Procedural Content Generation using Neural Networks

Boran Wang

BSc (Hons) Computer Science

Department of Computer Science

University of Manchester

Primary Advisor: Ke Chen

2024

Acknowledgements

This project cannot be completed without the constant support of my supervisor, my family and my friends. Firstly, I would like to thank my supervisor Ke Chen, for always keeping me on track, and offering me insights and direction when I struggled with reinforcement learning. Secondly, I would like to thank Ahmed Khalifa, Philip Bontrager, Sam Earle, Julian Togelius, for their inspirational paper on Procedural Content Generation via Reinforcement Learning (Khalifa et al., 2020) which laid the groundwork for this project. Finally, I thank my family and my girlfriend Tina, for giving me constant emotional support, inspirations, and bringing in new perspectives when I most needed them.

Abstract

This project investigates the integration of neural networks with procedural content generation (PCG) to automate and enhance the creation of dynamic game environments. Motivated by the limitations of manual content creation in game development, this research explores the feasibility of using neural networks for PCG. The project leverages recent advancements in machine learning, game design, and GPUs, to develop a framework that utilizes both shallow and deep neural networks for generating game content.

The methodology involves the design and implementation of neural network models, a detailed comparison of different architectures and learning algorithms, and the development of a custom game environment in Unity for real-world application and testing. The evaluation of the project outcomes was conducted through a series of experiments, focusing on the validity, difficulty, and playability of the generated content across different agents.

This project demonstrates that neural networks can significantly contribute to the efficiency and creativity of PCG, offering a scalable solution to the challenges of traditional game development processes. The project highlights the potential of combining neural networks with PCG to create more engaging and varied gaming experiences, suggesting a promising direction for future research and development in game design and AI.

Table of Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Objectives 3

1.3 Report Structure 4

2 Background **5**

2.1 Procedural Content Generation 5

2.2 Neural Networks 7

2.3 Reinforcement Learning 8

2.4 Training methods 9

2.4.1 Genetic Algorithms 10

2.4.2 Proximal Policy Optimization 11

Policy gradient method 12

Clipped Surrogate Objective 12

2.5 Relevant Work 13

2.6 Unity 15

3 Methods **17**

3.1 Multilayer Perceptron, Shallow and Deep Neural Networks 17

3.2 Genetic Algorithm 19

3.2.1 Chromosome 20

3.2.2 Fitness Function 20

3.2.3 Population 20

3.2.4 Selection 21

3.2.5 Crossover 21

3.2.6 Mutation 23

3.3 Reinforcement Learning for PCG 23

3.3.1 Markov Decision Process 24

3.3.2 Proximal Policy Optimisation 25

3.3.3 Observation and feature extraction 26
3.3.4 Reward 28

4 System Design and Implementation 30
4.1 Game Design 30
4.1.1 Rules 31
4.1.2 Agent-Map Interactions 31
4.1.3 Map Evaluation 32
4.2 Unity and Genetic Algorithms 34
4.2.1 SNN Snake Agent 35
4.2.2 Reinforcement learning and ML-Agents 37
Convolution Layers 38
Fully Connected Layers 39
4.3 Agent Evaluation 39

5 Experiments 40
5.1 SNN Snake Agent 40
5.1.1 Evaluation of SNN Snake 41
5.2 DNN Snake Agent 43
5.2.1 Evaluation of DNN Snake 44
5.3 CNN Snake 46
5.4 CNN Turtle 48
5.4.1 Reward 49
5.4.2 Catastrophic Forgetting 52

6 Conclusion 54
6.1 Achievements 54
6.2 Limitation 54
6.3 Future work 56

Appendix A PPO Surrogate Object function 60

List of Figures

2.1 Dungeon crawler Rogue, 1980 6
2.2 Simple neural network example and terminology, Figure adopted from (Sze et al., 2017) 7
2.3 An illustration of the multilayer perception architecture used in TD-Gammon's neural network. This architecture is also used in popular backpropagation learning procedures. Figure adopted from (Tesauro, 1995) 9
2.4 The system architecture for the PCGRL environment for content generation. Figure adopted from (Khalifa et al., 2020) 14
2.5 Sokoban level as 2D integer array. Figure adopted from (Khalifa et al., 2020) 14
2.6 Zelda generated examples using different representation for the same starting state. Figure adopted from (Khalifa et al., 2020) 15

3.1 Structure of networks and algorithms used in this by the agents that are in the demo of this project 17
3.2 Multilayer Perceptron with one hidden layer, which is used to train a selfdriving car in Unity (Berrocal, 2019) 18
3.3 Genetic Algorithm flowchart 19
3.4 Simplified multi-point crossover, where each number represent a weight on a Neural Network. In reality there are much more weights in each chromosome 22
3.5 Reinforcement Learning framework used in this project. 23
3.6 Example of an agent using raycasts to play Snake 27
3.7 Example of convolution layer and pooling layer 28

4.1 Screenshot of the game built for this project. The assets used in the game comes from Kenny's Tower Defense Kit (kenny, n.d.), which is Copyright free, and the animations are built using Unity's Particle System. 31
4.2 Grid Representation of the map, containing starting point for the Turtle Agent 32
4.3 Grid Representation of the map, containing starting point for the Turtle Agent 33
4.4 Flowchart of how the components interact with each other in the GA framework 34
4.5 All the important parameters in the GA framework 36
4.6 Example of how Unity interacts with scripts 36
4.7 Flowchart of how the MLAgents interact with the environment 37
4.8 Convolution layers used by the Turtle Agent 38

5.1 SNN Snake example 41
5.2 Training of 100 generations, with 10 chromosomes per generation. . . 42
5.3 Training of 100 generations, with 30 chromosomes per generation. . . 42
5.4 Architecture of the DNN Snake 43
5.5 Training result of five million steps 44
5.6 Map produced by both agents side by side 45
5.7 Training result after modifying reward. 46
5.8 Two maps generated by CNN Snake 47
5.9 Training result of the CNN Snake with updated reward function. . . . 47
5.10 Map Generated after updating CNN reward function 48
5.11 Example of how the turtle agent is used for map generation. The environment find the longest-shortest path generated by the turtle agent and use it as the map path. 49
5.12 Example of connecting components, the agent receives a reward when it connects two components together 50
5.13 Training result of the Turtle Agent 51
5.14 130 million steps, agent ignores the start state 51
5.15 Training result of the Turtle Agent 52
5.16 Example of Task A vs Task B 53
5.17 Result of the agent failing at 25 million steps 53
5.18 Result of the agent failing at 33 million steps 53

6.1 Result of an exploration focused agent, trained for 8 days 55

Chapter 1

Introduction

This project delves into the innovative intersection of procedural content generation (PCG) and neural networks, aiming to explore and expand the capabilities of artificial intelligence in game development. The essence of the project is to leverage neural networks to automate the creation of game content, thereby enhancing the depth and replayability of games. Overall, the core aspects of this project are:

- Developing map making agents capable of generating game content.
- Utilizing neural networks to interpret game environments and decision making.
- Exploring different neural network architectures and learning algorithms to optimize the efficiency and effectiveness of content generation.

Using raw pixel data and value based data from the environment as input, the goal of the agents is to learn the underlying pattern of what makes a map viable through trial and error. Agents would repeatably generate game maps and receive feedback from the environment using a reward system. The agents are evaluated based on the degree of over-fitting and the score of the each map, which is calculated using methods from Khalifa's work on Procedural Content Generation using Reinforcement Learning (Khalifa et al., 2020), which calculates a score based on the replayability, validity and difficulty of the map generated.

Neural networks have the potential to revolutionize game design and development. Using automating content generation, developers can significantly reduce the time

and resources required to create expansive game worlds. By integrating deep neural networks with PCG, this project offers a promising, scalable solution for creating adaptable game environments.

1.1 Motivation

The interest in procedural content generation (PCG) within game development is driven by its ability to enhance gaming experiences through the generation of varied and complex content without requiring a proportional increase in development resources. As the game industry continue to grow, with a total revenue of 262 billion across the globe and projected to rise 312 billion in 2027 (PwC, 2023), the limitations of manually creating game content become increasingly apparent. This is especially true for games that demand expansive worlds or highly customized player experiences.

Neural networks have shown great promise in their ability to process and interpret complex data patterns. Their application has extended into numerous areas, including image and language processing, and more recently, game development. The use of neural networks in PCG is inspired by the 2020 paper on PCG using reinforcement learning (Khalifa et al., 2020), which proposes a new content generation method that increases the efficiency of the content creation process, introducing a level of complexity and diversity that manually programmed methods might find difficult to match.

The hypothesis guiding this project is that by casting the problem of PCG as a Markov Decision Process, neural networks can make the process of generating game content more efficient by reducing the time and effort required. The increasing availability of high-performance computing, especially through GPUs, and the progress in neural network algorithms have made the application of neural networks to PCG a practical and appealing proposition. This project seeks to explore these possibilities, drawing inspiration from the work of entities like DeepMind and OpenAI, to contribute to the future of game development.

1.2 Objectives

The main goal of this project is to investigate the application of neural networks in procedural content generation for video games, by framing the process of map design as a iterative, learnable task. This involves several specific objectives aimed at understanding, developing, and evaluating neural network models in the context of PCG. The key objectives include:

- Building the base game: To create a game that serves as a test field for agent training. This test field should contain all the essential elements of a video game, and can demonstrate how the map making agents can be implemented in a games.
- Development of Neural Network Models: To create and implement neural network models designed for procedural content generation tasks. This involves both adapting existing neural network architectures and potentially developing new models to effectively address the specific needs of PCG in game development.
- Performance Comparison: To conduct an analytical comparison of various neural network architectures and learning algorithms, focusing on their performance in generating game content. This comparison aims to identify the most effective models and algorithms based on criteria such as efficiency, quality of output, and computational resource requirements.
- Model Evaluation and Optimization: Establish criteria for evaluating the quality of generated content and use these criteria to refine the neural network models and training environment. The objective is to optimize the models to enhance their capability to produce engaging, high-quality game content.

By achieving these objectives, the project explores innovative solutions in the field of game development, which can be used when few or no examples exist to train from. In this report, we explore three different architectures, which include shallow neural

networks, deep neural networks and convolution neural networks, and two network optimization methods, genetic algorithms and deep reinforcement learning. Agents are evaluated by the map they generated, and a score is given based on the map validity, playability and difficulty.

1.3 Report Structure

The report is organized into several key sections to provide a comprehensive overview of the project:

- Background research and literature review to set the context and highlight the significance of PCG and neural networks in game development.
- Detailed description of the project design, including the selection of neural network models, learning algorithms, and evaluation metrics.
- Discussion of the implementation process, challenges encountered, and solutions devised to overcome them.
- Evaluation of the project outcomes, comparison of different approaches, and analysis of the generated game content.
- Conclusion and reflection on the project findings, implications for future research, and the potential impact on the game development industry.

Chapter 2

Background

This chapter explores the connections of Procedural Content Generation (PCG), Neural Networks (NN), Reinforcement Learning (RL) and Genetic Algorithms (GAs) for game development. PCG has significantly advanced from its early use in games like "Rogue" (1980) to sophisticated systems that have shaped dynamic and expansive virtual worlds. Similarly, the evolution from Shallow to Deep Neural Networks moves towards more complex and capable AI systems. RL made significant progress with methods such as Proximal Policy Optimization (PPO), which enabling more stable and efficient learning. This project incorporates these advancements by employing state-of-the-art deep reinforcement learning methods and genetic algorithms to train neural networks within Unity, creating a scalable content generation method. Unity's environment and ML-Agents Toolkit facilitates the integration of these techniques to drive innovation in Procedural Content Generation for video games. The following sections explain how these elements are implemented in depth and the synergies they create in the context of this project.

2.1 Procedural Content Generation

Procedural Content Generation (PCG) is a technique used in game development to generate content algorithmically, which reduces development costs and increase replayability. Different PCG methods can be used to generate a wide range of content, including levels, maps, quests, items, and even story elements, which can offer

unique gaming experience without requiring game designers to manually create new content(Dahrén, 2021).

One of the earliest implementations of PCG can be traced back to "Rogue" (1980), a game in the dungeon crawler genre that utilized PCG for the creation of dungeon level, shown on figure 2.1. This technique has been adopted by a variety of dungeon crawlers, including "The Binding of Isaac" and "Dead Cells", marking the inception of the roguelike genre. This methodological approach to game design has encouraged the development of numerous games within the genre, which leverages PCG to enhance gameplay variety and replayability.

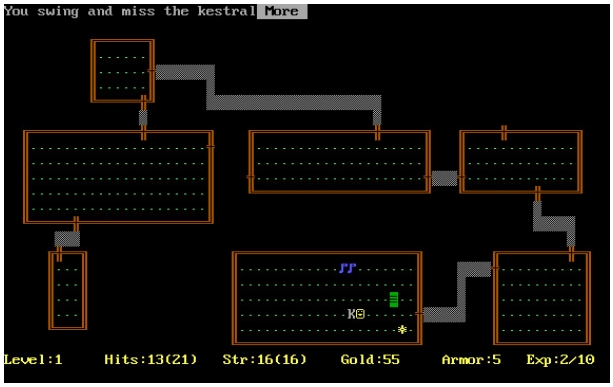


Figure 2.1: Dungeon crawler Rogue, 1980

The application of PCG has evolved from simply generating level randomly to sophisticated systems capable of creating complex, engaging and realistic environments. Games like “No Man’s Sky” have pushed the boundaries of PCG: by utilizing Perlin and simplex noise, it allows the player to explore 18 quintillion ($1.8e19$) unique planets and moons, each with its own wildlife, vegetation, and geological features. Similarly, a project detailed in an MIT blog post (Brummelen and Chen, 2018) explores the use of procedural generation and deep learning to create 3D worlds, using neural networks trained on terrain and color data to generate realistic landscapes. These examples demonstrate the advanced capabilities and potential of PCG in creating detailed and immersive digital environments.

2.2 Neural Networks

All of the agents trained in this project use Neural Networks for decision making. Inspired by the biological neural networks observed in animal brains, Neural Networks have the goal of replicating human cognitive and learning processes. The research on Neural Networks began with Shallow Neural Networks, which were characterized by their minimal number of layers between the input and output, shown on figure 2.2. These networks demonstrated the potential of machines to mimic basic human cognitive functions, such as pattern recognition and decision-making. Frank Rosenblatt's perceptron, developed in 1957, was one of the first neural networks and showed the ability to carry out basic pattern recognition tasks (Kanal, 2003). However, the

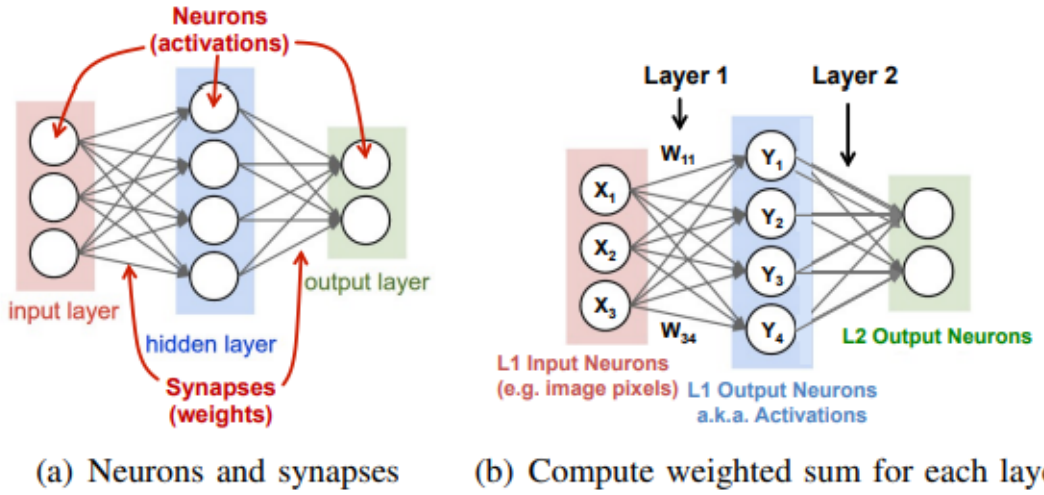


Figure 2.2: Simple neural network example and terminology, Figure adopted from (Sze et al., 2017)

limits of shallow neural networks were evident when researchers tried to tackle more complicated problems. Their simple structures were not able to capture the complex and hierarchical patterns present in real-world data. It resulted in a decrease in advancements due to decreased funding and doubt around AI research(Toosi et al., 2021). This was commonly referred as the AI winter.

Deep Neural Networks (DNNs) addresses this limitation by increasing the number of hidden layers and neurons in each layer. DNNs can learn complex and hierarchical

patterns in large data sets, which leads to advancements in image and audio recognition, natural language processing, and autonomous systems. In this project, both Shallow and Deep Neural Networks are used to perform content generation.

2.3 Reinforcement Learning

Reinforcement learning (RL) is implemented as one of the network weight optimization method, it is a machine learning method where an agent learns decision-making through interactions with an environment. Compared to supervised learning, this approach has the advantage of being able to train agents without training data, which is usually unavailable for independent video games. In Reinforcement learning, the agent performs actions in a specific state, it gets a reward or feedback from the environment, and transitions to a new state based on its actions. The agent's objective is to acquire a policy that maximizes the cumulative rewards over time. This process is typically referred as the Markov Decision Process.

The process of iterative learning had originated in the late 1980s with the creation of the fundamental temporal-difference (TD) learning algorithm. During this time, RL started to explore methods that utilizes the principles of Markov decision processes to gradually converge toward the best policy solutions. During this period, RL was defined by using basic algorithms such as tabular approaches and linear function approximators due to the limits in computational resources. In 1992, Gerald Tesauro applied TD-Lambda to create TD-Gammon, a program that reached an expert-human level in Backgammon(Tesauro, 1995). This marked an important milestone in the Reinforcement Learning history, which proved its ability to discover strategies in complex settings. Figure 2.3 contains the architecture used in TD-Gammon's neural network.

The modern era of reinforcement learning has been dominated by deep reinforcement learning (DRL), after DeepMind introduced the Deep Q-Network (DQN) method in 2013. DRL combines the decision-making structure of RL with the advanced function approximation abilities of deep neural networks, which allows agents to learn

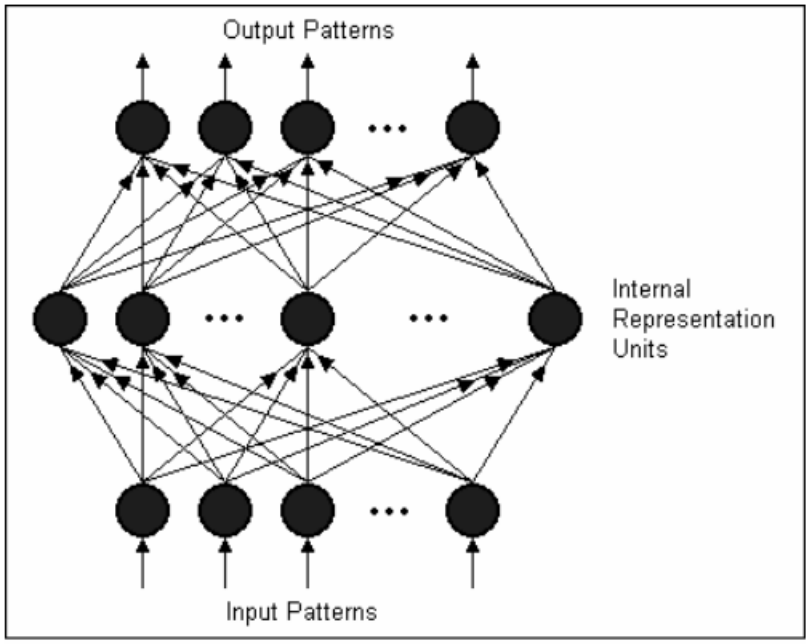


Figure 2.3: An illustration of the multilayer perception architecture used in TD-Gammon's neural network. This architecture is also used in popular backpropagation learning procedures. Figure adopted from (Tesauro, 1995)

directly from complex sensory inputs. This advancement is used to tackle previously unsolvable issues in several fields, such as playing at Atari games and progressing in robotics and autonomous systems (Osband et al., 2016). The transition to Deep Reinforcement Learning (DRL) is facilitated by significant improvements in computer resources, particularly GPUs, as well as progress in neural network structures and optimization techniques.

2.4 Training methods

This section provides a brief overview of the training algorithms used in this project. Genetic algorithms are widely used as a replacement for back-propagation in Unity to train agents that performs simple tasks, although it is not as effective compared to the best gradient methods, it can still be a promising learning method when implemented correctly (Schaffer, Whitley and Eshelman, 1992). Proximal Policy Optimization is a state-of-the-art reinforcement learning algorithm used to train deep neural networks, which are used to train agents capable of complex tasks. This

section only introduces the concept of these algorithms, detailed implementation will be included in the next chapter.

2.4.1 Genetic Algorithms

Genetic Algorithms (GAs) are a class of evolutionary algorithms inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). GAs simulate the process of natural selection. By encoding potential solutions as strings/chromosomes, we can simulate genetic combination, mutation, and selection to evolve to an optimal solution. In the context of training shallow neural networks, it can produce extremely variated results, and can avoid being trapped in a local minimum compared to algorithms like gradient decent (Ding, Su and Yu, 2011).

The Genetic Algorithm used in this project can be represented using the tuple $\langle Chr, Fit, Pop, Sel, Cross, Mut, \rangle$:

- Chr: Chromosome, encodes potential solutions to the problem, often using binary strings. This project uses shallow neural networks as representations of potential solutions.
- Pop: Population of chromosomes, with diversity among them to encourage exploration of a wide variety of potential solutions. Initially all solutions are randomly generated, each sub-sequence generation contains a mixture of selected and randomly generated chromosomes.
- Fit: Fitness function, used to evaluate how a given solution performs by calculating a fitness score for each chromosome in the population.
- Sel: Selection of the best chromosomes based on their fitness scores and removes chromosomes with low scores, reflecting the idea of natural selection.
- $Cross$: Crossover function, which is used to create new chromosomes for the next generation. It mimics biological reproduction by combining two parent chromosomes to generate an offspring.

- Mut: Mutation, introduces variation into the population by randomly altering the genes of each chromosome. This ensures genetic diversity within the population, preventing premature convergence to sub-optimal solutions.

This evolutionary algorithm begins by randomly generating a population of chromosomes. Each chromosome is evaluated and assigned a fitness score using the fitness function. The chromosomes with the highest fitness scores are selected for reproduction, generating new chromosomes for the next population. To prevent the algorithm from prematurely converging to sub-optimal solutions, each chromosome has a chance to mutate, which introduces variation into the population by randomly altering the genes of the chromosomes. This process of iterative learning is often used in Unity to train agent of various purposes, from playing games (Bullet, 2018) to self driving cars (Berrocal, 2019), and can be easily implemented without using external machine learning libraries. However, it only performs well when training on a static environments and performing simple tasks.

2.4.2 Proximal Policy Optimization

In 2017, OpenAI proposed a new family of policy gradient methods for reinforcement learning, called Proximal Policy Optimization. It tackles the issue of policy gradient methods being too sensitive to the step size, balancing efficient training and avoiding catastrophic failures. This is done by ensuring each parameter update stays within a trust-region of the previous parameter iteration (Schulman, Wolski et al., 2017). Compared to other approaches like Trust Region Policy Optimization(Schulman, Levine et al., 2015) and Actor-Critic with Experience Replay(Wang et al., 2016), PPO offers easier implementation and is more applicable in more general settings which is why PPO is becoming a widely adopted approach for deep reinforcement learning.

Policy gradient method

The policy gradient approach directly learns a policy by computing an estimation of the policy and optimizing its parameters using stochastic gradient ascent. One commonly used gradient estimator is given by:

$$\hat{g} = \mathbb{E}_t \left[\nabla_\theta \log \pi_\theta(a_t|s_t) \hat{A}_t \right]$$

Here's a break down of the components(Fristedt, 2023):

- $\hat{g}$: Estimate of the policy gradient. It represents the direction to adjust the policy parameters θ in order to maximize the expected reward.
- $\mathbb{E}_t[\cdot]$: Expected value over a distribution at time step t, often representing the average over a batch of samples from the environment.
- ∇_θ: Gradient with respect to the policy parameters θ, indicating the direction of steepest ascent in the parameter space to improve the policy.
- $\log \pi_\theta(a_t|s_t)$: Logarithm of the policy π under parameters θ, given the state s_t at time t and action a_t. The logarithm simplifies the multiplication of probabilities into a sum, which is easier to optimize.
- $\hat{A}_t$: Estimate of the advantage function at time t. The advantage function measures the benefit of taking action a_t in state s_t over the average action, indicating the relative quality of an action.

Clipped Surrogate Objective

In order to avoid unnecessary large policy updates that can cause unstable training, PPO utilizes a surrogate objective function denoted as:

$$L_{\text{PPO}} = \mathbb{E} \min \left(\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)} A_t,\ \text{clip} \left(\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon \right) A_t \right)$$

Where $\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$ is the ratio of the probability under the current policy to the probability under the previous policy for taking action a_t in state s_t, and ε is a hyper para-

meter (usually 0.2)(OpenAI et al., 2018). This function discourages bigger changes to the policy to avoid catastrophic forgetting. A more detailed breakdown of the function is in Appendix A.

2.5 Relevant Work

In 2020, a team at New York University introduces a method utilizing Deep Reinforcement Learning for Procedural Content Generation in games, which provided the fundamental ideas this project is based on. The research explores training agents within three games: The classic puzzle game Sokoban (Thinking Rabbit, 1982), a simplified version of Legend of Zelda (Nintendo, 1986) and an environment to generate a longest maze. All three games are presented as a 2D grid, shown on figure 2.5, where each cell in the grid represents a part of the level (e.g., wall, floor, player, goal). Using this environment, they trained three map-making agents: *Narrow*, *Turtle* and *Wide*. The observation space of each agent is a grid-based view of the current map layout, using the presented observation the agents performs an action that alters the map, then the augmented map is evaluated and used to calculate a reward for the agent, shown on figure 2.4. When an immediate reward is not available due to the complexity of evaluating a level's design quality in real-time, the system resorts to calculating a predicted or discounted future reward.

Here is an overview of the three agents:

- Narrow: The observation space of this agent is a 2D integer grid representing the map, and a cell on the grid, chosen by the environment. The agent can decide to change the cell to other values, or skip the change. If this agent want to change a specific cell on the grid, it have to wait for the environment to randomly choose the cell.
- Turtle: The observation space of this agent is a 2D integer grid representing the map, and its current location on the grid. The agent can move up, down, left or right one cell, or change the current cell in each action. This agent is more

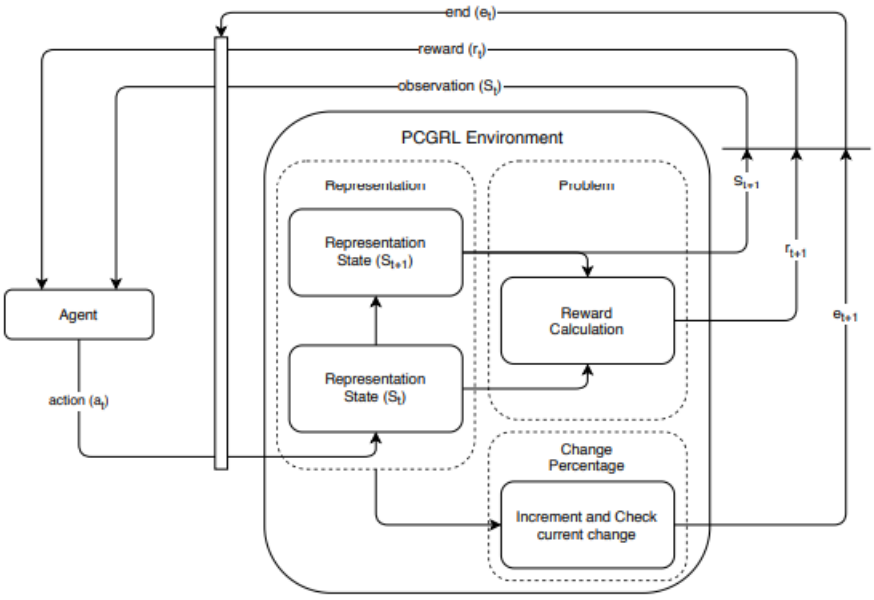


Figure 2.4: The system architecture for the PCGRL environment for content generation. Figure adopted from (Khalifa et al., 2020)

1	1	1	1	1	1	1
1	4	0	0	0	0	1
1	0	3	1	3	0	1
1	1	0	1	0	1	1
1	0	2	0	3	0	1
1	4	0	0	0	4	1
1	1	1	1	1	1	1

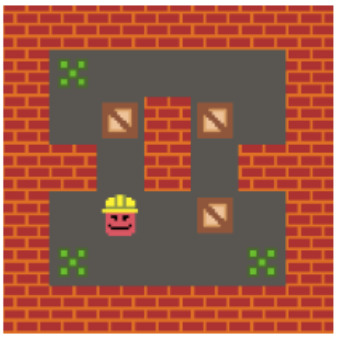


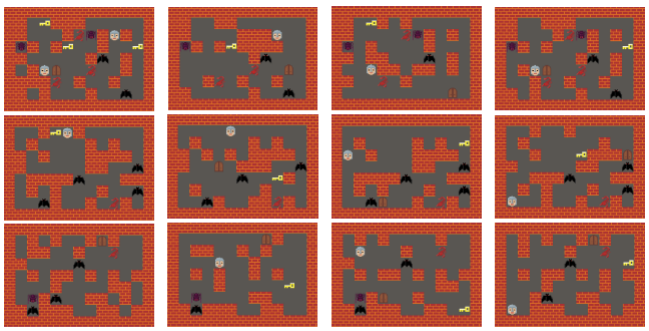
Figure 2.5: Sokoban level as 2D integer array. Figure adopted from (Khalifa et al., 2020)

powerful than Narrow as it can decide which cell it wants to change without needing to wait for the environment to randomly choose it, but requires more resource to train.

- Wide: The observation space of this agent is the 2D integer grid. At each step, the agent is allowed to change one cell on the grid, regardless of location. This

agent's action space is the same size as the observation space, which makes it the most powerful agent out of the three agents, allowing to perform more changes in the same number of steps.

These agents are trained using Convolution Neural Networks and PPO, a more detailed description will be included in the implementation section, where I replicated the Turtle agent to use in my own game. Using these agents, they were able to generate unique maps from randomized initial starting states, shown on figure 2.6. By presenting level generation as a learnable and sequential task, it introduces a new way to create diverse game content efficiently. This research showcases the practical application of RL in generating levels for varied game types, laying the path for future explorations in AI-driven game development and design. My project is heavily inspired by their work, and part of the project is to recreate the Turtle agent using the the environment design and evaluation methods mentioned in this research.



(a) Initial (b) Narrow (c) Turtle (d) Wide

Figure 2.6: Zelda generated examples using different representation for the same starting state. Figure adopted from (Khalifa et al., 2020)

2.6 Unity

A significant part of the project is building the game for Procedural Content Generation and finding the right resources. Unity is chosen for this project because it offers a wide range of development tools, detailed documentations for beginners and open sourced external libraries.

Unity was first introduced in 2005 as a cross-platform game engine for developing video games for desktop computers. Since then, it has evolved into a comprehensive platform for creating interactive content including video games, simulations, and other experiences for like virtual reality (AR/VR) applications. Its versatility and user-friendly interface has made it a popular choice among developers, ranging from individuals and indie studios to large-scale commercial game projects. Unity also provides extensive support for animations and physics, enabling the creation of dynamic, interactive environments. Its animation system allows for the seamless blending of animations, while the physics engine handles collisions, gravity, and other real-world behaviors which adds a layer of realism to the virtual worlds (Young, 2021).

Recent advancements in Unity have focused on the integration of machine learning and artificial intelligence, opening new avenues for creating intelligent behaviors and procedural content generation. The ML-Agents Toolkit, for instance, allows developers to train and implement sophisticated AI models directly within Unity which stimulates innovation in game mechanics, AI-driven characters, and dynamic environments.

Chapter 3

Methods

This chapter provides the details of methods outlined in the previous chapter. We will go through the framework and weight optimisation methods used in the agents for this project. Figure 3.1 contains the structure of the agents. This chapter begins with an introduction of Neural Networks and algorithms used in these agent, followed by a detailed description how these methods are used for Procedural Content Generation.

Agent	Network	Algorithm
Snake	Shallow Neural Network	Genetic Algorithm
Snake	Deep Neural Network	PPO
Turtle	Convolution Neural Network	PPO

Figure 3.1: Structure of networks and algorithms used in this by the agents that are in the demo of this project

3.1 Multilayer Perceptron, Shallow and Deep Neural Networks

Originally, Shallow Neural Networks (SNNs) are often referred to as single-layer perceptrons, which consist of a single layer of output nodes connected directly to the input features without any hidden layers in between. However, as the field of machine learning progresses, the term shallow neural network is commonly referred to networks that include one hidden layer or multiple hidden layers with minimal number of nodes, rather than networks with no hidden layers at all (Bianchini and

Scarselli, 2014). This shift in terminology is the result of increasing complexity and capabilities expected of neural networks, and the term "Shallow" is used to represent the network's limited learning capability. In this study, Shallow Neural Network will be used to represent Multilayer Perceptrons(MLP) with one or two hidden layers, with each layer containing ten or less nodes, and Deep Neural Networks will be used to refer Multilayer Perceptrons with two or more hidden layers, with each layer containing at least a hundred neurons.

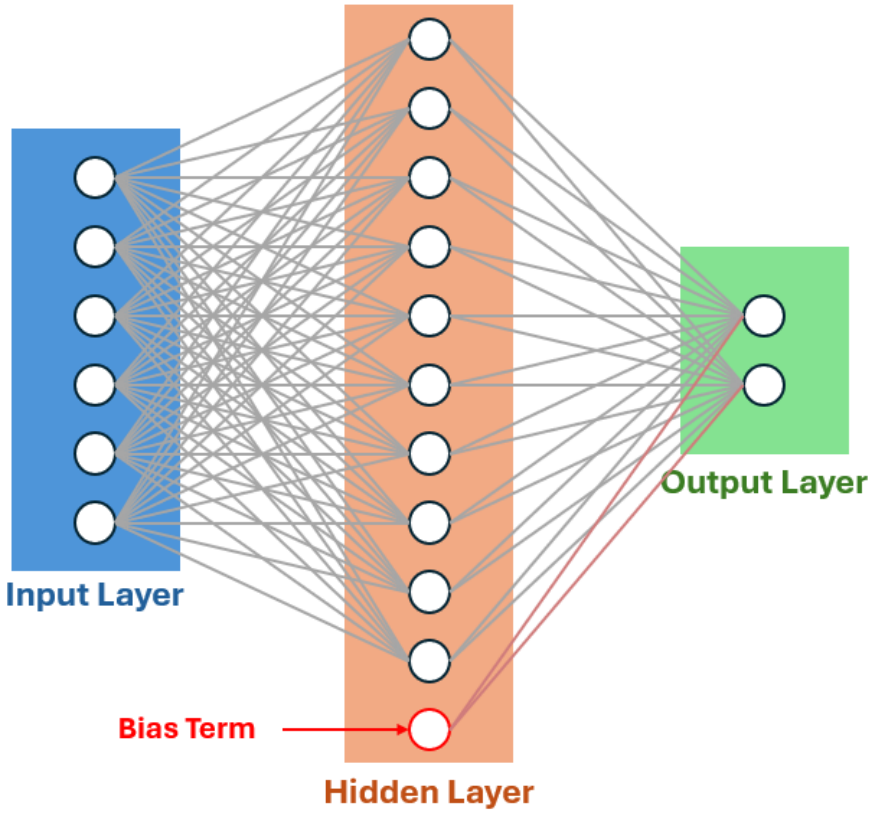


Figure 3.2: Multilayer Perceptron with one hidden layer, which is used to train a selfdriving car in Unity (Berrocal, 2019)

In a MLP, all the nodes in every layer is connected to all the nodes in the previous layer, this type of layer is referred as "fully connected layers", all the layers beside the input layer is fully connected, which allows the model to capture the relationship between inputs and outputs effectively. To further enhance the network's ability to model complex functions, a bias term is added to every hidden layer. This bias acts as an additional parameter alongside the weights and is used for adjusting the output

along the activation function, allowing the neuron to be more flexible in fitting the data.

In general, the more hidden layers and nodes a MLP has, the "deeper" it is said to be. Despite deeper networks have the potential to capture more complex patterns and relationships in the data due to their increased capacity for representation, this does not mean deeper networks are necessarily better. The depth of a network allows for the modeling of more nuanced interaction between input and outputs, but they are also more prone to over-fitting, especially when the amount of training data is limited. While techniques such as dropout, regularization, and batch normalization can mitigate this risk to some extent, the fundamental challenge remains.

3.2 Genetic Algorithm

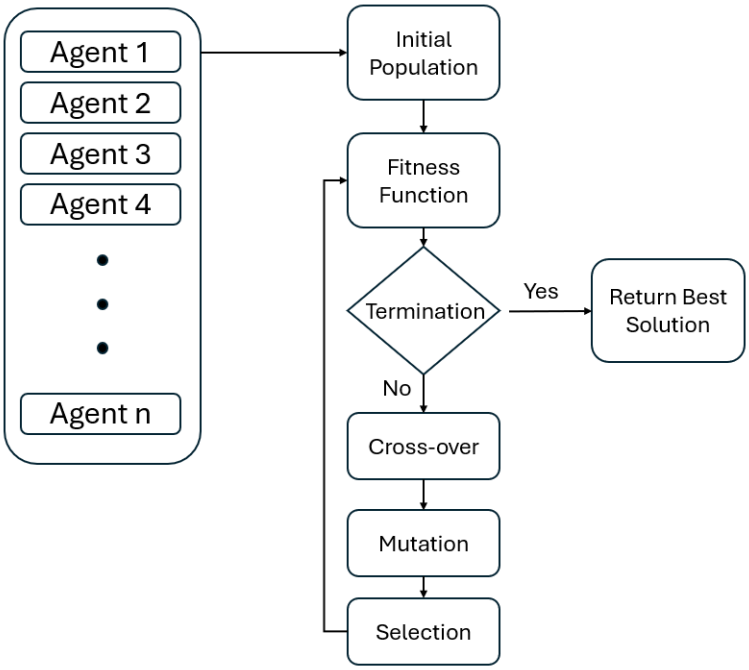


Figure 3.3: Genetic Algorithm flowchart

As mentioned in chapter 2, GA provides a framework for evolving solutions that can adapt and improve over generations. It is one of the most popular way to train agents in Unity as it offers a simple yet effective way for weight optimisation. This section

will describe each the Genetic Algorithm in detail. Overview of the framework is on figure 3.3.

3.2.1 Chromosome

Chromosomes serve as the blueprint for potential solutions. By encoding the neural network weights within chromosomes, GAs can systematically improve network performance across generations. The structure of chromosomes influences the genetic operations such as crossover and mutation, which effects the GAs ability to explore and exploit the solution space.

3.2.2 Fitness Function

The fitness function evaluates the suitability of the chromosomes as a solution to the problem. This function quantitatively assesses how well each individual within the population performs by assigning a fitness score that reflects its ability in solving the problem or achieving certain objectives. Designing an effective fitness function is crucial for the success of a GA, it must accurately reflect the goals of the problem while being computationally efficient to evaluate. A poorly designed fitness function can lead to slow convergence towards sub-optimal solutions or, in the worst case, complete failure to find a viable solution.

3.2.3 Population

In Genetic Algorithms, the population is used to balance between exploration of the solution space and training efficiency. A larger population size offers greater genetic diversity, which allows more exploration of the solution space. This diversity is important for avoiding premature convergence on sub-optimal solutions and ensures that the algorithm can find high-performing solutions across a wide range of possibilities. However, there is a trade-off between population size and computational power. Specifically, training time escalates when the size of the population increases, as each individual requires evaluation to determine its fitness. This evaluation pro-

cess, when multiplied across a large number of individuals, can significantly extend the duration of the training phase. The population size is a hyper-parameter that can be optimized for training efficiency.

3.2.4 Selection

The selection process drives the evolutionary progress towards optimal solutions. By choosing individuals from the current population based on their fitness, selection ensures that traits contributing to success are more likely to be passed down through generations. This process mimics natural selection, where the "survival of the fittest" principle guides the evolutionary trajectory. The efficiency of selection depends on the methods employed, such as tournament selection, roulette wheel selection, or rank-based selection. Although a well-tuned selection process accelerates convergence towards optimal solutions by focusing on the most promising chromosomes, it also risks premature convergence if the selected population lacks diversity. Thus, selection process contains a trade off between the exploration of new solutions and the exploitation of current successes. As with population size, the choice of selection strategy affects the training efficiency and effectiveness.

3.2.5 Crossover

The idea of Crossover is to mimic the reproduction process, off-springs of strong performing individuals have a higher chance of success. Various crossover strategies, for instance, single-point, multi-point, uniform, whole arithmetic recombination, and blend crossover (BLX-alpha), are used for different problems and solution representations. Single-point and multi-point crossover involve swapping segments of the parent genomes at one or more selected points, whereas uniform crossover considers each gene individually, inheriting it from either parent with equal probability. The effectiveness of these methods depends on their ability to mix parental genes in a way that balances the exploration of the solution space with the exploitation of current knowledge, thus converging towards increasingly refined solutions. The choice

of crossover technique and its parameters, such as the crossover rate, prevents premature convergence on sub-optimal solutions or excessive disruption of promising solutions.

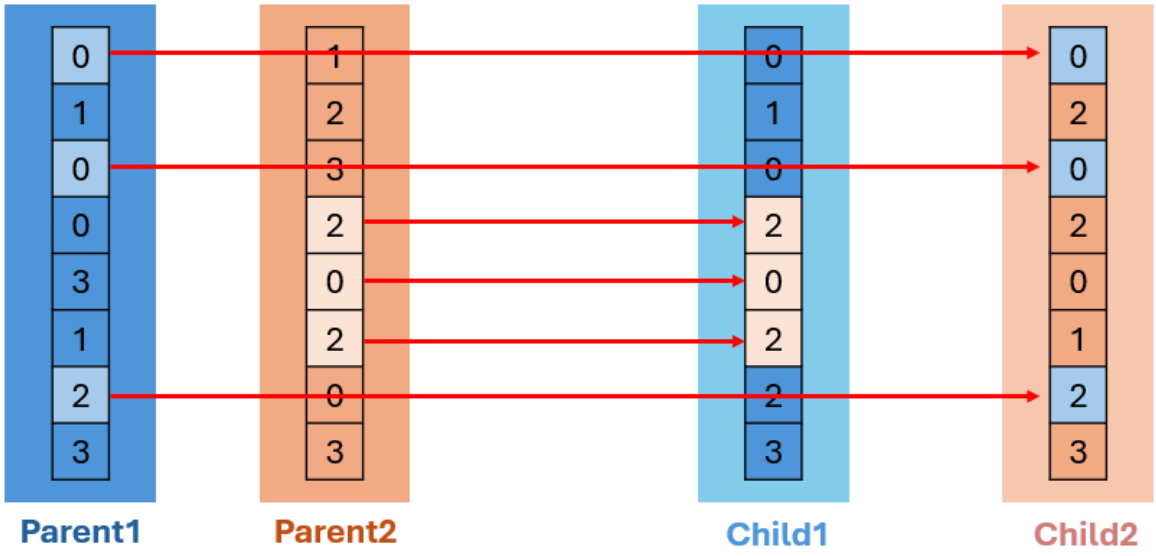


Figure 3.4: Simplified multi-point crossover, where each number represent a weight on a Neural Network. In reality there are much more weights in each chromosome

This project uses multi-point crossover, shown on figure 3.4. It offers distinct advantages over other crossover techniques. Unlike single-point crossover which swaps nodes at a singular location, multi-point crossover allows the interchange of several nodes between the parents, promoting a more intricate combination of their traits. By enabling multiple swapping points, multi-point crossover reduces the risk of disrupting beneficial gene combinations that have evolved, preserving and potentially enhancing advantageous traits across generations. Moreover, compared to uniform crossover, which treats each gene independently, multi-point crossover maintains chunks of genes together, keeping the inherent structure of the solution and possibly preserving functional gene blocks that contribute to high fitness levels. By blending genetic material more effectively and preserving the structural integrity of the genome, multi-point crossover balances exploring new areas of the solution space and exploiting existing evolutionary gains.

3.2.6 Mutation

Mutation is a mechanic that introduces variability into the population. By randomly altering nodes in each individual chromosome, mutation injects new traits into the population, which helps the algorithm to escape local optimal and explore more potential solutions. However, excessively disrupting the chromosomes can disrupt the evolutionary process, preventing the algorithm from converging to an optimal solution.

3.3 Reinforcement Learning for PCG

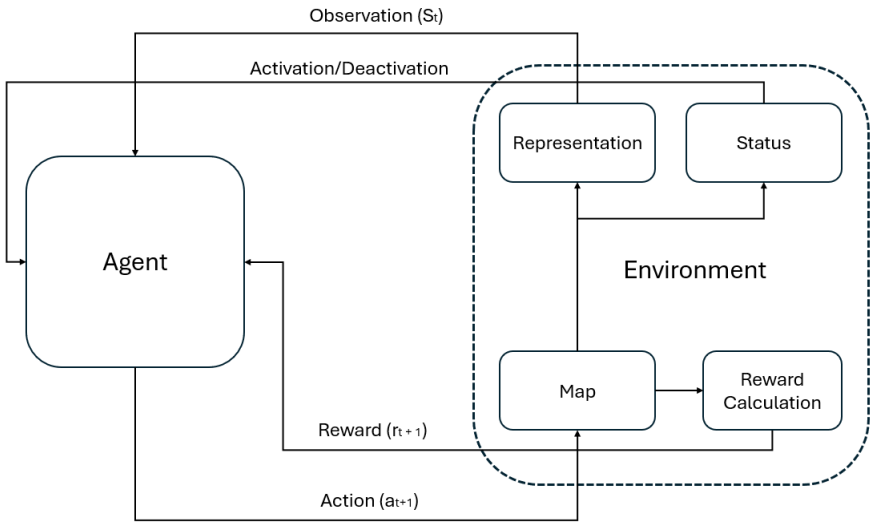


Figure 3.5: Reinforcement Learning framework used in this project.

In order to make the process of Procedural Content Generation as a iterative task instead of generating the whole map at once, the problem of PCG has to be modelled as a Markov Decision Process (MDP) (Khalifa et al., 2020), where the agent makes small and iterative changes to improve a map. This section will go through the framework of MDP, and how it is used to model the framework used in this project for map generation, shown on figure 3.5.

Agents designed in this project have three inputs from the environment: observation, reward and activation. The status of the agent is determined by the environment,

when the agent reach the maximal change percentage (detail in 4.2.1) or produce an invalid map, the environment kills the agent, calculate a reward based on the map produced and reset the map. Using the observation and reward signal, the agent derives an optimal policy that helps it find the best action in each state.

3.3.1 Markov Decision Process

A typical MDP can be represented as $\langle S, A, Pa, R, \gamma \rangle$:

- S: a set of states called the *state space*.
- A: a set of actions called the *action space*.
- $Pa(s, s')$: a transition function that calculates the likelihood of moving from one .state to another based on an action.
- R: Reward function, provides immediate reward after transitioning between states.
- γ: discount factor, used to calculated the discounted future rewards.

Given a state at time t, denoted as St, and the set of actions that can be performed at time t, denoted as At, we want to maximise our accumulative reward at the end, which can be estimated as a sum of current reward(if any) and the expected future reward(if any). This can be denoted as:

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

The strategy, or the behaviour of the agent at each state, can be denoted as a policy π_θ, which is a function parameterized by θ that maps states s to actions a:

$$\pi_\theta(a \mid s) = \mathbb{P}[A_t = a \mid S_t = s]$$

The goal of MDP is to find an optimal policy that can maximize G_t for all $St \in S$. Note that future rewards are discounted by γ, because they are estimates that contain a degree of uncertainty which make the agent prioritise more immediate rewards.

3.3.2 Proximal Policy Optimisation

After modelling the problem of PCG as a Markov Decision Process, we need to find the optimal policy that can maximize the expected reward. Proximal Policy Optimization (PPO) is a reinforcement learning algorithm belongs to the category of policy gradient methods, which optimises policy $\pi_\theta(a|s)$ directly. The fundamental goal is to find the parameters θ that maximize the expected return $J(\theta)$, given by:

$$J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}[R(\tau)],$$

where τ represents a sequence of states and actions performed, and $R(\tau)$ is the total sum of reward of τ, and $J(\theta)$ is the expected reward for following the policy π_θ over all possible τ.

The policy gradient theorem (detail in 3.4.2) provides an explicit formula for the gradient $\nabla_\theta J(\theta)$, allowing us to understand how the change in policy parameters θ affects the overall performance. The theorem states that the gradient of $J(\theta)$ with respect to θ can be represented as:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)\right],$$

Where ∇_θ is the gradient with respect to the policy parameters θ, indicating the direction of steepest ascent in the parameter space to improve the policy. $\nabla_\theta \log \pi_\theta(a_t|s_t)$ is known as the log policy gradient, it measures how sensitive the probability of taking a_t in state s_t is when changing θ. Using this representation, we can update the parameter θ using gradient ascent:

$$\theta_{\text{new}} = \theta_{old} + \alpha\nabla_\theta J(\theta)$$

where α is the learning rate.

On top of the standard policy gradient approach, PPO introduces modifications to to enhance stability and efficiency. It proposes an objective function that discourages

large deviations from the current policy, thus mitigating the risk of destabilizing updates. The clipped surrogate objective function is designed to keep the updated policy $\pi_{\theta_{new}}$ close to the old policy $\pi_{\theta_{old}}$ by clipping the probability ratio $r(\theta)$, which is defined as:

$$r(\theta) = \frac{\pi_{\theta_{new}}(a|s)}{\pi_{\theta_{old}}(a|s)}.$$

The clipped objective function $L^{CLIP}(\theta)$ is given by:

$$L^{CLIP}(\theta) = \mathbb{E}\left[\min(r(\theta)\hat{A}(s,a), \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\hat{A}(s,a))\right],$$

where $\hat{A}(s,a)$ is an estimator of the advantage function at state s and action a, and ϵ is a hyper parameter that determines the clipping threshold. This formulation restricts the policy update to a safe region, reducing the likelihood of large and potentially harmful updates.

3.3.3 Observation and feature extraction

The observation of an agent needs to contain all the information needed to solve the presented problem. This information can either be calculated by ourselves, or the agent can try to interpret the map like a human using Convolution Neural Networks (CNN). For example, when we train an agent to play a Snake game, the agents needs information about it's current location on the map, position of the fruit, and position of its tails. We can either put the entire game map into a few convolution layers, or quantify these information ourselves. One common approach is to use ray casts to let the agent learn its surroundings , shown on figure 3.6. In this project, both methods are used: the Turtle agent uses a CNN that contains the entire game map, while the Snake agent uses information calculated by the environment. In this section, we will go through the details of how a CNN works, more information about both agents will be in the Implementation chapter.

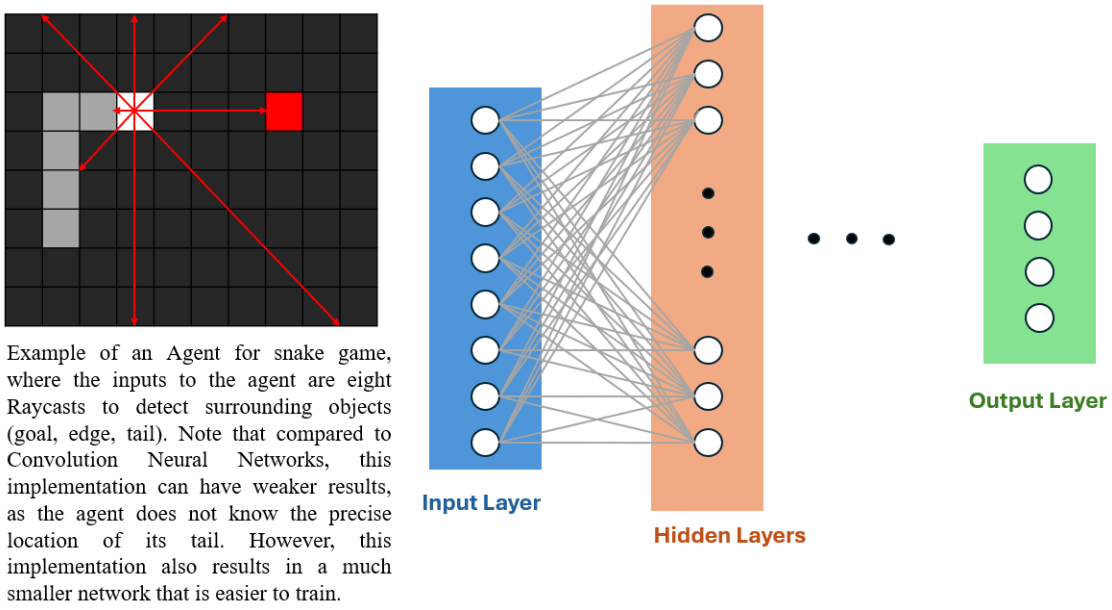


Figure 3.6: Example of an agent using raycasts to play Snake

When dealing with high-dimensional inputs, such as a 3D matrix of the game map, a multi-layer perceptron can be inefficient as the the number of parameters becomes too large to manage. This explosion of parameters not only makes the network computationally intensive but also prone to over-fitting. To address these challenges, a feature extractor needs to be implemented before feeding the observations onto the input layer. Convolutional neural networks emerge as a more effective solution. CNNs are designed to adaptively learn spatial hierarchies of features from input images by exploiting the spatially local correlation presented in images through the use of learnable filters and reducing the number of parameters needed compared to MLPs. CNNs operate through several types of layers, including convolutional layers, pooling layers, ReLU layers and fully connected layers towards the end:

- Convolutional layers: Applies a set of learnable filters to the input. Each filter is spatially small but extends through the full depth of the input volume, and every one of these computes a dot product of each filter with the input.
- Pooling layers: Produces non-linear down samples of the input, which helps to reduce the number of parameters and computation in the network.

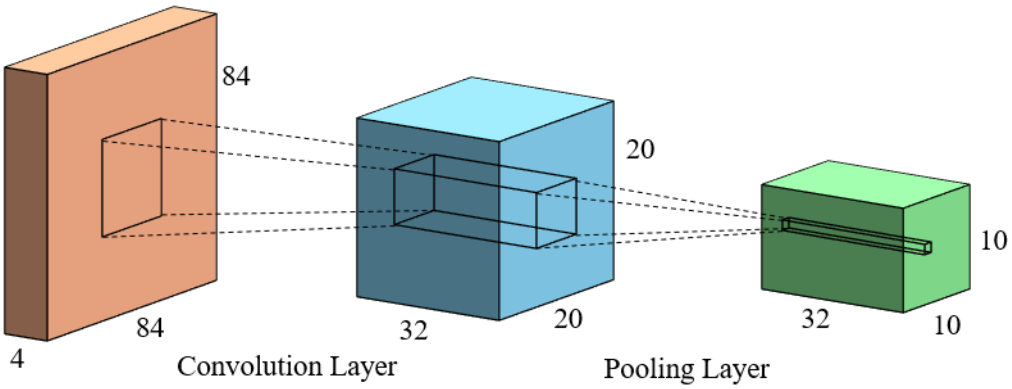


Figure 3.7: Example of convolution layer and pooling layer

- ReLU layers: Applies an activation function to the input, removes negative values by setting them to zero.
- Fully connected layers: Added at the end of a CNN, which resembles the traditional MLP architecture.

By carefully combining these filters, we can form a Convolution Neural Network. The decision on the number of filters in each convolutional layer of a CNN is influenced by a combination of factors including the task complexity, the depth of the layer within the network, computational constraints, and empirical findings from similar tasks.

3.3.4 Reward

When we train agents to play video games, the game score can be used as reward. However, when training agents to do general tasks such as level generation, the reward has to be calculated by the environment separately. Like other similar reinforcement projects, designing the right reward function is the most challenging part. A sparse reward leads to longer training time because the agent will need to reach the desired state by randomly exploring, while a dense reward often makes the agent behave in unexpectedly by maximizing reward in creative ways (L., 2024). For example, when a penalty is given to the Turtle agent for placing tiles in sub-optimal

locations, sometimes the agent will converge to not placing any tiles at all. To combat this, very sparse reward is used when a measurement of fitness is unavailable (Khalifa et al., 2020)(e.g. reward of 1 when the objective is reached, 0 otherwise), at the cost of significantly increasing the training time.

Chapter 4

System Design and Implementation

Unlike other research projects, this project contains lots of software engineering, as a game was developed during the time frame of this project. This chapter explains the how the methods mentioned in previous chapters are implemented, including training environments, evaluation methods and model design. The agents designed in this project Uses models from a wide range of similar projects, which allow me to focus on environment design and reward system design. Details about how the game is implemented will not be included, as it is not the main focus of this project. Scripts of the game can be found in the zip file uploaded to blackboard.

4.1 Game Design

A significant part of the project is designing the game for content generation, as a self-developed game has a larger degree of freedom for modification compared to other open-sourced games. The game developed for this project serves as a demo of how the Procedural Content Generation techniques can be used in video games. This section will go through rules of the game, how the agents interact with the map, and the methods used for map evaluation. A version of the game that uses a random path generator is included in the demo.

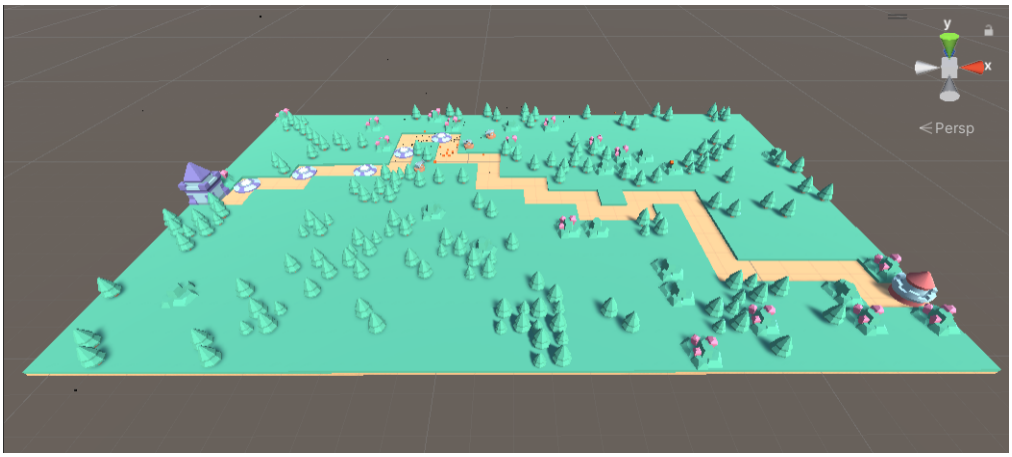


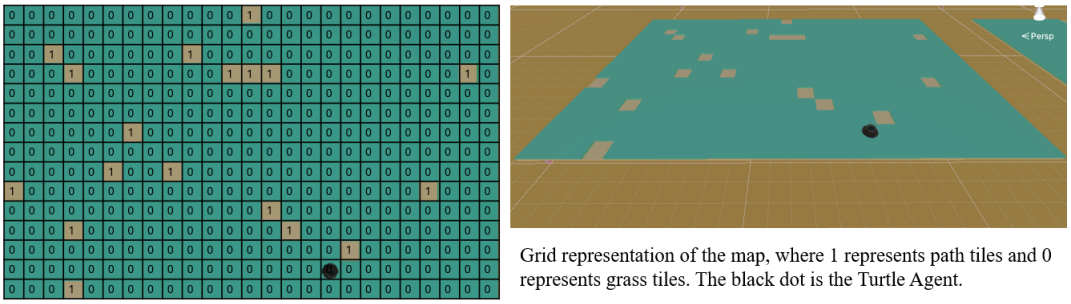
Figure 4.1: Screenshot of the game built for this project. The assets used in the game comes from Kenny's Tower Defense Kit (kenny, n.d.), which is Copyright free, and the animations are built using Unity's Particle System.

4.1.1 Rules

The game designed in this project is a tower defense game. On a 15*25 grid, UFOs follows a path from spawn to base, and the player can build towers to defeat UFOs. If more than ten UFOs reached base, the player loses. If all UFOs are defeated, the player wins. For aesthetic and game complexity, trees and stones are placed randomly on the map after the path is generated which prevents the player from building towers.

4.1.2 Agent-Map Interactions

Training an agent using reinforcement learning require interpreting and manipulating the game environment millions of times, which demostrates the importance of keeping the game representation simple. Inspired by Khalifa's work (Khalifa et al., 2020), the game used in this project will also be represented as a 2D integer array, where each cell contains an integer representing a game element (grass, path, spawn, base). This representation allows the agent to find and modify objects on the grid with O(m*n) time complexity, which decreases the time needed for each episode and saves valuable training time. The goal of an agent is to generate a path for the UFOs to follow. To avoid agents from over-fitting and making the same map every time, the agent



Grid representation of the map, where 1 represents path tiles and 0 represents grass tiles. The black dot is the Turtle Agent.

Figure 4.2: Grid Representation of the map, containing starting point for the Turtle Agent

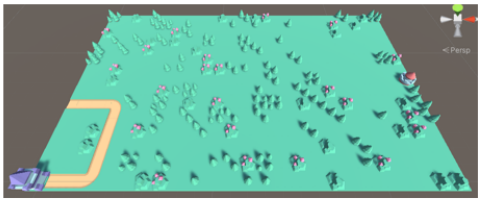
is presented with some randomness in the initial map. Beside randomness, previous studies suggest adding a change percentage of around 20 percent can also prevent over-fitting (Khalifa et al., 2020). Using this idea, agents automatically terminate when they changed more than $(15*25)/4 = 94$ tiles. In total, two types of agents are trained for path generation:

- Snake: At the start, the agent is presented with a grid filled with grass tile, with a goal prefab randomly placed on the grid. Using information from the grid, the agent can move one cell at a time, leaving a path behind every time it moves. Starting from a random position, the goal for the agent is to generate a longest path before reaching the base prefab.
- Turtle: Inspired by classic turtle graph languages, This agent is able to move freely on the grid and decide which tile it want to change. The agent is presented with a grid randomly populated with grass and tile, as shown on figure4.2, and the goal for the agent is to generate a longest-shortest path on the grid.

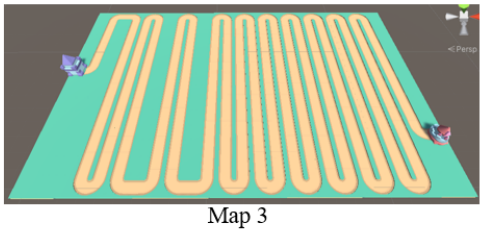
4.1.3 Map Evaluation

The most important part of this project is designing a way for map evaluation, which is required for both Genetic Algorithm and PPO. Using ideas from Khalifa's work (Khalifa et al., 2020), maps are evaluated on three aspects: validity, difficulty and play-ability:

- Validity: For a map to be valid, there must exist a path from the spawn prefab to base prefab, otherwise the UFOs can never reach base.
- Difficulty: The level generated needs to be winnable, and contain enough tiles for the player to build towers on.
- Play-ability: The path generated needs to be long enough for the map to be enjoyable to play.

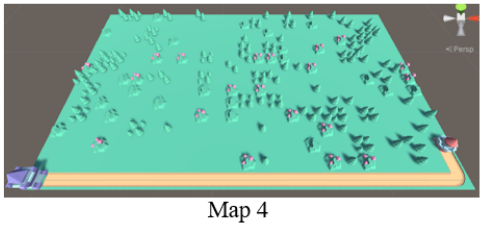


Map 1



Map 3

Map 2



Map 4

Map 1 and 2 are not valid, as they does not contain a path from spawn to base. Although both maps are not valid, we would consider Map 2 to be better than Map 1, as it is closer to the base prefab and have a longer path. Map 3 and 4 are valid, but not pleasant to play with, as the path are either too long or too short. Map 5 is valid and have a good balance of path length, which will be what we aim to create.

Map 5

Figure 4.3: Grid Representation of the map, containing starting point for the Turtle Agent

Based on these matrices, we can distinguish between the maps we want and we does not want. An example is included in figure 4.3. Now we have identified the features of map we want, we can assign a numeric score to them and assess them quantitatively. If an agent manages to reach the base prefab, a large score is rewarded, the agent also gets a small reward for each path prefab it places. To prevent the agent from filling the map with path tiles, we utilize the change percentage proposed by Khalifa

(Khalifa et al., 2020), which forces the agent to find the goal prefab before changing too much of the map. The base reward function is:

$$Reward = x * tilesN + (goal * y) - penalty$$

Where x is reward for placing a tile, tilesN is the number of tiles placed, goal is 1 if the agent manages to reach the base prefab, 0 if the agent fails, and y is the reward the agent get for reaching goal.

4.2 Unity and Genetic Algorithms

As mentioned in section 3.6, Unity offers a highly flexible platform with a wide range of development tools. Utilizing this flexibility, the Genetic Algorithm framework implemented in this project only uses an matrix multiplication package, the rest is implemented from scratch using C#. Figure 3.3 showcased details of the GAs, and this section will explain how it is implemented to interact with the game environment. 4.4. Each component in this framework interacts with each other, with the goal

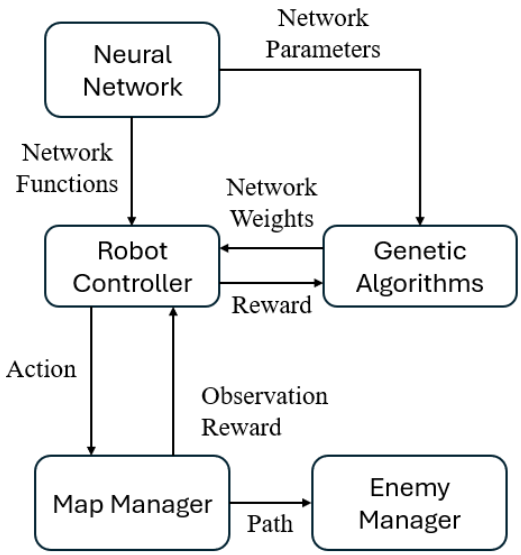


Figure 4.4: Flowchart of how the components interact with each other in the GA framework

of generating a neural network that maximizes reward based on Map Manager's evaluation.

1. The Genetic Algorithms initializes a population of agents, stored as weights in matrices.
2. One of the agents is loaded into the Robot Controller, which collects observations from the Map Manager.
3. The observation is fed into the input layer of the network, and the network returns an action as output.
4. The Robot Controller executes the output, which interacts with the Map Manager, and collects new observations after the action.
5. Processes 3 and 4 are repeated until the Map Manager decides to kill the agent, this happens either when the agent reaches a goal or fails. A score is assigned to the agent based on performance.
6. A new agent is loaded from the Genetic Algorithms. This is repeated until all agents in this population are evaluated.
7. The Genetic Algorithms generates the next population.

After obtaining a high performance agent, we can load it into the Robot Controller and generate a map onto the Map Manager, which can then be used to play the game. This framework is incredibly flexible, and all the methods and parameters mentioned in section 4.2 needs to be set manually. Figure 4.5 contains a list of all the parameters/functions, and appropriate range of values. After implementing the scripts, Unity offers an interface to monitor and edit the parameters at runtime (as shown on figure 4.6), which can be very useful when trying to understand how each parameter affects agent performance.

4.2.1 SNN Snake Agent

The objective for this agent is to move from a starting position to a goal position, while generating a longest path < 94 tiles before reaching the goal. Inspired by the SNN model used for training a self driving car in Unity 3.2, the Snake agent uses a

Parameter/functions	Range of Values	Explanations
Input layer size	-	Dependent on the observation collected from the agent
Number of hidden layers	1 ~ 2	Limited to reduce the number of parameters, as GA is not suitable for training deep NNet
Nodes in each hidden layer	1 ~ 10	
Population size	1 ~ 100	Limited to reduce training time, as each agent takes 5-10 second to evaluate
Crossover percentage	20 ~ 60	Percentage of the population that is descendants from last population, larger value encourages exploitation and smaller value encourages exploration
Best agent percentage	10 ~ 30	Percentage of the population that is carried to the next population, a rank-based selection is used.
Output layer size	-	Dependent on the action space of the agent
Reward/Penalty	-	Main function that influences agent behaviour, dependent on the agent goal.
Duration (Seconds)	5 - 10	Time taken before killing the agent, preventing bad performing agents taking up resources.

Figure 4.5: All the important parameters in the GA framework

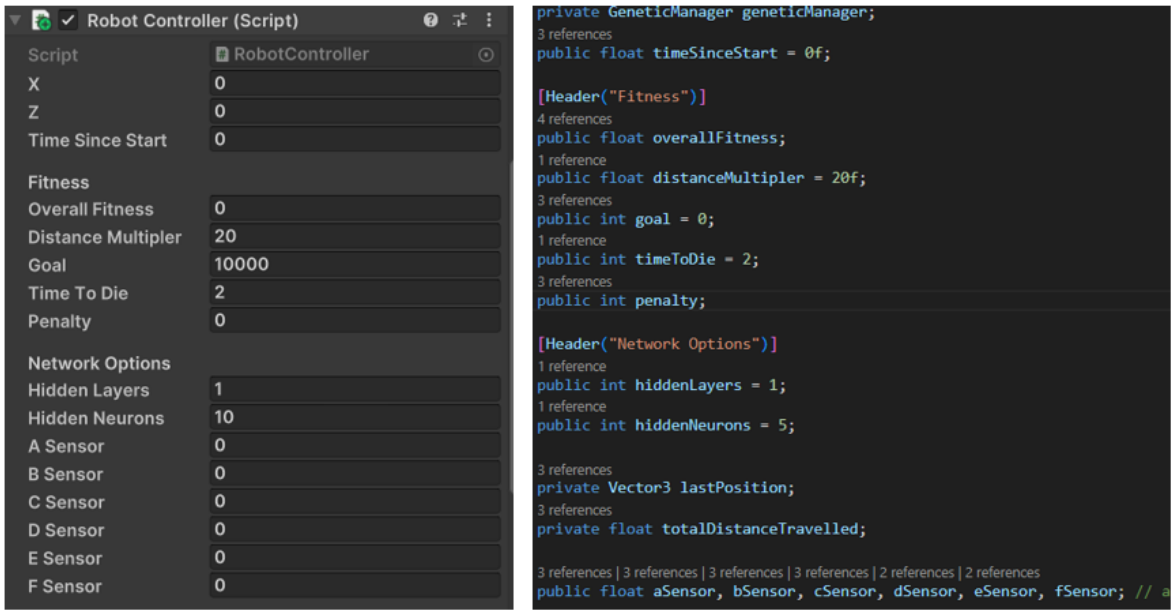


Figure 4.6: Example of how Unity interacts with scripts

similar framework. It contains six nodes in the input layer, two hidden layers with ten nodes in each layer, and two nodes in the output layer.

4.2.2 Reinforcement learning and ML-Agents

Reinforcement learning operates on a fundamental interaction loop between an agent and the environment. Unity, with its ML-Agents Toolkit, provides a flexible API that simplifies the process of training agents capable of navigating and learning from their surroundings extends this data flow principle. The ML-Agents environment is crafted using Unity's editor, where the developer can design interactive scenes that serve as the learning playground for agents. Each agent within Unity is equipped with a Brain (either a Neural Network trained via deep reinforcement learning or a heuristic model defined by the developer) that processes sensory inputs (observations) and outputs actions to perform in the environment. To progress through the learning episode,

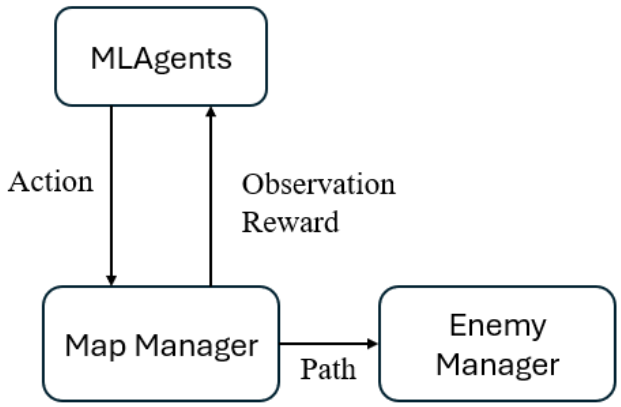


Figure 4.7: Flowchart of how the MLAgents interact with the environment

ML-Agents employs a step function, which is built within the Unity environment's simulation loop. When an agent takes an action, the Unity environment responds by updating the state of the world, providing new observations and rewards back to the agent. As a Unity official package, ML-Agents offer a number of features that make the Unity Engine more suitable for machine learning, including automatic cache cleanup, speeding up the game environment and allowing multiple agents to be trained at the same time. These features allow a much larger model to be trained in less time compared to using the GA framework. ML-Agents also include a number of builtin CNN, PPO and reward calculation function, which make agent training very

efficient and allow us to focus more on building the environment and experiments with the reward function.

Convolution Layers

CNN is used in the Turtle agent for feature extraction, consisting of three convolution layer. The input image is collected and resized using Unity's builtin camera, afterwards, it is fed into a series of three convolution layers, shown on figure 4.8. This

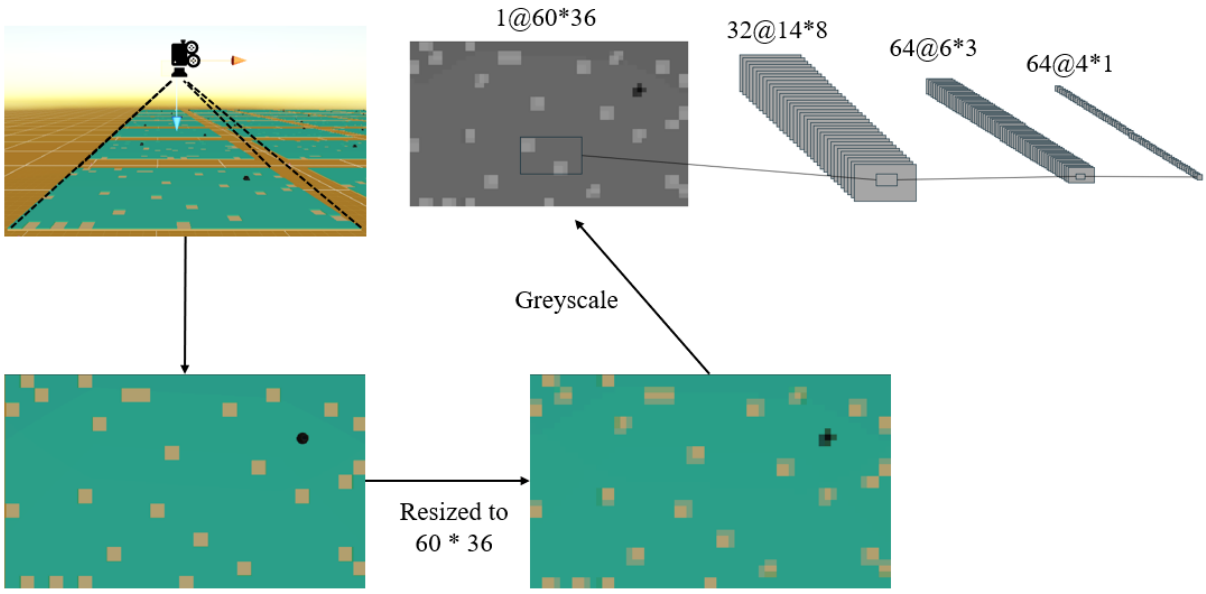


Figure 4.8: Convolution layers used by the Turtle Agent

architecture is a similar architecture used in the Turtle agent proposed by Khalifa (Khalifa et al., 2020), which is based on the CNN proposed by Mnih et al (Mnih et al., 2015), used to train agents capable of human-level control playing Atari games. The first convolution layer performs a convolution of 32 14x8 filter with a stride of 4. The second layer then performs a convolution of 64 6x3 filters of stride 2. The third layer convolves 64 4x1 filters of stride 1, producing the final output before the fully-connected network.

Fully Connected Layers

After extracting the features, fully connected layers are used to learn the pattern between the input data and desired output. Both agents use the same architecture, consisting of two layer with 128 neurons before one additional fully connected output layer for the action. This architecture is chosen because it is used in a lot of projects using ML-Agents, such as the rocket landing project (L., 2024), and is also recommended by the ML-Agents documentation. While the architecture have some effects on the training performance, the focus on this project is not to on finding the best architecture or training method, but on formulating PCG as a iteratable problem that can be solved using machine learning.

4.3 Agent Evaluation

With a lack of previous work, and without a standard benchmark to compare with, evaluating the agent performance becomes a challenge. Beside comparing the agents with each-other, and evaluating the agent using the reward system, I've implemented a map generator that uses a standard random number generator (RNG), which serves as a guideline for the agent. One goal of the agents is to generate paths that is longer than the RNG implementation, which contains at least 20 path tiles.

Another matrix for agent evaluation is over-fitting of the network, which is reflected on by the similarities between the maps an agent generates. This is evaluated based on the agent's initial sequence of actions: if the agent is repeatedly following a same sequence of actions that generates the same map every time, we know it is over fitting. The agents can avoid over-fitting by utilizing the random positions of the initial state and using the change percentage mechanism mentioned in the previous chapter.

Chapter 5

Experiments

This chapter is broken into four sections, with each section dedicated to one of the three agents included in the demo, and a CNN Snake agent. Each section will explain the observation space, action space and the reward function used, reasons behind the design, and the results of using that framework. An optimal agent would be able to generate valid maps with high playability, these are the maps that contains a path from the spawn prefab to the base prefab, with an optimal path length (at least 20) that ensures the game is not too easy or too difficult.

5.1 SNN Snake Agent

The first agent we will go through is the Shallow Neural Network Snake agent trained using Genetic Algorithm. The problem of PCG is simplified for this agent by keeping the spawn and base position static, as it uses a much less power network compared to the others.

This agents take six inputs: current X location, current Y location, distance to goal X, distance to goal Y, width and height of the map. The action space of this agent is up, down left and right, and the goal of the agent is to find the base prefab on the right of the map starting on the left. The agent has one hidden layers with ten nodes, this architecture is the same one used by the self-diving car project on figure 3.2. Building on the idea mentioned in the map evaluation section, the agent receives a reward of 20 for each tile it places, and a reward of 10000 if it successfully reaches the

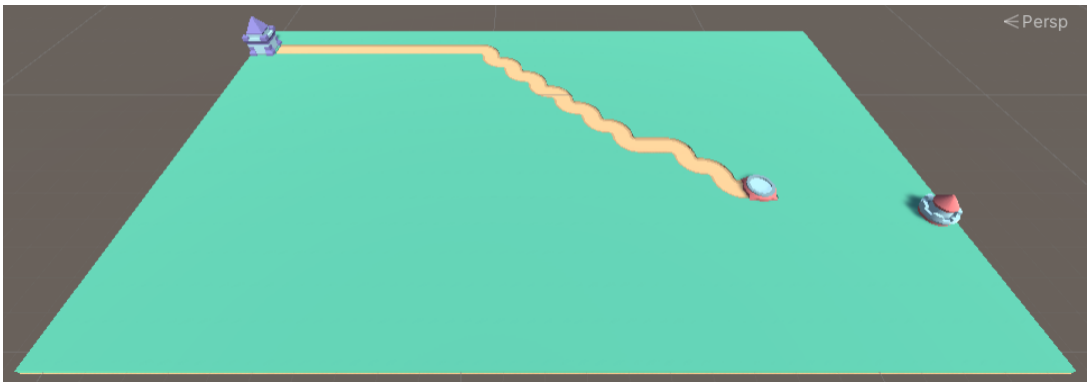


Figure 5.1: SNN Snake example

goal. One problem for this agent is that it does not have the information of the paths it already placed, causing it to overlap with itself. To combat this, instead of giving the agent information of all the path positions, which will significantly increases the network size, a penalty of 100 is given every time the agent overlaps with a path tile.

$$Reward = 20 * tilesN + (goal * 10000) - penalty$$

Given the simplicity of the task, instead of including a well-trained agent in the demo, the whole training environment is put into the demo to visualize the evolutionary process of this agent, labelled as "Genetic Algorithm Demo", as it usually takes less than a hundred generations to generate a decent map.

5.1.1 Evaluation of SNN Snake

Figure 5.2 contains the training result for SNN Snake included in the demo, which took around 30 minutes. The population size was set to 10, and best agent percentage was set to 20 percent and crossover percentage was set to 30 percent, details of these parameters can be found in figure 4.5. These parameters strongly focus on exploitation of strong solutions, which ensure that the demo converges to an optimal solution very quickly, and the process of evolution can be demonstrated in a reasonable time frame. After ten generations, the best agents in the population were able to find the base prefab consistently, with an average score of 7480. At this point, the best maps generated were valid, but lack playability as majority of times they

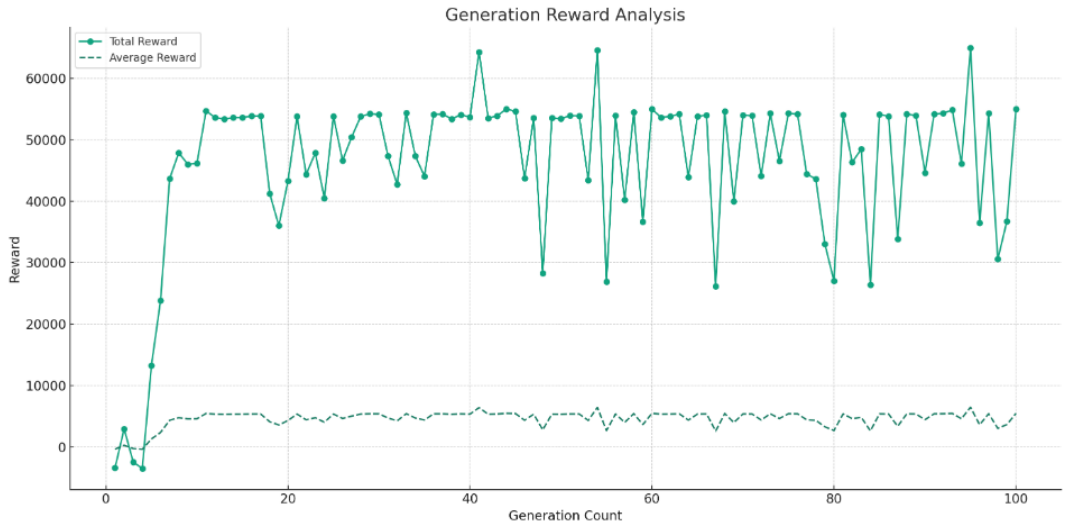


Figure 5.2: Training of 100 generations, with 10 chromosomes per generation.

were very short. The agent struggled to improve on the solution in future episodes, potentially due to the population in the demo lacks diversity. After adjusting these parameters to increase diversity, we can train an agent that performs better, at the cost of increasing the training time. Figure 5.3 includes the training result of the

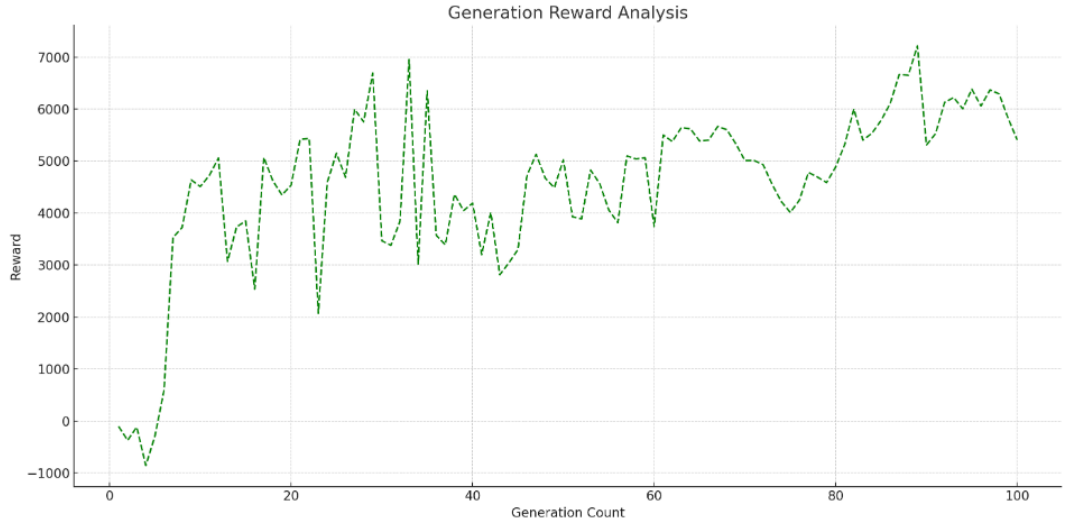


Figure 5.3: Training of 100 generations, with 30 chromosomes per generation.

same agent, with the population size being increased to 30, best agent percentage reduced to 10 percent and crossover percentage was set to 30 percent. The total training time was around 90 minutes. Note the average in each generation decreases compared to figure 5.2, and there are more variations between generations, this is

because the population is more diverse, and less successful individuals from previous population are used in the new one. The increase in diversity allows the model to find better solutions. After generation 100, the best agent in the population achieved a score of 11160, which was better than the results on figure 5.2, where the best agent achieved a score of 10640. However, the training time increased by 300 percent, as three times more individuals needed to be evaluated. These results highlight the trade-off between computational power and agent performance mentioned in section 3.1.

5.2 DNN Snake Agent

The second agent has the same observation space and action space as the SNN Snake agent, and a similar reward system. The main differences between this agent and the SNN Snake agent is the size of the network. As mentioned in section 4.2.2, this agent contains two hidden layers between the input and output layer, with 128 nodes in each hidden layer. Beside network size, this agent uses PPO instead of GA for weight optimization, and the spawn/base position is randomly generated between episodes, allowing the agent to generate a more diverse map.

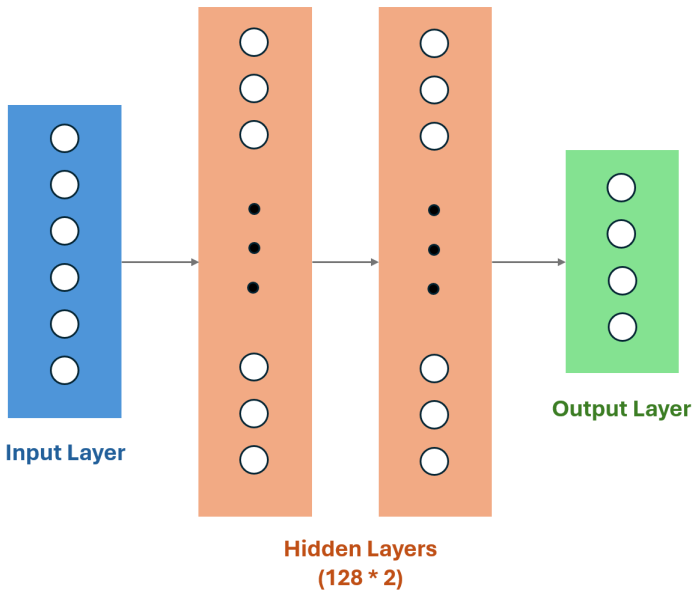


Figure 5.4: Architecture of the DNN Snake

Like the SNN Snake, the agent receives a reward of 5 for each tile it places, a reward of 100 if it hits the goal, and a penalty of 100 it if the agent overlaps with a path tile. If the agent didn't reach the goal position before exceeding the maximum change percentage (which is how much the agent is allowed to change the map, proposed by Khalifa (Khalifa et al., 2020)), it receives a penalty of 50.

$$Reward = 5 * tilesN + (goal * 100) - penalty$$

A trained agent is included in the demo of this project, labelled as "Snake Agent".

5.2.1 Evaluation of DNN Snake

Figure 5.5 showed the training result of the DNN Snake included in the demo. The agent was trained for five million steps; Each step is the cycle of reading an observation and executing an action, and the total training time is just over 8 hours. The beta value in the clipped surrogate objective function is set to 0.2 (as recommended by paper published by OpenAI (OpenAI et al., 2018)), and the learning rate is set to 0.0003, which is the higher range recommended by the ML-Agents documentation. The training time of this agent was long but still manageable, so these values were not adjusted to decrease training time. The agent stabilized around 2.5 million

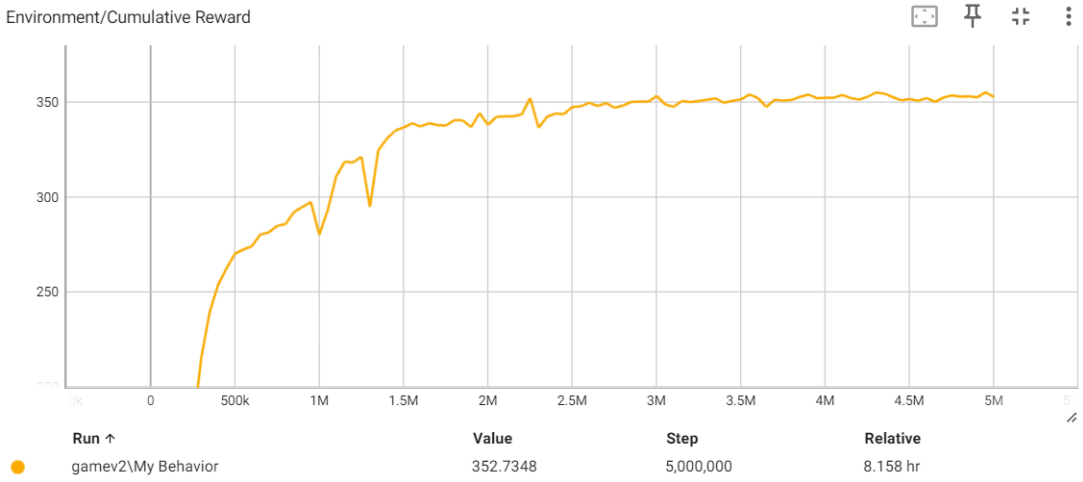


Figure 5.5: Training result of five million steps

steps, and was able to generate a valid map every time. Compared to the SNN Snake

agent, the map it was able to generate is slightly longer, and it was able to generate maps on previous unseen initial positions which made it the stronger agent. Figure 5.6 illustrated the maps created by both agents. The agent struggled to find a better

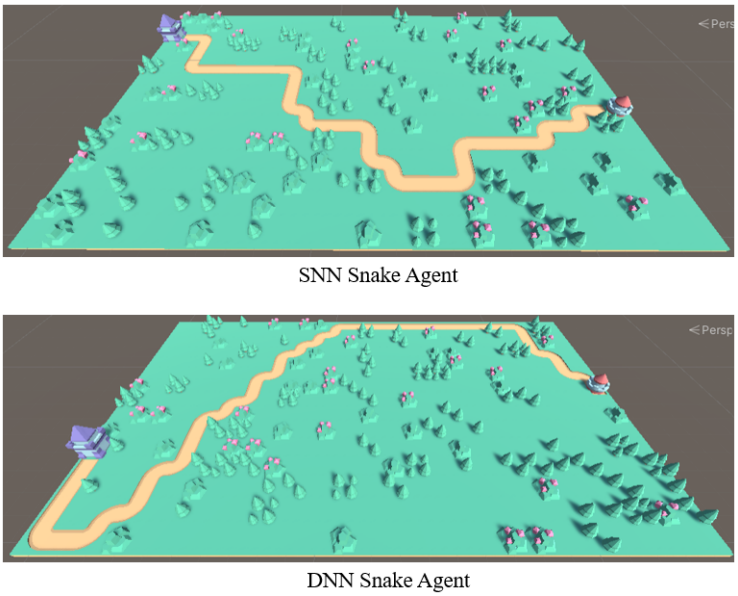


Figure 5.6: Map produced by both agents side by side

solution in future episodes, as the reward set for reaching the goal position was too high compared to placing more tiles, and the agent was penalized too much when exploring new solutions. To address this issue, the reward function was updated by reducing the ratio of reward for reaching goal to reward for placing a tile, in order to encourage the agent to explore more possible solutions: the reward for reaching the goal position is reduced to 10, and the reward for placing a tile is set to 1.

$$NewReward = 1 * tilesN + (goal * 10) - penalty$$

Figure 5.7 exhibited the result of updating the reward function. The training results became highly unstable, since the agent did not have the information on the path it already placed. A solution to this problem is to include information of the placed path into the network. However, because the number of path tiles are always changing, it is hard to encode them on the input layer of the network. A similar problem was faced by other researchers when training an agent to play snake, where the agent

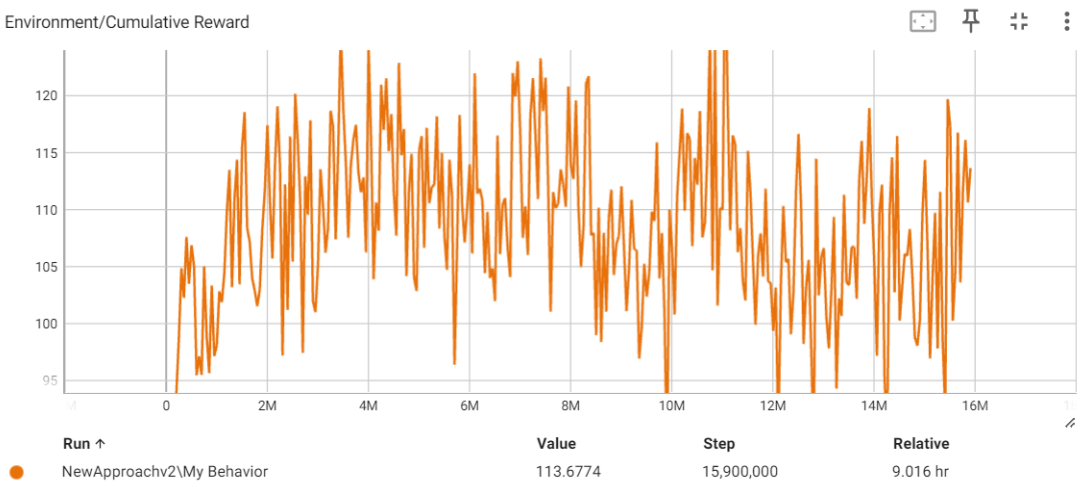


Figure 5.7: Training result after modifying reward.

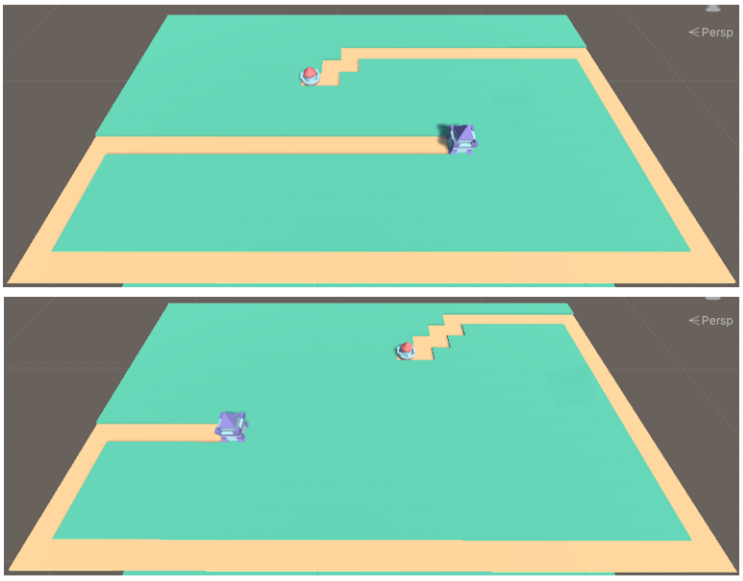
needed the information of the tail positions in order to avoid them. Viau proposed a solution by using raycasts like figure 3.6, which helped the agent to determine the objects in front of it (Viau, 2018). Another solution was using a convolution neural network. Compared to raycasts, using a CNN has the benefits of knowing global positions of the tail positions, which helps the agent to avoid dead ends and make better decisions. We will go through the CNN Snake agent implementation in the following section.

5.3 CNN Snake

As an attempt to further improve the Snake agent, I implemented a Snake agent that used CNN. This agent uses the architecture mentioned in figure 4.8, followed by two fully connected layer with 128 nodes in each layer and the output layer. This new architecture was able to capture the information of the tail positions, and generated longer maps than the DNN Snake agent. Without having to worry about the agent overlapping with itself, starting position of the spawn and base prefabs were randomly placed on the map, allowing the agent to include more diversity in the map generated. However, the agent became more prone to over-fitting, when using the same reward function as the DNN Snake agent:

$$Reward = 1 * tilesN + (goal * 10) - penalty$$

The agent overfitted to the same series of action before going to the goal position. This over-fitting problem in CNN was mentioned in multiple studies, and I found



Maps generated by CNN Snake

Figure 5.8: Two maps generated by CNN Snake

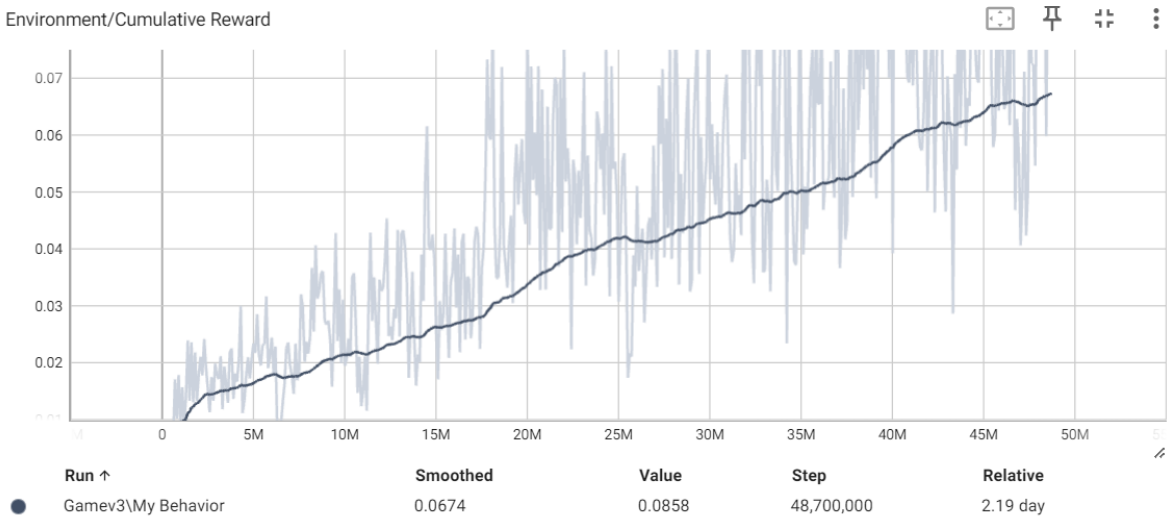


Figure 5.9: Training result of the CNN Snake with updated reward function.

two potential solutions for this problem: simplify the reward system (L., 2024) or include more randomness in the initial state (Khalifa et al., 2020). As mentioned in section 3.3.4, using a sparse reward can potentially avoid over-fitting at the cost of increasing training time. Utilizing this idea, the reward function was updated

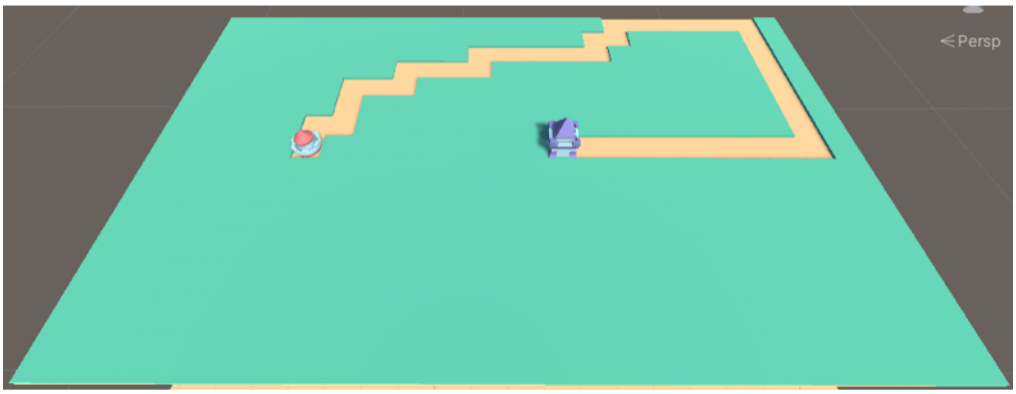


Figure 5.10: Map Generated after updating CNN reward function

once again, to 1 if the agent manages to reach the goal with a path longer than 50, 0 otherwise. Figure 5.9 showed the training results after updating the reward function. Compared to figure 5.7, the training was much more stable. However, the agent still overfitted, as the agent moved toward the right edge of the map regardless of the starting position. Changing the reward function reduced some amount of over-fitting as the agent did not stick to the edge as much, as shown on figure 5.10. If we want to further reduce over-fitting, we need to include more randomness at the starting position (e.g. include tiles the agent must pass before reaching the goal position), which requires redesigning the environment entirely.

5.4 CNN Turtle

The last agent that is included in the demo of this project is a Turtle agent which uses CNN. Instead of generating a path between two prefabs, this goal of this agent is to generate a longest-shortest path between any two points of the map(longest path between any two points when using a shortest path finder). Then the environment uses the generated path as the game map, placing the spawn and base prefabs on both ends and spawn enemies between them. This agent uses the same CNN architecture in figure 4.8, followed by two fully connected layers with 128 nodes in each layer and the output layer. The main difference between this agent and the CNN Snake agent is that it does not automatically place tiles at its location, it has one additional action in the action space. At each step, the agent can either move up, down, left

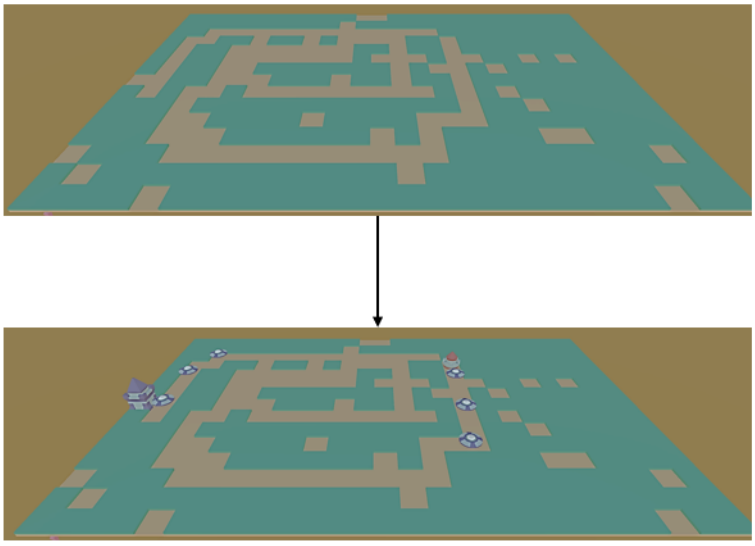


Figure 5.11: Example of how the turtle agent is used for map generation. The environment find the longest-shortest path generated by the turtle agent and use it as the map path.

or right on the map, or change the tile it is currently on. After the agent reaches maximum change percentage, or reaches the maximum step count (set to 1500, used to terminates agent that does not place any tiles), the episode ends.

5.4.1 Reward

The reward function used for the previous agent is not suitable anymore, as tiles placed on the map do not equal to tile used as path between spawn and goal. In the 2020 paper on PCG-RL (Khalifa et al., 2020), an identical turtle agent was used to generate a longest maze, the reward function wasn't explicitly stated, but the goal of the agent was described as *"modify a 2D map of solid and empty spaces such that the longest shortest path between any two points in the map increases by at least X tiles (where X = 20 in our experiments) and all the empty spaces are connected."* This statement focuses on two aspects of agent performance: "longest shortest path between any two points" and "all the empty spaces are connected", which is the foundational idea we use to design the reward function.

Using a really sparse reward is no longer suitable for this task, as it is nearly impossible for the agent to connect all the empty spaces together by randomly exploring.

Instead, a dense reward is used to encourage the agent to connect more empty space together. Using basic graph theory, each path tile is viewed as a node, and path tiles that are connected together forms a connected component. When the agent reduce the number of connected components, it gets closer to connecting all the empty spaces together, and is rewarded. At the start of the episode the number of

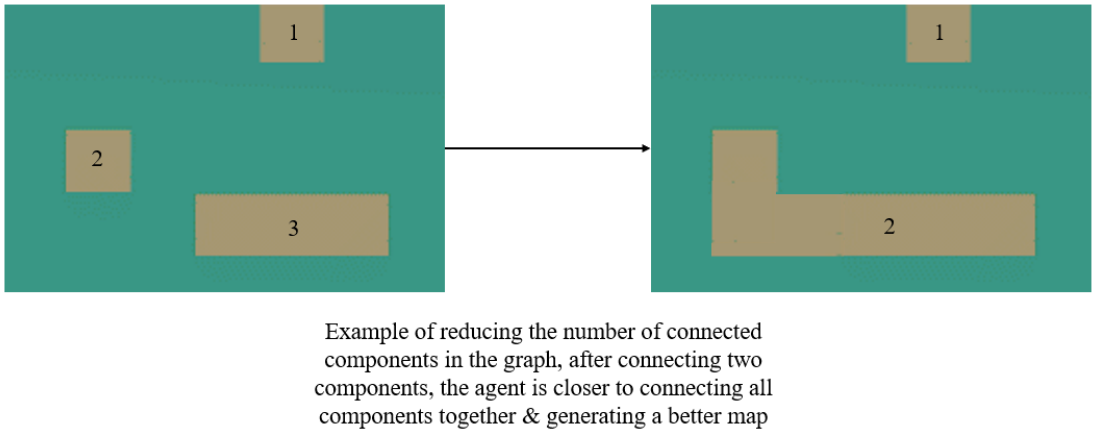


Figure 5.12: Example of connecting components, the agent receives a reward when it connects two components together

connected components are recorded by calling a depth first search algorithm on all the nodes, and if the number of connected components is reduced at the end of the episode, the agent is rewarded. The reward function used by the Turtle agent in the demo of this project is :

$$Reward = 1 * (pathL - L) + 1 * (initialCC - finalCC) + penalty$$

where pathL is the length of the longest shortest path, L is the minimal length the agent need to generate, which is set to 30. If the agent is not able to generate a path longer than 30, it receives a penalty. InitialCC is the number of connected component at the start of the episode, and finalCC is the number of connected components at the end of the episode. A large negative penalty is given to the agent if it does not place any tiles. Figure 5.13 contains the training result of using this reward function, which is stopped after 130 million steps because the agent start to over-fit. To reduce over-fitting, more randomly placed tiles were included in the

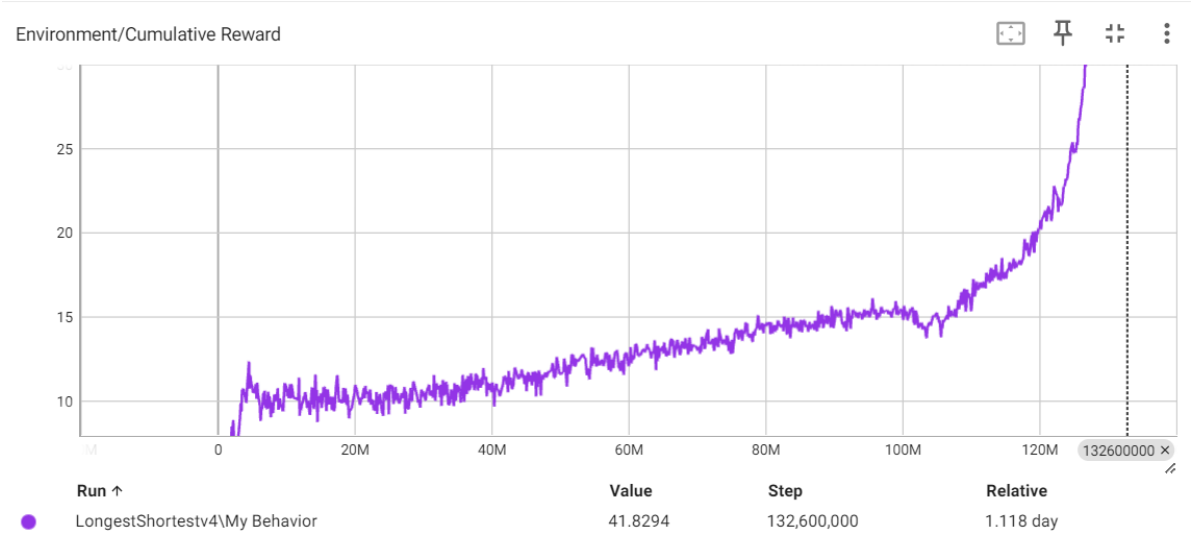


Figure 5.13: Training result of the Turtle Agent

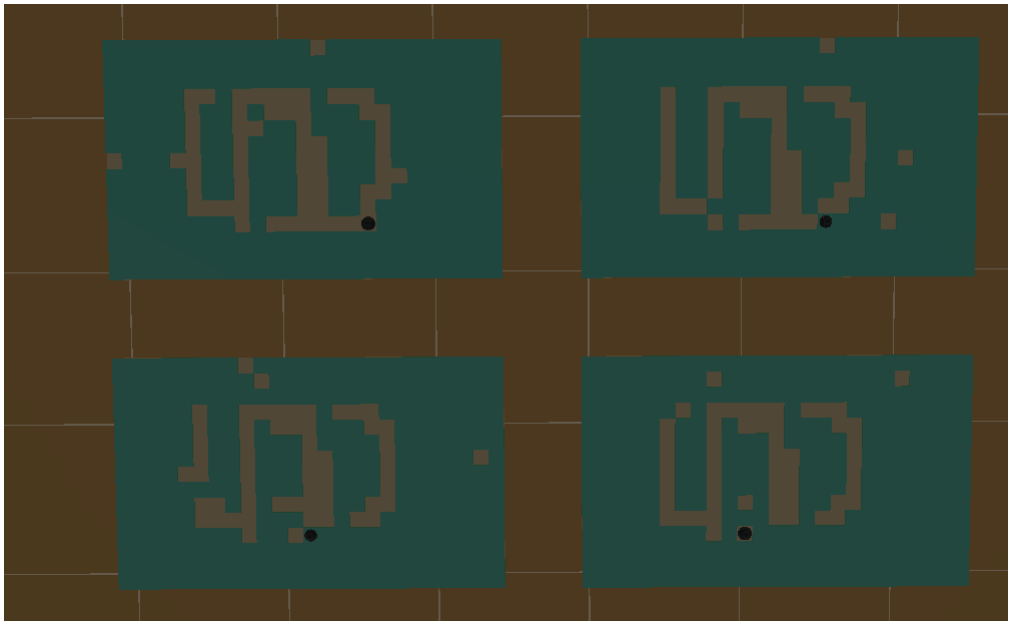


Figure 5.14: 130 million steps, agent ignores the start state

start state. Figure 5.15 shows the training result after modifying the starting state. The agent was able to achieve some success at 40 million steps, but after 45 million steps, the reward suddenly dropped to zero. This is due to a phenomenon called the "catastrophic forgetting", which is common problem faced in deep reinforcement learning. However, the map produced by the agent at 40 million steps showed more variability, and was included into the demo of this project.

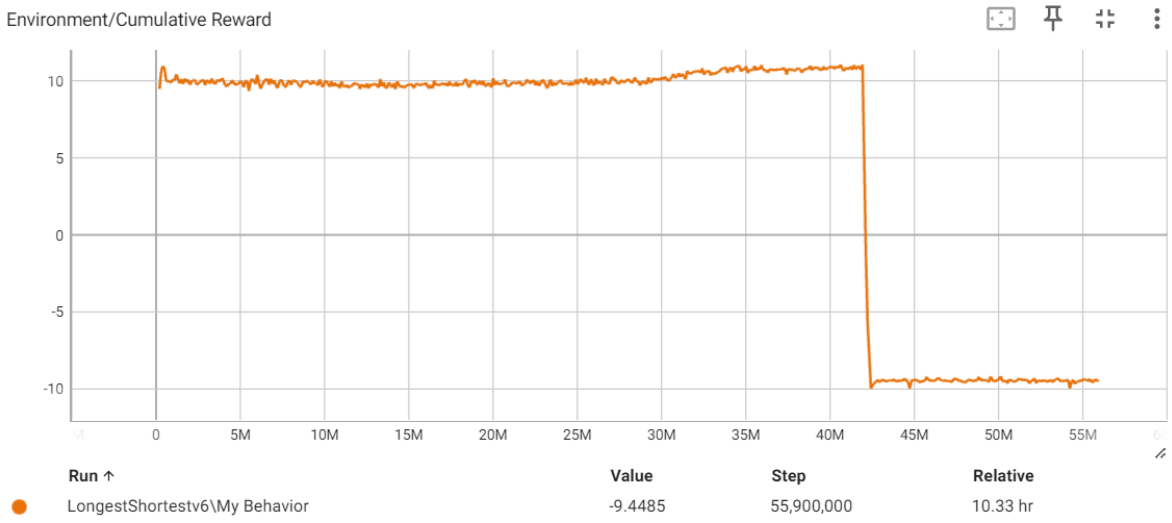


Figure 5.15: Training result of the Turtle Agent

5.4.2 Catastrophic Forgetting

This problem was faced multiple times after increasing the number of random tiles at the start state when training the Turtle agent. Catastrophic Forgetting occurs when a model, after being trained on one task (Task A) and then trained on a new task (Task B), performs well on Task B but its performance on Task A significantly deteriorates. In this case, task A would be generating the longest shortest path and task B would be connecting all the components together. When the agent learns to connect components together, it is rewarded for placing all the path tiles in a giant blob. However, this cause the agent to receive a penalty for not generating a long path. When the agent tries to generate a long path, it risks creating new connected components as it is placing tiles away from the blob. An example is illustrated in figure 5.16. This phenomenon happens randomly, figure 5.17 and 5.18 illustrate two training session of the agent with the same PPO parameters. In the second training session, the penalty was increased in an attempt to prevent the agent failing. As mentioned in 3.3.2, PPO's clipped surrogate objective function can reduce the chances of this happening. By reducing the ϵ value, we limit how much the policy is allowed to change. In figure 5.15 the ϵ value was reduced to 0.1 which allowed it to remain stable for 40 million episodes.

Receive reward for connecting all components together, but penalized for generating a short path

Receives reward for generating a long path, but penalized for not connecting all components together

Figure 5.16: Example of Task A vs Task B

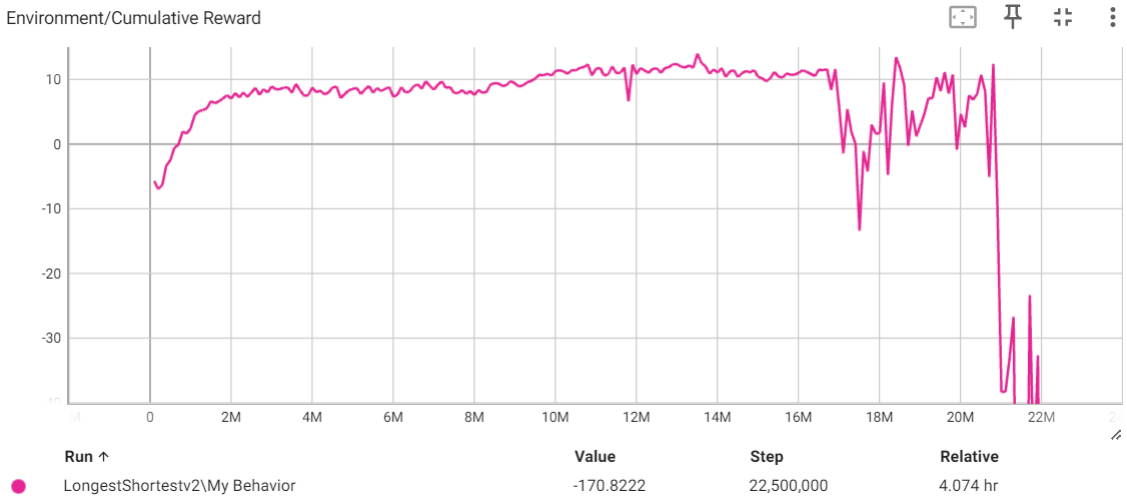


Figure 5.17: Result of the agent failing at 25 million steps

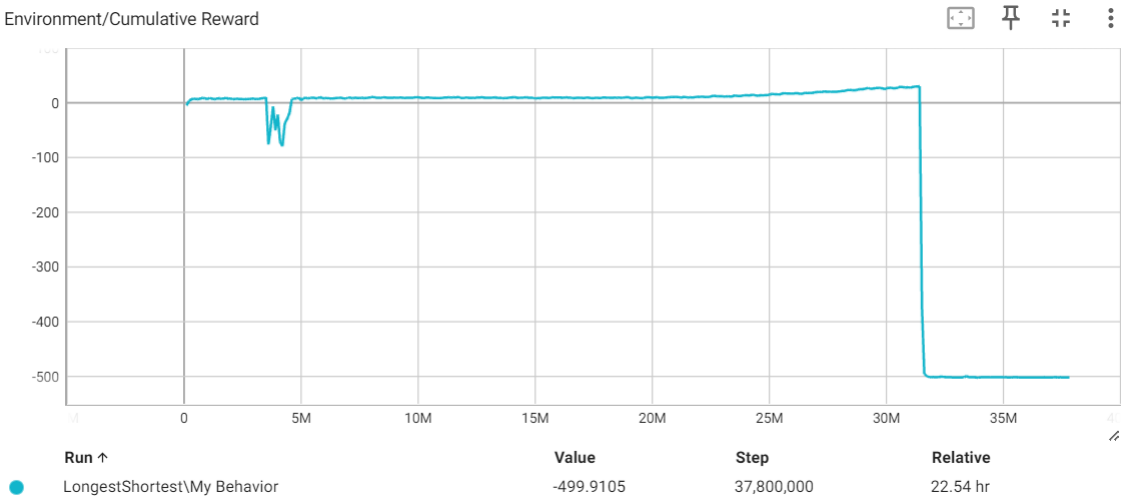


Figure 5.18: Result of the agent failing at 33 million steps

Chapter 6

Conclusion

6.1 Achievements

This project explored the synergy between neural networks, reinforcement learning (RL), specifically Proximal Policy Optimization (PPO), genetic algorithms, and their application within the Unity platform for map generation. A major hurdle in the development of this project is the process of learning basis of reinforcement learning and researching relevant works, and unlike other research projects, this project contains a lot of software engineering. A substantial part of this project was learning how to use Unity to build the base game and developing interactive environments for reinforcement learning. Using the Unity platform, this project is built and hosted on Unity Play website, which includes the base game, a demo of the evolutionary process of the genetic algorithm, and demos of the Snake and Turtle agent that are capable to generate maps dynamically. This project successfully demonstrated that by integrating these advanced computational techniques, it's feasible to surpass the limitations of traditional PCG methods and generate new game content without using historical data.

6.2 Limitation

One of the primary limitations encountered is not having enough computational power when training deep neural networks within a real-time game development

environment. All the training was done on my home PC and the time it took to train an agent is substantial: it could take up to days as demonstrated on figure 6.1. This does not only limits the amount of experiments that can be done within the

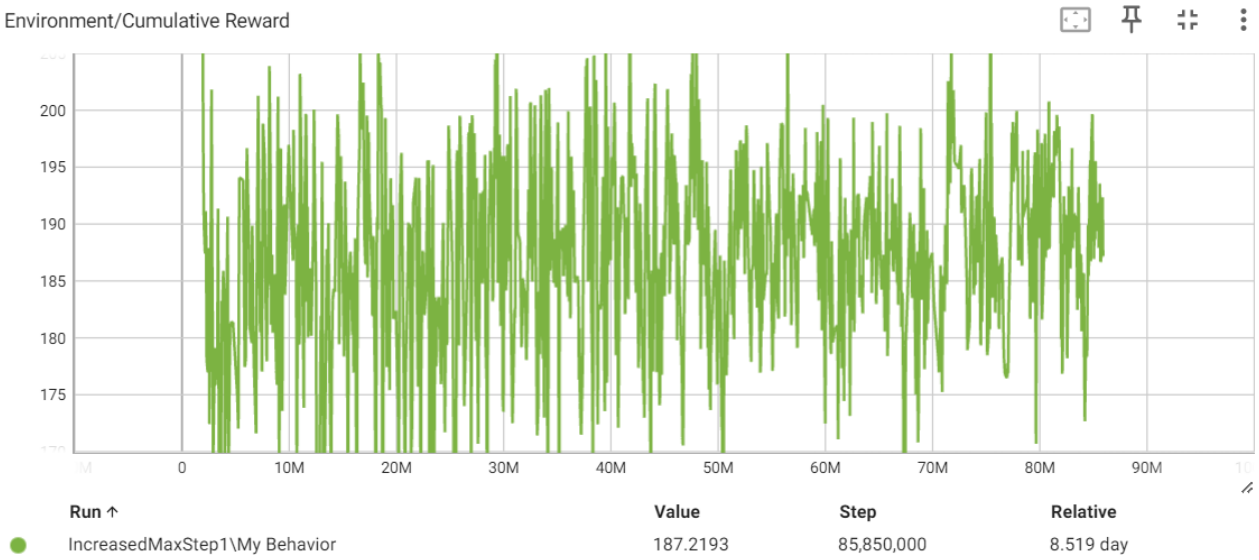


Figure 6.1: Result of an exploration focused agent, trained for 8 days

time frame of this project, but also limits the approaches that can be experimented with, as agents that focus on exploring the solution space are too time-consuming to train. If I had more time, I would create new environments for the CNN Snake agent, and try to find a solution to prevent catastrophic forgetting when training the Turtle agent.

Another limitation came from lack of previous work, although Khalifa's work in 2020 (Khalifa et al., 2020) laid the ground work for this project, it did not contain the details that were required to replicate their findings. By interpreting their work, I was able to generate decent results with the Snake agent, but struggled to prevent over fitting when trying to replicate their Turtle agent, which is potentially due to inefficient training setup or environment design. More research and careful considerations are needed when designing the training environment if I want to replicate a similar Turtle agent, which I intend to pursue going forward.

If I can do this project again, I would spend more time focusing on speeding up

training time before doing experiments. Through out the duration of this project I slowly improved my training environment, and by the end of the project, I was able to reduce the training time by ten folds compared to the first agent I trained, which is done by using four environments and training a hundred agents in each environment. I would be able to conduct more experiments efficiently, that can potentially stabilize agent performance if I discovered the faster training methods at a earlier stage.

6.3 Future work

Due to the limitations, I wasn't able to produce the results I would've liked for the Turtle agent and CNN Snake agent. One area for future work is to use redesign the environments for these agent in a way that can prevent the agent from generating similar maps. Another important area for further work is refinement of training approaches. The time it took to train agents, highlighted by the eight-day training period for an exploration-focused agent, suggests that making these processes more efficient could dramatically speed up development cycles. Future work could explore more advance optimization strategies, including fine tuning hyper-parameters and employing distributed training methods. Such approaches could potentially cut down on training time while enhancing the agents' ability to adapt and perform.

References

Berrocal, Diego (Feb. 2019). *Self Driving Car Simulation Unity 3D using Genetic Algorithms and Neural Networks - Unity 3D & C#*. Accessed: 11/03/2024. YouTube. URL: https://www.youtube.com/watch?v=m8fYPy9eiOo (cit. on pp. v, 11, 18).

Bianchini, Monica and Franco Scarselli (2014). 'On the Complexity of Neural Network Classifiers: A Comparison Between Shallow and Deep Architectures'. In: *IEEE Transactions on Neural Networks and Learning Systems* 25.8, pp. 1553–1565. DOI: 10.1109/TNNLS.2013.2293637 (cit. on p. 17).

Brummelen, Jessica Van and Bryan Chen (2018). *Procedural Generation: Creating 3D Worlds with Deep Learning*. Accessed: 2024-03-01. URL: https://www.mit.edu/~jessicav/6.S198/Blog_Post/ProceduralGeneration.html (visited on 1st Mar. 2024) (cit. on p. 6).

Bullet, Code (Dec. 2018). *A.I. Learns to play Flappy Bird*. Accessed: 11/03/2024. URL: https://www.youtube.com/watch?v=WSW-5m8lRMs (cit. on p. 11).

Dahrén, Martin (2021). 'The Usage of PCG Techniques Within Different Game Genres'. In: (cit. on p. 6).

Ding, Shifei, Chunyang Su and Junzhao Yu (2011). 'An optimizing BP neural network algorithm based on genetic algorithm'. In: *Artificial Intelligence Review* 36, pp. 153–162. DOI: 10.1007/s10462-011-9208-z (cit. on p. 10).

Fristedt, Anton (June 2023). 'Optimizing Reinforcement Learning Algorithms Using Design Of Experiments'. Master's thesis. Lund, Sweden: Department of Computer Science, Lund University (cit. on pp. 12, 60).

Kanal, Laveen N. (Jan. 2003). 'Perceptron'. In: *ResearchGate*. Available online: https://www.researchgate.net/publication/262288029. (Visited on 1st Mar. 2024) (cit. on p. 7).

kenny (n.d.). URL: https://kenney-assets.itch.io/tower-defense-kit (cit. on pp. vi, 31).

Khalifa, Ahmed et al. (2020). 'PCGRL: Procedural Content Generation via Reinforcement Learning'. In: *arXiv preprint arXiv:2001.09212* (cit. on pp. i, v, 1, 2, 14, 15, 23, 29, 31, 32, 34, 38, 44, 47, 49, 55).

L., Alexander T. (2024). *Landing Starships Simulation*. Version v1.2. GitHub. URL: https://github.com/alxndrTL/Landing-Starships (visited on 21st Mar. 2024) (cit. on pp. 28, 39, 47).

Mnih, Volodymyr et al. (2015). 'Human-level control through deep reinforcement learning'. In: *Nature* 518, pp. 529–533. DOI: 10.1038/nature14236 (cit. on p. 38).

OpenAI et al. (2018). 'Learning Dexterous In-Hand Manipulation'. In: *The International Journal of Robotics Research* (cit. on pp. 13, 44, 60).

Osband, Ian et al. (2016). 'Deep Exploration via Bootstrapped DQN'. In: *30th Conference on Neural Information Processing Systems (NIPS 2016)*. Stanford University, Google DeepMind. Barcelona, Spain (cit. on p. 9).

PwC (2023). *Gaming industry emerging technology trends*. Accessed: 2024-04-18. URL: https://www.pwc.com/us/en/tech-effect/emerging-tech/emerging-technology-trends-in-the-gaming-industry.html (visited on 18th Apr. 2024) (cit. on p. 2).

Schaffer, J. David, Darrell Whitley and Larry J. Eshelman (1992). 'Combinations of Genetic Algorithms and Neural Networks: A Survey of the State of the Art'. In: *Proceedings of the International Workshop on Combinations of Genetic Algorithms and Neural Networks (COGANN-92)*, pp. 1–37. DOI: 10.1109/COGANN.1992.273950. URL: https://ieeexplore.ieee.org/document/273950 (cit. on p. 9).

Schulman, John, Sergey Levine et al. (2015). 'Trust Region Policy Optimization'. In: *arXiv: Learning* arXiv:1502.05477 (cit. on p. 11).

Schulman, John, Filip Wolski et al. (2017). 'Proximal Policy Optimization Algorithms'. In: *arXiv: Learning* arXiv:1707.06347 (cit. on pp. 11, 60).

Sze, Vivienne et al. (2017). 'Efficient Processing of Deep Neural Networks: A Tutorial and Survey'. In: *arXiv* abs/1703.09039. URL: https://arxiv.org/abs/1703.09039 (cit. on pp. v, 7).

Tesauro, Gerald (Mar. 1995). 'Temporal Difference Learning and TD-Gammon'. In: *Communications of the ACM* 38.3. Copyright © 1995 ACM. URL: https://www.csd.uwo.ca/~xling/cs346a/extra/tdgammon.pdf (visited on 1st Mar. 2024) (cit. on pp. v, 8, 9).

Toosi, Amirhosein et al. (2021). 'A BRIEF HISTORY OF AI: HOW TO PREVENT ANOTHER WINTER'. In: *PET Clinics* (cit. on p. 7).

Viau, Greer (2018). *SnakeAI*. https://github.com/greerviau/SnakeAI. Accessed: 2024-03-31 (cit. on p. 46).

Wang, Ziyu et al. (2016). 'Sample Efficient Actor-Critic with Experience Replay'. In: *arXiv: Learning* arXiv:1611.01224 (cit. on p. 11).

Young, Chris J. (2021). 'Unity Production: Capturing the Everyday Game Maker Market'. In: *Game Production Studies*. Ed. by Olli Sotamaa and Jan Švelch. Amsterdam University Press. DOI: 10.5117/9789463725439_ch07. URL: https://www.jstor.org/stable/j.ctv1hp5hqw.10 (cit. on p. 16).

Appendix A
PPO Surrogate Object function

$$L_{\text{PPO}} = \mathbb{E} \min \left(\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)} A_t,\ \text{clip} \left(\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}, 1-\varepsilon, 1+\varepsilon \right) A_t \right)$$

- L_{PPO}: The Proximal Policy Optimization objective function, used to update the policy.
- $\mathbb{E}$: The expectation, indicating an average over a distribution of data. This is typically approximated by sampling a batch of data.
- $\min(\cdot, \cdot)$: The minimum operator, used here to prevent the objective from becoming too large.
- $\frac{\pi(a_t|s_t)}{\pi_{\text{old}}(a_t|s_t)}$: The ratio of the probability under the current policy to the probability under the previous policy for taking action a_t in state s_t.
- A_t: The Generalized Advantage Estimation at time t, which provides an estimate of how much better an action is compared to the average action as predicted by the current policy.
- $\text{clip}(\cdot, 1-\varepsilon, 1+\varepsilon)$: A function that clips its input value to the range specified by $1-\varepsilon$ and $1+\varepsilon$, to avoid excessively large updates.
- The clipped objective: It represents the core idea behind PPO, ensuring that updates to the policy are significant enough to learn but not so large that they destabilize training.

Source: (Schulman, Wolski et al., 2017), (OpenAI et al., 2018) and (Fristedt, 2023)