DEVELOPMENT OF A 2D PLATFORMER GAME AND MACHINE LEARNING MODEL

by

MATHEW MICHAEL DAWSON
URN: 6743842

A dissertation submitted in partial fulfilment of the
requirements for the award of

BACHELOR OF SCIENCE IN COMPUTER SCIENCE

May 2025

Department of Computer Science
University of Surrey
Guildford GU2 7XH

Supervised by: Daniel Gardham

I declare that this dissertation is my own work and that the work of others is acknowledged and indicated by explicit references.

Mathew Michael Dawson
May 2025

Abstract

This dissertation explores deep reinforcement learning in video games by developing a Deep Q-Network (DQN) agent capable of autonomously playing a custom 2D platformer game. The game is built using the Godot, a lightweight open-source game engine, and the model is implemented using the PyTorch deep learning framework. The model is lightweight enough to run on the CPU, but can be accelerated using a GPU. Experimental results show the DQN agent successfully learned effective strategies, achieving times comparable to human players. The agent's performance improved with training, demonstrating the potential of deep reinforcement learning in video games.

Acknowledgements

I would like to thank my supervisor Dr Daniel Gardham for being on board with my idea from our first meeting, as well as his continued guidance and support, had it not been for him I would likely have picked something much more straightforward and within my comfort zone. I would also like to thank my friends and family for encouraging me along the development of this project. Thanks also go to the Godot contributors who work tirelessly, for free, on this wonderful game engine.

Contents

1 Introduction **11**

1.1 Chapter Overview 11

1.2 Project Background 11

1.3 Project Overview 12

1.4 Project Aim and Objectives 12

1.5 Limitations 12

2 Literature Review **14**

2.1 Traditional approaches to AI in video games 14

2.1.1 Finite State Machines 14

2.1.2 Behaviour Trees 15

2.1.3 Goal-Oriented Action Planning 17

2.2 Reinforcement Learning 18

2.2.1 RL Fundamentals and the Markov Decision Process 18

2.2.2 The Bellman equation 20

2.2.3 Deep Reinforcement Learning 20

2.2.3.1 Experience Replay 21

2.2.3.2 Epsilon Decay 22

3 Problem Analysis **23**

3.1 Problem Analysis 23
3.1.1 Video Game Development 23
3.1.2 Network Communication Interface 23
3.1.3 AI Integration into Gameplay 24

4 Design Decisions 25

4.1 Version Control System 25
4.2 Choosing Technologies 25
4.2.1 Choosing a Game Engine 25
4.2.2 Choosing an ML Framework 26
4.2.3 GPU Acceleration 26
4.3 Design of the Game 27
4.4 Spriting and Graphics 29

5 Implementation 31

5.1 Building a Basic Game Environment 31
5.1.1 Procedural Map Generation 33
5.2 Debug Information 34
5.3 State Observation System 35
5.3.1 Observation Scope and Representation 35
5.3.2 Implementation Mechanics 37
5.4 Determining reward 37
5.5 Implementing Networking 38
5.5.1 Communication Protocol Design 38
5.5.2 Operational Modes 39
5.5.3 Communication Cycle Implementation 39
5.6 Implementing the Model 40

5.6.1 Model Architecture 41

5.6.2 Training Algorithm 42

5.6.2.1 Replay Buffer 42

5.6.3 Training Process 43

5.7 Integration of AI into Gameplay 43

6 Evaluation **45**

6.1 Functional Evaluation 45

6.2 AI Performance Evaluation 45

6.2.1 Results 46

6.3 Analysis of results 46

6.4 Conclusion 46

List of Figures

2.1 A simple FSM for an enemy NPC 15

2.2 A simple BT for an enemy NPC 16

2.3 Example of a GOAP planning sequence 18

2.4 A simple MDP Graph example 19

2.5 Architecture of a Deep Q-Network 21

4.1 Classifying tiles by type 28

5.1 Basic Godot game environment 32

5.2 Environment's node hierarchy 32

5.3 Pillars 34

5.4 Snake 34

5.5 Platforms 34

5.6 Procedurally generated map types 34

5.7 Game running with debug information turned on 35

5.8 Visualisation of the observation window 36

List of Tables

6.1 Success Rate (%) and Average Completion Time (s) on Platforms Map 46

6.2 Success Rate (%) and Average Completion Time (s) on Snake Map 46

6.3 Success Rate (%) and Average Completion Time (s) on Platforms Map 46

Abbreviations

AI	Artificial Intelligence
DQN	Deep Q Network
ML	Machine Learning
RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
MDP	Markov Decision Process
NPC	Non-Player Character
FSM	Finite State Machine
BT	Behaviour Tree
GOAP	Goal-Oriented Action Planning
PPO	Proximal Policy Optimisation
A3C	Asynchronous Advantage Actor-Critic
CPU	Central Processing Unit
GPU	Graphics Processing Unit

Chapter 1

Introduction

1.1 Chapter Overview

This chapter will focus on introducing the project with an overview along with its objectives and limitations.

1.2 Project Background

In the area of video games, Artificial Intelligence (AI) has been used for many years to create non-player characters (NPCs) that can interact with players in a believable way. This was first seen in the game "Nim" in 1948 (Wikipedia 2025). This is often done using finite state machines (FSMs) or behaviour trees, which allow NPCs to react to player actions in a way that seems intelligent (Carpenter 2019). However, these methods can be limited in their ability to adapt to new situations, due to being based on pre-defined rules and behaviours. For example, developing an FSM for a procedurally generated game, or one with another amount of randomness involved, can be difficult or even impossible, as the FSM must be able to handle all possible situations that may arise. This is the problem that I will be experimenting with and attempting to address in this project. Reinforcement Learning (RL) provides an alternative approach to video game AI. Unlike traditional methods, RL agents learn through interaction with their environment by receiving rewards for desirable actions. This allows them to develop adaptive strategies without explicitly programmed rules. Building upon RL, Deep Reinforcement Learning (DRL) combines traditional RL algorithms with deep neural networks, enabling agents to process complex visual inputs and learn effective policies from high-dimensional data. DRL has demonstrated remarkable capabilities in video games, as seen in systems like OpenAI's DQN that mastered Atari games (Team 2023) and AlphaGo which defeated world champions in Go (DeepMind 2023). The adaptive nature of DRL makes it particularly promising for procedurally generated or dynamic game environments where traditional AI approaches struggle.

1.3 Project Overview

This project will begin with research and literature review into traditional approaches to AI in video games, as well as reinforcement learning and deep reinforcement learning techniques. Following this research phase, a simple game environment will be developed using the Godot game engine, designed specifically to test and showcase the capabilities of an RL agent. The project will then implement and train an RL agent to operate within this environment, focusing on creating an NPC that can learn and adapt to changes within the game rather than following predetermined patterns.

1.4 Project Aim and Objectives

The primary aim of this project be to attempt a non-standard machine learning based apporach to video game AI.

- Research and understand how existing AI in video games works, then the fundamentals of Reinforcement Learning (RL) and its application in video games.
- Design and implement a video game environment suitable for testing RL agents.
- Develop and train an RL agent to interact with the game.
- Evaluate the performance of the RL agent.
- Implement the RL agent as part of the game, and have it interact with players.

1.5 Limitations

This project will have the following limitations:

- The game will be custom made, rather than an existing one.
- The game will have simple graphics and gameplay, acting more as a "front-end" for the model and it's integration which will be the main focus of the project.
- The model will be lightweight, and will need to run on a single CPU and/or GPU. It must also not consume too much memory. This is to ensure that I can train and run it on my hardware.

Chapter 2

Literature Review

This chapter contains a review of traditional approaches to AI in video games. It then explores RL and deep learning fundamentals, algorithms and data structures, as well as some of their existing applications to video games.

2.1 Traditional approaches to AI in video games

This section explores some existing traditional methods of implementing AI in video games.

2.1.1 Finite State Machines

A Finite State Machine (FSM) is a computational model that has been foundational in video game AI development for decades. FSMs represent an agent's behavior as a set of discrete states, with well-defined transitions between these states triggered by specific conditions or events (Kanade 2021).

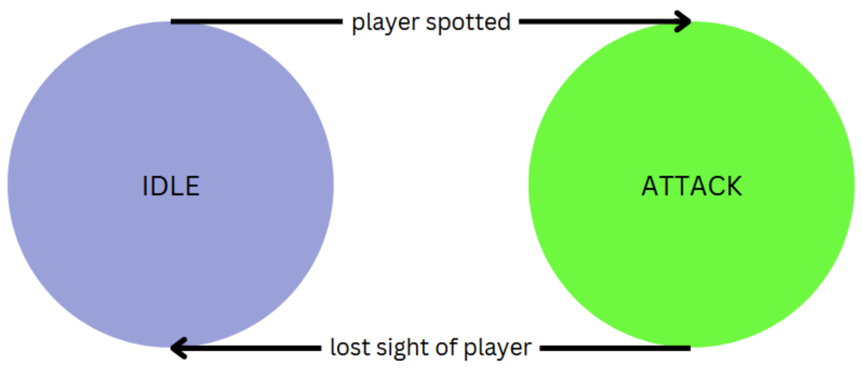


Figure 2.1: A simple FSM for an enemy NPC

Each state encapsulates a particular behavior or action pattern, while transitions define the rules governing when an agent should change its behavior. The simplicity and predictability of FSMs make them particularly suitable for controlling NPCs with straightforward behavioral patterns, as they are computationally efficient and easily debuggable. However, FSMs face significant limitations as complexity increases: the number of states and transitions can grow exponentially, leading to the "state explosion" problem that makes maintenance challenging. Additionally, FSMs struggle with handling concurrent behaviors and can appear rigid when compared to more dynamic AI approaches. Despite these limitations, FSMs remain prevalent in game development due to their intuitive implementation and reliable performance for many common AI tasks.

2.1.2 Behaviour Trees

Behaviour Trees (BTs) represent a significant advancement over FSMs in game AI architecture, offering a hierarchical, modular approach to decision-making. Originally developed for robotics and adopted by the game industry in titles like Halo 2 (Carpenter 2019), BTs organize agent behaviors into a tree structure where leaf nodes represent atomic actions and internal nodes control flow through various composites such as sequences, selectors, and parallels. This structure enables developers to create complex, reusable behavior patterns that can be visually represented and intuitively understood. Unlike FSMs, BTs naturally handle concurrent actions and gracefully manage behavior prioritization through their hierarchical evaluation. BTs excel at creating responsive AI that can react to changing game conditions while maintaining coherent behavior patterns. They facilitate an incremental development approach, allowing designers to progressively refine AI by adding branches without disrupting existing functionality. While BTs require more initial design consideration than FSMs, their scalability, maintainability, and ability to represent sophisticated decision-making logic have made them the standard approach for contemporary game AI systems, particularly in action, strategy, and open-world games where adaptable NPC behavior is critical to player experience.

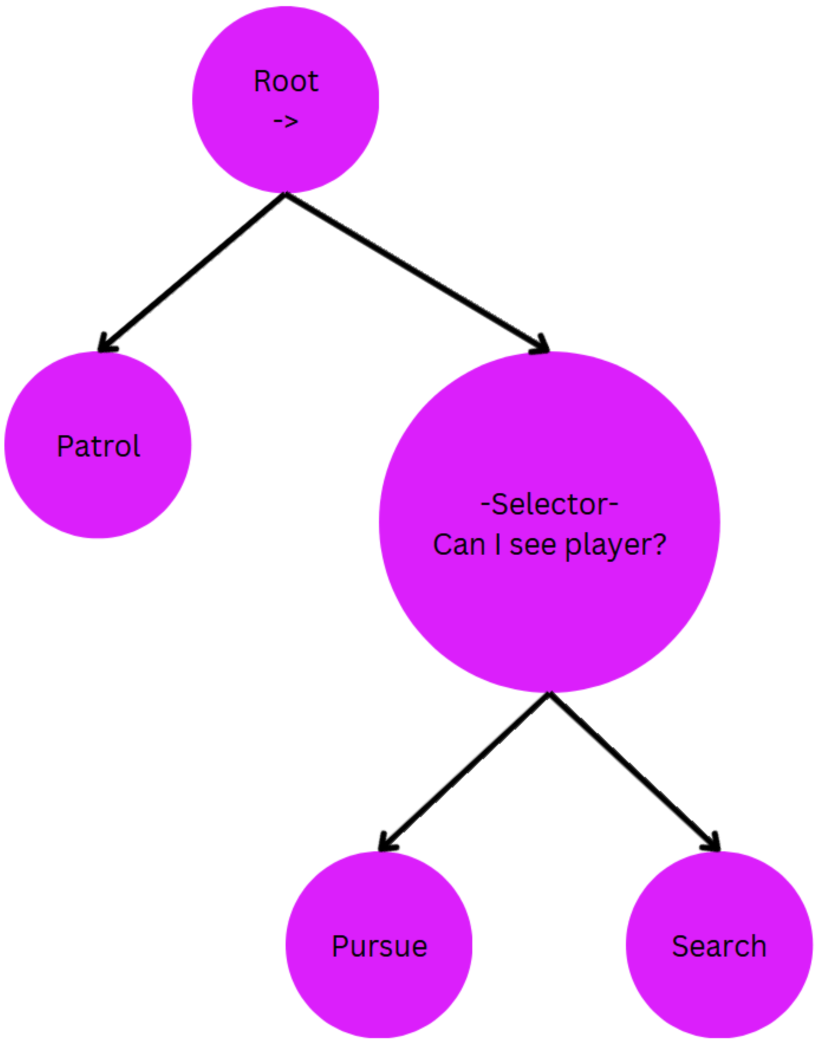


Figure 2.2: A simple BT for an enemy NPC

2.1.3 Goal-Oriented Action Planning

Goal-Oriented Action Planning (GOAP) represents a more dynamic approach to AI decision-making compared to FSMs and BTs. GOAP employs principles from automated planning and means-end analysis to create AI agents that formulate plans to achieve specific goals. Unlike more rigid systems, GOAP agents dynamically determine action sequences by considering current world states, available actions with preconditions and effects, and desired goal states. In a GOAP system, each action is associated with both preconditions (requirements that must be satisfied before the action can be taken) and effects (how the action changes the world state). The AI agent uses planning algorithms, commonly A* search, to find the optimal sequence of actions that transforms the current state into the goal state. This approach allows NPCs to solve problems creatively and adapt to unexpected changes in the game environment. GOAP gained prominence through its implementation in F.E.A.R. (2005) (Thompson 2020), where it produced enemies capable of contextually appropriate tactical behaviors like seeking cover, flanking the player, and coordinating with allies. The system's strength lies in its separation of goals (what the agent wants to achieve) from the specific methods to achieve them. While GOAP offers exceptional adaptability and problem-solving capabilities, it comes with higher computational costs and increased implementation complexity compared to FSMs and BTs. Despite these challenges, GOAP remains valuable for games requiring sophisticated AI that can respond intelligently to dynamic and unpredictable gameplay scenarios.

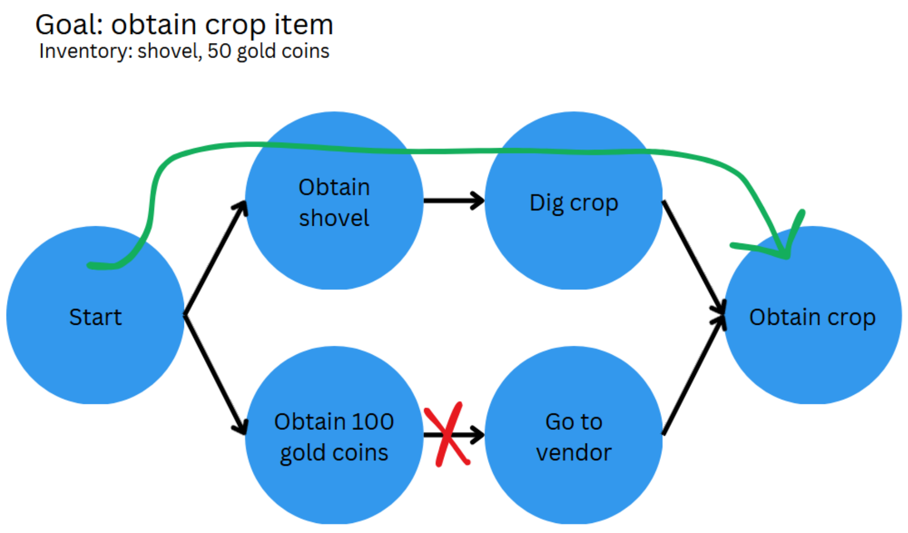


Figure 2.3: Example of a GOAP planning sequence

2.2 Reinforcement Learning

This section explores fundamental RL concepts and algorithms.

2.2.1 RL Fundamentals and the Markov Decision Process

Reinforcement Learning (RL) represents a departure from traditional game AI approaches by focusing on learning optimal behaviors through trial and error interaction with an environment. Unlike FSMs, BTs, or GOAP systems that rely on pre-programmed rules, RL agents improve their decision-making capabilities through experience.

At its core, RL is formalized as a Markov Decision Process (MDP) consisting of:

- A set of states S representing all possible situations an agent may encounter
- A set of actions A that the agent can take
- Transition probabilities $P(s'|s, a)$ defining the likelihood of moving to state s' after taking action a in state s
- A reward function $R(s, a, s')$ providing feedback on the quality of decisions

- A discount factor $\gamma \in [0, 1]$ determining the importance of future rewards

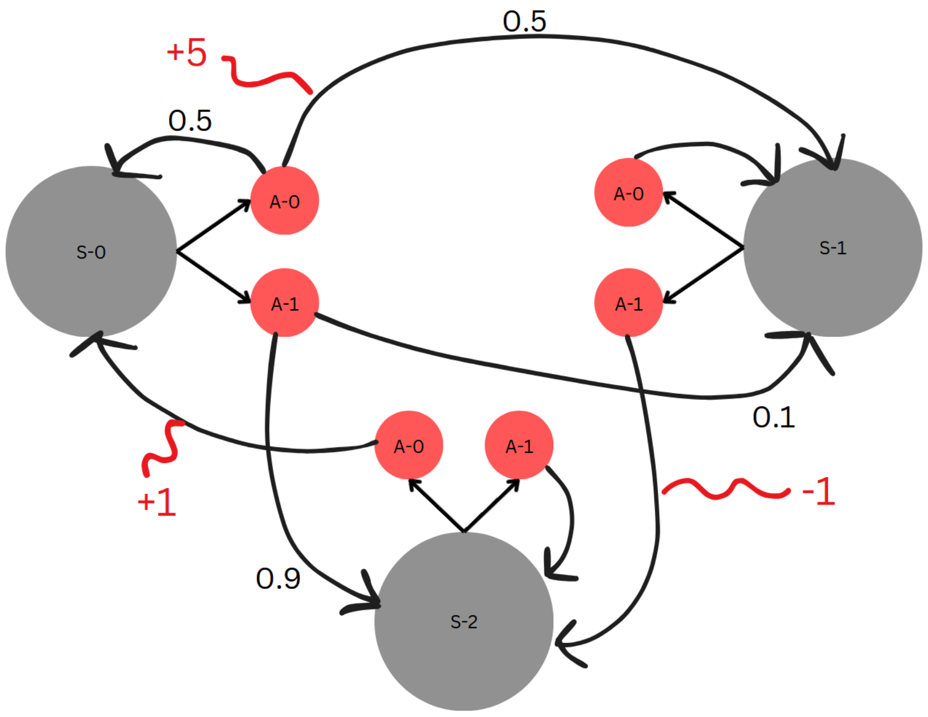


Figure 2.4: A simple MDP Graph example

S Nodes represent states and A nodes represent actions. Some actions can result in more than one state, the transition probabilities are marked in black. Some actions result in a positive or negative reward, marked in red.

The agent's objective is to learn a policy π that maps states to actions in a way that maximizes the expected cumulative reward. This optimisation process balances immediate rewards against long-term consequences, addressing the fundamental exploration-exploitation dilemma: whether to capitalize on known good strategies or explore new possibilities that might yield better results. RL algorithms generally fall into three categories: value-based methods (like Q-learning), policy-based methods (such as policy gradients), and actor-critic approaches that combine aspects of both. The choice of algorithm depends on factors including the complexity of the state space, whether the environment is fully observable, and computational constraints. Unlike traditional AI techniques, RL offers adaptability to unexpected situations and can discover novel strategies beyond designer expectations. However, these advantages come with challenges including high sample complexity (requiring many environment interactions), difficulty in specifying appropriate reward functions, and potential convergence to suboptimal solutions when trained in limited scenarios.

2.2.2 The Bellman equation

The Bellman equation provides a mathematical foundation for solving Markov Decision Processes (MDPs) by establishing recursive relationships between value functions of different states. It serves as the cornerstone of many RL algorithms. The Bellman equation defines the optimal value function $V^*(s)$ for each state s as:

$$V^*(s) = \max_a \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^*(s') \right]$$

This recursive formulation elegantly captures the principle that the value of a state equals the reward obtained from the best action plus the discounted expected value of the successor state. For policies, the Bellman expectation equation becomes:

$$V^\pi(s) = \sum_a \pi(a|s) \left[R(s,a) + \gamma \sum_{s'} P(s'|s,a) V^\pi(s') \right]$$

Similarly, the action-value function $Q^*(s,a)$ represents the expected return starting from state s, taking action a, and thereafter following the optimal policy:

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

Many RL algorithms iteratively apply these equations to converge to optimal solutions. Value iteration repeatedly updates state values using the Bellman optimality equation until convergence. Policy iteration alternates between policy evaluation (calculating values for the current policy) and policy improvement (selecting better actions based on updated values). While solving the Bellman equation exactly requires complete knowledge of the environment's dynamics (transition probabilities and rewards), most practical RL algorithms like Q-learning approximate solutions through sampling and iterative updates, enabling agents to learn optimal policies through experience rather than explicit environment models.

2.2.3 Deep Reinforcement Learning

Deep Reinforcement Learning (DRL) combines classical RL with deep neural networks to handle high-dimensional state spaces that would be intractable with traditional tabular methods.

This integration enables RL to operate effectively in complex environments with visual inputs, continuous action spaces, and intricate state representations common in modern video games. The breakthrough Deep Q-Network (DQN) algorithm, introduced by DeepMind in 2015, demonstrated superhuman performance on Atari games using only pixel inputs and game scores. DQN employs several key innovations including:

- Experience replay, which stores and randomly samples past experiences to break correlation between sequential samples
- Target networks that stabilize training by reducing moving target problems
- Convolutional neural networks that process visual information effectively

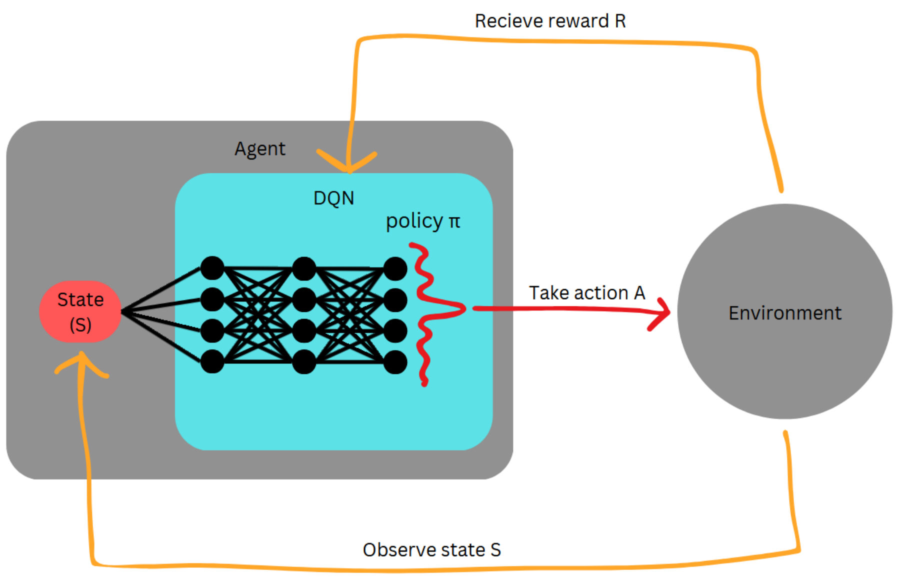


Figure 2.5: Architecture of a Deep Q-Network

2.2.3.1 Experience Replay

Experience replay is a fundamental technique in deep reinforcement learning that addresses the challenge of correlated samples in sequential learning. While traditional online RL updates parameters based on immediate experiences, experience replay stores transitions (s_t, a_t, r_t, s_{t+1}) in a buffer and samples them randomly during training.

This mechanism provides several crucial benefits:

- It breaks temporal correlations between consecutive training samples that can lead to unstable learning
- It increases data efficiency by allowing experiences to be used multiple times
- It reduces the variance of updates by averaging over different states and actions

The replay buffer typically maintains a fixed-size queue that stores the N most recent experiences. During the learning phase, mini-batches of transitions are randomly sampled from this buffer to update the neural network parameters, following:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a'} Q(s',a') - Q(s,a)]$$

More sophisticated variants include prioritized experience replay, which samples important transitions more frequently based on their temporal-difference error, and hindsight experience replay, which retrospectively generates additional training signals from failed attempts.

2.2.3.2 Epsilon Decay

Epsilon-decay represents a critical strategy in exploration-exploitation balance for reinforcement learning algorithms like DQN. During training, agents follow an ϵ-greedy policy where they select the highest-valued action with probability $1-\epsilon$, and choose a random action with probability ϵ. Epsilon decay gradually reduces this exploration parameter over time according to a schedule:

$$\varepsilon_t = \varepsilon_{\text{end}} + (\varepsilon_{\text{start}} - \varepsilon_{\text{end}})e^{-\lambda t}$$

where $\varepsilon_{\text{start}}$ is the initial value (often near 1.0), ε_{end} is the minimum value (typically 0.01-0.1), and λ determines the decay rate. This approach ensures thorough exploration of the environment initially, when the agent's knowledge is limited, while gradually transitioning to exploitation of learned strategies as training progresses. Properly tuned epsilon decay significantly impacts learning efficiency and final policy quality, particularly in environments with sparse rewards or complex optimal strategies.

Chapter 3

Problem Analysis

This chapter contains an analysis of the problem and breaks it down into sub problems. It then sets out the functional and non-functional requirements.

3.1 Problem Analysis

The problem can be broken down into three main components:

3.1.1 Video Game Development

Creating a basic video game involves designing core gameplay mechanics, implementing player controls, creating a game environment, and establishing basic game rules and objectives. This requires selecting an appropriate game engine that supports the planned features and performance requirements. Core gameplay mechanics and rules must be defined to provide an engaging player experience, while visual assets and user interface elements need implementation to create a cohesive visual identity. Player control systems need to be intuitive and responsive, allowing for meaningful interaction with the game world. Finally, clear win/lose conditions need establishment to provide players with goals and motivation.

3.1.2 Network Communication Interface

Connecting an external AI agent to the game requires establishing a reliable communication channel. This involves designing a networking protocol that efficiently transmits necessary game state information and receives AI decisions. A server-client architecture must be implemented to facilitate this exchange, along with data serialization and parsing mechanisms to ensure proper interpretation of messages on both ends. Synchronization between game state and AI agent is critical to maintain consistency, while also managing network latency and potential disconnections to ensure smooth gameplay even under non-ideal network conditions.

3.1.3 AI Integration into Gameplay

Incorporating the AI agent as part of the gameplay loop involves defining the AI's role within the game context, whether as an opponent, ally, or environmental element. Interfaces for AI action and observation must be created to allow the AI to both perceive the game state and affect it meaningfully. Game state representation needs careful implementation to provide the AI with relevant information in a consumable format. The AI's capabilities must be balanced with player experience to ensure engaging but fair gameplay. Additionally, feedback mechanisms must be designed to showcase AI behavior, allowing players to understand and respond to AI actions appropriately.

Chapter 4

Design Decisions

Over the course of the development of my project, I had to make numerous design decisions for each part. This chapter lays out and explains the design decisions I made, along with justifications for them, and details the implementation process that followed these decisions.

4.1 Version Control System

Throughout the development of the project, I will use the industry-standard Git version control system. This allows me to keep snapshots of the project over time, so I can revert if anything goes wrong, as well as keeping the project backed up to GitHub.

4.2 Choosing Technologies

4.2.1 Choosing a Game Engine

To develop the game frontend, I needed to choose a game engine or game development framework. Throughout my research, I considered three different options, Godot, Unity and Pygame. I selected Godot over Unity and Pygame after careful consideration of my project's requirements. While Unity offers powerful features, its proprietary nature, heavyweight installation, and steep learning curve made it less suitable for my academic project. Unity's reliance on the C Sharp programming language would also have required additional time investment to master. Pygame, despite its Python compatibility that would have allowed native interfacing with

PyTorch, lacks a graphical development environment and many built-in features that would speed up development. In contrast, Godot offered the perfect balance: it's open-source and free, extremely lightweight (only 130MB), uses a Python-like language (GDScript) that was quick to learn, features an intuitive interface I was already familiar with, and provides excellent 2D capabilities that matched my game requirements. Additionally, Godot's plain text resources integrate seamlessly with Git, its cross-platform compatibility ensures the game works on both Windows and Linux environments (critical for working between home and university), and its networking features provide essential connectivity to the backend model.

4.2.2 Choosing an ML Framework

For the model backend, there were only two real options - PyTorch and Tensorflow. I chose PyTorch over TensorFlow for my machine learning framework due to several key advantages. While TensorFlow offers a robust ecosystem and production-ready deployment tools, PyTorch's more intuitive API and dynamic computation graph better suited the experimental nature of this project. TensorFlow's static graph design, though efficient for production, would have hindered the rapid prototyping and debugging I needed during development. PyTorch's Python-first approach also aligned better with my existing workflow, offering seamless integration with Python data science libraries like NumPy and Pandas. The framework's extensive documentation, community support, and built-in tools for neural network development made implementation more straightforward. Additionally, PyTorch's native support for GPU acceleration provided the necessary computational power for model training, and its reinforcement learning libraries offered ready-made solutions for developing game AI agents. TensorFlow's steep learning curve and more verbose syntax would have increased development time without providing significant benefits for this particular academic project.

4.2.3 GPU Acceleration

During development, I investigated GPU acceleration options for the machine learning components of my project. Since I have an AMD graphics card that isn't compatible with CUDA, I experimented with **pytorch-directml** as an alternative acceleration method. DirectML, developed by Microsoft, functions as a translation layer that enables GPU acceleration of machine learning workloads through DirectX. This implementation provides acceleration capabilities for

AMD graphics cards but comes with the limitation of only functioning on Windows operating systems. However, after thorough testing, I found this additional layer unnecessary as my CPU handled both model training and execution without performance issues. Nevertheless, to ensure optimal performance across different hardware configurations, the final implementation includes automatic detection for CUDA support. When running on NVIDIA hardware with appropriate drivers, the application automatically leverages GPU acceleration to enhance model training speed and inference performance. This design decision maintains the project's versatility across various computing environments while optimizing performance when possible.

4.3 Design of the Game

For the game component of my project, I decided to develop a 2D tile-based platformer, similar to Super Mario Bros. This decision was driven by several practical considerations. First, 2D platformers offer a good balance between implementation complexity and gameplay depth, making them ideal for academic projects with limited timeframes. The tile-based approach simplifies level design, collision detection, and environmental interactions while still allowing for creative gameplay mechanics. The tile-based approach is also ideal for RL, as the environment is formed of tiles that can be classified by type, numbered, and then fed directly into a model.

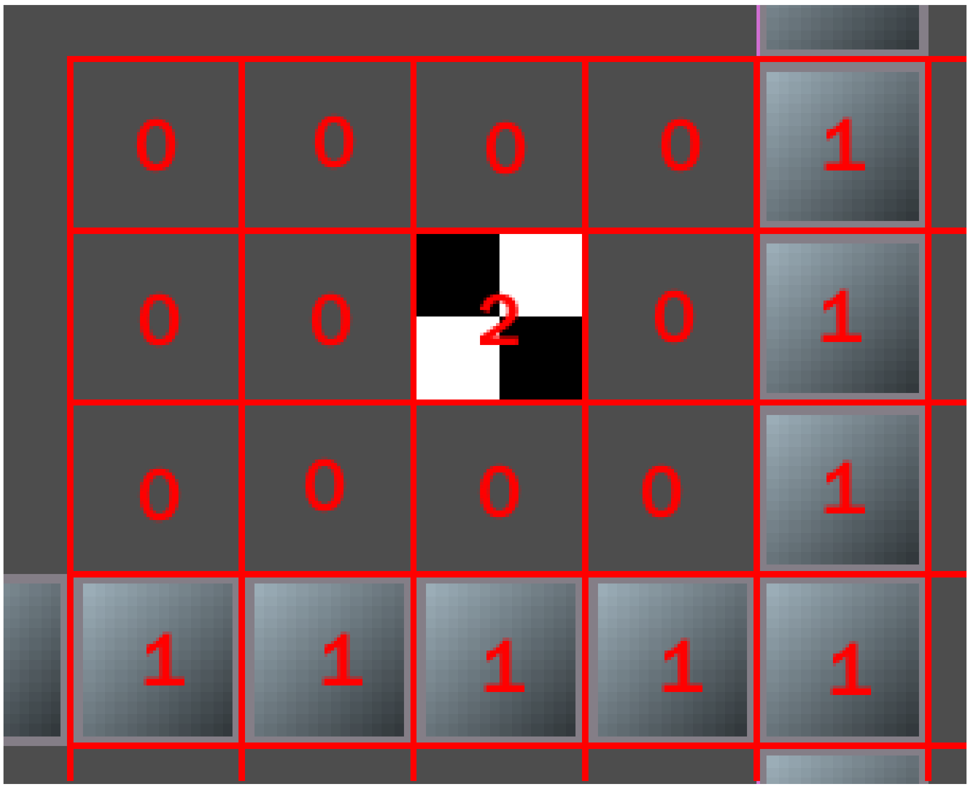


Figure 4.1: Classifying tiles by type

From a technical perspective, 2D platformers require fewer computational resources than 3D games, enabling me to focus more processing power on the machine learning components rather than graphics rendering. Godot's excellent 2D capabilities made this genre particularly suitable, as the engine provides robust tools for sprite animation, tilemap management, and physics-based movement. Additionally, platformers offer clear, discrete states and actions that translate well to reinforcement learning problems. The character can move left, right, jump, or perform other specific actions, while the game state can be represented efficiently for the ML model to process. This clarity in the action space makes platformers an excellent testbed for AI development, as the model must learn fundamental gaming concepts like obstacle avoidance, timing, and spatial navigation. The tile-based structure also facilitates procedural level generation, allowing me to create varied environments for training the AI without manually designing each level. This was particularly important for developing robust models that could generalize across different scenarios rather than overfitting to specific level layouts.

I decided to design the game around a clear, measurable objective: completing a time-based obstacle course. This design choice was deliberate as it provides several advantages for both gameplay and AI development. From a gameplay perspective, a timed obstacle course offers an intuitive goal that requires no elaborate explanation - players simply need to reach the end as quickly as possible. This straightforward objective creates natural replay value as players (human or AI) attempt to optimize their route and execution to achieve faster completion times. From a machine learning standpoint, this objective is ideal because it creates a well-defined reward function where success can be quantified precisely through time measurements. The faster an agent completes the course, the better its performance, providing a clear optimization target for reinforcement learning algorithms. This design also allows for incremental learning progression. An AI agent can first learn the fundamental task of reaching the goal, then gradually optimize for speed - mirroring how human players typically approach such challenges. The continuous nature of the time metric (as opposed to binary success/failure) gives the learning algorithm more nuanced feedback about small improvements in performance. Additionally, timed courses naturally incorporate key platforming challenges like precise jumping, obstacle avoidance, and route optimization, creating a rich environment for AI learning while remaining accessible to human players for comparison.

4.4 Spriting and Graphics

A small part of this project involves the creation of graphics, so we can see what is going on. In 2D games these are referred to as sprites. For the graphical elements of the game, I opted to create all assets using Aseprite, a specialized pixel art editor. This decision aligned perfectly with both the technical requirements of the project and the aesthetic direction I wanted to pursue. I deliberately chose a pixel art style for several reasons. First, the simplicity of pixel graphics reduced development time, allowing me to focus more on the AI components that were the primary focus of my research. Second, pixel art's grid-based structure naturally complemented the tile-based nature of the platformer, creating visual coherence between the mechanics and aesthetics. Finally, the minimalist style ensured that game elements remained visually distinct and easily recognisable. Using Aseprite, I created character sprites for the player character and AI character, as well as tiles for terrain and the finish line. I maintained a consistent color palette throughout to ensure visual harmony, limiting myself to a small set of colors that clearly differentiated between game elements while providing adequate contrast. For creating the user interface I used two fonts: PressStart2P, and the default Godot font provided with the engine.

Both are open source and free to use.

Chapter 5

Implementation

5.1 Building a Basic Game Environment

My first task was to build a basic game environment within Godot that could then be built further upon. This consists of:

- **Player Character**: I implemented a character with basic movement controls including running left/right, jumping, and falling with appropriate physics, using Godot's built in 2D physics system.
- **Camera**: The game's camera is a Camera2D node that remains static providing a view of the whole map.
- **Tile Based Map**: I created a tilemap system using Godot's TileMapLayer node, allowing me to construct levels from reusable tiles representing walls, floors and ceilings.
- **Kill Plane**: At the bottom of the map, there is a player collidable box that will reset their position if it is touched. This ensures that the player cannot fall out of the map endlessly.
- **Finish Line and Timer**: Each map contains a finish line tile. As soon as the player is instantiated, their timer starts and will stop once they touch the finish line.

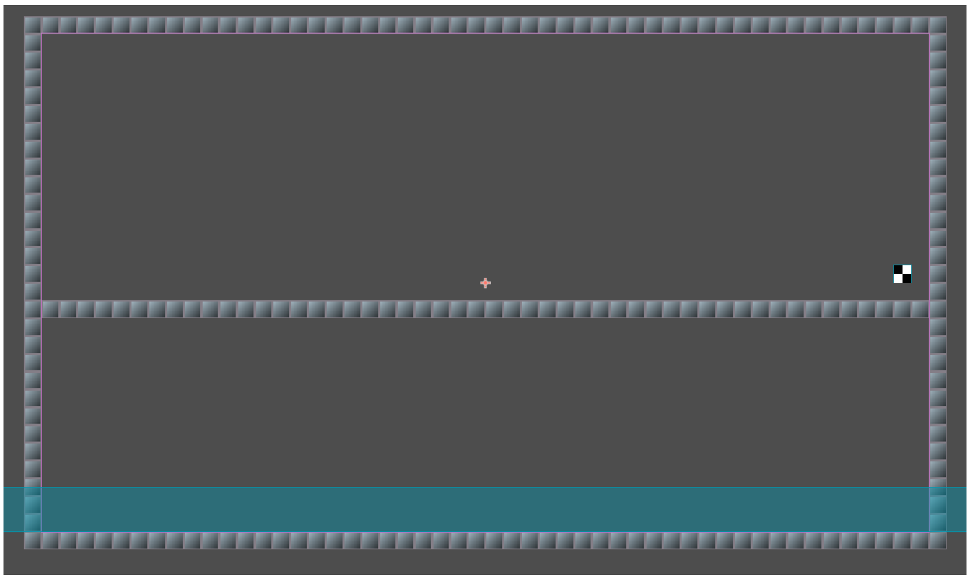


Figure 5.1: Basic Godot game environment

Godot's node based architecture allowed me to neatly lay these all out.

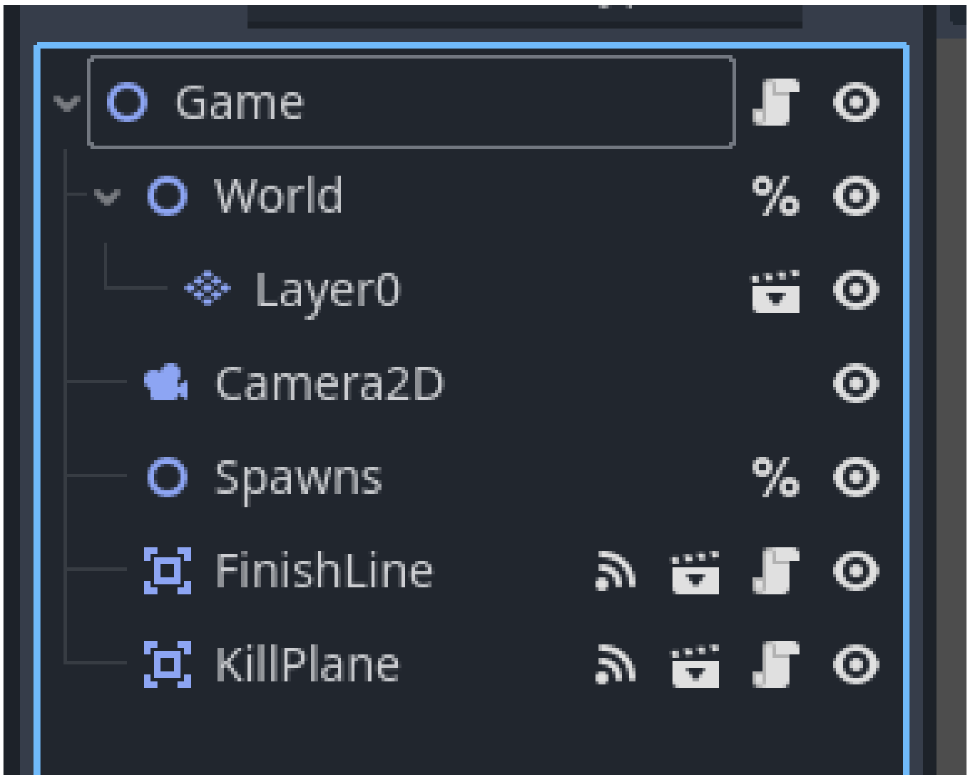


Figure 5.2: Environment's node hierarchy

Building upon this foundational environment, I implemented a separate AI Player node to serve as the agent controlled by the reinforcement learning model. This AI Player functions as a physics-affected body with identical physical properties to the player character, ensuring that both human and AI participants operate under the same environmental constraints. The key distinction is that while the human player responds to keyboard or controller inputs, the AI Player receives its movement commands exclusively from the reinforcement learning model via the network communication system. The separate node implementation also facilitates the competitive gameplay structure, allowing the AI to demonstrate a complete run before the human player attempts the same procedurally generated level. To facilitate the training process, the AI Player keeps track of "stuck time". This is a counter that starts counting up if the AI has not made horizontal progress towards the goal. After 2 seconds have elapsed, the AI is reset to it's starting point and a fail is counted. This helps the training process as I would often find that the model would get stuck during the early stages of training.

5.1.1 Procedural Map Generation

To evaluate the agent's adaptability across diverse environments, I implemented a procedural map generation system that could create varied challenge scenarios. This system produces three distinct map types:

- **Pillars**: Generates vertical obstacles of variable height and width between the player starting position and the goal. This design tests the agent's ability to navigate vertical challenges through precise jumping.
- **Snake**: Creates a randomized horizontal tunnel pathway that winds between the starting position and the goal. This configuration evaluates the agent's capacity to follow constrained paths.
- **Platforms**: Places discontinuous platforms at randomized heights across the map. This challenging layout requires the agent to make calculated jumps between platforms while avoiding fatal falls through the gaps.

The procedural generation system ensures that each training episode presents the agent with a unique layout while maintaining consistent difficulty parameters. This approach prevents the model from simply memorizing specific map solutions and instead encourages the development of generalized navigation strategies. The map type can be selected by the user from the main menu with a drop down selector box.

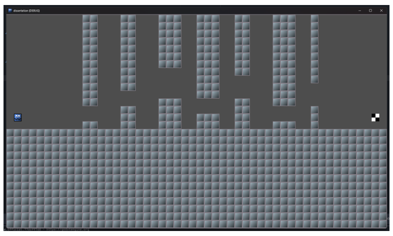


Figure 5.3: Pillars

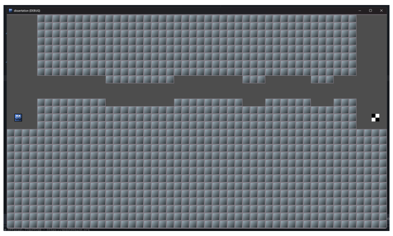


Figure 5.4: Snake

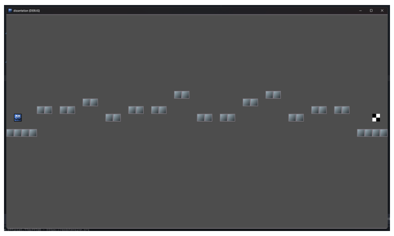
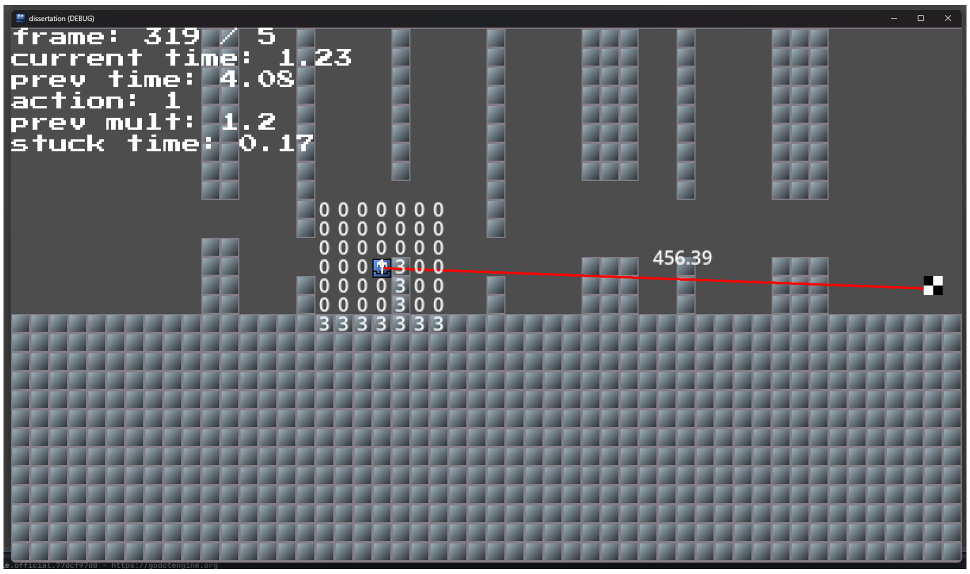


Figure 5.5: Platforms

Figure 5.6: Procedurally generated map types

5.2 Debug Information

To facilitate the development process, I implemented comprehensive debug information within the game environment. This included a visual representation of the agent's current observation state, allowing direct inspection of what the model "sees" from the 7×7 grid centered on its position. Distance metrics were also continuously displayed, showing the agent's proximity to the goal and enabling real-time evaluation of progress. Additional timing metrics were incorporated to track performance data relevant to the agent's decision-making processes. These debugging tools proved invaluable during development, providing immediate visual feedback on the observation system's functionality and the agent's environmental perception. These features can be toggled via a switch on the main menu.



```
frame: 319 / 5
current time: 1.23
prev time: 4.08
action: 1
prev mult: 1.2
stuck time: 0.17
```

```
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0 0 0 0 0
0 0 0   3 0 0
0 0 0 0 3 0 0
0 0 0 0 3 0 0
3 3 3 3 3 3 3
```

Figure 5.7: Game running with debug information turned on

5.3 State Observation System

For the reinforcement learning agent to make informed decisions, it needs to perceive its environment effectively. This required designing an observation system attached to the AI Player that captures relevant environmental information while maintaining computational efficiency.

5.3.1 Observation Scope and Representation

While capturing the entire environment state would provide the agent with complete information, such an approach proved prohibitively expensive from a computational perspective. A full environment representation would significantly increase the input dimensionality of the neural network, require substantially more training time to converge, demand greater memory resources during both training and inference, and scale poorly with larger environment sizes. These computational constraints led me to pursue a more efficient observation approach that would balance environmental awareness with practical implementation considerations.

After experimentation with various observation window sizes, I determined that a 7×7 grid centered on the player offered an optimal balance between environmental awareness and computational efficiency. This window size provides sufficient context about nearby obstacles, pathways, and goals while keeping the state representation compact and manageable. The constrained field of view also encourages the agent to develop more generalizable navigation strategies rather than memorizing specific map layouts, potentially improving transfer learning capabilities across different environments. The agent perceives a 7×7 grid centered on its position, providing sufficient contextual awareness of the immediate surroundings.

This observation window uses a simple numerical encoding scheme:

- **0**: Empty space / traversable areas
- **1**: Player character (Always at the centre of the grid)
- **2**: Finish line
- **3**: Walls / non-traversable terrain

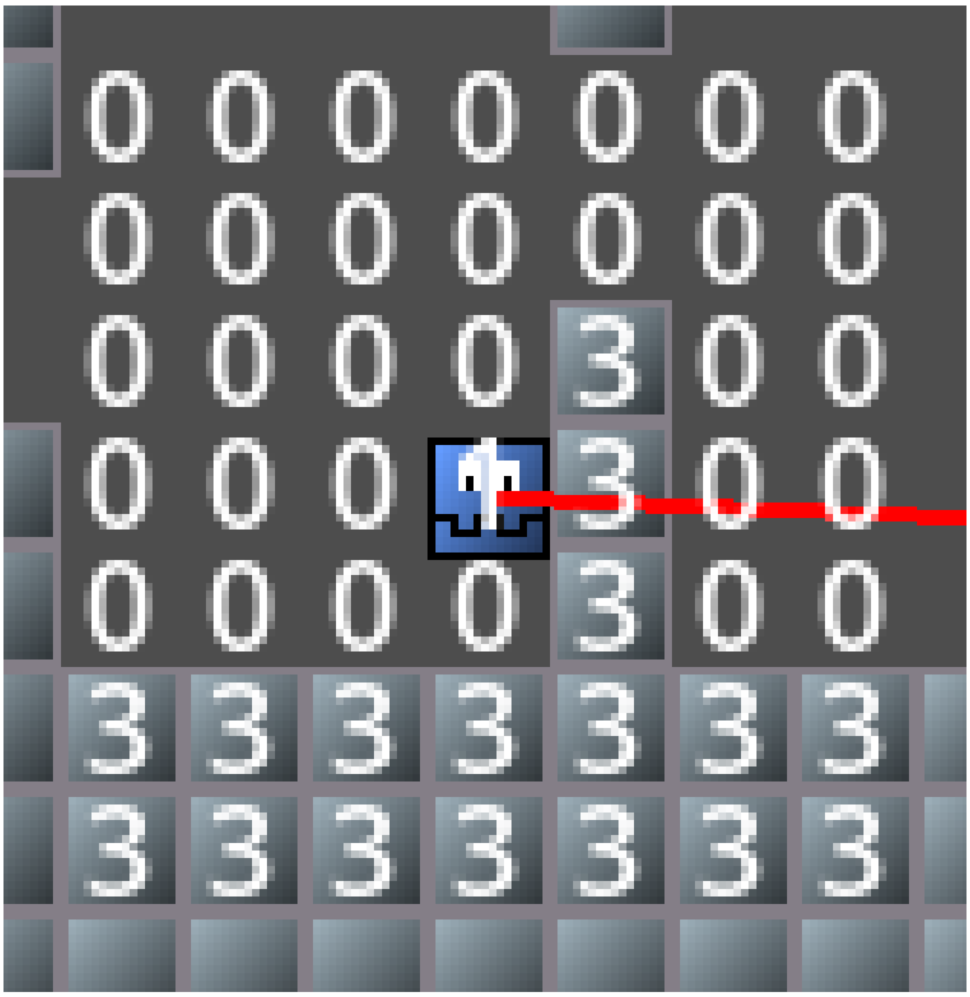


Figure 5.8: Visualisation of the observation window

5.3.2 Implementation Mechanics

The observation mechanism operates on each frame update, scanning the surrounding tiles relative to the agent's position. For each coordinate within the 7×7 grid, the system queries the tilemap to determine the tile type at that location. This information is then encoded according to the numerical scheme and assembled into a two-dimensional array. This process is run twice, once for the initial observation, and again for the second observation used alongside the reward.

This representation offers several advantages:

- **Translation invariance**: The agent learns based on relative positions rather than absolute map coordinates
- **Computational efficiency**: The fixed-size input simplifies network architecture and reduces processing requirements
- **Informational clarity**: The categorical encoding provides clear distinctions between critical environmental features

Once constructed, this state representation is transmitted to the reinforcement learning model, which uses it as input for determining the next optimal action.

5.4 Determining reward

Designing an effective reward function was critical for the reinforcement learning agent's success. After extensive experimentation, I implemented a straightforward distance-based reward mechanism that provides immediate feedback after each action:

- **+1 reward** when the agent moves closer to the finish line
- **-1 penalty** when the agent moves further from the finish line
- **+1.2/0.8 additional reward** upon reaching the goal. The higher reward is granted when the agent reaches the goal faster than the previous attempt, and vice versa.

This reward function is run once each frame, alongside the second observation, after the model has made an action.

This simple approach encourages the agent to make continuous progress toward the objective while maintaining computational efficiency. The reward function balances immediate guidance with the freedom to discover optimal paths independently. During development, I explored several alternative reward structures, including proportional multipliers ($\times1.2/\times0.8$) and larger magnitude rewards (+600 for goal completion, -300 for falling). However, these higher values led to training instability, causing the model to struggle with convergence. The proportional multipliers proved beneficial and were retained, while the reward magnitudes were scaled back to ±1 to ensure stable learning. I also experimented with line-of-sight rewards using raycasting techniques, where the agent would receive additional rewards when establishing visual contact with the goal. After testing, this approach was ultimately not incorporated into the final implementation as it did not significantly improve performance while adding computational overhead.

A significant limitation of this reward structure is that it discourages exploratory backtracking, effectively requiring level designs to maintain a linear progression toward the goal. This constraint arises because the agent receives consistent penalties when moving away from the finish line, even when such movements might be strategically necessary to navigate complex environmental features. While this simplifies the learning process for straightforward courses, it potentially limits the agent's ability to discover optimal solutions in more complex, non-linear environments where temporary retreat may be advantageous.

5.5 Implementing Networking

To facilitate communication between the Godot game environment and the Python-based reinforcement learning model, I developed a robust networking system using TCP sockets and JSON encoding. This approach allows the game state to be effectively transmitted to the model for processing, and action decisions to be received back for execution within the game environment.

5.5.1 Communication Protocol Design

After evaluating several options for inter-process communication, I selected a TCP socket-based approach with JSON encoding for several key reasons:

- **Reliability**: TCP's connection-oriented nature ensures that no state observations or action commands are lost during transmission

- **Cross-platform compatibility**: The socket-based approach works seamlessly across different operating systems. The JSON encoding also ensures that the state and reward data is encoded and decoded the same on both the Godot side and the Python side.
- **Low implementation overhead**: Both Godot and Python offer built-in libraries for TCP communication without requiring additional dependencies

To implement this, I used the *socket* library on the python side, and a *StreamPeerTCP* node on the Godot side.

5.5.2 Operational Modes

The Python component of the system was designed with two distinct operational modes:

- **Training Mode**: Incorporates epsilon-greedy exploration strategies and periodically saves model checkpoints to preserve training progress. This mode focuses on model improvement through experience gathering and optimization.
- **Play Mode**: Loads the latest saved checkpoint and plays deterministically with epsilon set to zero, demonstrating the current capabilities of the trained model without further exploration.

5.5.3 Communication Cycle Implementation

As we are working within a game context, to get the state and reward from any given action, we must wait until the next frame of the game for the result of the action to be processed.

The core networking loop on the Godot side follows a structured pattern of state transmission and action reception:

1. First, the game engine calculates and sends the reward from the previous action along with the current state observation
2. It then awaits confirmation that the Python script is ready for the next cycle
3. Upon confirmation, the game sends a fresh observation of the current environment state

4. Finally, it receives the model's selected action (encoded as an integer: 0 for left movement, 1 for right movement, 2 for jump) and executes this action within the game environment

This communication cycle repeats once each frame during gameplay, maintaining a consistent feedback loop between the game environment and the learning model. The implementation ensures that state observations, rewards, and actions are synchronized properly, preventing timing issues that could otherwise lead to misaligned learning experiences.

The code excerpt below illustrates the essential communication pattern written in GDScript:

```
# Transmit previous reward and current state
var model_reward : float = reward()
peer.put_float(model_reward)
peer.poll()
var next_state : Dictionary = observe()
var next_state_encoded : PackedByteArray =
    JSON.stringify(next_state).to_utf8_buffer()
peer.put_data(next_state_encoded)
peer.poll()

# Await ready signal from Python
peer.poll()
var _isready : String = peer.get_string(5)

# Send observation for decision making
var visibleTiles : Dictionary = observe()
var visibleTilesEncoded : PackedByteArray =
    JSON.stringify(visibleTiles).to_utf8_buffer()
peer.put_data(visibleTilesEncoded)
peer.poll()

# Receive and execute model's action decision
peer.poll()
var model_action : int = peer.get_u8()
action(delta, model_action)
```

This networking implementation provides the critical infrastructure that allows the reinforcement learning agent to interact with the game environment, facilitating the entire learning process.

5.6 Implementing the Model

For the reinforcement learning component, I implemented a Deep Q-Network (DQN) architecture with convolutional layers to process the spatial information from the game environment.

5.6.1 Model Architecture

After considering various neural network architectures, I selected a convolutional DQN model that could efficiently process the 2D grid observation data from the game environment:

- **Input Layer**: Accepts the 7×7 grid observation (49 values) reshaped into a 2D format
- **Convolutional Layer**: A 2D convolutional layer with 16 filters, kernel size of 3×3, stride of 1, and padding of 1, enabling the network to detect spatial patterns in the environment
- **Hidden Layers**: Two fully-connected layers with 128 neurons each and ReLU activation functions
- **Output Layer**: A fully-connected layer with 4 outputs corresponding to the possible actions (move left, move right, jump, do nothing)

The convolutional layer was particularly important for this implementation as it allows the model to recognize patterns in the spatial arrangement of walls, empty spaces, and the goal within the observation window. This spatial awareness is critical for navigating the platformer environment effectively. Below is a code listing of the Pytorch model definition.

```
class DQN(nn.Module):
def __init__(self, n_observations, n_actions):
    super(DQN, self).__init__()
    # Reshape the input for Conv2D
    self.input_dim = int(np.sqrt(n_observations))

    # Convolutional layer
    self.conv1 = nn.Conv2d(1, 16, kernel_size=3, stride=1, padding=1)

    # Calculate the size after convolution for the fully connected layer
    conv_output_size = 16 * self.input_dim * self.input_dim

    self.fc1 = nn.Linear(conv_output_size, 128)
    self.fc2 = nn.Linear(128, 128)
    self.fc3 = nn.Linear(128, n_actions)

def forward(self, x):
    # Reshape input to [batch_size, channels, height, width]
    batch_size = x.size(0)
    x = x.view(batch_size, 1, self.input_dim, self.input_dim)

    # Apply convolution and activation
    x = F.relu(self.conv1(x))

    # Flatten for fully connected layers
    x = x.view(batch_size, -1)
```

```
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

5.6.2 Training Algorithm

I implemented a standard Q-learning algorithm with a target network for stable training. The training strategy incorporated several key techniques to enhance learning and stability. Epsilon-greedy exploration was implemented, beginning with a high exploration rate that gradually decreased over time, creating an effective balance between exploring new strategies and exploiting learned knowledge. This approach allowed the agent to discover potentially valuable actions early in training while increasingly focusing on optimal strategies as training progressed. A target network was employed as a separate network with parameters periodically updated from the main network, providing stable Q-value targets during training and mitigating the risk of divergence that can occur in Q-learning when the same values are both predicted and used as training targets. The Adam optimizer was selected with a learning rate of 0.001 for its adaptive learning rate capabilities, allowing for efficient parameter updates that dynamically adjusted based on gradient history. This optimizer helped navigate the complex loss landscape more effectively than traditional stochastic gradient descent. Finally, Mean Squared Error was used as the loss function between predicted Q-values and target Q-values, providing a differentiable metric that penalized large prediction errors more heavily, guiding the network toward more accurate action-value estimations and ultimately more optimal decision-making policies.

5.6.2.1 Replay Buffer

I implemented an experience replay buffer to enhance training stability and efficiency. This buffer stores transitions (state, action, reward, next_state) as the agent interacts with the environment:

```
class ReplayMemory(object):
    def __init__(self, capacity):
        self.memory = deque([], maxlen=capacity)

    def push(self, state, action, next_state, reward):
        self.memory.append((state, action, next_state, reward))
    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)
```

However, after extensive testing, I found that in this specific implementation, the replay buffer provided minimal benefits. The relative simplicity of the environment and model architecture meant that the agent achieved sufficient performance through direct online updates. Furthermore, with the limited training duration in the experimental setting, the buffer rarely reached capacity before training concluded. This observation aligns with research suggesting that for simpler environments with limited state spaces, the overhead of experience replay may not always justify its benefits.

5.6.3 Training Process

The training process operated on a cycle-by-cycle basis with the game environment:

1. Receive the current state observation from the Godot environment
2. Select an action using the epsilon-greedy policy
3. Send the selected action to the game environment for execution
4. Receive the reward and next state observation
5. Update the Q-network.
6. Periodically update the target network parameters

During training, the epsilon value (controlling exploration vs. exploitation) decayed from an initial value of 0.9 to a minimum of 0.1, allowing the agent to gradually transition from predominantly random exploration to informed decision-making based on learned values.

5.7 Integration of AI into Gameplay

The final integration of the AI component into gameplay implements a straightforward competitive framework. The system follows a simple procedural flow: The game procedurally generates a new stage layout when a level begins. First, the AI agent performs a complete navigation run through this generated environment, establishing a baseline completion time. Following the agent's run, the player is then challenged to complete the identical stage, with the AI's performance time presented as the target to beat. This implementation creates a direct competitive dynamic between human and artificial intelligence performance. While time constraints limited the development of more sophisticated integration features, this approach effectively demonstrates the practical application of the reinforcement learning model within an actual gameplay context. The competition framework provides an intuitive way for players to benchmark their skills against the trained agent, while also showcasing the agent's capabilities across the procedurally generated environments.

Chapter 6

Evaluation

This chapter presents an evaluation of the implemented system, focusing on functionality and performance.

6.1 Functional Evaluation

All implemented components of the system function as intended. The game mechanics, AI behaviour, and interface elements operate correctly with no significant bugs or issues observed during testing.

6.2 AI Performance Evaluation

To evaluate the performance of the trained models and demonstrate emergent intelligent behavior, I employed a systematic evaluation methodology. Models were trained for 2000, 4000, and 6000 iterations separately on the different map types. Additionally, a combined model was trained for 6000 iterations on each map type (18000 total iterations). Performance testing involved 100 independent attempts for each trained model on each map, with two key metrics recorded: success rate (percentage of attempts where the agent successfully reached the goal) and average completion time in seconds. If on an attempt the agent gets stuck and does not finish, then it's time was not counted. The evaluation aimed to determine how training iteration count correlates with performance improvements across different environmental challenges. We should expect to see the success rate increase and the completion time decrease as training increases.

6.2.1 Results

Table 6.1: Success Rate (%) and Average Completion Time (s) on Platforms Map

Training Iterations	Success Rate (%)	Avg. Completion Time (s)
2000	66	4.0
4000	87	3.7
6000	99	3.6
Combined (18000)	99	3.6

Table 6.2: Success Rate (%) and Average Completion Time (s) on Snake Map

Training Iterations	Success Rate (%)	Avg. Completion Time (s)
2000	60	3.9
4000	85	3.7
6000	98	3.6
Combined (18000)	97	3.6

Table 6.3: Success Rate (%) and Average Completion Time (s) on Platforms Map

Training Iterations	Success Rate (%)	Avg. Completion Time (s)
2000	45	4.2
4000	84	3.8
6000	89	3.7
Combined (18000)	92	3.7

6.3 Analysis of results

6.4 Conclusion

Bibliography

Carpenter, S. (2019), ‘Simulacrum: Building believable behaviors in video game ai’, `https://medium.com/@selcarpe/https-medium-com-simulacrum-28477a0e759e`. Accessed: 26/04/25.

DeepMind (2023), ‘Alphago’, `https://deepmind.google/research/breakthroughs/alphago/`. Accessed: 26/04/25.

Kanade, V. (2021), ‘What is finite state machine (fsm)?’, `https://www.spiceworks.com/tech/tech-general/articles/what-is-fsm/`. Accessed: 26/04/25.

Team, K. (2023), ‘Deep Q-learning for atari breakout’, `https://keras.io/examples/rl/deep_q_network_breakout/`. Accessed: 26/04/25.

Thompson, T. (2020), ‘Building the ai of f.e.a.r. with goal-oriented action planning’, `https://www.gamedeveloper.com/design/building-the-ai-of-f-e-a-r-with-goal-oriented-action-planning`. Accessed: 26/04/25.

Wikipedia (2025), ‘Artificial intelligence in video games’, `https://en.wikipedia.org/wiki/Artificial_intelligence_in_video_games`. Accessed: 26/04/25.