete transfer.

Result 2: Transfer within agent type is stronger than between agent type

Next, transfer is stronger between agents with the same architecture than between agents with different architectures. This can be seen in Figure 6.11b where the agent’s probability of reaching the goal is lower (i.e. stronger transfer) when evaluated on failure cases from agents with same architecture. For example, MERLIN agents on MERLIN failure cases averaged 0.42, while they averaged 0.58 on A2C failure cases. Likewise, A2C agents averaged 0.63 on A2C failure cases and 0.70 on MERLIN failure cases. This suggests that agents share some biases due to their architecture.

Result 3: A2C agents are less susceptible to transfer than MERLIN agents

Interestingly, while the A2C agents have a less sophisticated architecture and perform worse on the navigation tasks on average, they are less susceptible to transfer than the MERLIN agents. Specifically, as shown in Figure 6.11b, the median performance of the A2C agents is consistently higher than their MERLIN counterparts. For example, MERLIN agents achieve 0.42 and 0.58 on MERLIN and A2C failure cases respectively, while A2C agents achieve 0.70 and 0.63 on the same respective failure cases. This suggests that there may exist a trade-off between model complexity, average-case performance, and worst-case performance, even when training on procedurally generated environments.
(a) Probability of reaching the goal across hyperparameters with same architecture. (b) Probability of reaching the goal within the same architecture and across architectures.

Figure 6.11: **Transfer of catastrophic failures across agents & architectures.** Mazes that lead to failure in one agent lead to failure in other agents. This holds for agents with the same architecture (with different hyperparameters) and for agents with significantly different architectures. (a) Transfer results across five A2C agents with different hyperparameters. (b) Transfer results across five A2C agents and five MERLIN agents. A probability below 0.98 (the average-case baseline performance) indicates some level of transfer, with a lower probability indicating stronger transfer.

Figure 6.12: **Full pairwise transfer results.** Average probability of a trained agent reaching the goal in mazes optimised for another agent. As above, a probability below 0.98 indicates some level of transfer, with a lower probability indicating stronger transfer. ‘A’ and ‘M’ correspond to A2C and MERLIN agents respectively.
6.5 Summary

This chapter demonstrated that despite the strong average-case performance often reported of reinforcement learning agents trained on procedurally generated environments, worst-case analysis can uncover environment settings which agents have failed to generalise to.

The main contribution was the definition of adversarial environment settings which was introduced alongside a general search algorithm for finding such settings that lead to significantly reduced agent performance. Using this algorithm, it was shown that not only do catastrophic failures exist for state-of-the-art navigation agents, but simple catastrophic failures also exist which are robust to various agents while being easily solvable by humans. Furthermore, it was also demonstrated that settings which lead to failure transfer across agents with different training hyperparameters and architectures.
Chapter 7

Improving Agent Robustness to Adversarial Environments

In this chapter, the following question is addressed:

*How can adversarial examples be incorporated into training to improve the performance of agents?*

Previously, in Chapter 6, the issue of adversarial environment settings was highlighted and it was demonstrated that state-of-the-art reinforcement learning agents are not robust to such settings. To address this, this chapter presents a number of methods for efficiently finding adversarial environment settings which are then used to investigate how the robustness of agents can be improved.

**Contributions** To summarise, the contributions of this chapter are as follows:

1. The definition of adversarial environment settings is expanded to cover in-distribution and out-of-distribution adversarial examples (Section 7.3.1).

2. Three methods for finding in-distribution and out-of-distribution adversarial examples are presented (Section 7.3.2 and 7.3.3).

3. Adversarially augmenting the training set of agents is shown to be an effective method for improving their robustness to adversarial examples (Section 7.4).
7.1 Introduction

While failure caused by adversarial environment settings (i.e. adversarial examples) is often cheap and without any lasting consequences in simulation, there are many real-world applications where failing can have catastrophic consequences. For example, in robotics they can cause expensive physical damage, while for self-driving cars they can lead to crashes and potential loss of life. In such applications, it is critical to address failures before they happen by reducing their likelihood of occurring.

However, finding adversarial examples is difficult as they are typically rare, and therefore existing search methods (such as random sampling) are severely inefficient [48]. Furthermore, as adversarial examples can be either in-distribution (i.e. from the training distribution) or out-of-distribution (i.e. outside of the training distribution), alternative methods are required to find and generate each type. Together, these points make it challenging to not only address the failures, but also to simply find them in the first place.

To address this challenge and help improve agent robustness, this chapter studies a navigation agent which is tasked with reaching the goal in procedurally generated gridworld mazes. Notably, an adversarial training approach is introduced which adversarially augments the training set of agents with examples which lead the agent to failure. Alongside this, two methods for finding in-distribution adversarial examples are presented with their efficiency over random sampling highlighted. In addition, the local search method from the previous chapter is adapted to facilitate the study of adversarial examples which are iteratively perturbed to be further out-of-distribution.

The benefits of the introduced adversarial training approach are demonstrated by applying it first to in-distribution examples and then to out-of-distribution examples. For both, the introduced methods for finding and generating such examples are used to adversarially augment the initial training set, thereby producing new sets for agents to train on. By training on these augmented sets, the robustness of agents is significantly improved to both types of adversarial examples, though at the cost of marginally decreased reward and longer training time.
7.2 Setting

Throughout this chapter, the robustness of an agent is investigated and improved on first-person navigation tasks in a gridworld environment. In this section, the environment and agent are both described.

7.2.1 Environment

The navigation tasks are performed in the Gridworld Maze Navigation environment (see Section 4.2.1 for full details). Each task takes place in a procedurally generated maze where the agent must find and reach the goal within the timespan of a single episode. In Figure 7.1 below, an example of such a task is visualised with an example agent trajectory (which solves the task) alongside the agent’s observations.

![Gridworld Maze Navigation Environment](image)

(a) Example navigation task. (b) Partially observable agent observations.

Figure 7.1: Navigation task. (a) Example procedurally generated maze along with an agent’s trajectory from its starting location (cyan triangle) to the goal (purple circle). (b) Frames from top left to bottom right corresponding to the agent’s visual observations as it moves along its trajectory towards the goal.

There are two key reasons for using this specific environment to investigate how agent robustness can be improved. First, similar to the environment studied in Chapter 6, the navigation tasks and their associated environment settings are meaningful and intuitive. Therefore, the environment is an ideal candidate for investigating how agents can be made more robust. Second, by moving from the 3D world of the previous chapter to the gridworld here, the training time of agents is significantly reduced (from 6e6 episodes over 120 hours to 6e4 episodes over 1.5 hours). This reduction is vital for practically investigating methods to improve robustness as such methods typically require both (1) incorporating a large number of adversarial examples into
training relative to the number of training examples, and (2) significantly increasing
the number of training steps.

7.2.2 Agent

The agent evaluated in this chapter uses the reinforcement learning algorithm Advantage Actor-Critic (A2C) [2] and is implemented in PyTorch. By default, the following architecture and training setup is used as the baseline. Later, in Section 7.4, the training set, number of training steps, and the network’s capacity are all varied to investigate how the agent’s robustness can be improved.

7.2.2.1 Architecture

The network architecture is as follows: the agent’s visual observation is first processed through two convolutional layers with 16 and 32 channels, $2 \times 2$ kernel sizes, and a stride of 1, followed by a rectified linear unit (ReLU) activation function. The output is then concatenated with the agent’s facing direction (one hot encoded). This is then passed through an LSTM layer followed by a single fully-connected layer to produce policy logits, from which an action is sampled, and value output. The agent is configured to use 16 parallel workers, 10 unroll steps, 64 hidden units, a discount factor of 0.99, and a learning rate of 0.0007 with the RMSprop optimizer [93].

7.2.2.2 Training

Every episode, the agent is trained on a procedurally generated maze. To improve the consistency of training across runs, and investigate the influence of mazes on the agent’s performance, these mazes are sampled from a predefined set of mazes which are generated beforehand by the procedural generator (as opposed to directly sampling from the generator every episode). This predefined set of mazes is referred to as the agent’s training set.

In total, the baseline agent is trained for $2e7$ steps and therefore experiences 60,000 unique mazes across a full training run. Notably, training for this number of steps takes approximately 1.5 hours on a standard consumer desktop and reaches
> 99% optimal average-case performance. In comparison to the agents trained in the previous chapter, training is significantly faster (from 120 hours to 1.5 hours) and produces more optimal agents. As a result of this reduced training time, it is now computationally feasible to investigate significant increases in the training length and training set size of the agents (as will be investigated in Section 7.4).

7.3 Methods for Finding and Generating Adversarial Environment Settings

In this section, adversarial environment settings are defined in the context of the navigation environment and different methods for finding such settings both in- and out-of-distribution are presented.

7.3.1 Adversarial Environment Settings

From the previous chapter, an environment setting can be defined as adversarial if it causes a significant negative deviation in agent performance from their expected behaviour. This can be the result of a substantial reduction in reward or through an undesirable event occurring.

In the context of the navigation environment considered in this chapter, an adversarial environment setting is defined as a maze where the agent typically fails to navigate to the goal in the duration of an episode. Specifically, these are defined as mazes where the agent’s probability of reaching the goal is less than 50%, with such mazes being referred to as adversarial mazes (and known more generally as adversarial examples).

As the mazes are procedurally generated and can then be modified, they can be further defined as either in-distribution or out-of-distribution, where distribution is the procedurally defined distribution generated by the procedural maze generator. The rest of this section details a number of methods which can be used for finding and generating both types of adversarial examples.
7.3.2 Finding In-Distribution Adversarial Examples

An *in-distribution* example is defined as any environment setting which can be sampled from the procedurally defined distribution, and therefore can be seen (but not necessarily is seen) by an agent during training.

There are two key reasons why in-distribution adversarial examples exist. First, as the space of possible environment settings is vast, it can be prohibitively expensive to train the agent on every setting (even if each example is only seen once). For example, the $9 \times 9$ maze generator used by the environment considered here can produce over ten million unique mazes which would require millions of episodes to exhaustively train on each maze once (which still does not guarantee that the agent can then solve all mazes seen during training). Next, the space of possible environment setting is also not uniformly distributed. As a result, training and testing via random sampling leads to many rare and challenging settings being under-seen by the agent, resulting in skewed or even misleading performance results.

In the rest of this section, two different methods for finding in-distribution adversarial examples are presented. First, the typically used method of random sampling is introduced. Next, an alternative method based on rejection sampling is proposed which uses a discriminator to classify examples.

7.3.2.1 Method 1: Random Sampling

The standard baseline, and currently the most common way to train and evaluate agents, is through randomly sampling environment settings. This method works by randomly sampling settings from the procedurally defined distribution and then evaluating the agent on each sample. As a result, this method is inefficient and time-consuming, especially if adversarial examples are rare due to an agent having a low failure rate.

7.3.2.2 Method 2: Rejection Sampling

To overcome the inefficiency of random sampling, which requires evaluating the agent on every sampled environment setting, rejection sampling can be used to select which
examples to evaluate the agent on. Through the use of a discriminator which classifies settings as either adversarial or non-adversarial, evaluation can be performed more efficiently by only evaluating on settings which are predicted to be adversarial.

Rejection sampling is a useful approach when it is cheap to generate environment settings yet expensive to evaluate agents on them. For example, the cost of real-world evaluations tends to be dominated by the cost of running the agent in the real-world, rather than generating the settings themselves. However, the drawback to this approach is that it requires either expert domain knowledge to handcraft a discriminator or a large set of previously evaluated settings in order to train a discriminator. For the latter, one such set is the agent’s training data (which is typically available), and therefore it can be possible to alleviate this drawback.

**Handcrafted Discriminator** If domain knowledge is available about the environment, such as which features are correlated with the adversity of settings, a discriminator can be handcrafted using this knowledge.

From a visual inspection of adversarial mazes, it appears that agents tend to fail when the goal is far away from their starting location and hidden in a small room. Based on this, the discriminator calculates a score for each maze using the following domain-specific features:

1. Shortest Path Length ($S$): The length of the shortest path from the agent’s starting location to the goal’s location.
2. Goal Visibility ($V$): The percentage of positions in the maze from which the goal’s location can be seen by the agent.

Presented in Figure 7.2 are the distributions of these features for both known non-adversarial and adversarial mazes. As can be seen in Figure 7.2a, adversarial mazes have a significantly longer shortest path length to the goal than non-adversarial mazes ($15.7 \pm 3.5$ compared to $6.8 \pm 3.9$). Furthermore, as shown in Figure 7.2b, a large number of adversarial mazes have minimal goal visibility while non-adversarial mazes typically have more visible goals. Together, these findings suggest that there is a correlation between these features and maze adversity, therefore validating their use.
in a handcrafted discriminator. However, it should be noted that while the findings indicate correlation, they do not imply causation.

![Graph showing distribution of shortest path length and goal visibility for non-adversarial and adversarial mazes.](image)

(a) Shortest path length from the agent’s starting location to the goal’s location. (b) Percentage of positions from which the goal can be seen by the agent.

Figure 7.2: **Distribution of maze features.** Distribution of shortest path length and goal visibility for non-adversarial and adversarial mazes. Notably, adversarial mazes tend to have a longer shortest path from the agent’s starting location to the goal and a less visible goal.

To calculate a maze’s score, the discriminator divides the maze’s shortest path length by the visibility of the maze’s goal (i.e. $\frac{S}{V}$). The value of this score is then compared against an optimised threshold ($x$) to determine whether the maze should be labelled as adversarial or not. Specifically, if $\frac{S}{V} > x$ then the maze is predicted to be adversarial, else it is predicted to be non-adversarial.

Presented in Figure 7.3 are several mazes which have been classified by the handcrafted discriminator: four non-adversarial examples and four adversarial examples.

![Mazes from handcrafted discriminator.](image)

Figure 7.3: **Mazes from handcrafted discriminator.** Example non-adversarial and adversarial mazes as predicted by the handcrafted discriminator.

Due to the heuristics used, mazes that have a highly visible goal and a relatively short path to the goal are likely to be predicted as non-adversarial. Conversely, mazes with a long path to a hidden goal are likely to be predicted as adversarial.
Trained Discriminator  An alternative to handcrafting a discriminator is to instead train a neural network to predict whether environment settings are adversarial. This network can then be used to quickly and cheaply evaluate environment settings, determining whether or not to evaluate the agent on them.

The advantage of this method is that it is domain agnostic and does not require expert knowledge in the form of handcrafted heuristics. However, this comes at the cost of the network requiring a significantly larger set of evaluated settings in order to accurately train the network.

Presented in Figure 7.4 are several mazes which have been classified by the trained discriminator: four non-adversarial examples and four adversarial examples.

![Mazes from trained discriminator](image)

Figure 7.4: Mazes from trained discriminator. Example non-adversarial and adversarial mazes as predicted by the trained discriminator.

As can be seen, the examples bear a resemblance to those predicted by the handcrafted discriminator (despite the trained discriminator only training on the maze itself without any path length values being provided). Specifically, there are clear differences in both the goal’s visibility and the shortest path length between the two sets of examples. However, the difference is less noticeable in comparison to the examples predicted by the handcrafted discriminator.

7.3.3 Generating Out-of-Distribution Adversarial Examples

An out-of-distribution example is any environment setting drawn far away from in-distribution. Importantly, while these examples are not in the distribution defined by the procedural generator, and therefore not seen during training, they are still valid tasks. Therefore, it can be considered important to understand how agent performance is affected on such tasks, as well as how agent performance on them can be improved.
In the rest of this section, the local search algorithm from the previous chapter is reintroduced as a method for generating out-of-distribution adversarial examples. Notably, several key changes to the algorithm are highlighted, and a consistent way of measuring the size of adversarial perturbations is explained.

7.3.3.1 Method: Local Search

In this chapter, out-of-distribution adversarial examples are generated using a modified version of the local search algorithm introduced in the previous chapter (Section 6.3.2). Specifically, in-distribution mazes are provided to the algorithm and then iteratively modified through the addition and removal of walls, moving the examples further out-of-distribution each iteration.

The modified local search algorithm is presented in Algorithm 2 with two notable differences in comparison to the original algorithm introduced in Chapter 6. First, a single maze is provided as the initial candidate instead of randomly sampling a set from the training distribution (line 1). This allows a consistent evaluation set of mazes to be used across different agents. Second, at each iteration only a single wall is added or removed, and previously modified walls cannot be modified again (line 3). This enables a consistent way of quantifying the size of modifications and measuring how out-of-distribution examples are across iterations and independent runs.

**Algorithm 2** Modified method for finding environment settings which lead to failure

1: best_candidate ← EVALUATE(initial_candidate, num_evaluations, trained_agent)
2: for iteration i = 1 to num_iterations do
3:     candidates ← MODIFY_ONCE(best_candidate, num_candidates)
4:     best_candidate ← EVALUATE(candidates, num_evaluations, trained_agent)
5: end for
6: return best_candidate

To denote the size of the modification from the original maze to the maze at iteration $i$, the symbol $\epsilon$ is used. As an example, consider the in-distribution maze $X$. If one wall is added to the maze to produce $X_1$, then $\epsilon = 1$. Next, if one wall is then removed in $X_1$ to produce $X_2$ then $\epsilon = 2$ (1 for the first added wall plus 1 for the removed wall). Therefore, at iteration 2 the size of the perturbation $\epsilon$ is 2.
The method works as follows: given an environment setting (i.e., maze) and a modify function (i.e., add/remove a single wall), independently apply the function to the setting to generate a number of candidate settings (line 3). Next, evaluate the agent on each candidate setting (line 4). Finally, repeat for the next iteration using the setting where the agent received the lowest reward as the setting to which the function is applied.

The result of this method is that at iteration 1, after a single modification, $\epsilon = 1$. At iteration 2, $\epsilon = 2$, and so on. Importantly, this is consistent across runs which therefore facilitates the comparison of (approximate) worst-case performance of agents as the size of the adversarial perturbation ($\epsilon$) is iteratively increased. An example of this procedure, and the mazes it produces each iteration, is presented in Figure 7.5.

Figure 7.5: Example of the modified local search procedure. Example mazes produced using the local search procedure across 6 iterations, with the original maze iteratively becoming further out-of-distribution. The wall added/removed each iteration is indicated in red and the agent’s probability of solving the maze is listed above each maze (‘Solve %’).

As can be seen, the method iteratively reduces the agent’s probability of solving the task across each iteration – from the agent reaching the goal every attempt (100% solve rate) to the agent never reaching the goal (0% solve rate). This is done by first placing walls around the goal to reduce its visibility ($\epsilon = 1, 2$). Next, the shortest path length is increased by blocking off the quickest route ($\epsilon = 3$). Finally, minor changes are performed to influence the agent’s route through the maze and further obscure its vision of the goal ($\epsilon = 4, 5, 6$).
7.4 Improving Agent Robustness

In this section, adversarial training (through adversarially augmenting the agent’s training set) is introduced and investigated as a method for improving agent robustness against both in-distribution and out-of-distribution adversarial examples.

7.4.1 Adversarial Training

While procedural generation can provide significant variation for agents during training, and ultimately improve their ability to generalise to previously unseen settings, it is not without its own drawbacks. As highlighted in the previous chapter, the procedural generators used to train and evaluate agents can be flawed. For example, they can output biased, ‘uninteresting’, or redundant environment settings, or simply be unable to generate all valid settings.

When training reinforcement learning agents on procedurally generated environments, it is important to be aware of these potential flaws as they will influence the policies learned by the agents – from the obvious (such as the agents performing poorly) to the more subtle (such as the agents learning a biased strategy). As an example of the latter, a bias can be observed in the procedural maze generation algorithm used in this chapter. Specifically, as shown earlier in Figure 7.2a, approximately 15% of the generated mazes contain the agent and goal next to one another (an overly frequent yet trivial representation of a navigation task). As a result of their frequency, the learned navigation strategy of the agents is not optimal for solving an unbiased set of navigation tasks.

The issue with flaws like this one is that they artificially boost the reported agent performance due to the presence of many similar (and therefore redundant) examples which can be trivially solved. Furthermore, due to the prevalence of such examples, the strategy learned during training is negatively impacted as the agent will overfit to these common yet poorly generated examples.

To address these flaws and improve agent robustness, an agent can be adversarially trained by searching for examples where the agent fails to perform as expected and
incorporating them back into training. In the context of this chapter, this process involves augmenting the agent’s training set with a large number of adversarial mazes. As a result, adversarial training causes the agent to train more frequently on rare and challenging mazes, with the objective of the agent learning a more robust navigation strategy due to the agent’s reduced ability to overfit to the redundant and poorly generated mazes present in original distribution.

7.4.2 Improving Robustness Against In-Distribution Adversarial Examples

7.4.2.1 Adversarial Training Setup

In order to adversarially train agents, a set containing in-distribution adversarial examples is first required. In this section, the methodology for constructing these sets is described, covering details on both of the discriminators used as well as the changes to the agent’s training setup.

Adversarially Augmented Sets To construct the two adversarially augmented sets, the methods presented in Section 7.3.2 were used to find in-distribution examples which were predicted to be adversarial. Specifically, examples were sampled from the procedurally defined distribution and then labelled by the handcrafted and trained discriminators. The examples which were predicted to be adversarial were then added to the adversarially augmented set according to the corresponding discriminator which was used (i.e. \( D_{\text{Handcrafted}} \) for the handcrafted discriminator and \( D_{\text{Trained}} \) for the trained discriminator).

Below, the baseline set and adversarially augmented sets are summarised:

- \( D_{\text{Baseline}} \): The standard training set of 120,000 mazes which were randomly sampled from the procedurally defined distribution (Section 7.2.2.2).

- \( D_{\text{Handcrafted}} \): The first adversarially augmented training set comprising 60,000 sampled mazes which were predicted to be adversarial by the handcrafted discriminator, plus 60,000 mazes from \( D_{\text{Baseline}} \).
• $D_{\text{Trained}}$: The second adversarially augmented training set comprising 60,000 sampled mazes which were predicted to be adversarial by the trained discriminator, plus 60,000 mazes from $D_{\text{Baseline}}$.

To fairly compare the differences between training on each of these sets, the total number of mazes in each was fixed to 120,000. For the augmented sets, this was achieved by including half of the mazes from the baseline training set (i.e. $D_{\text{Baseline}}$). Therefore, each set contained 120,000 mazes consisting of approximately half non-adversarial and half (predicted) adversarial examples. Notably, including the mazes from the baseline set also led to a significant improvement in reward and step efficiency for the trained agents in comparison to training only on adversarial examples.

It is also worth noting that random sampling was not used to find any adversarial examples to augment the training set. This decision was made due to random sampling’s severe inefficiency at finding adversarial examples, whereby to create an adversarial set of the same size as the training set (i.e. 120,000 examples) it would have taken a prohibitively long time (approximately 2 months or $1000 \times$ longer than training).

**Handcrafted Discriminator** To augment the training set with 60,000 adversarial examples, thereby producing $D_{\text{Handcrafted}}$, the handcrafted discriminator calculates a score for each maze (i.e. its shortest path length divided by its goal visibility). If this score is greater than the optimised threshold of 44.2 then the maze is predicted to be adversarial and therefore added to the new set, else the maze is labelled as non-adversarial and subsequently rejected. The threshold’s value was optimised on a hold-out set of 50,000 mazes.

**Trained Discriminator** To augment the training set with 60,000 adversarial examples, thereby producing $D_{\text{Trained}}$, the trained discriminator uses a neural network to predict whether mazes are adversarial or non-adversarial. As before, mazes that are predicted to be adversarial are added to the new set, and those that aren’t are
rejected. The network itself is trained for 50 epochs on a set containing 6,666 mazes comprising 3,333 adversarial examples and 3,333 non-adversarial examples.

The network’s architecture is as follows: the $9 \times 9 \times 3$ encoded maze is first processed through two convolutional layers with 20 and 50 channels, $3 \times 3$ kernel sizes, and a stride of 1, followed by a rectified linear unit (ReLU) activation function. The output is then passed through a single fully-connected layer with 500 hidden units followed by a ReLU activation function. Finally, the output is passed through a softmax function which corresponds to whether the example is either adversarial or non-adversarial.

**Adversarially Trained Agents** The agents in this section are trained from scratch as before (Section 7.2.2.2), with a couple of key changes. First, the mazes sampled during training are now drawn from the adversarially augmented sets described earlier as opposed to the original procedurally generated training set. Second, due to the increased complexity and challenge of solving adversarial mazes, the number of training steps is increased from $2e7$ to $6e7$ ($3 \times$). For a fair comparison, the baseline’s number of training steps is also equivalently increased.

### 7.4.2.2 Results from Adversarial Training

The robustness of an agent is measured using the *Inverse Adversarial Rate* (IAR): the average number of samples required to find one adversarial example. For instance, if IAR = 100 then 100 examples need to be sampled (and then the agent evaluated on) in order to find one adversarial example. As a result, a highly robust agent that rarely fails will have a high IAR, and an agent that often fails will have a low IAR.

To quantify IAR, as well as other performance measures such as reward and episode length (steps), the agent is evaluated on a hold-out set of 50,000 mazes which were not seen during training. Specifically, the agent’s robustness is considered against adversarial examples that are (1) sampled randomly, and (2) obtained through rejection sampling using a discriminator.
Robustness against examples sampled via random sampling  

Presented in Table 7.1 are the evaluation results of the adversarially trained agent against examples sampled via random sampling. Notably, the agent’s robustness is significantly improved, with IAR increasing from 635 to 2632 (+314%) and 1786 (+181%) when trained on $\mathcal{D}_{\text{Handcrafted}}$ and $\mathcal{D}_{\text{Trained}}$ respectively. In other words, 4.1× and 2.8× more examples need to be sampled in order for the agent to fail the same number of times as the baseline. This comes at the cost of a reduction in reward from 0.929 to 0.922 (-0.76%) and 0.914 (-1.64%), as well as an increase in the time it takes the agent to solve the maze from 14.8 steps to 15.4 (3.9%) and 16.3 (+10.0%).

Table 7.1: **Robustness results against random sampling.** Agent performance on the hold-out evaluation set of 50,000 mazes when trained on each training set. Importantly, training on the adversarially augmented sets ($\mathcal{D}_{\text{Handcrafted}}$ and $\mathcal{D}_{\text{Trained}}$) leads to significant improvements in agent robustness over training on the baseline set ($\mathcal{D}_{\text{Baseline}}$).

<table>
<thead>
<tr>
<th>Training Set</th>
<th>Inverse Adversarial Rate</th>
<th>Reward</th>
<th>Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathcal{D}_{\text{Baseline}}$</td>
<td>635</td>
<td>0.929</td>
<td>14.9</td>
</tr>
<tr>
<td>$\mathcal{D}_{\text{Handcrafted}}$</td>
<td>2632 (+314%)</td>
<td>0.922 (-0.76%)</td>
<td>15.4 (+3.9%)</td>
</tr>
<tr>
<td>$\mathcal{D}_{\text{Trained}}$</td>
<td>1786 (+181%)</td>
<td>0.914 (-1.64%)</td>
<td>16.3 (+10.0%)</td>
</tr>
</tbody>
</table>

While randomly sampling from the procedurally defined distribution can be an inefficient way of evaluating agents, it does have the advantage of being an unbiased adversary. As a consequence of this, the concern of exactly what the agent is more robust to is alleviated, with these results demonstrating that it is possible to train an agent to be more robust to the same distribution that it was originally evaluated on.

Robustness against examples sampled via rejection sampling  

Presented in Table 7.2 are the evaluation results of the adversarially trained agent against examples sampled via rejection sampling using the two discriminators. As can be seen, the agent’s robustness is again significantly improved with IAR increasing between 190% and 703% across all training steps. Interestingly, this improvement in robustness becomes larger as the agent is trained for longer, highlighting that while training for
longer can yield minor improvements in robustness, it is made more effective when combined with other methods such as adversarial training.

Table 7.2: **Robustness results against rejection sampling.** Inverse Adversarial Rate (IAR) of agents on adversarial examples from the evaluation set as predicted by the discriminators. Importantly, adversarial training significantly improves robustness over training on the baseline set ($D_{Baseline}$) and the gained improvement increases with the number of training steps.

<table>
<thead>
<tr>
<th>Evaluation Set Discriminator</th>
<th>Training Set</th>
<th>Number of Training Steps</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$D_{Baseline}$</td>
<td></td>
</tr>
<tr>
<td>Handcrafted</td>
<td>21</td>
<td>105</td>
</tr>
<tr>
<td>$D_{Handcrafted}$</td>
<td>61 (+190%)</td>
<td>560 (+433%)</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>33</td>
</tr>
<tr>
<td>Trained</td>
<td>37 (+428%)</td>
<td>265 (+703%)</td>
</tr>
</tbody>
</table>

A key takeaway from these results is that they demonstrate that it is possible to train agents to be more robust to specific adversaries who are trying to lead them to failure. In this example, the discriminators are the adversaries, and they both become less effective at leading the agent to failure as the agent is progressively trained on each of the adversarially augmented sets.

**Performance of Discriminators** One of the challenges in creating an adversarial set is efficiently finding examples which cause the agent to fail. The reason for this is that while it may be cheap to generate settings, evaluating agents on them is computationally expensive and can take a significant amount of time.

An example of this inefficiency can be seen in the robustness results for the baseline agent trained earlier (Table 7.1) where random sampling discovered only 1 adversarial example for every 635 examples sampled and evaluated (i.e. IAR = 635). In contrast, both the handcrafted and trained discriminators introduced in this chapter were significantly more efficient – reducing IAR to 105 (-84%) and 33 (-95%) respectively (Table 7.2). These results highlight the benefits that can be gained by using alternative methods, such as discriminators, to find adversarial examples.
7.4.3 Improving Robustness Against Out-of-Distribution Adversarial Examples

7.4.3.1 Adversarial Training Setup

In order to adversarially train agents, a set containing out-of-distribution adversarial examples is first required. In this section, the methodology for constructing this set is described, followed by details on the two variations investigated in the adversarially trained agent.

Adversarially Augmented Set To adversarially augment the training set with out-of-distribution adversarial examples, the modified local search method from Section 7.3.3.1 was used. Specifically, local search was ran 30,000 times on the baseline agent, taking the first and last adversarial example from each independent run to produce 60,000 unique adversarial examples. Combined with 60,000 in-distribution mazes from \( D_{\text{Baseline}} \), this created a training set of 120,000 unique mazes with an approximately 1:1 non-adversarial to adversarial example ratio.

Adversarially Trained Agent For the agent itself, two key variations are investigated in comparison to the baseline in an effort to improve robustness alongside adversarial training. First, the number of hidden units is varied to increase the network’s capacity – from the baseline of 64 (1\( \times \)) per hidden layer to 128 (2\( \times \)) and 256 (4\( \times \)). Second, the number of steps that the agent is adversarially trained for is varied – from the baseline of 2e7 (1\( \times \)) to 1e8 (5\( \times \)) and 5e8 (25\( \times \)).

Notably, rather than adversarially training each agent variation from scratch, an agent is pre-trained for each variation using the standard training method and then these agents are adversarially trained. For example, an agent with 256 hidden units is non-adversarially trained for 2e7 steps and then adversarially trained for a further 2e7 steps (represented as 1\( \times \) in Figure 7.6b). By taking this approach, the following results demonstrate that it is possible to improve the robustness of previously non-adversarially trained agents without having to start from scratch. This broadens the
applicability of these results as it means existing agents (such as those in the related literature) can be made more robust without having to restart the agent’s training from the beginning.

7.4.3.2 Results from Adversarial Training

To measure the robustness of each agent variation, every adversarially trained agent is evaluated by independently performing local search starting from the same fixed set of 100 in-distribution non-adversarial examples. During this evaluation process, the agent’s probability of reaching the goal is measured as the size of the adversarial perturbation (\(\epsilon\)) is increased from 0 to 10.

The results of this procedure for both variations (network capacity and training length) are visualised in Figure 7.6. Of note is that the agent variations in Figure 7.6a are all trained until convergence and the agent variations in Figure 7.6b all have 256 (4\(\times\)) hidden units. This was done as both variations were required to observe noticeable improvements in agent performance.

![Figure 7.6](image)

(a) Performance as network capacity is varied. Baseline is the non-adversarially trained agent with 64 hidden units. 1\(\times\), 2\(\times\), and 4\(\times\) are adversarially trained agents with \(x\times\) as many hidden units.

(b) Performance as training length is varied. Baseline is the non-adversarially trained agent trained for 2\(e\)7 steps. 1\(\times\), 5\(\times\), and 25\(\times\) are adversarially trained agents trained for \(x\times\) as many steps.

Figure 7.6: **Robustness results against local search.** Probability of the agent reaching the goal as the local search procedure iteratively modifies the evaluation mazes with larger adversarial perturbations. Performance is shown as (a) the network’s capacity is varied, and (b) the agent’s training length is varied. Measured across 100 evaluation mazes. The horizontal dashed line represents the 50% threshold below which mazes are defined to be adversarial.
As can be seen above, agent performance dramatically decreases as the size of the adversarial perturbation is increased. In particular, while all agents have a greater than 99% probability of reaching the goal when there is no perturbation ($\epsilon = 0$), they all consistently experience a drop in performance as the perturbation’s size is increased. For example, by the largest perturbation ($\epsilon = 10$), all agents have less than a 65% probability of reaching the goal (at best), with the baseline agent performing the worst by a notable margin.

Importantly, in the context of improving agent robustness to adversarial examples, it can be seen that adversarial training is able to significantly improve the agent’s performance across all sizes of perturbations when combined with a larger network capacity and longer training length. Specifically, by quadrupling the number of hidden units and adversarially training for 500 million steps (25× the baseline), the agent’s probability of reaching the goal at $\epsilon = 10$ is more than doubled from 30% to 65% – a marked improvement in robustness as the agent’s failure rate is more than halved even on highly perturbed adversarial examples.

Interestingly, as shown in Figure 7.6a, while an increased model capacity leads to the greatest improvements in robustness, it is not necessary for improving robustness. For instance, adversarially training an agent with the same number of hidden units for more training steps can lead to improvements across all sizes of perturbations (see line 1×). Furthermore, as shown in Figure 7.6b, adversarially training an agent for an insufficiently long time (see line 1×) leads to a negligible improvement in performance. This suggests that while both variations are important, the increased training length is a crucial component for improving robustness.

Taken together, these results demonstrate that while it is possible to adversarially train an agent to be more robust to out-of-distribution adversarial examples, it is both expensive and time consuming relative to the conventional method of training agents. This is due to the cost of generating a large number of adversarial examples to sufficiently augment the training set, as well as the extra training time required by the agents to learn on the more challenging examples.
7.5 Summary

This chapter investigated how adversarial examples could be incorporated into training to improve the performance of agents. Using an expanded definition of adversarial environment settings to cover in-distribution and out-of-distribution adversarial examples, the robustness of agents to both types of settings was improved through adversarial training.

The core contribution was the proposed adversarial training approach which involved adversarially augmenting the training set of agents with adversarial examples, as well as the results from its application. Due to the inefficiency of random sampling for finding such examples, several alternative methods were introduced for efficiently finding and generating them: rejection sampling using either a handcrafted or trained discriminator, and a variant of the local search algorithm from the previous chapter.

Results on navigation tasks were presented which showed that adversarial training can significantly improve the robustness of agents to adversarial examples. Notably, in-distribution robustness was improved by over 300% at the cost of marginally reduced reward and efficiency at solving the tasks. Furthermore, by also increasing the network capacity and training time of agents, their failure rate on highly perturbed out-of-distribution adversarial examples was more than halved.
Chapter 8

Influencing Emergent Agent Behaviour Using the Environment

In this chapter, the following question is addressed:

*Which emergent behaviours can be evaluated and influenced through variations in the environment?*

The previous two chapters investigated agents which were trained and evaluated on various navigation tasks. However, there are a range of other tasks and emergent behaviours which are of interest. To this end, this chapter investigates how agent behaviour can be efficiently evaluated and influenced on two distinct environments.

**Contributions** To summarise, the contributions of this chapter are as follows:

1. A World Agent is introduced which can generate worlds (i.e. environment settings) based on the performance and behaviour of agents by searching the latent space of a trained deep generative model (Section 8.3).

2. Worlds are discovered that lead to minimal and maximal agent performance on two distinct environments through efficient agent evaluation (Section 8.5).

3. The behaviour of agents is influenced to be more desirable, leading to safer and fairer agents at the trade-off of reward (Section 8.6).
8.1 Introduction

There are a wide range of environments and agent behaviours which are of interest in the application of reinforcement learning. For example, in the real-world it is desirable for self-driving cars to be safe and agents which interact with humans to be fair. However, the environments that agents interact with are often not nicely parametrised, nor is the relationship between the environment and the emergent agent behaviour obvious. Together, these make it challenging to search for environment settings and meaningfully modify them to get the desired agent behaviours to emerge.

To address this challenge, this chapter proposes a general method for strategically evaluating and influencing reinforcement learning agents. Concretely, a generative agent called the World Agent is introduced into the training and evaluating procedure of the reinforcement learning agents, giving it the ability to modify the distribution of worlds (i.e. environment settings) that the agents see. It does this by using deep generative modelling to learn a latent representation of the environment which it can then optimise in response to the behaviour and performance of the agents. As an example, such optimisations could be to maximise the reward of the agents or minimise the occurrence of an undesirable behaviour (such as aggressive actions).

The capabilities of the proposed World Agent are demonstrated by applying it to two distinct use-cases: a single-agent racing environment and a multi-agent resource gathering environment. For both, the World Agent is used to efficiently and consistently sample worlds which lead to minimal and maximal agent reward. Notably, it finds challenging worlds which cause agents to perform poorly, as well as highlight simple worlds where the agents perform best. In addition, an analysis of two optimisation methods is included and their trade-off between performance and sample efficiency is discussed. To conclude, the World Agent is used to update the training distribution of the agents based on their performance, influencing their learned behaviour to be safer and fairer at the cost of reward.
8.2 Setting

Throughout this chapter, the ability to evaluate and influence agent behaviour is investigated on two distinct environments. In this section, both of these environments and their corresponding agents are described.

8.2.1 Environments

To thoroughly investigate the World Agent and its capabilities, two different environments are considered: (1) the Particle Racing environment containing single-agent driving tasks, and (2) the Resource Harvest environment containing multi-agent social dilemmas. These are both outlined below with full details available in Chapter 4.

8.2.1.1 Single-Agent: Particle Racing

The single-agent driving tasks take place in the Particle Racing environment, with the agent tasked with driving around procedurally generated tracks as quickly as possible. Notably, the agent is positively rewarded proportional to its speed, negatively rewarded for crashing (i.e. leaving the track), and receives a partial observation of its local surroundings in addition to its velocity and position. In Figure 8.1 below, an example of such a task is visualised alongside the agent’s visual observation.

Figure 8.1: Driving task. (a) Example procedurally generated track where the blue circle represents the agent and the white dashed box indicates the agent’s visual observation window. (b) Raw and processed frames corresponding to the agent’s visual observation from (a).

The motivation behind using the Particle Racing environment in the context of this chapter is two-fold. First, the generated driving tasks are intuitive, with the spatial...
structure of the task (i.e. layout of the track) having an understandable impact on the expected behaviour of the agent. For example, slow down when approaching a corner and speed up when on a straight. These intuitive responses further help to highlight how changes to the distribution of the task’s structure can impact the agent’s behaviour. Second, the procedural generator for creating the tracks is open-source [99], has been used in related work [100], and is not parametrised. As a result, it presents an opportunity to demonstrate the capabilities of the proposed World Agent in encoding and optimising the generated tracks which was previously not possible.

8.2.1.2 Multi-Agent: Resource Harvest

The multi-agent social dilemmas take place in the Resource Harvest environment, with each of the four agents individually tasked with collecting resources in a procedurally generated level. Notably, while agents are positively rewarded for collecting resources, the recovery rate of each resource is proportional to the number of nearby resources. As a result, there exists a trade-off for the agents between collecting all of the resources before the other agents and waiting for the resources to recover in order to sustainably collect them over the entire episode (hence, a social dilemma).

In Figure 8.2 below, an example of such a social dilemma is visualised alongside each of the four agents’ observations.

![Figure 8.2: Social dilemma.](image)

(a) Example social dilemma.  
(b) Partially observable agent observations.

Figure 8.2: Social dilemma. (a) Example procedurally generated level with three rooms enclosed by walls (grey) and containing resources (green). Four agents are represented as red and blue, with the blue agent’s visual observation window highlighted by the dashed white square. (b) Frames corresponding to each of the agents’ north-facing visual observations from (a).
The motivation behind using the Resource Harvest environment in the context of this chapter is two-fold. First, the multi-agent nature of the environment allows for complex social behaviours to emerge which can be influenced and quantified [78]. For example, the conflict between agents as they compete for resources and the resulting (in)equality that emerges. Second, it has been recently demonstrated that the behaviour of agents and the resulting social outcomes can be influenced through changes to the environment’s/task’s structure [78]. Notably, the authors modified the environment by hand, and therefore an exciting next step is to investigate whether this process can be automated using the World Agent proposed here.

8.2.2 Agents

In this chapter, two different deep reinforcement learning algorithms are used for the agents based on whether the environment contains only a single-agent (in the case of the Particle Racing environment) or contains multiple agents (in the case of the Resource Harvest environment). Here, these agents are briefly described with further references provided for full details.

8.2.2.1 Single Agent: ACKTR

For the single-agent Particle Racing environment, the ACTKR algorithm [101] is used based on the open-source PyTorch implementation [102] with its default settings. In total, the agent is trained for 10,000 episodes.

8.2.2.2 Multi-Agent: Deep Q-Network (DQN)

For the multi-agent Resource Harvest environment, each agent is independently controlled by its own Deep Q-Network [1]. The agents’ settings are based on those from related work on variants of the same environment [78] with the notable difference of a reduced capacity due to the smaller environment and observation size. In total, the agents are trained for 5,000 episodes.
8.3 World Agent

In this section, the World Agent is introduced and its application explained.

8.3.1 Overview

The proposed World Agent uses a deep generative model to encode the space of environment settings into a searchable latent space. Using this, the World Agent is able to adapt the distribution of environment settings (referred to as ‘worlds’ throughout this chapter) which the agents see during training and evaluation based on how they perform across a range of different metrics.

To help explain the World Agent, its application is separated into the following three stages which are visualised in Figure 8.3 and expanded upon in this section:

1. Generate Worlds: Sample worlds using the pre-trained deep generative model.

2. Train Agents: Train reinforcement learning agent(s) on the sampled set of generated worlds.

3. Optimise Worlds: Iteratively generate and optimise worlds to maximise a specified agent-based metric.

![Figure 8.3: World Agent overview. Three stage application of the World Agent.](image)

(a) Generate worlds via sampling generator.  
(b) Train agents on sampled worlds.  
(c) Optimise worlds given trained agents.

Figure 8.3: World Agent overview. Three stage application of the World Agent.
(a) Stage 1 – Generate Worlds: sample latent vectors (z) from the latent space as input to the generator \( G \) to produce a set of worlds \( (w_1, ..., w_M) \) from world distribution.  
(b) Stage 2 – Train Agents: use sampled set of worlds (W) to train RL agents A \( M_i = M \).  
(c) Stage 3 – Optimise Worlds: iteratively sample the latent space to find \( z^* \) using optimiser \( O \), where \( z^* \) is an optimal point in the latent space which maximises the World Agent’s objective. Notation: \( z_i \) corresponds to the latent vector of world \( w_i \) and the World Agent’s evaluation of \( w_i \) is given by the metric \( M_i \).
As will be seen later, one iteration through these three stages can be used to evaluate agents (see Section 8.5), with additional iterations making it possible to influence the emergent behaviour of agents towards desired behaviours (see Section 8.6).

### 8.3.2 Stage 1: Generating Worlds

The first stage of the World Agent’s application is the generation of worlds. Here, rather than individually optimising every aspect of a world (for example at an individual pixel level), a deep generative model is used to compress the complex distribution over the world space $\mathbf{W}$ into a tractable distribution over the latent space $\mathbf{z}$. This makes it possible to efficiently optimise the environment by searching within this lower dimensionality latent space. In Equation 8.1 below, it is shown how a world $\mathbf{w}_i$ can be sampled from the latent space by passing a sample $\mathbf{z}_i$ to the generator $\mathbf{G}$, where $\theta_g$ are the generator’s weights:

$$\mathbf{w}_i = \mathbf{G}(\mathbf{z}_i; \theta_g), \quad \mathbf{z}_i \sim q(\mathbf{z} | \theta_e) \quad (8.1)$$

To learn $\theta_g$ and $\theta_e$, a Variational Autoencoder (VAE) is trained on a dataset of worlds created by a handcrafted procedural generator. As such procedural generators are typically rule-based or involve few parameters, they are challenging to optimise. Deep generative models, like VAEs, address this by learning to compress the complex parameter set of the procedural generator into a tractable distribution over the latent space, making it possible to optimise worlds efficiently. Notably, a VAE was chosen (over a GAN for example), due to their recent success in learning World Models [100]. Presented in Figure 8.4a is an example of how the VAE is trained on procedurally generated tracks for the Particle Racing environment.

After training the generative model, the learned decoder is used to form the generator $\mathbf{G}$ as demonstrated in Figure 8.4b. Here, a sample $\mathbf{z}_i$ is passed to the decoder which produces a sample world. This is followed by an additional processing step to ensure the world is valid for the environment (for example, being solvable) and produces the output world. Section 8.4.1 describes the process for each environment.
8.3.3 Stage 2: Training Agents

The next stage in the World Agent’s application is the training of the reinforcement learning agents which interact in and with the generated worlds. Here, each agent tries to learn a policy which maximises their own individual total expected discounted reward across the provided distribution of worlds. Importantly, the reward function of the agents does not have to align with the World Agent’s objective, making it possible for worlds to be optimised for a range of different metrics irrespective of the reinforcement learning agents’ objectives.

Throughout the three stages, it is assumed that this is the only stage during which agents can learn and therefore change their behaviour over time. However, while learning agents are considered throughout, rule-based and pre-trained agents can also be used (with this stage therefore being omitted). Outside of this stage, when agents are being evaluated on worlds, their policies are assumed to be fixed.

For a given world $w_i$ and its latent representation $z_i$, the behaviour of the agents is summarised in their corresponding set of trajectories $T_i$. These can then be quantified into various metrics $M$ which are discussed in Section 8.3.5.

8.3.4 Stage 3: Optimising Worlds

The third and final stage is the optimisation of worlds based on the behaviour of agents. To find optimal worlds, an optimiser is used to sample from the latent space
of the generator. Samples are selected with the goal of maximising the World Agent’s objective function (i.e. metric $\mathcal{M}$), where the optimisation task:

$$z^* = \arg\max_z \mathcal{M}(G(z; \theta_g), T(z)),$$  

(8.2)

is over the latent space. This objective function depends on the behaviour of the agents (the trajectories $T$) and the generated worlds ($\{G(z_i; \theta_g)\}_{i=1}^M$), both of which are functions of the latent space.

For this optimisation task there exists a trade-off between sample efficiency and performance. If sampling is expensive (for example, if an episode takes a significant amount of time or resources), then it is preferable to minimise the number of samples required. On the other hand, if sampling is fast and cheap then a more optimal method is preferred. To investigate this, two different optimisation methods are compared in Section 8.5.3.

8.3.5 Metrics

The standard metric considered and optimised for is the reward that the agents receive, with the World Agent’s objective set to either maximising or minimising this reward. For the single-agent Particle environment, this is simply the agent’s total reward received in an episode, while for the multi-agent Resource Harvest environment this is the sum total of rewards received by all agents in an episode (i.e. group or collective reward).

It is also possible to optimise for other environment-specific metrics as will be shown later in Section 8.6. For example, the crash rate in Particle Racing and conflict in Resource Harvest.

8.4 Setup

In this section, the setup of the World Agent’s generator is described for each environment. This is followed by an overview of the two optimisation methods investigated as the World Agent’s optimiser.
8.4.1 Generator

To encode the space of worlds, a Variational Autoencoder (VAE) is trained on a procedurally generated dataset and the learned decoder is used in the World Agent’s generator. Both the encoder and decoder of the VAE are parametrised by 2-layer fully-connected neural networks with 1024 & 512 hidden units and $\dim(z) = 10$.

**Particle Racing**  The generator for the Particle Racing environment is setup as follows. First, the procedural generator from the Car Racing environment [42] is used to generate a dataset of 10,000 tracks. Next, these generated tracks are each compressed to $64 \times 64$ in size and used to train the VAE. The resulting decoder is then used as the World Agent’s generator with an additional processing step to ensure the reconstructed tracks are valid. Here, the decoder’s output is mapped to the most similar track in the original dataset as calculated by the minimal pixel-wise distance.

**Resource Harvest**  The generator for the Resource Harvest environment is setup as follows. First, the procedural generator introduced in Section 4.5.1 is used to generate a dataset of 10,000 levels. Next, these generated levels are used to train the VAE. The resulting decoder is then used as the World Agent’s generator with an additional processing step to ensure consistency. Specifically, the decoder’s output is processed so that the number of resources is the same across all levels (with extra/missing resources randomly removed/added as necessary).

8.4.2 Optimiser

Two different optimisation algorithms are used and compared for optimising worlds. For both, each world is evaluated 8 times and the returned metrics averaged.

**Covariance Matrix Adaptation Evolution Strategy (ES)**  The first optimisation algorithm, and main one used throughout this chapter due to its effectiveness, is the Covariance Matrix Adaptation Evolution Strategy [85, 103]. In particular, the open-source implementation [104] is used with a population size of 21.
Bayesian Optimisation (BO)  The second optimisation algorithm investigated is Bayesian Optimisation which is implemented using GPyOpt [105] with default settings and compared to ES in Section 8.5.3.

8.5  Evaluating Agents

In this section, the performance of agents trained on randomly sampled worlds is evaluated for both environments. First, the results from evaluating on randomly sampled worlds are presented. This is then followed by a comparison with an evaluation of the same agents using the introduced generative World Agent to strategically optimise worlds according to the behaviour of the agents.

Note, to improve readability all reported metrics are normalised to be between 0 and 1. To support this, an explanation of the observed agent behaviour is included to demonstrate that the agents have learned a sensible policy.

8.5.1  Evaluating Agents on Randomly Sampled Worlds

To begin, the trained agents are first evaluated on 1,000 randomly sampled worlds for both environments, with the resulting rewards visualised in Figure 8.5.

(a) Histogram of agent rewards.  
(b) Box plot of agent rewards.

Figure 8.5: Agent performance on randomly sampled worlds. Distribution of agent rewards on both environments presented as (a) a histogram, and (b) a box plot.

As can be seen, agent performance is notably different for each of the environments. For Particle Racing, a vast majority of the sampled worlds have a high reward,
yielding an overall average reward of 0.87. This corresponds to the agent successfully navigating multiple loops on a diverse range of tracks, therefore suggesting that the agent has learned a sensible policy. In contrast, the results for Resource Harvest are consistently more varied, yielding an overall average reward of 0.61 distributed predominantly between 0.2 and 0.9. This large spread is likely due to several confounding factors including the stochastic nature of the environment, the emergent interactions between four independent agents, and – importantly – the strong influence of the resources’ spatial arrangement.

Together, these results highlight the potential issues that need to be considered when drawing conclusions about the behaviour of agents which have been trained on randomly generated worlds. By training and evaluating agents on such worlds, the biases in the procedurally defined distribution can lead to misleading results which make it difficult to understand where and how agents are performing – a point which was initially raised in Chapter 6 and investigated further in Chapter 7.

Specifically, in the Particle Racing environment the average world has few corners and therefore the optimal policy is to drive quickly most of the time. This leads to an agent which performs well on average, but can fail easily on the few rarer worlds which contain more challenging corners. Furthermore, in the Resource Harvest environment the strong compounding influences of the resources’ spatial arrangement (i.e. many resources grouped together will respawn faster) are rarely seen during training and evaluation. As a result, the agents rarely experience such worlds and therefore do not learn a policy for performing well in these (arguably) more relevant worlds.

8.5.2 Evaluating Agents on Strategically Optimised Worlds

Next, the generative World Agent proposed in this chapter is used to evaluate the trained agents by strategically optimising worlds based on the performance metrics of the agents. In particular, the World Agent is used to efficiently finds worlds where the agents perform worst (minimum reward) and perform best (maximum reward) for both the Particle Racing and Resource Harvest environments. These results are then compared to the reported average reward from the previous section.
8.5.2.1 Particle Racing

Presented in Figure 8.6 are the strategically optimised worlds found by the World Agent which minimise and maximise the trained agent’s reward on the Particle Racing environment. In addition, the reported average reward for evaluating the same agent on randomly sampled worlds is also included (from Figure 8.5).

![Particle Racing](image)

(a) Minimum world. (b) Agent rewards. (c) Maximum world.

Figure 8.6: **Evaluation results on Particle Racing.** Optimised worlds which minimise (left, 0.13) and maximise (right, 0.99) agent reward for the Particle Racing environment, as well as the reported average reward (0.87).

As can be seen, the World Agent is able to find rare worlds which consistently lead to the agent crashing. Specifically, the worst discovered world leads to a minimal reward of only 0.13, an 85% reduction from the expected average reward of 0.87. In terms of agent behaviour, the agent is reduced from being able to complete several loops of the track within an episode on average to consistently crashing before completing a single loop. This finding therefore suggests that there are certain localised spatial configurations that the agent has not learned to solve sufficiently, and further supports the worst-case analysis findings presented in Chapter 6.

In contrast to this discovered minimal world, the optimised world where the agent performs best has no surprising corners, and is instead a simple rectangular shape. As a result, the reward is near optimal at 0.99, a 14% increase from the expected average reward of 0.87. Here, the difference in behaviour is that the agent is able to consistently complete multiple loops of the track within a single episode without crashing, and is also able to do so at a higher speed than average.
8.5.2.2 Resource Harvest

Presented in Figure 8.7 are the strategically optimised worlds found by the World Agent which minimise and maximise the trained agent’s reward in the Resource Harvest environment. In addition, the reported average reward for evaluating the same agents on randomly sampled worlds is also included (from Figure 8.5).

![Resource Harvest](image)

<table>
<thead>
<tr>
<th>Reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Min</td>
</tr>
<tr>
<td>0.05</td>
</tr>
</tbody>
</table>

(a) Minimum world.  
(b) Agent rewards.  
(c) Maximum world.

Figure 8.7: Evaluation results on Resource Harvest. Optimised worlds which minimise (0.05, left) and maximise (0.92, right) agent group reward for the Resource Harvest environment, as well as the reported average reward (0.61).

As can be seen, the World Agent is able to find a spatial arrangement of resources and walls such that the reward of the agents is heavily diminished. Specifically, the worst discovered world leads to a minimal reward of only 0.05, a 92% reduction from the expected average reward of 0.61. This was achieved by spreading out the resources so that they do not gain the recovery bonus from nearby resources, therefore reducing the total number of resources which can be harvested within the timespan of an episode.

In contrast to this minimum world, the optimised world where the agents perform best (i.e. reward is maximised) has few walls and groups the resources together. As a result, all of the resources benefit from the recovery bonus leading to more which can be harvested within an episode. Consequently, the reward is notably higher at 0.92 which is a 51% increase from the average reward of 0.61.
8.5.3 Analysis of Optimisation Methods

The key component of the World Agent is the optimisation process by which worlds are optimised based on the performance of the agents. Due to the World Agent’s modular design, various different optimisation methods are supported and can therefore be interchanged based on their trade-offs.

To investigate the trade-offs between optimisation methods, Figure 8.8 presents the distribution of agent rewards on worlds sampled by each optimiser with the World Agent’s objective set to finding worlds which minimise the agent’s reward in the Particle Racing environment. Specifically, the sample efficiency and performance of the Covariance Matrix Adaptation Evolution Strategy (ES) algorithm and Bayesian Optimisation (BO) are compared after 50 samples and 1000 samples. Additionally, the baseline of randomly sampling worlds (Random) is also included for comparison.

As can be seen, both the Evolution Strategy (ES) and Bayesian Optimisation (BO) are able to find a large number of worlds which lead to significantly lower reward than average. Interestingly, and most notably, BO initially performs the best after fewer samples (i.e., more mass on lower rewards, therefore more sample efficient) while
ES barely improves upon Random. However, as the number of samples increases, the performance of ES improves relative to BO, eventually finding more optimal solutions than those found by BO and performing significantly better than Random.

From these results, it can be concluded that there is a useful trade-off between performance and sample efficiency which can be employed based on the cost of sampling. In particular, if the cost of evaluating agents on worlds is expensive (and therefore few samples can be obtained), then more sample efficient methods such as Bayesian Optimisation are preferable. Conversely, if evaluation is cheap then Evolution Strategies offer a strong alternative as they are able to find more optimal solutions while requiring many more samples.

8.6 Influencing Agents

In the previous section, the introduced World Agent was shown to be able to efficiently optimise and find worlds which lead to specific agent behaviours. A natural question which therefore arises from this is whether these worlds can be incorporated back into training to reinforce these behaviours further.

To investigate whether emergent agent behaviour can be influenced in this way, this section uses the World Agent to adjust the training distribution of the agents by sampling worlds which maximise a desired behaviour. Referred to as Influenced Training, the training process involves first constructing new training datasets which have the desired agent behaviour (specific to each environment), and then training fresh agents on these new datasets. This contrasts to the Standard Training method where agents are simply trained on randomly sampled worlds.

The rest of this section investigates the effectiveness to which Influenced Training can encourage agents to be safer and fairer in the Particle Racing and Resource Harvest environments respectively. Furthermore, to demonstrate that the behaviours learned from Influenced Training can transfer, all evaluation results are reported on the same distribution of randomly sampled worlds as the previous experiment and compared to the Standard Training method.
8.6.1 Influencing Agents to be Safer

As discovered in Section 8.5, there exists many possible worlds in the Particle Racing environment where the agent fails to safely traverse the track within an episode due to the agent crashing. For safety-critical applications such as self-driving cars, this behaviour is not desirable and therefore should be minimised. To this end, this section investigates whether Influenced Training can be used to train safer agents which crash less frequently. In particular, by exposing the agents to worlds during training which cause crashes more frequently as discovered by the World Agent.

8.6.1.1 Setup

**Metrics** In addition to the reward received by the agent within an episode, the agent’s *Crash Rate* is also recorded. Here, a crash is defined as having occurred when the agent leaves the track (visualised in grey) and touches the grass (visualised in green), with the crash rate therefore corresponding to the proportion of episodes where a crash occurs.

**Influenced Training Dataset** In an attempt to reduce this undesirable behaviour, the World Agent is used to discover worlds where the agent is likely to crash, using such worlds to form the new training dataset. Specifically, the World Agent’s objective is set to maximise the driving agent’s crash rate, with the worlds exceeding a 50% crash rate used to form the Influenced Training dataset.

8.6.1.2 Results

Presented in Figure 8.9 are the influenced training results on Particle Racing with the objective of training safer agents. In particular, two example influencing worlds are shown in Figure 8.9a alongside the agent’s performance results in terms of reward and crash rate for each training method in Figure 8.9b.

As shown on the left, the influencing worlds which lead to a high crash rate tend to follow the same structure as those seen in Section 8.5 – containing a number of sharp corners and unexpected deviations in the track. On the right, agent performance
Figure 8.9: **Influenced training results on Particle Racing.** (a) Two example worlds which were discovered using the generative World Agent and included in the influenced training dataset. (b) Crash rate and reward using each training method, evaluated on the same distribution. Notably, training on influencing worlds (which feature a large number of sharp corners), leads to a lower crash rate at the expense of a lower reward.

Trained on these influencing worlds is compared to training on randomly sampled worlds. Notably, Influenced Training significantly lowers the crash rate from 0.11 to 0.04 when compared to Standard Training, reducing the frequency of crashes from every 9 episodes to every 25 episodes (an 170% improvement). This improvement comes at the cost of a 5% reduction in reward due to the agent driving slower and therefore taking longer to complete a loop of the track. In the context of safety, the newly trained agent’s behaviour is more desirable as it crashes less frequently and is therefore safer.

### 8.6.2 Influencing Agents to be Fairer

In a number of multi-agent systems, situations can arise where certain agents have more power than their peers (such as by reacting faster or having access to additional capabilities). Consequently, this can lead to the more powerful agents receiving a larger share of the available resources at the expense of the weaker agents, an outcome which may not always be appropriate. As an example, in many human-agent interactions it may be desirable for the humans to receive an equal share of the resources despite the potentially superior capabilities of the agents. To this end, this
section investigates whether Influenced Training can be used as a method to train a more powerful agent to behave more fairly in the Resource Harvest environment.

8.6.2.1 Setup

Environment Changes To facilitate this investigation, the Resource Harvest environment is updated such that one of the four agents has access to an additional action which can be used to monopolise the resources. In particular, the agent is allowed to perform the \textsc{Tag} action which fires a 3-block wide beam forward from the agent, tagging out any other agent hit from the episode for 25 steps. Consequently, this more powerful agent is encouraged to tag out other agents as much as possible, reducing competition and therefore privatising the resources for themselves.

Metrics In addition to the group reward received by the agents within an episode, two social outcome metrics based on [78] are also recorded:

1. \textit{Conflict:} the average number of steps where agents are tagged out.
   \[
   C = \frac{\sum_{i=1}^{N} \sum_{t=1}^{T} I(a_i)}{NT} \quad \text{where} \quad I(a) = \begin{cases} 1 & \text{if } a = \text{tagged out}, \\ 0 & \text{otherwise}. \end{cases}
   \]

2. \textit{Equality:} the distribution of rewards across agents.
   \[
   E = 1 - \frac{\sum_{i=1}^{N} \sum_{j=1}^{N} |R_i - R_j|}{2N \sum_{i=1}^{N} R_i} \quad \text{(i.e. 1 - Gini coefficient)}
   \]
   where \(N\) is the number of agents, \(T\) is the duration of an episode, and \(R_i\) is the total reward received by agent \(i\) in an episode.

Influenced Training Dataset In an attempt to reduce this undesirable behaviour, the World Agent is used to discover worlds where the impact of the more powerful agent’s \textsc{Tag} action is minimised, using such worlds to form the new training dataset. Specifically, the World Agent’s objective is set to minimise conflict, and the resulting worlds with minimal conflict are used to form the Influenced Training dataset.
8.6.2.2 Results

Presented in Figure 8.10 are the influenced training results on Resource Harvest with the objective of training fairer agents. In particular, two example influencing worlds are shown in Figure 8.10a alongside the agents’ performance results in terms of conflict, (group) reward, and equality for each training method in Figure 8.10b.

Figure 8.10: Influenced training results on Resource Harvest. (a) Two example worlds which were discovered using the generative World Agent and included in the influenced training dataset. (b) Conflict, reward, and equality using each training method, evaluated on the same distribution. Notably, training on influencing worlds – which isolate the more powerful tagging agent (indicated in blue) – leads to lower conflict and more equality at the expense of a lower group reward.

As shown on the left, the influencing worlds which lead to low conflict tend to isolate the more powerful tagging agent from the other agents using the walls (represented as the blue square in the top left of the world). On the right, the performance of agents trained on these influencing worlds is compared to those trained on randomly sampled worlds. Notably, Influenced Training results in significantly lower conflict (from 0.51 to 0.05, a 90% reduction) and higher equality (from 0.33 to 0.78, a 136% increase) compared to Standard Training. This arises as the tagging agent doesn’t learn to associate its aggressive TAG action with increased reward, leading to a reduction in its use. These improvements come at the cost of reduced group reward (from 0.68 to 0.46, a 32% reduction) as the resources cannot be privatised and are therefore less likely to recover within an episode. In the context of fairness, the newly
trained agent’s behaviour is more desirable despite its increased capabilities, with the trade-off of lower overall group reward.

8.7 Summary

This chapter proposed a general method for finding and generating environment settings (referred to as worlds throughout). Through the optimisation of worlds based upon agent performance, a range of emergent behaviours were evaluated and reinforced by adapting the distribution seen by the agents.

The key contribution was the proposed World Agent followed by the evaluation and influencing results from its application. The method itself used a deep generative model to encode the procedurally generated environments, facilitating its application to two distinct environments and three different metrics based on agent behaviour.

Results were presented on both single- and multi-agent procedurally generated environments showing that the World Agent can efficiently and consistently discover worlds which minimise and maximise agent performance. Furthermore, it was also demonstrated that the emergent behaviour of agents can be influenced using the same method by adapting the training distribution of agents. Through this training procedure, agents were influenced to be safer and fairer at the cost of reward, highlighting the method’s ability to optimise for alternative measures of success other than reward – such as social benefit – by varying the environment.
Chapter 9

Conclusion and Future Work

9.1 Summary of Contributions

This thesis investigated the use of procedurally generated environments for strategically training and evaluating deep reinforcement learning agents. The key contributions of the thesis addressed the following four research questions around which the broader investigation was structured:

1. How does training on a procedurally generated environment affect an agent’s performance on unseen settings at test time?

2. Where do agents trained on procedurally generated environments fail to generalise?

3. How can adversarial examples be incorporated into training to improve the performance of agents?

4. Which emergent agent behaviours can be evaluated and influenced through variations in the environment?

By addressing these research questions, this thesis demonstrated the benefits of procedural generation, highlighted the failures that can emerge from their use, presented methods for alleviating such failures, and introduced a general method for evaluating and influencing agent behaviour. In the rest of this section, the specific contributions of each chapter are summarised.
Chapter 5 began the technical content of the thesis by demonstrating the benefits of training agents on procedurally generated environments. Specifically, by supplying a diverse range of environment settings during training, the agents were encouraged to learn a more general strategy rather than overfitting to specific settings. This led to improved agent performance on unseen settings at test time, addressing Research Question 1.

The takeaway contributions of this chapter were as follows. First, it was shown that agents only need to be trained on a fraction of the full space of possible environment settings to achieve strong average-case performance on unseen settings at test time. Next, by investigating navigation tasks in two contrastingly complex environments, this finding was demonstrated to hold across varying levels of environmental complexity including longer episode length, higher resolution observations, and an increased number of possible actions. Finally, training on randomly sampled setting was shown to be a simply yet effective method for achieving strong performance on unseen settings, performing comparably with an existing curriculum-based approach.

Chapter 6 highlighted the failures that can emerge when agents are trained on procedurally generated environments. This was achieved by performing an in-depth analysis into the worst-case performance of state-of-the-art deep reinforcement learning agents. Notably, examples were found where agent performance was significantly reduced due to a failure to generalise, addressing Research Question 2.

The primary contribution of this chapter was the introduced definition of adversarial environment settings alongside an intuitive algorithm for finding such settings that lead to failure. By applying this algorithm, it was shown that not only do catastrophic failures exist for state-of-the-art navigation agents, but also that simple catastrophic failures exist which are robust to various agents while still being easily solvable by humans. Furthermore, it was demonstrated that settings which lead to failure transfer across agents with different training hyperparameters and architectures, a finding that is consistent with the complementary literature on adversarial examples in computer vision.
Chapter 7 presented an adversarial training approach for alleviating the failures to generalise which were first highlighted in the previous chapter. In particular, the robustness of agents was improved by adversarially augmenting their training set with adversarial environment settings. This led to improved agent performance on both in-distribution and out-of-distribution examples, addressing Research Question 3.

The contributions of this chapter were threefold. First, the definition of adversarial environment settings was expanded to cover in-distribution and out-of-distribution examples. This was followed by the introduction of two methods for finding and generating such examples – an efficient search algorithm for finding in-distribution examples (which was shown to be better than random sampling), and a consistent local search procedure for generating progressively further out-of-distribution examples. Finally, it was demonstrated that adversarially augmenting an agent’s training set can significantly improve their in-distribution and out-of-distribution robustness, even against highly perturbed adversarial examples. However, this comes at the cost of overall reward, efficiency at solving the task, and considerably longer training time.

Chapter 8 concluded the thesis by proposing a general method for automatically finding and generating environment settings (i.e. worlds). Through the optimisation of worlds based upon agent performance, a range of emergent behaviours were evaluated and reinforced by adapting the distribution seen by the agents. This led to trained agents exhibiting more desirable behaviours, addressing Research Question 4.

The key contribution of this chapter was the proposed World Agent and the results from its application. To demonstrate the method’s generality, two distinct environments were investigated alongside several agent behavioural metrics such as crash rate, fairness, and equality. Notably, it was shown that not only can worlds be automatically optimised to maximise these metrics, but also that the resulting agent behaviours can be further reinforced by incorporating such worlds back into training. Using this training procedure, agents were influenced to be safer and fairer at the cost of reward, highlighting the method’s ability to optimise for alternative measures of success other than reward – such as social benefit – by varying the environment.
9.2 Future Work

In this section, several promising directions for future work are discussed.

9.2.1 Training Robust Agents

One of the motivating use-cases presented in this thesis was the deployment of agents in safety-critical domains. To this end, Chapters 6 and 7 presented analysis and training methods which yielded positive results using illustrative environments. In order to extend these results to real-world safety critical domains, both the methods proposed and the environments considered will need to be scaled up. As a result, the following two directions are proposed for future work.

Continuous adversarial training. Chapter 7 demonstrated that adversarially augmenting an agent’s initial training set with more challenging and adversarial examples can reduce the agent’s failure rate. Given this success, a logical next step is to investigate a continuous adversarial training setup where adversarial examples for the agent are continuously discovered during training and then incorporated into the training set as the agent learns. The hope here is that the agent’s failure rate would constantly decrease as training progresses through discovered adversarial examples, eventually reaching a failure rate that is low enough for the agent to be deployed in safety critical applications.

More realistic training environments. To help bridge the gap between simulated results and real-world applicability, more realistic environments can be used to train and evaluate agents. For example, TORCS [47] would logically extend the driving environment used in Chapter 8, with the long-term goal of using more complex simulators such as CARLA [106] and AirSim [107]. In addition, as a consequence of using more sophisticated environments, this direction will also require further work on developing efficient methods for finding adversarial examples.
9.2.2 Open-Ended Environments

Throughout this thesis, the environments used to train agents were all procedurally generated using a fixed set of predetermined parameters and rules. Notably, these were designed by humans each with a specific task in mind (such as navigating through a maze and driving around a track). As a result, the space of possible environmental challenges was constrained, by both human design and the environment’s encoding, and consequently the diversity and complexity of the emergent agent behaviour was also similarly constrained. To help alleviate these constraints, future work on open-ended environments is proposed along the following two directions.

**Designing open-ended tasks.** While the environments were constrained to a specific task by human design, they do not have to be. Instead, tasks can be designed which are both open-ended and have a high complexity ceiling, thereby supporting the emergence of more sophisticated agent behaviours and solutions. For example, rather than assigning reward based upon the achievement of specific objectives, agents can instead be rewarded for staying alive [50]. In combination with an environment containing a variety of actions and tasks which interact with the lifetime of agents, complex behaviours could emerge and be further influenced using methods such as those presented in Chapters 7 and 8 of this thesis.

**Developing more expressive environment representations.** To support open-ended environments which can still be optimised, a more expressive environment representation is required. Specifically, an encoding which has the capacity to represent an increasingly diverse set of tasks as training progresses. One approach for this is to represent the environment as a directed graph of mechanics which can be progressively modified, grown, and pruned throughout training. For example, [49] generated a simple version of such a graph with their procedural generator to construct layouts of increasing complexity. Another promising approach is generate the environment using a neural network, similar to that done in Chapter 8, but additionally combined with reinforcement learning [108, 109].
9.2.3 Generating Effective Learning Environments

When training agents, it is not always possible for them to learn to solve tasks directly from scratch. Instead, a training curriculum may be necessary. In the context of this thesis, this can be achieved by allowing the environment to adapt with the agent, tailoring itself to the agent’s current behaviour and capabilities (as proposed in Chapter 8). This raises the question of how to generate effective learning environments. To this end, the following research directions are proposed for future work.

Predicting effectiveness. While automated curriculum learning has recently been used to train agents in procedurally generated environments, success has been limited. Notably, existing methods have typically relied on handcrafted measures of difficulty [14, 96] or only been usable in continuous domains [110]. To address this, search methods (such as those in Chapter 7) can be used to predict the training effectiveness of environment settings, with promising settings used as a training curriculum.

Open-ended training. Rather than training agents to solve known target tasks, training can instead be open-ended with the broad objective of producing increasingly sophisticated agents as training progresses. As an example, the POET algorithm [43] was recently proposed which simultaneously generates increasingly complex environments and agents that can solve them. Along this direction, there are numerous open questions on how to generate effective open-ended learning environments such as how to maximise and maintain both environmental complexity and behavioural diversity.

Extending to multi-agent settings. Following the multi-agent results presented in Chapter 8, an exciting direction for future work is to extend open-ended learning further by considering environments populated by multiple agents. In such settings, the presence of multiple learners competing and cooperating in the same environment can lead to an emergent autocurriculum of complexity and innovation as agents seek to outperform each other [111, 112]. By combining both agent and environment autocurricula, this long-term direction has significant potential for future research.
References


