

Replacing traditional AI in video games using Machine Learning Agents

Danielis Golubovskis (170798389)

in supervision of Dr Gary Ushaw

MComp Computer Science (Games Engineering) I610

School of Computing, Newcastle University, Newcastle Upon Tyne, UK

Word Count: 12,161  
Date of Submission: xxxxx

Abstract

Self-learning AI, namely the deep learning subset of Machine Learning, promises to create more interesting, dynamic, and realistic game experiences by training machines to perform more complicated tasks in game environments. AI researchers at DeepMind, OpenAI, and more, are hard at work teaching software to play ever-more sophisticated games from classic Atari games to titles as advanced as Valve‘s Dota 2. However, despite the incredible results of such projects as OpenAI‘s ‚OpenAI Five‘, we are yet to see such sophisticated AI in commercial video games. This dissertation addresses the lack of sophisticated AI in video games and investigates the feasibility of training intelligent agents in a game environment by attempting to train agents capable of replacing traditional hard-coded AI using Unity‘s open-source machine learning toolkit.

Declaration

“I declare that this dissertation represents my own work except where otherwise stated.”

Acknowledgments

I would like to thank my supervisor, Dr Gary Ushaw, and the Games Engineering staff, Dr Graham Morgan and Dr Rich Davidson for the advice and support provided during the course of this project.

I would also like to thank Michele Condò at Ubisoft for taking the time to answer my questions and give me valuable insight into machine learning in the current video game industry.

Table of Contents

Abstract 1

Declaration 2

Acknowledgments 3

Table of Contents 4

1 Introduction 7

1.1 The Problem 7

1.2 Motivation 7

1.3 Approach 7

1.4 Aims and Objectives 8

1.5 Outcome 8

1.6 Outline 9

2 Background Research 10

2.1 Research Strategy 10

2.2 Artificial Intelligence 10

2.2.1 What is Artificial Intelligence? 10

2.3 Machine Learning 11

2.3.1 What is Machine Learning? 11

2.3.2 The Three Flavours of Machine Learning 11

2.4 AI in Video Games 11

2.4.1 Candy Crush – supervised learning 13

2.4.2 OpenAI Five – reinforcement learning 13

2.4.3 Clustering Game Behavior Data – unsupervised learning 13

2.5 First Part Summary 13

2.6 Unity 14

2.7 Unity Machine Learning Agents Toolkit 14

2.8 Possible Approaches 14

2.8.1 Reinforcement Learning and Imitation Learning 14

2.8.2 The Chosen Approach 15

2.9 Part Two Summary 15

3 Methodology 15

3.1 Making a New Learning Environment 15

3.1.1 Designing a Learning Environment 15

3.1.2 Defining the Game Rules 18

3.1.3 A Game of Hide and Seek 18

3.2 Defining the Agent Behaviours 18

3.2.1 Observations 18

3.2.2 Rewards 18

3.2.3 Training Configurations and Hyperparameters 19

3.2.4 Finite State Machine Seeker 19

3.2.5 Machine Learning Seeker Agent 22

3.2.6 Machine Learning Hider Agent 24

3.3 Defining Observable Parameters 25

3.4 Summary 26

4 Training 26

4.1 Observing Training 26

4.2.1 Version 0.1 – Training The Hider Alone 27

4.2.2 Version 0.2 – Added Walls 29

4.2.3 Version 0.3 – Added Raycast Observations 29

4.2.4 Version 0.3.1 – Raycast Observations Only With Obstacles 30

4.2.4 Version 0.3.2 – Comparing Raycast and Vector Observations 31

4.2.5 Version 0.3.2.1 – Increased Neural Network Layers and Nodes 32

4.2.6 Version 0.4 – Trying To Learn To Avoid Obstacles 32

4.2.8 Version 0.4.1 – Removed Raycasts and Increased Training Time 34

4.2.8 Version 0.5 - Training Both Agents Together 34

5 Evaluation 38

5.1 Trained AI and Traditional AI Movement 39

5.2.1 Hider Agent 39

5.2.2 FSM Agent 40

5.2.3 Trained Agent 42

5.2 Hider Agent performance against Traditional AI and Trained AI 44

5.3 Trained AI and Traditional AI Computational Resource Usage 45

6 Conclusion 46

6.1 Objectives Achievement 46

6.2 What Has Been Learned 47

6.2.1 Can Machine Learning Agents Replace Traditional AI ? 47

6.2.2 To What Extent Can Machine Learning Agents Replace Traditional AI? 48

6.2.3 The Pros and Cons of Training Machine Learning Agents 49

6.2.4 Why Are We Not Using Machine Learning Agents In Video Games? 49

6.3 What Has Been Learned 51

6.4 Personal Development 51

6.5 Future Work 51

References 53

1 Introduction

1.1 The Problem

Game developers often use algorithms like the Finite State Machine (FSM) to make their NPCs look intelligent. These hand-written NPCs often result in repetitive behaviour, loss of immersion, or abnormal behaviour the developers did not account for. Despite the incredible achievements of self-learning AI in self-driving cars, computer vision, language processing and even in video game environments, we don‘t see such sophisticated AI being used in commercial video games.

1.2 Motivation

While FSMs are hard-coded, machine learning is the ability of a system to learn and improve from experience, without being explicitly programmed to do so. Machine learning algorithms can offload a lot of the work that a game developer currently needs to perform. Currently, perfecting an AI can take days of hardcoding [1]. NPC character control and other things like the generation of unique environments could be automated if reliable algorithms are developed. The fact that machine learning has not taken over traditional AI in the video game industry made me very curious to find out what are the challenges that machine learning in video games faces.

1.3 Approach

Strongly inspired by OpenAI‘s paper „Emergent Tool Use from Multi-Agent Interaction“ [2] which observed agents discovering progressively more complex tool use while playing a simple hide-and-seek game, a similar approach of training agents to play a variation of the game of hide-and-seek was chosen. Other existing projects and research papers that also attempt to train sophisticated AI in video game environments often use large-scale systems, are worked on by large teams that specialize in AI and use custom, closed source software. My approach, however, differs from these projects in several ways:

* Only open-source tools are used
* Attempted on a much smaller scale of a single system (laptop computer)
* Includes scripting a FSM agent to directly compare to a trained agent

1.4 Aims and Objectives  
The project aims to investigate to what extent open-source machine learning toolkits such as Unity’s ‘ML-Agents’ can be used to train autonomous agents to replace traditional Finite State Machine agents in a game environment.

To meet my aim, I broke it down into smaller objectives and made sure they were SMART (Specific, Measurable, Achievable, Relevant, and Time-bound). The objectives are as follows:

1. Use online courses, published articles as well as Unity’s ML-Agents documentation to research and identify a technique for implementing the AI.
2. Establish a set of game rules and agent behaviours to achieve the desired ‘hide and seek’ behaviour.
3. Build at least 3 game level prototypes on which the training will take place.
4. Define a set of observable parameters for the hider and seeker agents that will be used to compare their performance to a fully scripted agent.
5. Build a FSM seeker, implement and train the hider and seeker agents using the rules and behaviours defined in objective 2.
6. Evaluate if, and to what extent the behaviours of FSM AI can be reproduced using Machine Learning agents using the observable parameters defined in objective 4.

Most of the objectives are done in preparation before the implementation (objectives 1 – 4). Firstly, adequate research into AI and machine learning must be initially completed to provide an understanding of both subjects and to identify a technique to implement the AI. The purpose of the second objective is to keep the environment controlled. For example, the game of hide and seek has many variations such as standard hide and seek, team up, sardines, hotter/colder, object hunt etc. A precisely defined rule set and behaviours would eliminate any ambiguity and allow for easy replication of the game in other projects and environments. The purpose of objective 3 is to allow for broader testing on how environments may influence training while objective 4 is needed to define the parameters that would allow to directly compare agents in these environments. The fifth objective is the broadest and covers the entire implementation step from scripting to training agents. This objective depends on objective 2 and 3, however only one environment is needed to begin training, thus objective 3 can be finished during the implementation phase. As the description suggests, the final objective evaluates whether the behaviour of traditional AI NPCs can be accurately reproduced using machine learning and uses the parameters from objective 4 to do so.

1.5 Outcome

In the end, both the hider and seeker agents were successfully trained to play a simple game of hide and seek. While the performance of the trained agent was not up to par with the scripted agent, it could still be further improved with more training time and alternative training methods. Despite the shortcomings in performance, the trained agents took much less programming time by reusing existing code and fitting the entire code in a single script file. Unity is continuing the development of its machine learning toolkit which promises to improve training and provide access to more computational power via cloud training soon. Further information on agent performance, training, issues of machine learning and other findings will be expanded on in the appropriate chapters.

1.6 Outline

**Introduction**

The introduction gives the reader some context and motivation for the project to build a basic understanding of what to expect before continuing.

**Background Research**

The background research chapter introduces the fields of AI and machine learning as well as the techniques commonly used to implement AI using machine learning in video games by providing concrete examples.

**Methodology**

This chapter discusses the programming and design decisions made before the implementation phase such as environmental design, defining the rules and implementing them in code. This chapter covers objectives 2, 3 and 4.

**Training**

*This chapter covers the training phase by splitting it into versions. Details and changes of every version are discussed and backed with graphed results.*

**Evaluation**

The trained models obtained from the previous chapter are reimplemented into the environment and tests are conducted on the observations described previously. These observations are then used to identify the outcome of the final objective.

**Conclusion**

The final chapter will reiterate what has been achieved and expand on the evaluation by summarizing the final thoughts on what has been learnt, as well as reflect as to what could have been done differently.

2 Background Research

2.1 Research Strategy

The background research is split into two parts. The first part covers the material for understanding the subjects of AI and machine learning as well as the different approaches to applying machine learning in video games. The second part of the background research includes material on Unity and its ‚ML-Agents‘ toolkit which will help identify the approach most suitable for the project.

2.2 Artificial Intelligence

Artificial intelligence is a wide-ranging branch of computer science with many closely related fields and topics. As such, it can be challenging to take in the great deal of information. The first piece of material that helped me the most at making a start at tackling this subject was [3]. What I found most useful was the introductory chapter which talks about the definition of AI, its philosophy and related fields.

2.2.1 What is Artificial Intelligence?

As it turns out, it‘s not an easy question to answer. The popularity of AI in media is in part due to the fact that people have started to use the term to refer to things that used to be called by other names. Everything from statistics to business analytics and even manually coded if-then rules is being called AI.

First and foremost, there is no officially agreed definition of AI. The field is being constantly redefined when some topics are classified as non-AI, and new topics emerge. This is why some may jokingly refer to AI as „cool things that computers can‘t do“. The irony is that under this definition, AI can never make any progress: as soon as we find a way to solve a task with a computer, it stops being an AI problem. Fifty years ago, for instance, automatic methods for search and planning were considered to belong to the domain of AI. Nowadays such methods are taught to every computer science student.

Another reason for the confusion is that AI challenges our idea of „intelligence“. For example, playing chess and solving mathematical exercises can seem to be difficult, requiring years of practice to master, which is why some of the initial AI research concentrated on these kinds of tasks, as it may have seemed at the time that they encapsulate the essence of intelligence. As it turned out that playing chess is very well suited to computers, which can follow a set of fairly simple rules and compute alternative move sequences at a rate of billions of computations a second. This lead to computers beating the reigning human world champion Gary Kasparov back in 1997 [4]. On the other hand, it can be hard to appreciate how difficult some tasks are; like picking up an object. Grasping objects involves many observations, precise movements of muscles and variables like the weight of an object. While easy for us humans, it is extremely hard for robots, and it is an area of active study. An example of this includes Google‘s robotic grasping project [5].

One attempt at a definition that I agree with the most would be to list properties that are characteristic of AI, in this case, autonomy and adaptivity.

*Autonomy – The ability to perform tasks in complex environments without constant guidance by a user.  
Adaptivity – The ability to improve performance by learning from experience.*  
Therefore, we can define AI as „Computer systems that are capable of performing tasks in complex environments without constant guidance“.

While AI must be autonomous, it does not necessarily have to be adaptive to be classified as AI. The adaptivity of AI comes from the field of machine learning.

2.3 Machine Learning

To build a high-level understanding of machine learning the following materials were used [6] and [7]. These materials describe the main steps of machine learning including data gathering, data preparation, choosing an algorithm, training, evaluation, parameter tuning and prediction.

2.3.1 What is Machine Learning?

Machine learning is a branch of artificial intelligence that is responsible for the vast majority of the artificial intelligence advancements heard in the media. Machine learning algorithms use statistics and are trained to find patterns and features in massive amounts of data in order to make decisions and predictions based on that data and improve their accuracy without being explicitly programmed to do so. The data that machine learning uses encompasses many things – numbers, words, images, etc. Machine learning powers many of the services we use today – from content recommendation on Netflix, Youtube, and Spotify to image processing on social media or self-driving cars.

2.3.2 The Three Flavours of Machine Learning

The following material [8] introduces the three main machine learning paradigms - supervised learning, unsupervised learning and reinforcement learning. In supervised learning, the most common of the three, the data is labelled to tell the machine exactly what patterns to look for. In unsupervised learning, the data is unlabeled and the machine extracts the patterns and classifies them by itself. Reinforcement learning, however, learns by trial and error to achieve a clear objective. The idea is that an agent starts by performing random actions and learns the correct behaviour by being either rewarded or penalized depending on whether its actions help or hinder it from achieving its objective.

2.4 AI in Video Games

In video games, artificial intelligence is used to generate responsive, adaptive or intelligent behaviours primarily in non-player characters (NPCs). Developers often use algorithms like the Finite State Machine (FSM) algorithm to make these NCPs look intelligent [9]. In a FSM, the developers generalize all possible situations that an AI could encounter and then program a specific reaction for each situation.

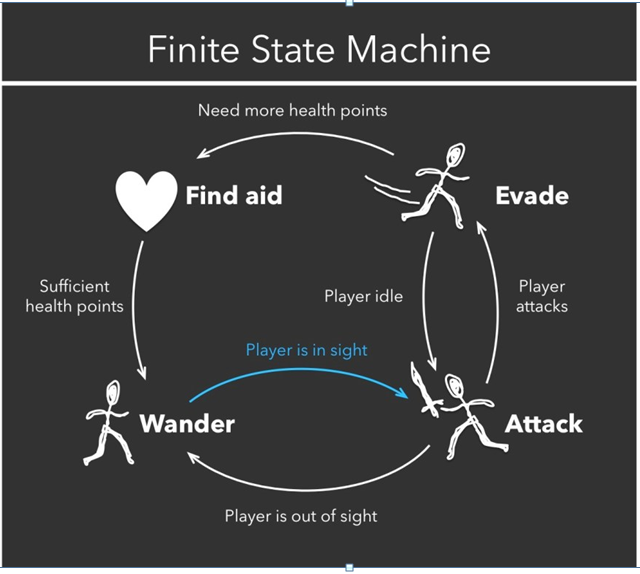


Figure 1. Finite State Machine model example (Taken from source [9])

The obvious drawback of the FSM design is that all NPCs’ behaviours are pre-programmed. These handwritten decision trees often result in “artificial stupidity” such as repetitive behavior, loss of immersion, or abnormal behavior in situations the developers did not plan for [10]. A more advanced method used to enhance the personalized gaming experience was the Monte Carlo Search Tree (MCST) algorithm. This is the AI strategy used in Deep Blue, the first computer program to defeat a human chess champion in 1997. At each point in the game, the algorithm considers all the possible moves it could take, then consider all the possible human player moves in response, then consider all its possible response moves, and so on. This way, an AI with MCST can calculate thousands of possible moves and choose the ones with the best playback. However, some games are far too complicated to consider all possible moves. Therefore, in such games as Sid Meier's Civilization, the algorithm instead selects a set of random possible moves to choose from. Therefore, the outcomes become much more random and unexpected.

Despite the efforts in making NPCs look intelligent with these algorithms, these characters lacked the ability to learn and adapt to the players’ input. This is why machine learning has become a hot research topic in video games.

2.4.1 Candy Crush – supervised learning  
King Digital Entertainment, the creators of Candy Crush Saga have developed an approach to learn and deploy human-like playtesting in computer games based on deep learning from player data [11]. The approach uses supervised learning on a Convolutional Neural Network (CNNs) to learn and predict the most „human“ action given a position in the game. Furthermore, the network can predict the difficulty levels of new content. The human data was gathered from Candy Crush Saga and Candy Crush Soda Saga, however, the method is general and suited for many games, in particular where content creation is sequential. Most interestingly, though, they show that this approach gives more accurate predictions when compared to the Monte Carlo Search Tree at a fraction of the computation time.

2.4.2 OpenAI Five – reinforcement learning

OpenAI, an artificial intelligence research laboratory, has succeeded to train five neural networks (OpenAI Five) to play Dota 2 and defeat professional players in a 5v5 match [12]. OpenAI Five plays 180 years worth of games against itself every day, learning via self-play. OpenAI used a scaled-up version of the Proximal Policy Optimization reinforcement algorithm to train the neural networks, running on 256 GPUs and 128,000 CPUs. Using separate LSTM (Long Short Term Memory) cells for each game hero and no human data, the algorithm learns recognizable strategies. More so, the project has proven that reinforcement learning can yield long-term planning on a large but achievable scale.

2.4.3 Clustering Game Behavior Data – unsupervised learning

Unsupervised methods such as clustering can be used to deal with massive, high-dimensional game data, such as player behaviour [13]. This paper discusses the methods of deriving insights from large amounts of high-dimensional data that people generate while playing games. Game analytics use these clustering algorithms to identify groups of players of similar behaviours or detect game features that constitute such behaviours. This data is used to help improve a game‘s design, ensure optimal user experience, identify valuable players or those at risk of leaving a game, personalize or adapt gameplay, as well as compare and benchmark games and improve AI.

2.5 First Part Summary

The material shows that artificial intelligence must be autonomous – it must be able to handle tasks without constant guidance. Such AI is most prominent in video games and is implemented with Finite State Machine algorithms. However, artificial intelligence can also be adaptive – learn from its experience to achieve its goals. This kind of AI is powered by machine learning which can be clustered into three main categories – supervised learning, unsupervised learning, and reinforcement learning. The examples of these learning approaches being used in video games reveal some very important details:

* Machine learning algorithms can outperform traditional AI approaches
* The same machine learning approach can be generalized to work with many games
* Reinforcement learning has been proven to be able to yield great results on a large but achievable scale.
* The most similar example project which trains agents uses reinforcement learning, which suggests that this is the desired approach for this project.

2.6 Unity

The Unity game engine is one of the world‘s leading platforms for creating interactive, real-time content which has been used to create over half of the world‘s games [14]. One of the reasons why Unity became so successful is that its pricing structure is developer-friendly. Their packages are designed for everyone – from complete beginners, hobbyists to large development teams. More importantly, Unity provides a clean and user-friendly interface and enables creators by giving them the right tools. One of these tools is the ‚Unity Machine Learning Agents‘ toolkit.

2.7 Unity Machine Learning Agents Toolkit

Unity‘s ML-Agents toolkit is an open-source project that enables games to serve as environments for training intelligent agents [15]. The toolkit is based on the PyTorch machine learning framework and provides implementations of state-of-the-art algorithms to enable game developers of all skill levels. Researchers can also use the provided Python API to train Agents using reinforcement learning, imitation learning, neuroevolution, or any other methods. These trained agents can be used for multiple purposes, including controlling NPC behaviour in a variety of settings such as multi-agent and adversarial, automated testing of game builds and evaluating different game design decisions pre-release.

2.8 Possible Approaches

The documentation separates its training methods into two separate categories: Environment-agnostic and Environment-specific. Environment-agnostic training methods are such that can be applied regardless of the specifics of the learning environment. These methods are reinforcement learning and imitation learning. Environment-specific methods can only aid in training in specific types of environments. Such methods include Self-Play, MA-POCA, Curriculum Learning, Environment Parameter Randomization.

At this stage of the project, it is difficult to predict whether any of the environment-specific methods could be of use. Thus, the choices were narrowed down to reinforcement learning and imitation learning.

2.8.1 Reinforcement Learning and Imitation Learning

In reinforcement learning, the end goal for the Agent is to discover a behaviour (a Policy) that maximizes a reward. Typically, a reward is defined by the environment and corresponds to reaching some goal.

Rather than attempting to train via trial-and-error, in imitation learning the agent is given a set of demonstrations of the desired behaviour. For example, these demonstrations can be the recorded moves from a game controller. The training methods then train the agent to mimic the actions shown in the demonstration as closely as possible.

Imitation learning can either be used alone or in conjunction with reinforcement learning. If used alone it can provide a mechanism for learning a specific type of behaviour (i.e. a specific style of solving the task). If used in conjunction with reinforcement learning it can dramatically reduce the time the agent takes to solve the environment. This can be especially pronounced in sparse-reward environments.

2.8.2 The Chosen Approach

Although both methods are promising, in the end, I chose to go with the simpler of the two – reinforcement learning. It is easier to set up and does not require demonstrations. Moreover, the OpenAI Five project has proven the effectiveness of the Proximal Policy Optimization (PPO) algorithm - a type of reinforcement learning algorithm – which is implemented in ML-Agents. With this, the first objective of the project has been reached.

2.9 Part Two Summary

One of the most important things about Unity‘s ML-Agents is that it is completely open-source, free to use, and is extensively documented. The toolkit provides the users with the implementations of different algorithms for many of the learning approaches such as reinforcement learning and imitation learning.

3 Methodology

Using the ML-Agents Toolkit in a Unity project involves the following basic steps:

1. Creating learning environments
2. Training & Inference
3. Embed the model into the game

This chapter covers the first step and discusses what design and programming decisions have been made throughout the project and why. Steps 2 and 3 will be discussed in the training and evaluation chapters.

3.1 Making a New Learning Environment

Creating a learning environment involves two steps: making the environment itself and designing the agents.

3.1.1 Designing a Learning Environment

The learning environment is where the agents live in. An environment can range from a simple physical simulation containing a few objects to an entire game ecosystem. The agents in the environment are represented as GameObjects, typically, an object in the scene that represents the Agent in the simulation (a character model, for example).

As mentioned before, I took inspiration from OpenAIs paper in which they observed agents playing a simple game of hide-and-seek and create my version of the hide-and-seek environment and agents. To reach objective 3 I built three different environments of different sizes and obstacle layouts on which the training will take place.

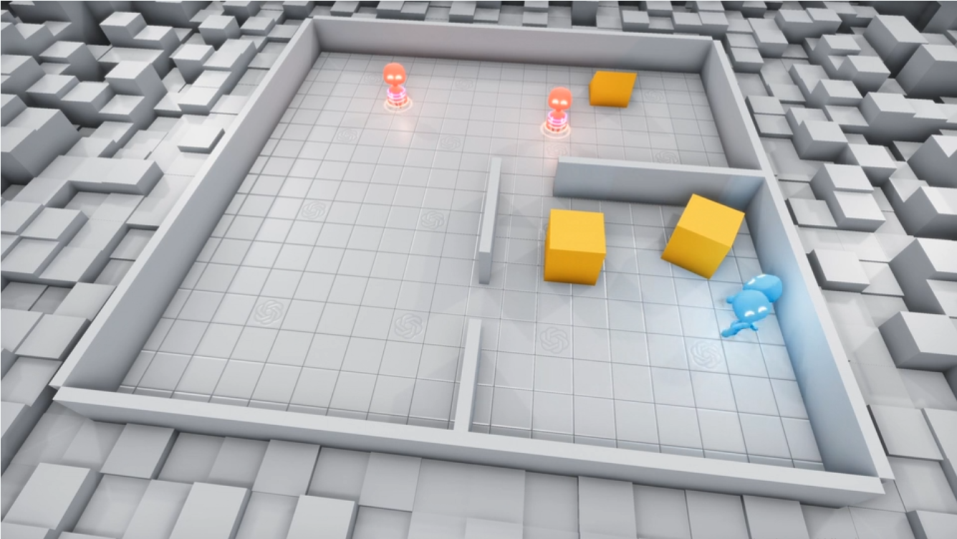


Figure 2. Learning environment created by OpenAI

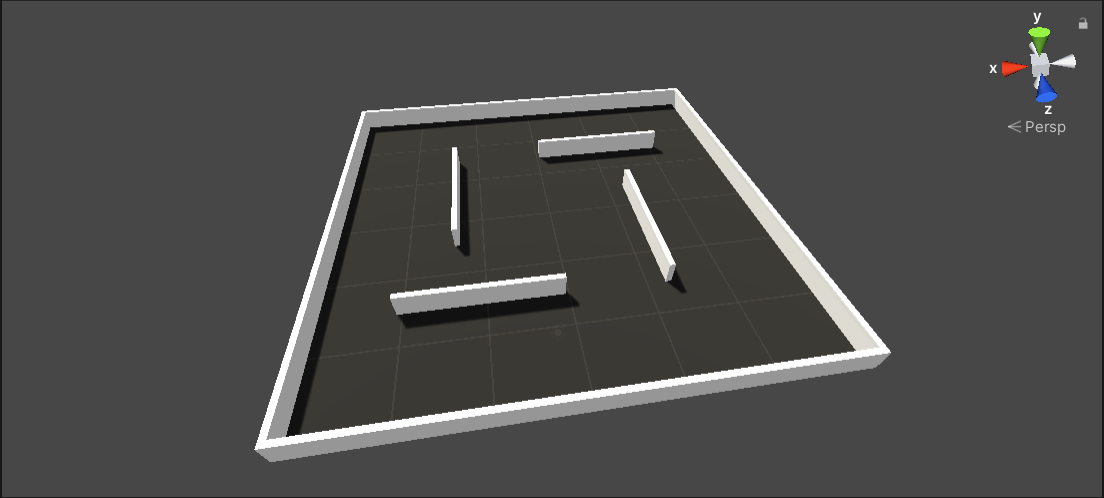


Figure 3. First learning environment created for this project

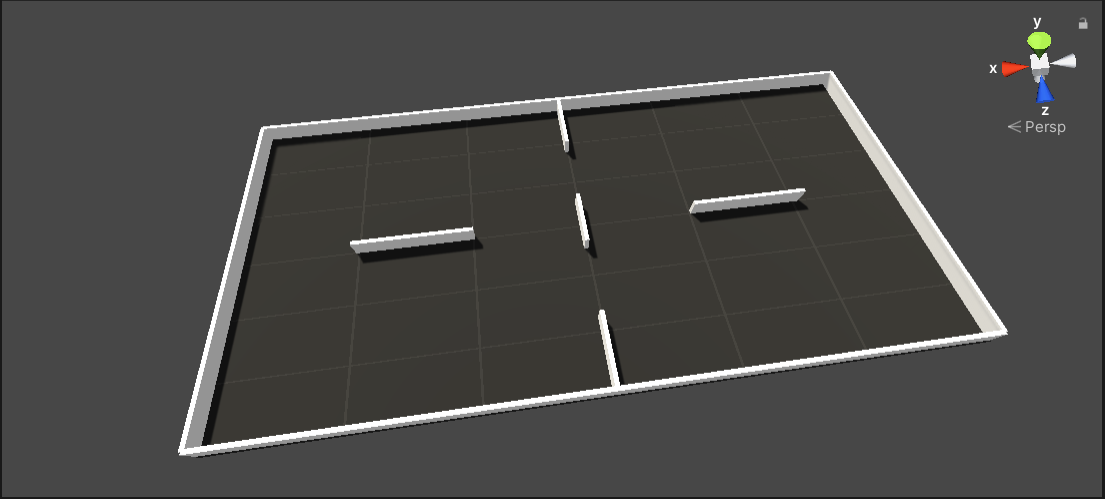


Figure 4. Second learning environment created for this project

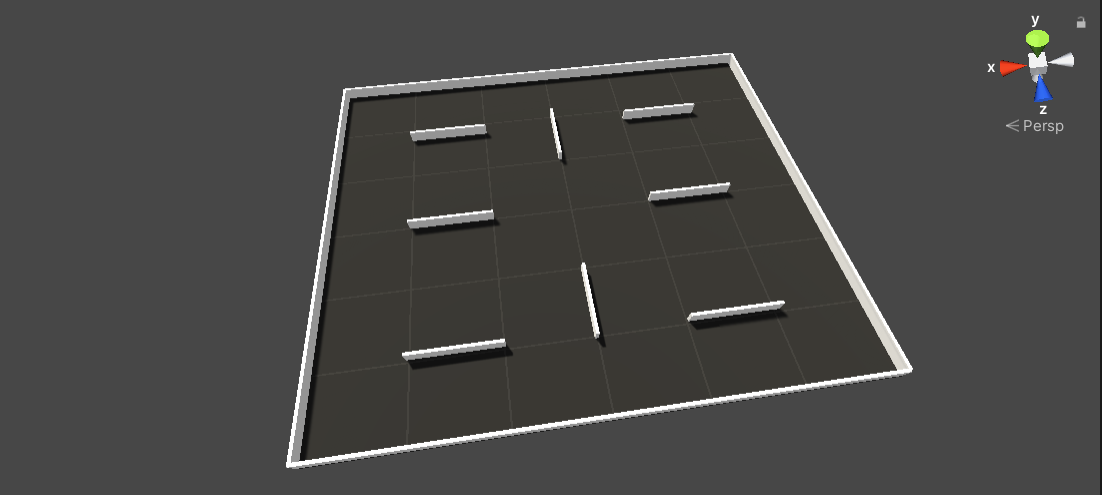


Figure 5. Third learning environment created for this project

To populate the environment with a hider and a seeker an agent model from Unity’s ML-Agents example projects was used.



Figure 6. Seeker agent game model



Figure 7. Hider agent game model

3.1.2 Defining the Game Rules

To reach objective 2 the next step is to define the game rules and agent behaviours.

3.1.3 A Game of Hide and Seek

The agents will be playing a variation of the game of hide-and-seek. The players are split into two teams – the hiders and the seekers. A target location is placed in a random position in the environment. The goal of the hiders is to reach the target without getting caught (touched) by the seeker. The goal of the seekers is to wander around the environment and catch the hiders when they are spotted. The seekers do not know the location of the target, thus they cannot guard one spot. Once the target is reached or the hider is caught the game is reset.

3.2 Defining the Agent Behaviours

3.2.1 Observations

An agent is an entity that can observe its environment, decide on the best course of action using those observations, and execute those actions within its environment. For an agent to learn, the observations should include all the information an agent needs to accomplish its task. Without sufficient and relevant information, an agent may learn poorly or may not learn at all. An Agent passes its observations to its Policy – a class that abstracts the decision making logic from the agent itself. The Policy then makes a decision and passes the chosen action back to the agent. Unity provides several ways of generating observations:

**Vector Observations**Vector observations are best used for aspects of the environment that are numerical and non-visual. They can hold Integers, booleans, floats, as well as some common Unity data types such as Vector3 and Vector2. This type of observations is perfect for storing object positions (x,y and z coordinates as a Vector3 data type).

**Visual Observations**Visual observations are provided to the agent via a camera sensor that collects image information and transforms it into a 3D Tensor which is fed into the Convolutional Neural Network (CNN) of the agent policy. Visual observations are useful when the observation is difficult to describe numerically.

**Raycast Observations**Several rays (or spheres) are cast into the physics world, and the objects that are hit determine the observation vector that is produced. These ray sensors are highly customizable and are best used when there is relevant spatial information for the agent that doesn't require a fully rendered image to convey.

3.2.2 Rewards

In reinforcement learning, the end goal for the Agent is to discover a behaviour (a Policy) that maximizes a reward. An agent needs to be provided with one or more reward signals to use during training. Typically, a reward is defined by the environment and corresponds to reaching some goal. These are referred to as extrinsic rewards. Rewards, however, can be defined outside of the environment as well, to encourage the agent to behave in certain ways or to aid the learning of the true extrinsic reward. These rewards are intrinsic reward signals. The total reward that the agent will learn to maximize can be a mix of extrinsic and intrinsic reward signals.

3.2.3 Training Configurations and Hyperparameters

ML-Agents training utility uses a YAML configuration file that contains all the configurations and hyperparameters to be used during training. The set of configurations and hyperparameters to include in the file depends on the agents and the specific training method. Hyperparameter values can have a big impact on training performance. The contents of the file define parameters such as the neural network configurations, trainer configurations, reward signals, and so on.

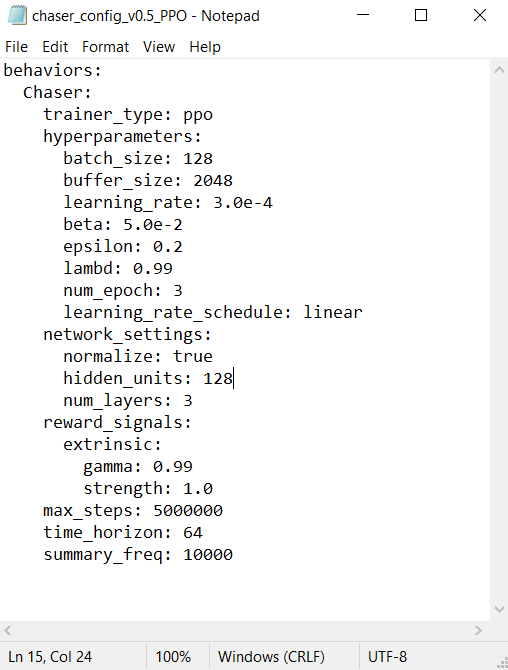


Figure 8. Example hyperparameters file

3.2.4 Finite State Machine Seeker

To attempt to replace a traditional AI, one must be first created using the FSM model. Only the seeker agent has been coded using this approach as it would take less time and would still allow for a comparison between a FSM agent and a trained agent. The agent‘s behaviour consists of three states: Idle, Patrol, Pursue.

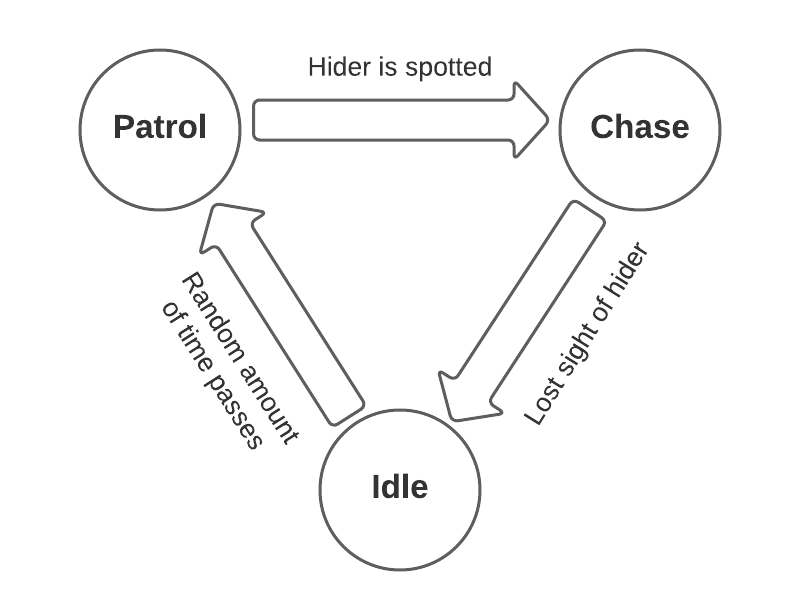


Figure 9. Seeker agent FSM model

To allow the agent to navigate the environment Unity‘s in-built NavMeshAgent component was used. Once attached to a mobile character it allows it to navigate the scene using the NavMesh. To handle the patrolling state, an array of waypoints were scattered around the environment and passed to the agent as a list of destinations.

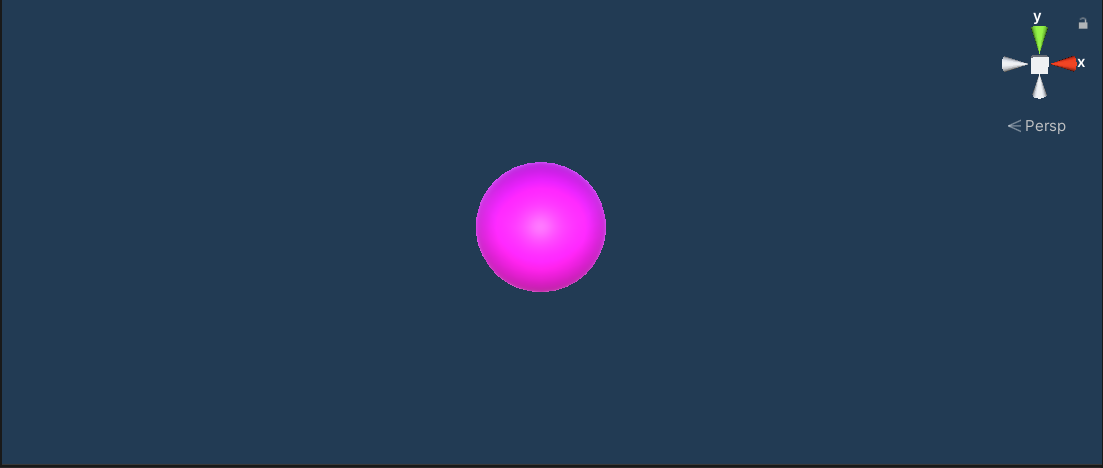


Figure 10. Waypoint game model

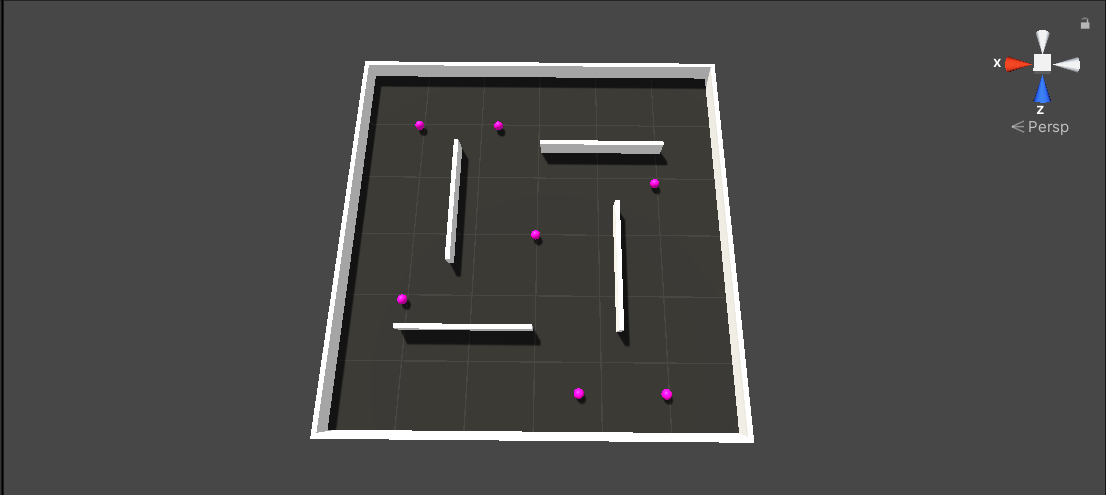


Figure 11. Learning environment with waypoints

To be able to spot the hider the agent is given a cone of sight which is achieved by keeping track of the distance between the hider and seeker and the angle. A viewDistance and viewAngle variables set the minimum distance and angle required for the hider to be within the sight code. The hider, however, can be obstructed by walls, thus once the viewDistance and viewAngle tests pass, several rays are cast from the left edge of the view angle to the right one. If at least one ray hits the hider, its position is fed into the seeker’s NavMeshAgent component as a destination and the chase begins until the sight is lost. Once the sight is lost, the agent waits a random amount of time before resuming to patrolling the waypoints.

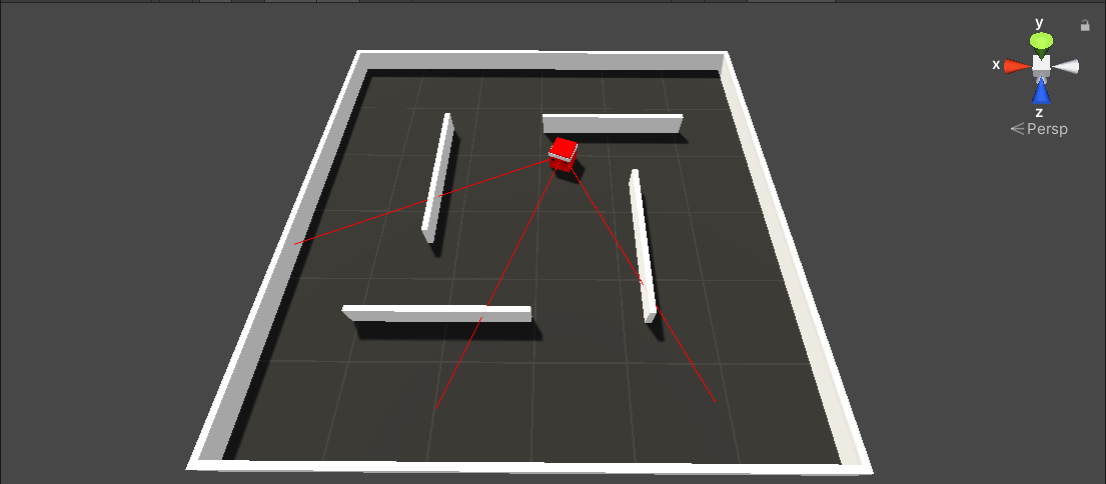


Figure 12. Seeker agent with cone of sight

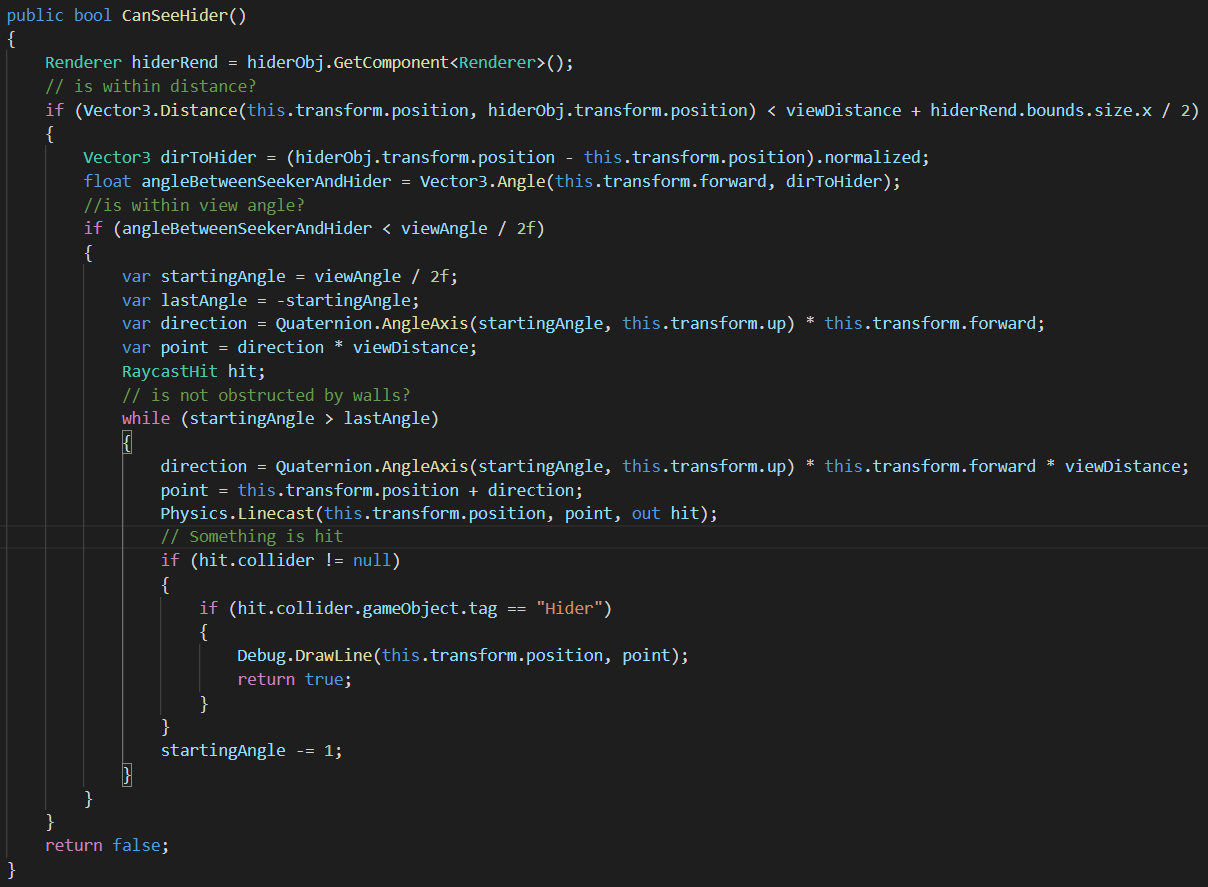


Figure 13. Code snippet of the hider detection function

3.2.5 Machine Learning Seeker Agent

As mentioned previously, machine learning is the ability of a system to learn and improve from experience. The machine learning agent has to achieve the same behaviour of patrolling and chasing without being explicitly programmed to do so. This behaviour is achieved through a set of rewards and observations.

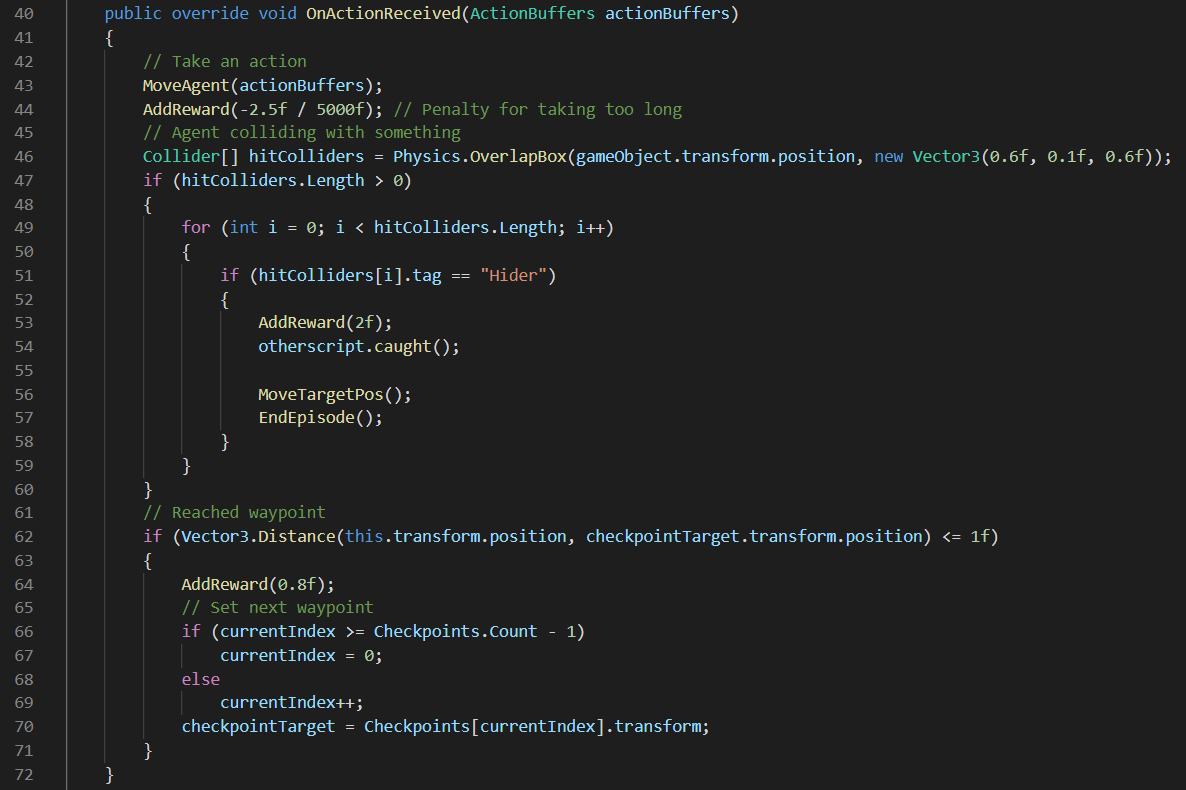


Figure 14. Code snippet showing the seeker agent reward system

The agent holds a method which is called every time it has to take an action. Line 43 simply calls the method that handles movement and is passed an actionBuffer – a number generated by the neural network in a pre-determined range from 0 to 4 which corresponds to moving in any direction (cases 1-4), or not taking any action (case 0). The agent receives a small reward penalty for each action taken (Line 44). This punishes the agent for aimlessly wandering too much without reaching the next waypoint. Lines 46-60 keep track of collisions between the agent and other objects. If the other object is a hider, the agent is given a reward and the game episode is ended, starting a new game. Lines 61-72 manage the waypoints and give a small reward for reaching the next waypoint, therefore teaching the agent to patrol the level.

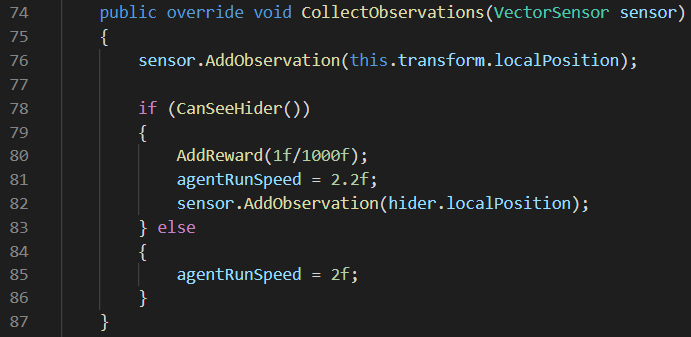


Figure 15. Code snippet showing seeker agent observations

For the observations, the agent must always be aware of its position in the environment (Line 76). This agent uses the same code to detect whether the hider is within the cone of sight and once it returns true the agent keeps receiving a small reward (Line 80) for keeping the hider within its sight. The hider‘s location is fed into the agent‘s sensor and its speed is increased to simulate running (Lines 81-82).

This simple setup of observations and rewards is enough to replicate the desired FSM behaviour and requires very little coding because a big portion of the code was copied over.

3.2.6 Machine Learning Hider Agent

There is no FSM model to base the hider agent on, however, the desired behaviour can be derived from the game rules: collecting target and avoiding seeker. Once again, the behaviour is achieved by a set of reward and observation parameters.

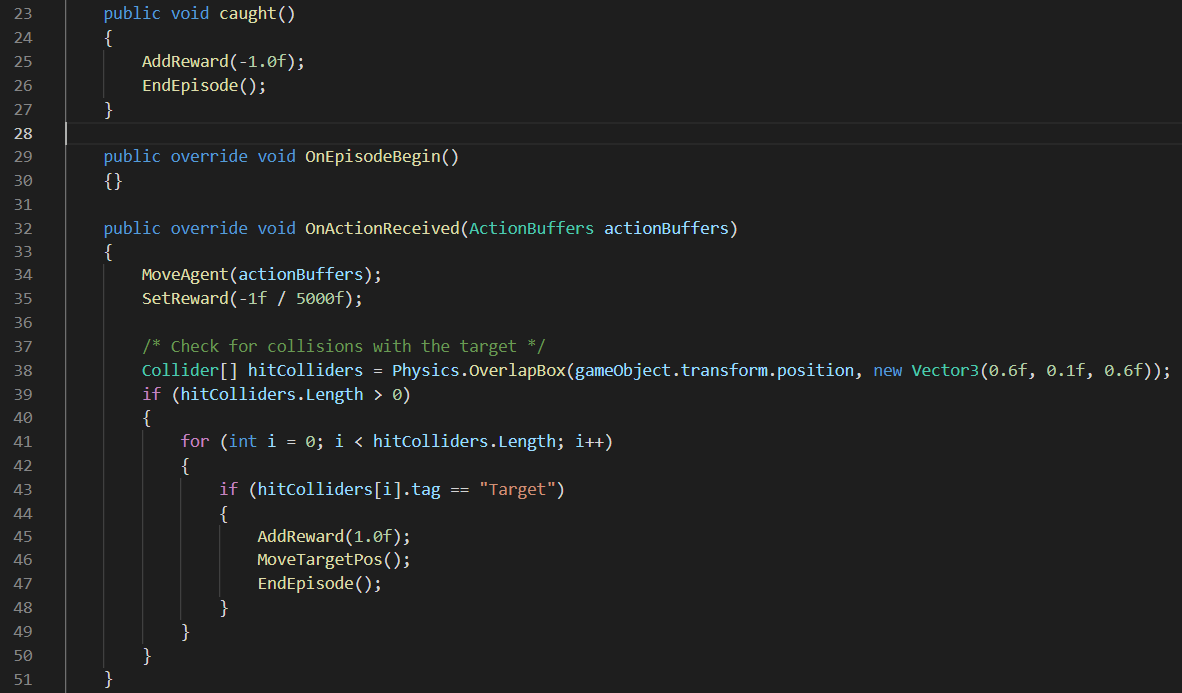


Figure 16. Code snippet showing hider agent reward system

The agent learns to avoid the seeker by being punished for getting caught (Line 23-27). The OnActionReceived method is near identical to the seeker – first, the agent makes a move (Line 34). A small penalty is applied to wandering too much (Line 35). Collisions are checked to detect when the target is reached, in which case a reward is given, the target is moved to a random position and the game is reset (Lines 38-51).

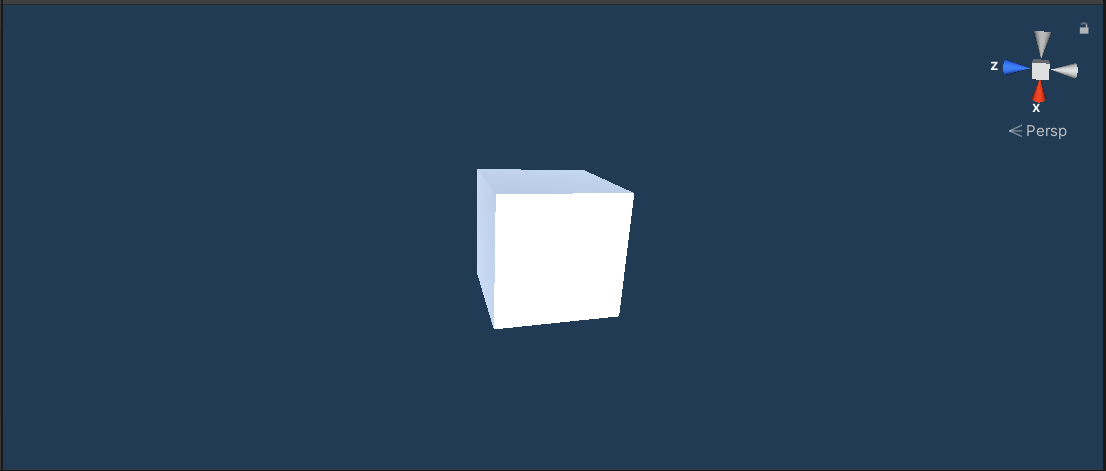


Figure 17. Hider's target game model

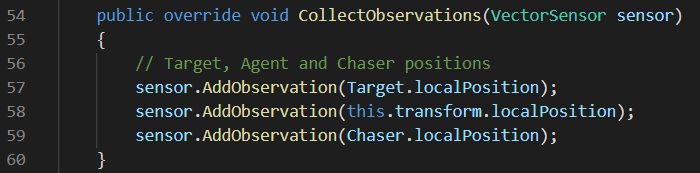


Figure 18. Code snippet showing hider agent observations

The agent only requires three sets of observations – itself, the seeker agent, and the target. Unlike the seeker, this agent is always aware of the position of the enemy player.

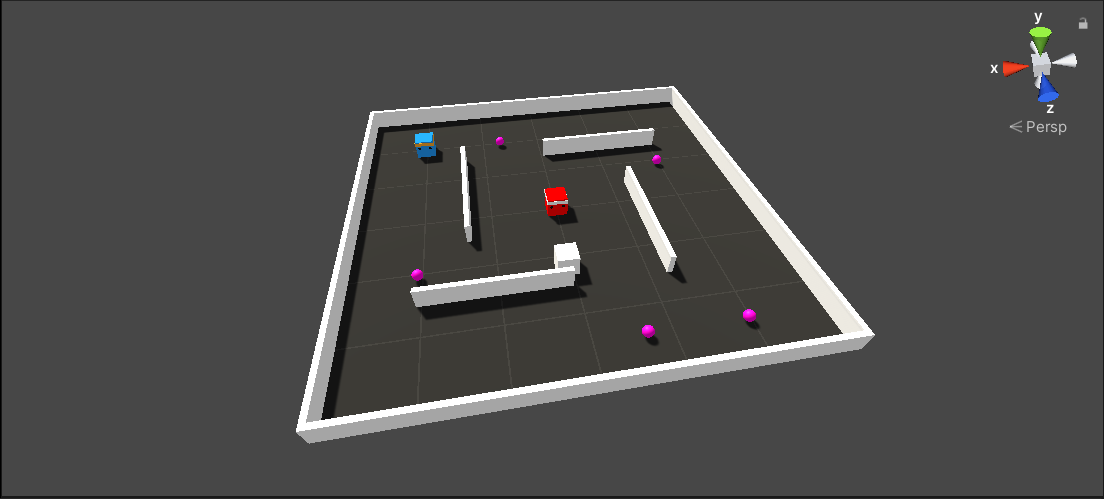


Figure 19. Learning environment with both agents, waypoints and target

3.3 Defining Observable Parameters

To be able to compare the FSM agent and the machine learning agent, and to complete objective 4 a set of observable parameters needs to be defined which will be used for comparison.

The following are the parameters to be observed in this project:

* Hider agent success rate in reaching target uncaught vs FSM agent (% from 100 game rounds)
* Hider agent success rate in reaching target uncaught vs machine learning agent (% from 100 game rounds)
* Hider agent average time to reach target vs FSM agent (seconds across 100 game rounds)
* Hider agent average time to reach target vs machine learning agent (seconds across 100 game rounds)
* Computational resource usage of traditional agent and trained agent
* FSM agent proficiency in performing its tasks (Observed visually)
* Machine learning agent proficiency in performing its tasks (Observed visually)

3.4 Summary

In this chapter, we have defined the basic hide-and-seek game rules, as well as defined the desired behaviours of the hider and seeker agents. We have also defined a set of 6 observable parameters which will be used to compare the FSM agent and the machine learning agent. With this, objectives 3, 4 and 5 have been achieved.

4 Training

This chapter covers the training phase by splitting it into versions and mentions all details and changes between versions. Any issues or failed experimentations are also mentioned.

4.1 Observing Training

The ML-Agents toolkit saves statistics during the learning sessions that can be viewed with a Tensorflow utility named Tensorboard. The training program saves the statistics for the environment, the policy, policy loss and self-play.



Figure 20. Example tensorboard statistics

In this project we will be observing the two main statistics:

* Environment/Cumulative Reward - The mean cumulative episode reward from all agents that share the same policy.
* Environment/Episode Length - The mean length of each episode (single game) in the environment for all agents.

The desired training result will show a steady increase in cumulative reward overtime followed by a plateau once the highest reward is reached and a decrease in episode length.

4.2.1 Version 0.1 – Training The Hider Alone

The first version included only the hider and the target in the first environment with no obstacles. The purpose here was to test the toolkit for the very first time to see if the desired behaviour and statistics can be achieved. For this and any subsequent versions, 16 copies of the learning environment were placed into the world. Each copy can learn independently and contribute to the overall training of a single model, speeding up the process 16-fold.

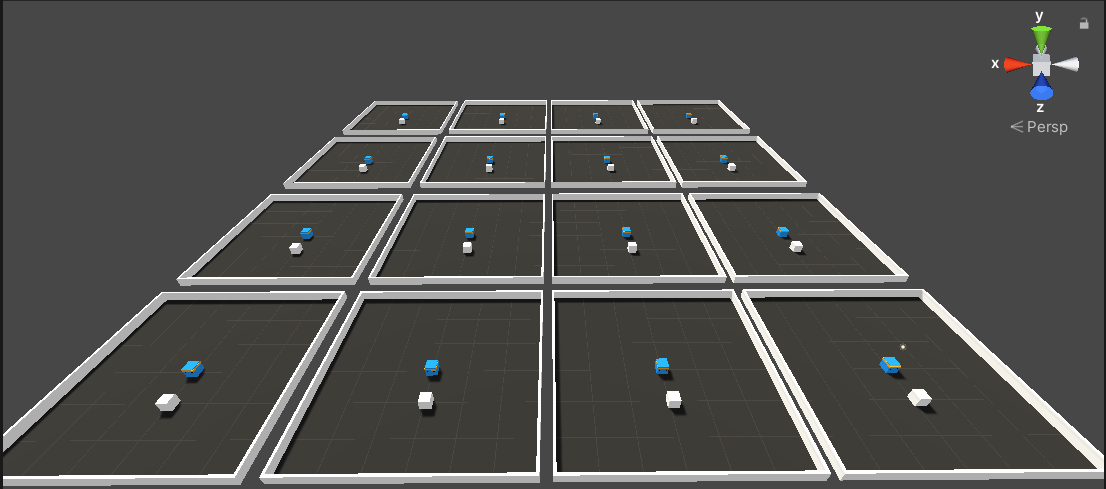


Figure 21. Training environments 1

The goal of the hider was to learn to grab the target, which would move to a random position each time it was picked up.

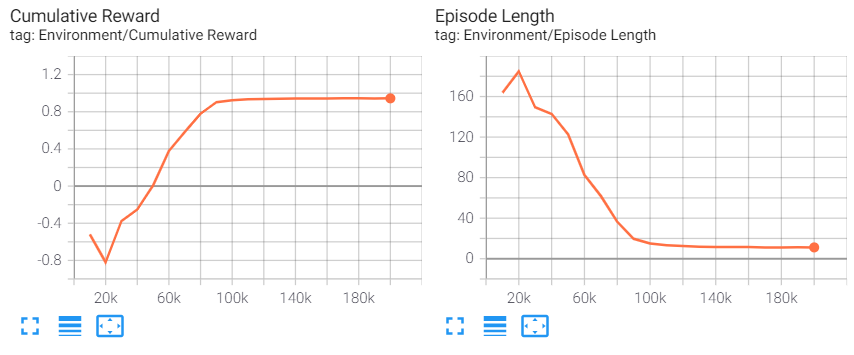


Figure 22. Training results 1

Only given the position of itself and the target, as well as a small reward for picking up the target, the agent learned to pick up the target in just 100,000 steps which only took 1m 21s of training time. At that point, the reward has plateaued and the episode length reached its lowest point, meaning the agent was very efficient in moving from target to target without any wandering.

4.2.2 Version 0.2 – Added Walls

In this version, obstacles were introduced in the environment to see how the agent would cope.

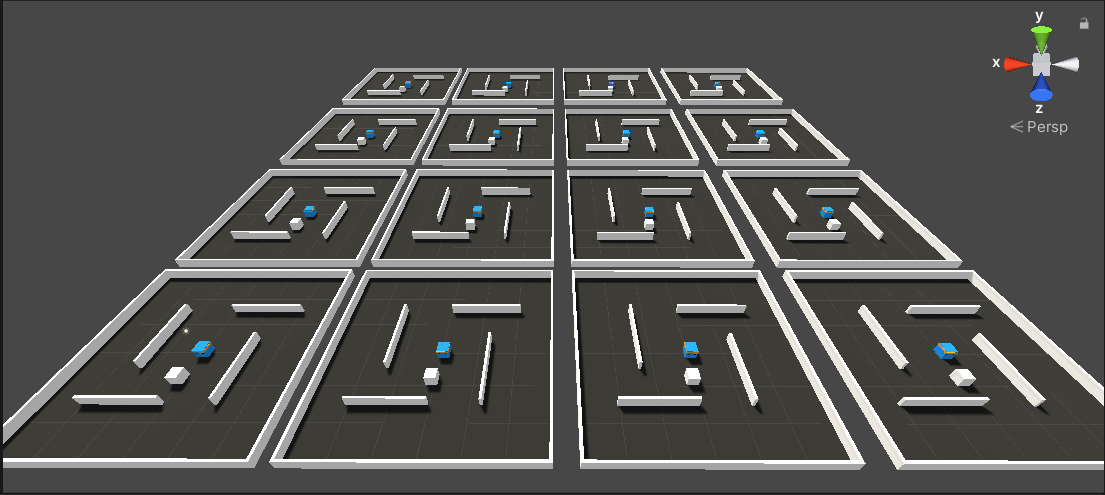


Figure 23. Training environments 2

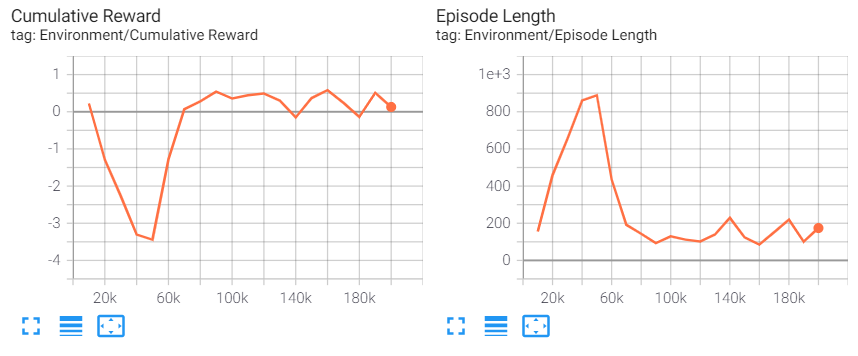


Figure 24. Training results 2

This time, the agent struggled to do its task. When the target appeared behind a wall, the agent would end up hitting and grinding against the wall as it had no clue how to get around it. This is seen by the large spikes in the cumulative reward and episode length, where the agent would take too long being stuck behind a wall and accumulating negative rewards. The training episode ran for 200,000 steps which took 3m 8s.

4.2.3 Version 0.3 – Added Raycast Observations

From the previous results, it was clear that the agent needed a way to sense its environment. As mentioned previously, the toolkit provides several ways of generating observations. I decided to add raycast observations to the agent.

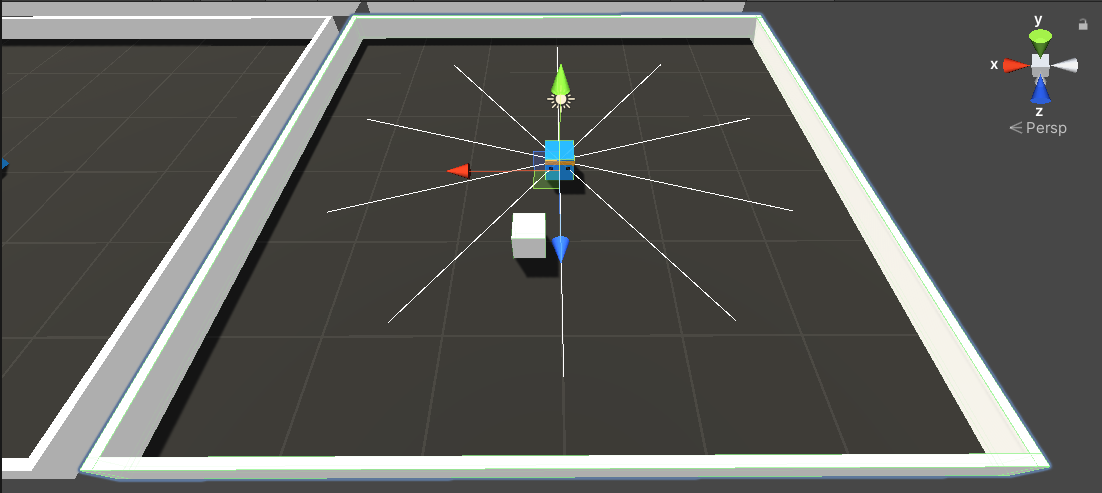


Figure 25. Hider agent with raycasts sensors

The rays would be shot in multiple directions around the agent. If an object was hit by the ray, its information such as the type of object and its position would be passed to the agent’s sensor. With this, the expectation was to allow the agent to be aware of when its path is obstructed by an obstacle.

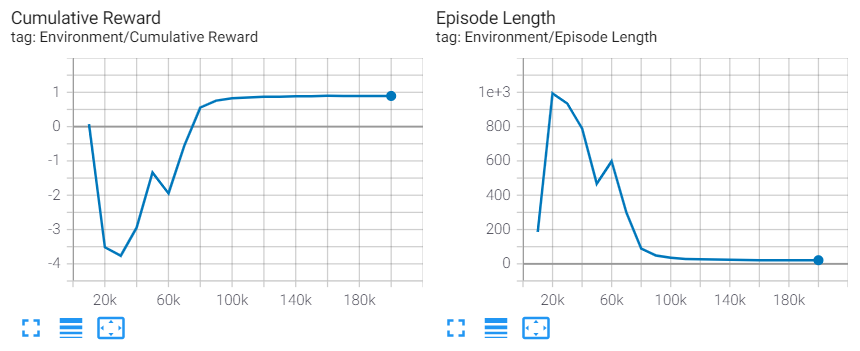


Figure 26. Training results 3

Repeating the training with only the raycasts in the first environment without any obstacles again showed very similar results as with vector observations for the target alone. It once again took around 100,000 steps to learn to do its task which took about 1m 26s of training time.

4.2.4 Version 0.3.1 – Raycast Observations Only With Obstacles

With the promising results from the last test I attempted to train the agent on the same environment with obstacles with raycast observations only.

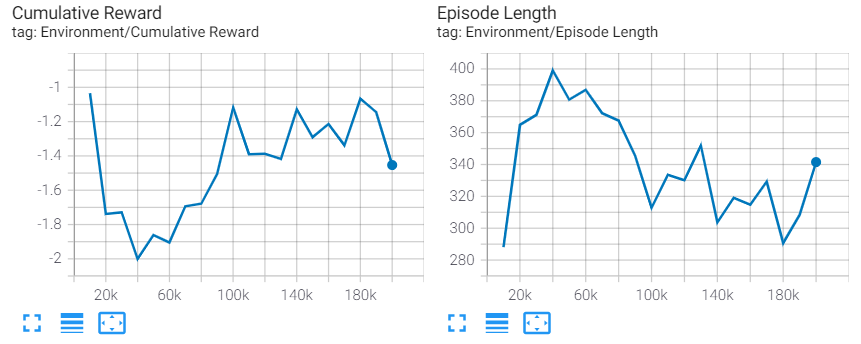


Figure 27. Training results 4

The results were similar to version 0.2 with vector observations only. Interestingly, the total training time to reach 200,000 steps increased from 3 minutes to 5 minutes. This is due to the number of raycasts needing to be updated every step.

4.2.4 Version 0.3.2 – Comparing Raycast and Vector Observations

This time I wanted to compare the agent’s performance with both vector observations and raycasts against vector observations only on the environment with obstacles. 

Figure 28. Training results 5

The results were unlike anything before as the cumulative reward reached 10 times lower lows than before. It looked like the agent which only had vector observations (green line) had gotten stuck on a wall from the very beginning and was not able to recover. The agent that also had raycast observations (red line) experienced several major dips in reward and spikes in episode length, however was able to recover. From this, I believed that the combination of raycasts and vector observations worked better.

4.2.5 Version 0.3.2.1 – Increased Neural Network Layers and Nodes

The instability of the training reward caused concerns and I couldn’t draw any conclusions just yet. Instead, I posted about the issue on the ML-Agents forums. The suggestions that I received were to increase the number of layers and nodes in the neural network, as well as increase the training time.

For this version, I started by increasing the number of neural network layers to 3 from 2 and doubled the number of nodes from 128 to 256.

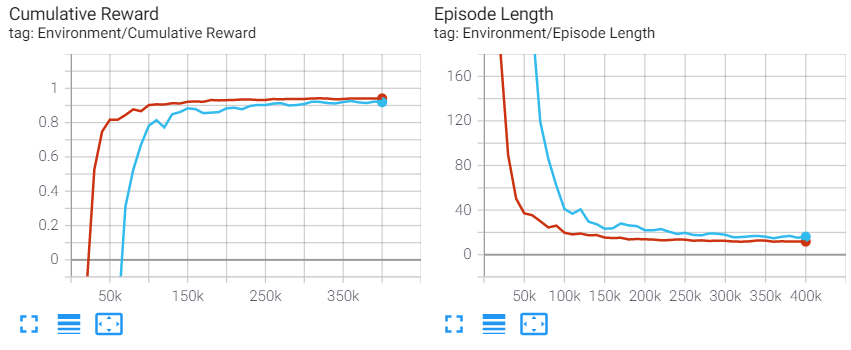


Figure 29. Training results 6

The training was much more stable and effective, reaching a high cumulative reward. The agent with raycasts (red line) took fewer steps to reach the optimal cumulative reward, however, the agent without the raycasts (blue line) took less time to train. The agent that relied on vector observations alone finished the 400,000 steps in 7m 37s, while the other agent took 10m 28s.

4.2.6 Version 0.4 – Trying To Learn To Avoid Obstacles

Both agents in the previous version still had issues with getting stuck on walls. Despite raycasts not showing any significant advantage in the previous versions, I have attempted several solutions in this version. I have added a negative reward to the hider agent for touching the walls. The agent still needed to be able to understand what it is touching that causes the negative reward, thus it was given raycasts, as before. This time, however, they were reduced in length and were adjusted to only recognize wall objects while the position of the target was fed through a vector observation.

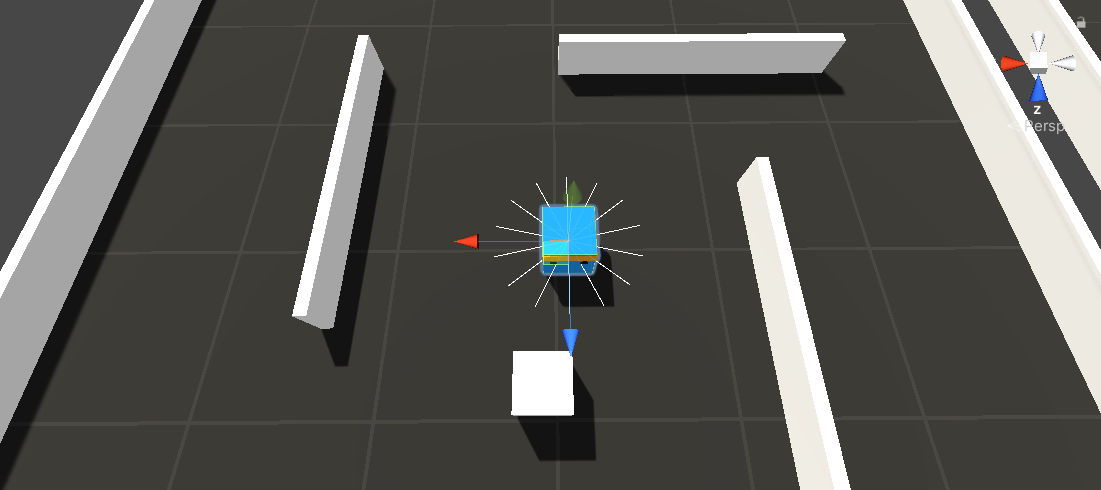


Figure 30. Hider agent with short raycast sensors

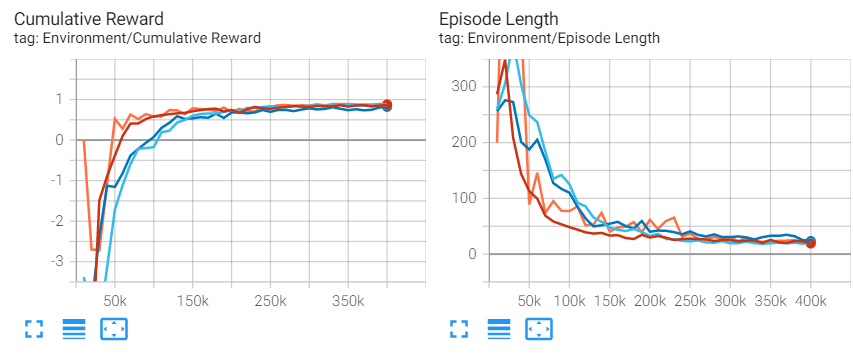


Figure 31. Training results 7

In total, I experimented with 4 different setups.

1. Added negative reward for touching walls (orange line)
2. Added negative reward and raycasts (purple line)
3. Added another neural network layer (red line)
4. Doubled experience buffer hyperparameter (blue line). The buffer size is the number of experiences to collect before updating the policy model. This corresponds to how many experiences should be collected before we do any learning or updating of the model.

All of these training sessions looked near identical to version 0.3.2.1. In the first setup, the agent was still getting stuck on walls, however did not accumulate a large negative reward because the penalty was only given only once when the agent first touched the walls. By touching the wall once and grinding against it the agent avoided getting more negative rewards. With the second setup, the raycasts helped the agent to avoid touching the walls. However, it would still try to go straight into the wall instead of walking around it.

4.2.8 Version 0.4.1 – Removed Raycasts and Increased Training Time

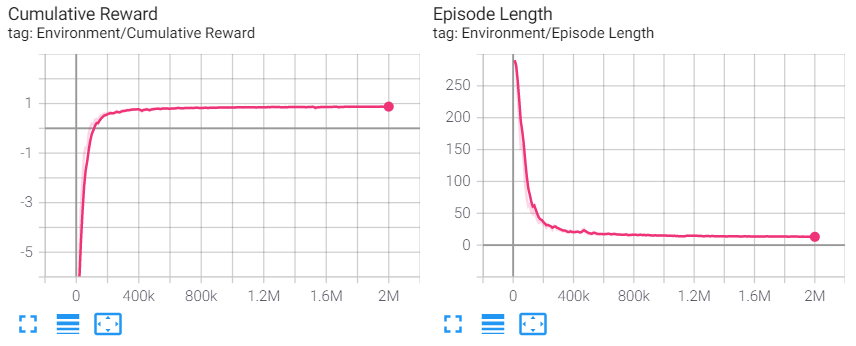
With the previous results, I have decided to remove the raycasts entirely. When posting about the issue once more on the ML-Agents forums it was pointed out that 400,000 steps are not nearly enough for a training session. With this, I have increased the training steps to 2,000,000 and ran the training session with vector observations only. 

Figure 32. Training results 8

The training took 41m 17s to finish. Surprisingly, the agent eventually became much more efficient in walking around the walls without having any observations to detect them. While there were some occasions where it would still take some time to walk around a wall, these moments were much less frequent and would likely disappear with more training time.

4.2.8 Version 0.5 - Training Both Agents Together

The previous version showed the success of training a single hider agent in an environment with obstacles. With the lessons learned previously, this version attempts to train a hider and seeker agents together for the first time.

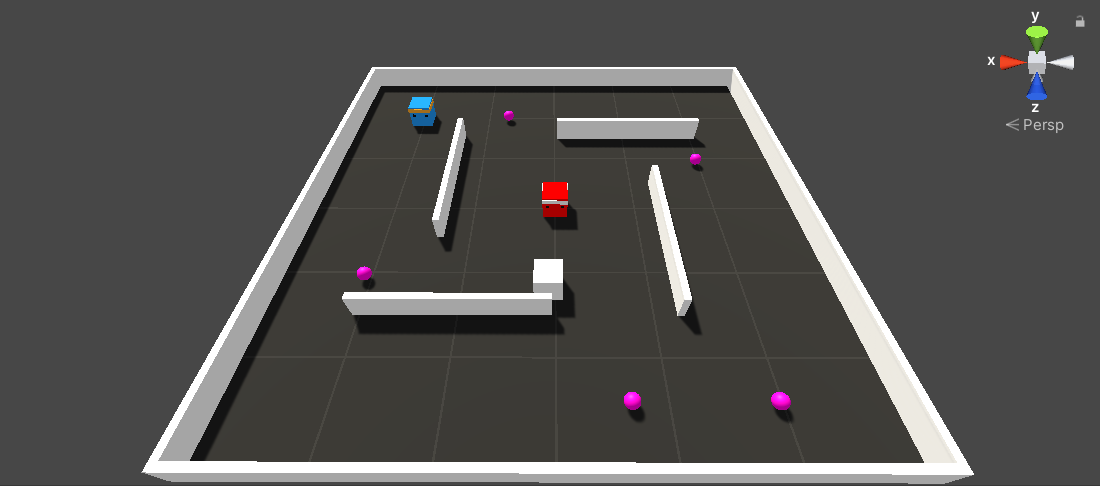


Figure 33. Full training environment 1

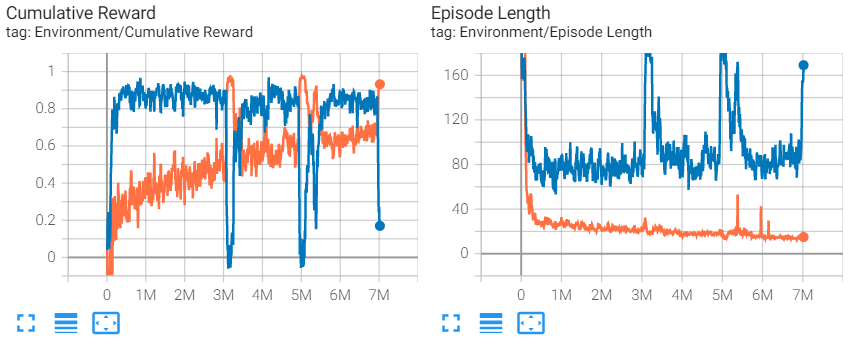


Figure 34. Training results 9

The first round of training took place in the first environment and the training steps were further increased to 7,000,000. The reward of the seeker is shown in the blue line, while the hider is orange. The training took 5h 38m 43s to complete. I am still uncertain what caused the large spikes in the rewards graph. These spikes show a rapid change in reward between the hider and seeker, where the hider comes out on top for a some time, until the seeker overtakes again.

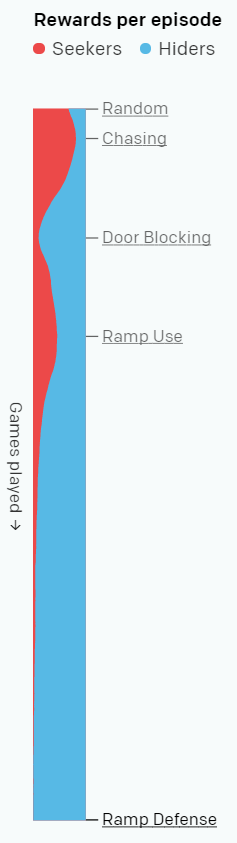
There are two possible explanations for this massive spike in rewards. First and foremost, it could simply be an issue with the ML-Agents toolkit or the tensorboard graphing tool. It could however be a sign of multi-agent competition. OpenAI’s paper [2] has observed a very similar occurrence in the hider and seeker rewards. Once a new behaviour emerged in the hider agents, the rewards balance shifted in favour of the hiders. Given enough time, the seekers would learn to adapt to the new behaviour and once again shift the reward balance back.

Figure . Training results from OpenAI's project

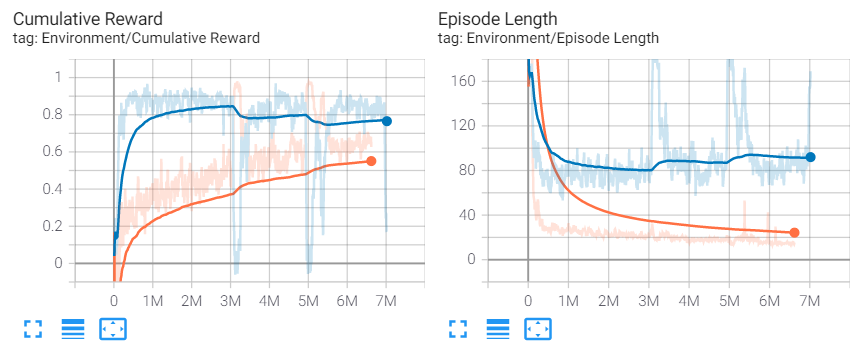
The seeker agent had an advantage from the start, as the hider would move around randomly, sometimes bumping into the seeker on its own. Smoothing out the graph shows a steady increase in the hider’s reward as it learns to avoid the seeker more efficiently, while the reward of the seeker keeps decreasing which suggests that the reward values would converge at some point.

Figure 36. Training results 9 smoothed

The trained models were then moved to new environments. The ML-Agents toolkit allows resuming training at any point, regardless of the environment. When the models were embedded into the agents in the new environments they were significantly worse at performing their tasks. Despite that, they would still perform their behaviours – the seeker would chase and patrol and the hider would try to run and reach the target.

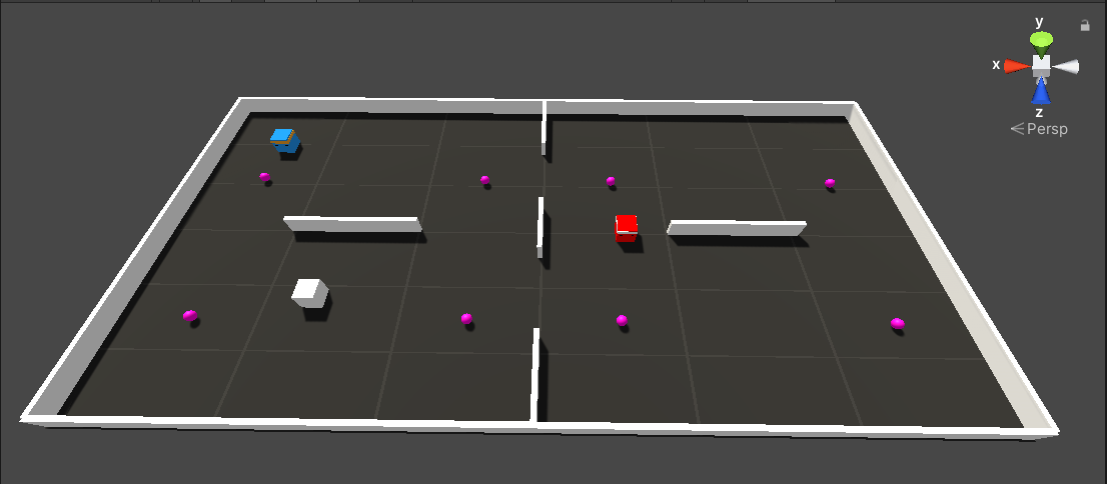


Figure 37. Full training environment 2

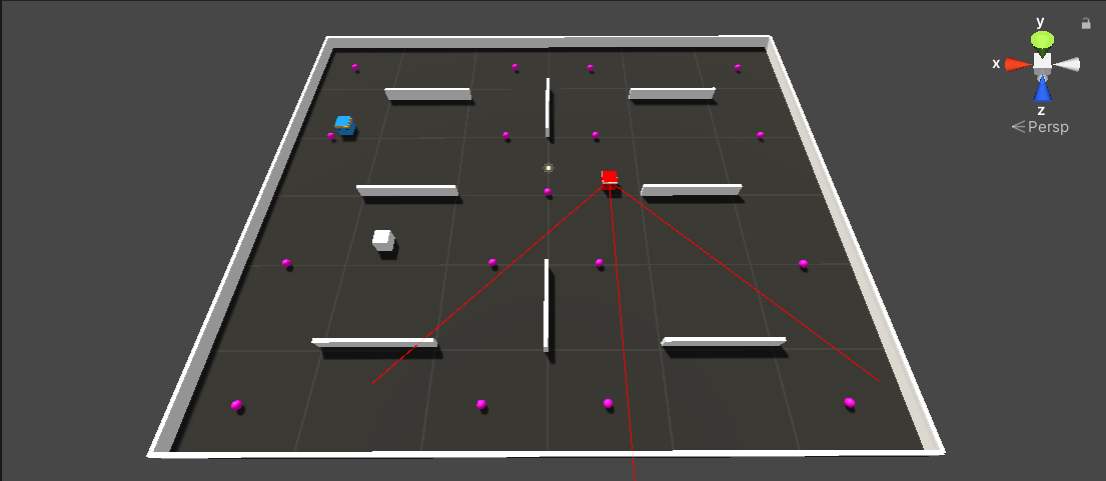


Figure 38. Full training environment 3

Instead of a complete re-training, the models were trained further to adapt to the new environment.

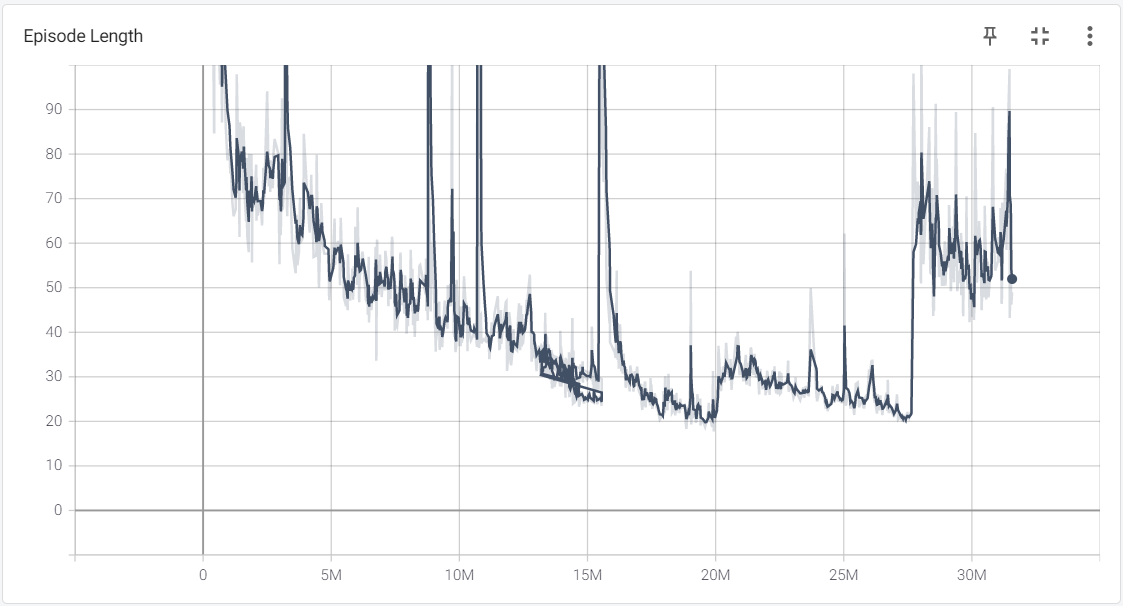


Figure 39. Hider agent training results 10 episode length

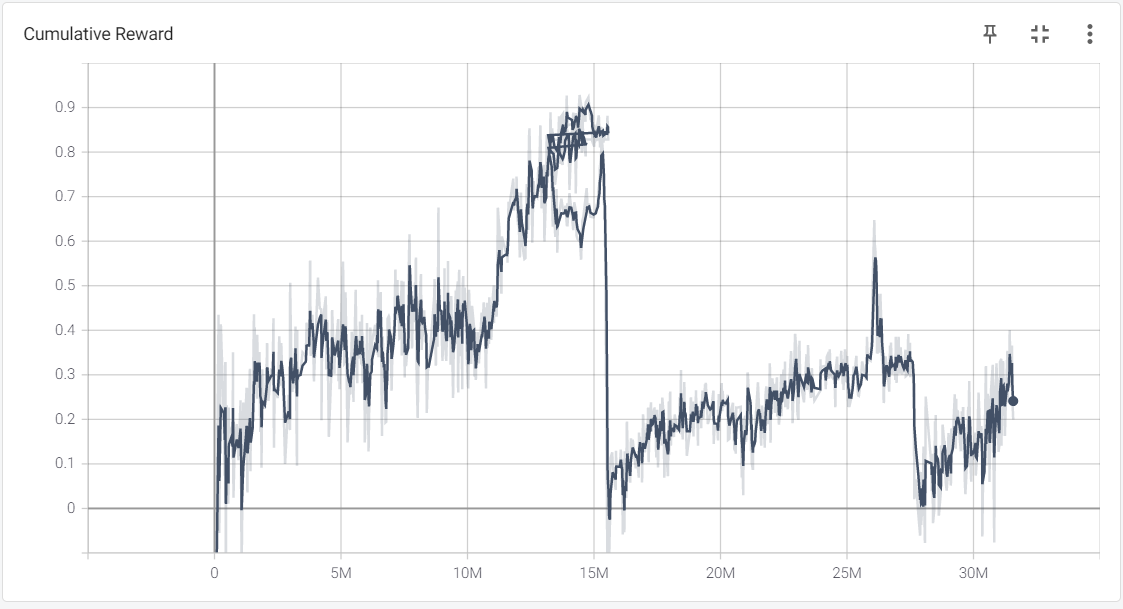


Figure 40. Hider agent training results 10 cumulative reward

The graphs above show the cumulative reward and episode length of a hider‘s model throughout the training in three environments. The model was left to train further on the first environment until around 15,000,000 steps from the previous version and was moved to the second environment. This caused a large drop in reward. Given enough time the agent started to adapt to the new environment and its cumulative reward kept growing. At around 27,000,000 steps the agent was once again put into another environment. This time, the cumulative reward started to climb back up much quicker, suggesting that with more experience it becomes easier to adapt to new environments. The total time spent training the model was around 23 hours of which 12 were spent in the first environment, 8 in the second and 3 in the last environment.

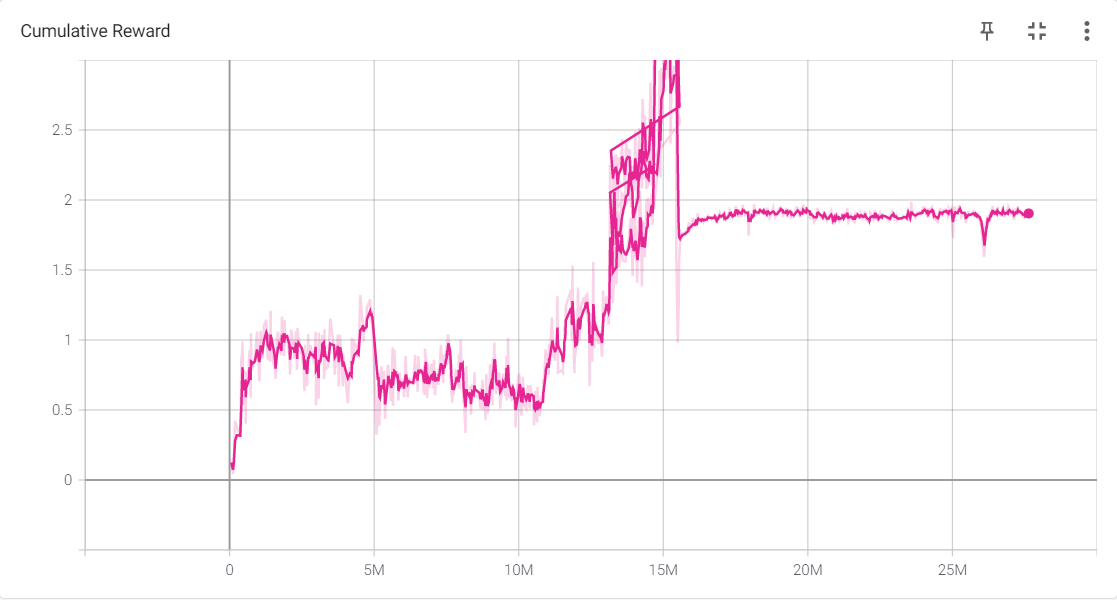


Figure 41. Seeker agent training results 10 cumulative reward

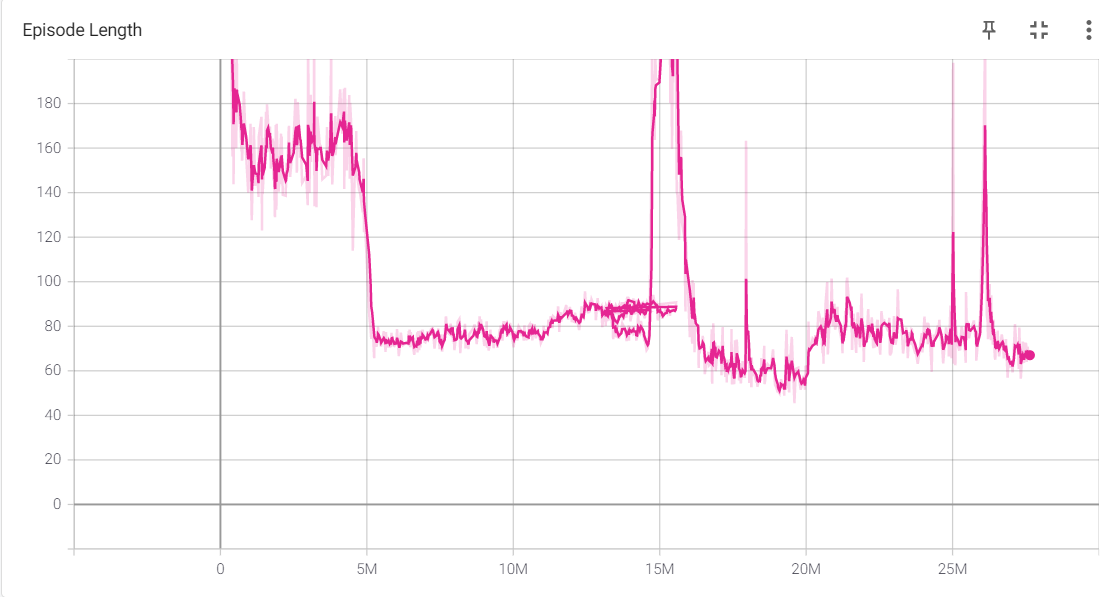


Figure 42. Hider agent training results 10 episode length

A similar drop in cumulative reward can be seen in the seeker‘s model. Interestingly, the reward did not climb back up with time and instead plateaued. A plateaued reward graph means that the model has already learned to perform its task as efficiently as it can learn. Moving to the second environment caused a big spike at 15,000,000 steps while moving to the third environment caused a much less significant spike at 27,000,000 steps. These are unexpected results and I am unsure what could have caused this. It could mean that some tasks are easier to adapt to than others.

5 Evaluation

The agents are evaluated by embedding the trained neural brain models into a series of environments to conduct tests that investigate a set of observations. First, the agents are observed visually using trail paths. These trails will visualise how the agents interact, as well as allow for direct comparison in agent movement and proficiency in their tasks. Then the trained and scripted agents are compared by measuring the performance of the hider against both agents in all environments. This is done by recording the percentage of successful targets reached by the hider agent and the average time taken to reach the target. Lastly, the trained and scripted agents will be compared by the computational resource usage.

5.1 Trained AI and Traditional AI Movement

5.2.1 Hider Agent

The hider agent became proficient in reaching its target with a minimal amount of training time. Given enough time the agent also learned to avoid obstacles despite not having any visual observations passed onto its sensors to detect them. A well-trained agent should produce a trail that wanders across the entire environment while avoiding obstacles such as walls.

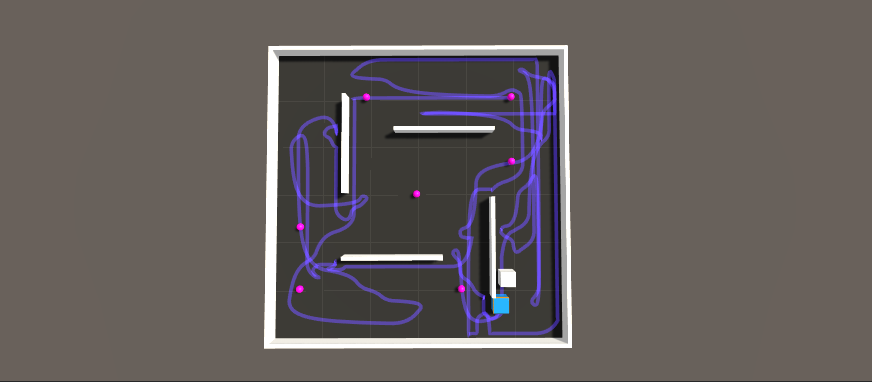


Figure 43. Hider agent movement trail in environment 1

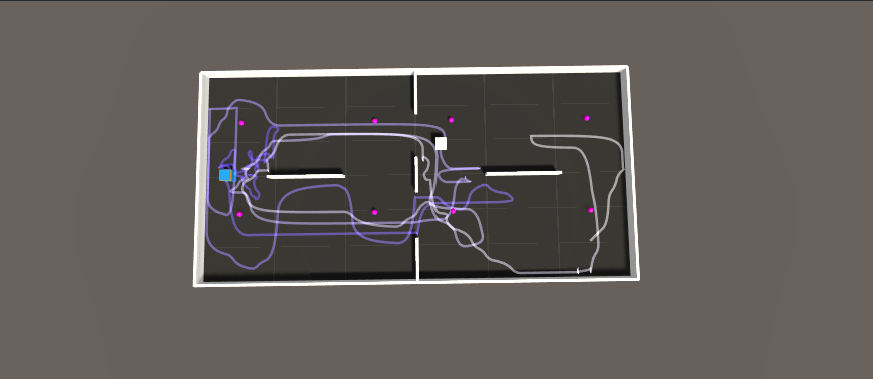


Figure 44. Hider agent movement trail in environment 2

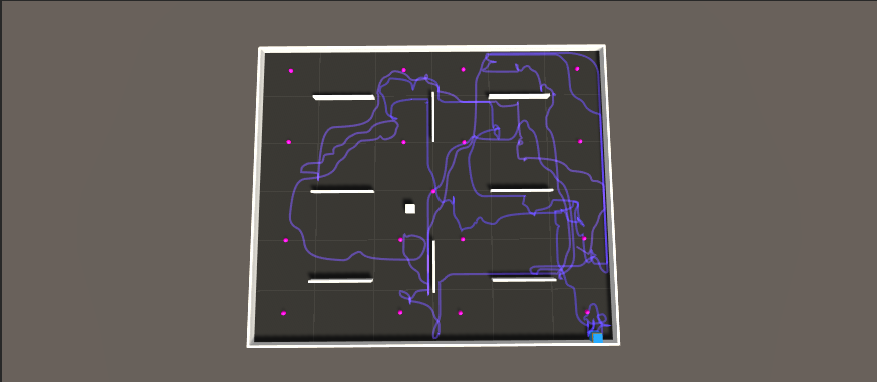


Figure 45. Hider agent movement trail in environment 3

5.2.2 FSM Agent

Since the agent was fully scripted, its movements were guaranteed to be precise and follow the fastest path to the goal. Its trail should show the most optimal patrolling path which a successfully trained agent should follow as closely as possible.

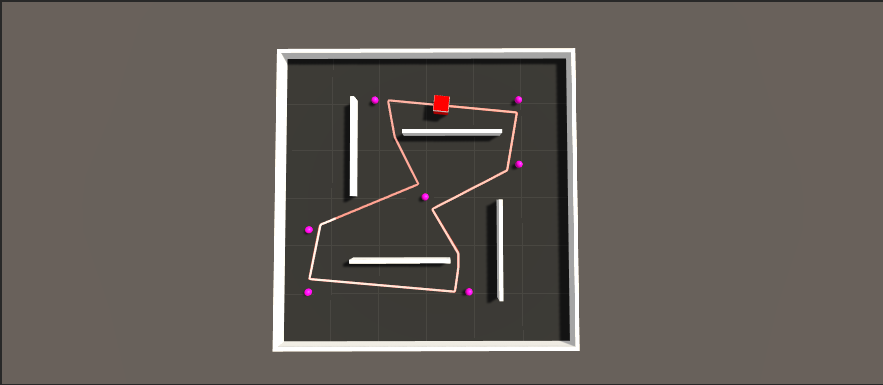


Figure 46. FSM agent movement trail in environment 1

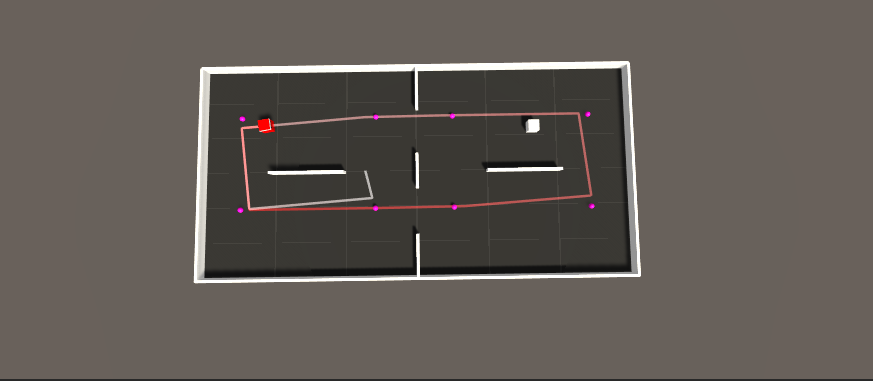


Figure 47. FSM agent movement trail in environment 2

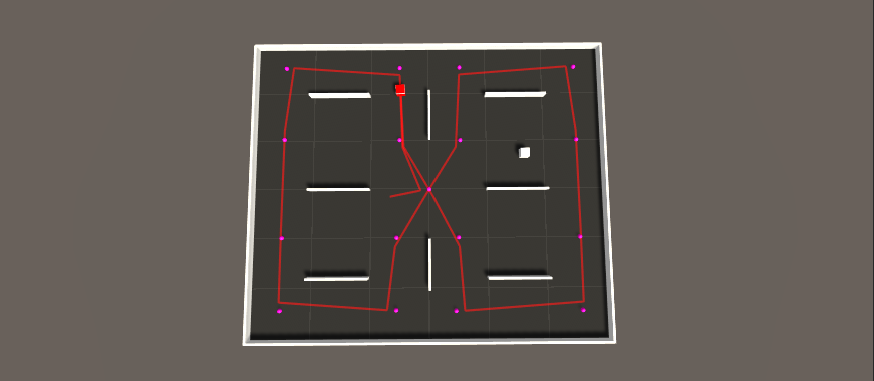


Figure 48. FSM agent movement trail in environment 3

The agent is also perfect in performing chasing as it can immediately react to seeing the hider and keep it in the middle of its cone of sight while moving directly towards it. The chasing can be seen in deviations of the trail from the patrolling path shown above. The instances where the hider is caught can be seen in its sudden change in the trail path as the agent is teleported away to a new location.

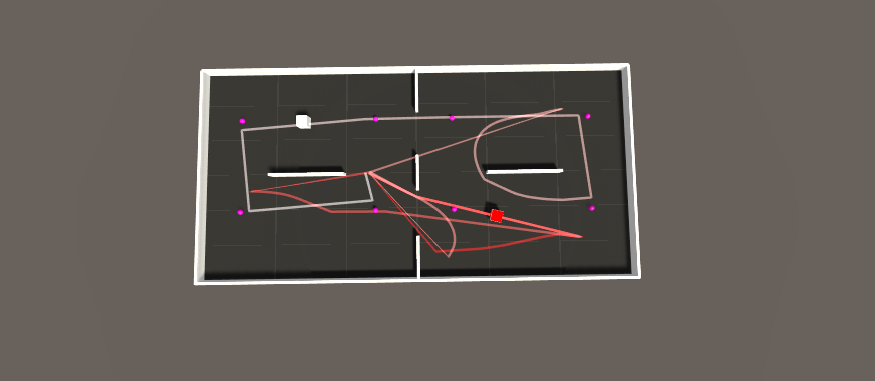


Figure 49. FSM agent movement trail in environment 2 with chasing

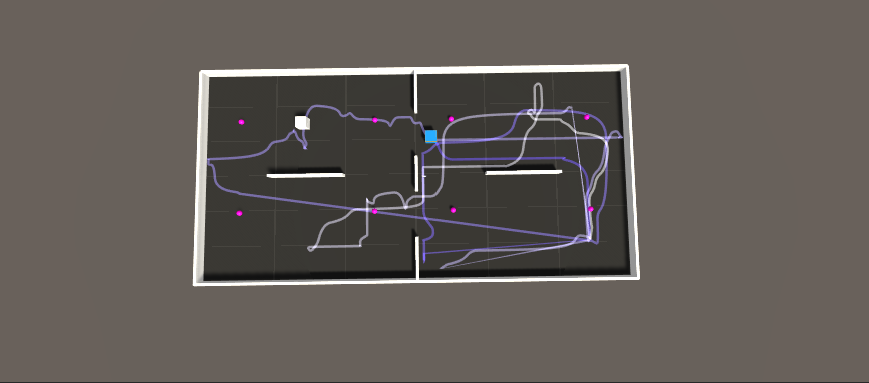


Figure 50. Hider agent movement trail in environment 2 with chasing

5.2.3 Trained Agent

When trained exclusively to patrol the environment, the agent took very little time to learn (less than 5 minutes) to go step through every checkpoint in the correct order. The movement is not as precise and accurate as with the FSM agent, however, given the little amount of training the results can easily be improved.

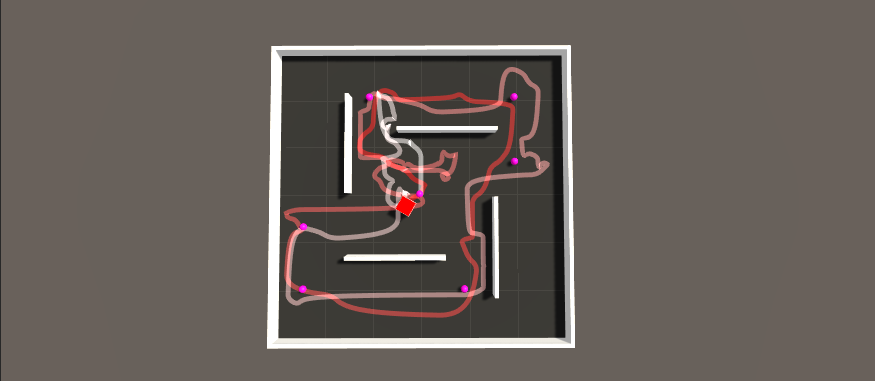


Figure 51. Seeker agent movement trail in environment 1

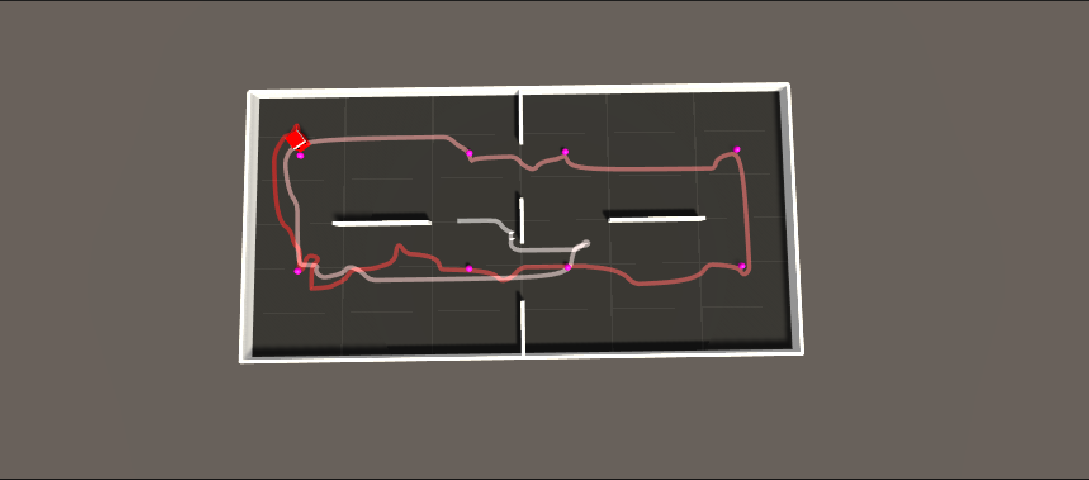


Figure 52. Seeker agent movement trail in environment 2

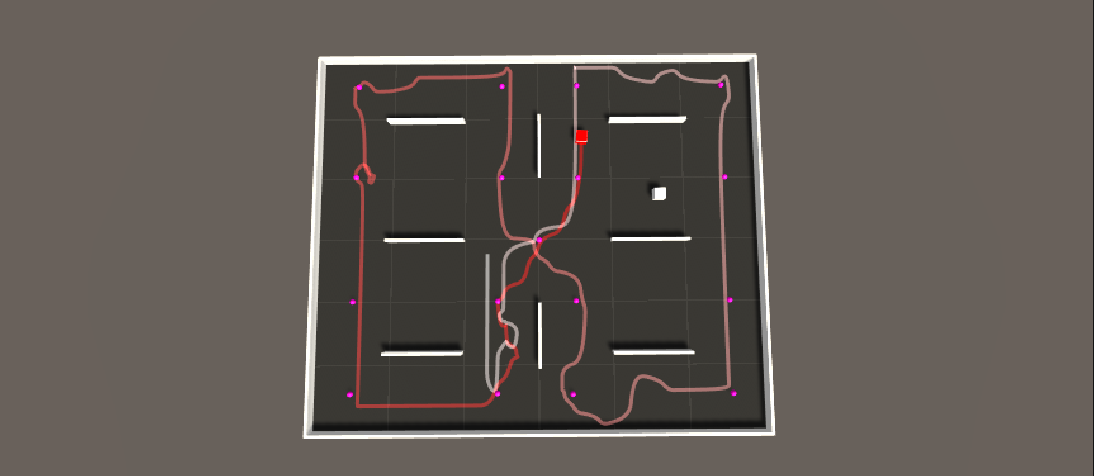


Figure 53. Seeker agent movement trail in environment 3

Training the agent to patrol the environment and chase the hider, however, was a much more troublesome task. The agent would struggle to balance the two tasks given to it by either focusing on the checkpoints alone or chasing only. In the case of learning to chase only, the agent would fail to patrol the environments without the hider agent as shown in the screenshots below.

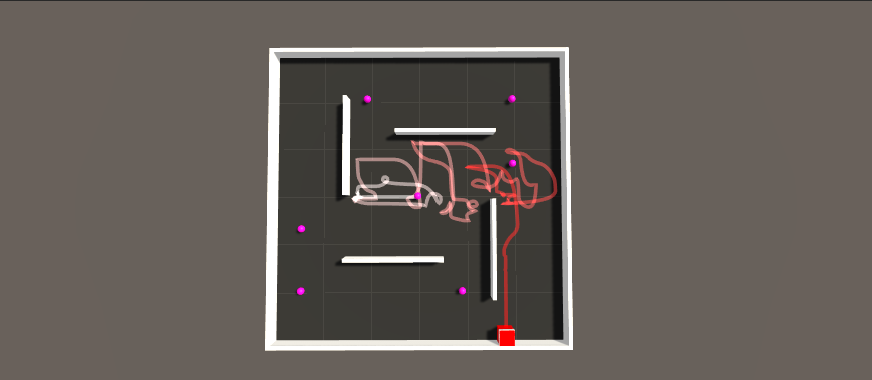


Figure 54. Seeker agent bad movement trail in environment 1

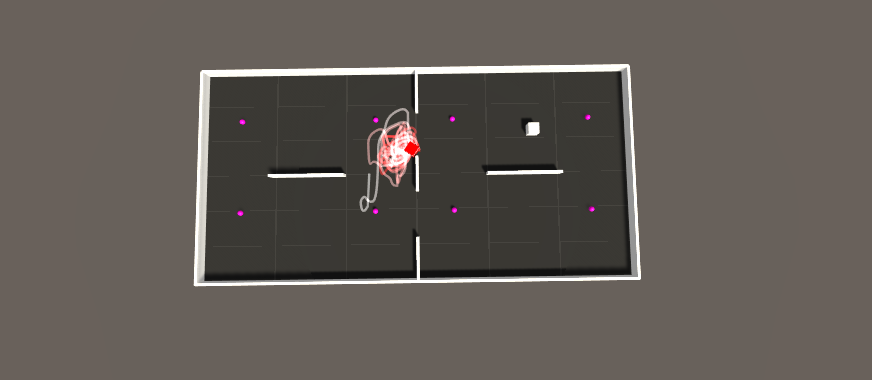


Figure 55. Seeker agent bad movement trail in environment 2

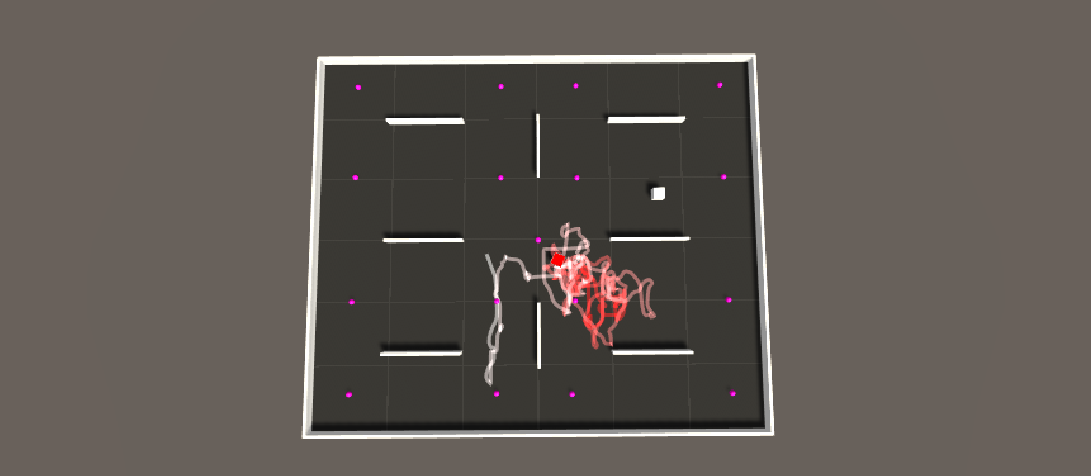


Figure 56. Seeker agent bad movement trail in environment 3

After making a thread about the issue on the ML-Agents forums it was pointed out that the issue was in the code implementation of the reward system. The agent receives a reward for two possible actions: catching the hider and reaching the next checkpoint. The greater the reward for an action, the more the agent will prioritize it. At the beginning of the project, the rewards for reaching a checkpoint and catching the hider were of equal value. This resulted in the agent arbitrarily prioritizing either one of these tasks. As shown previously, learning to reach the checkpoints was an easy task – much easier than chasing and catching the agent. In this case, the rewards had to be rebalanced to make sure the reward is much greater for the more difficult task and smaller for the easier one. The rebalancing process was done by trial and error to find the most optimal set of reward parameters, as well as some other changes in the code. The changes in the rewards resulted in a more promising trained model that managed to not only patrol the environment but also chase and catch the hider agent much like the FSM seeker.

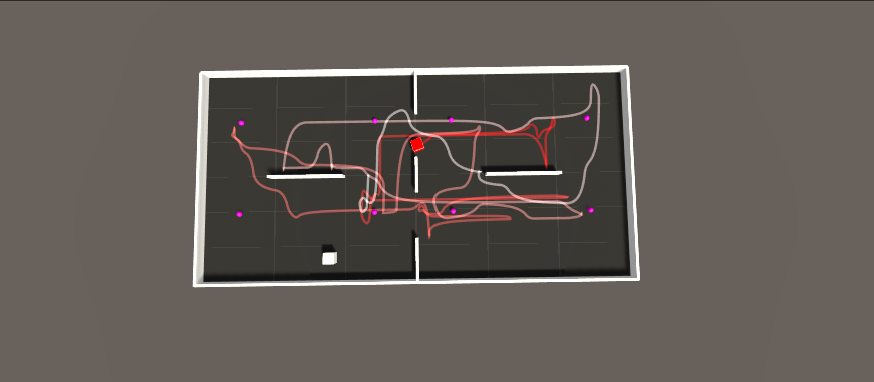


Figure 57. Seeker agent movement trail in environment 2 with chasing

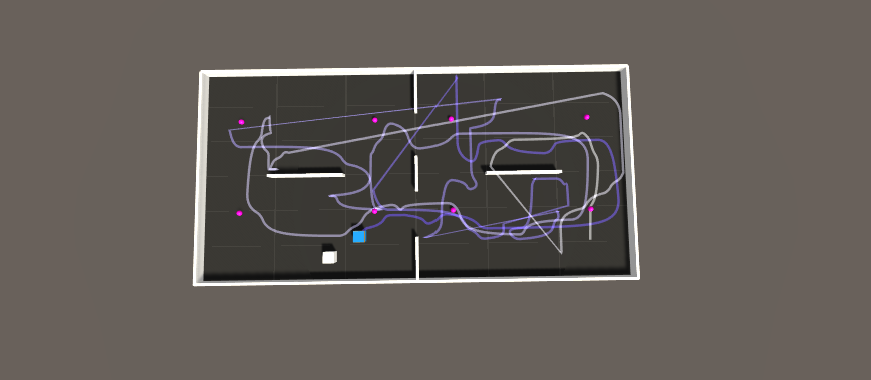


Figure 58. Hider agent movement trail in environment 2 with chasing

5.2 Hider Agent performance against Traditional AI and Trained AI

The latest models for the hider and seeker were placed into all three training environments and the game ran for 100 rounds to collect the observations. A total of two observations were recorded – the number of targets reached and the average time taken to reach the target.

**Environment 1**

|  |  |  |
| --- | --- | --- |
| Observation | Versus traditional AI | Versus trained AI |
| Targets reached (100 tries) | 73 | 72 |
| Average time taken (s) | 2.13 | 2.97 |

Table . Hider agent performance in the first environment

**Environment 2**

|  |  |  |
| --- | --- | --- |
| Observation | Versus traditional AI | Versus trained AI |
| Targets reached (100 tries) | 58 | 68 |
| Average time taken (s) | 3.04 | 2.44 |

Table . Hider agent performance in the second environment

**Environment 3**

|  |  |  |
| --- | --- | --- |
| Observation | Versus traditional AI | Versus trained AI |
| Targets reached (100 tries) | 53 | 62 |
| Average time taken (s) | 4.55 | 6.33 |

Table . Hider agent performance in the third environment

Except for the first environment, the hider agent struggled more to reach the target uncaught against the scripted agent. This can be explained when comparing the movement trails of the scripted and trained agents. The scripted agent moves in a perfectly straight line while the trained agent tends to rotate or even turn around. Such unpredictable moving patterns increase the probability of spotting the hider and is more difficult for the hider to learn to avoid.

5.3 Trained AI and Traditional AI Computational Resource Usage

A unity scene was run with 16 copies of the same environment with a hider agent and different seeker agents. There was no real difference in resource usage observed between the trained and scripted agents. In both cases, the CPU usage was fluctuating around 19-25% while the RAM usage was identical.

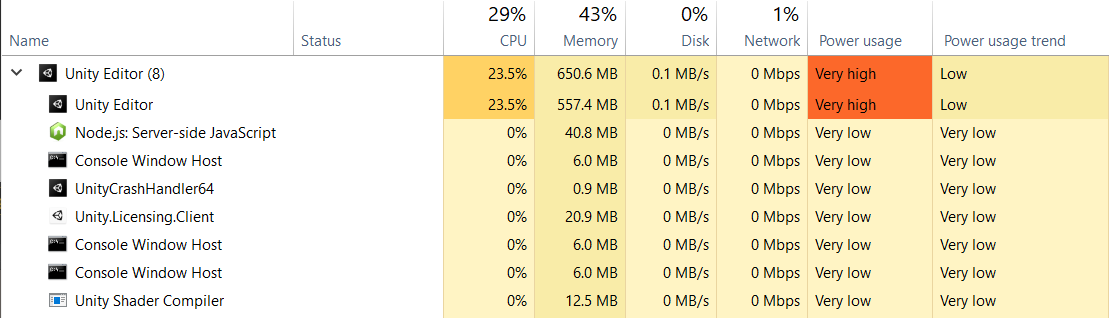


Figure 59. Unity resource usage with trained AI agents

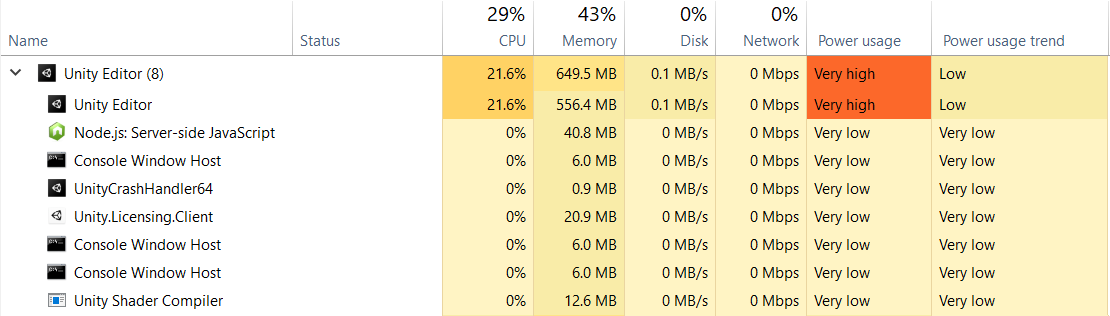


Figure 60. Unity resource usage with a FSM seeker

6 Conclusion

6.1 Objectives Achievement

At the beginning of the project, a total of six objectives were set. I will go through each objective and explain how they were met.

1. Use online courses, published articles, case studies as well as Unity’s ML-Agents documentation to research and identify a technique for implementing the AI.

I believe this objective has been fully met and I am very pleased with it. The first part of the background research chapter covers both artificial intelligence and machine learning, how they are used in video games. This has been the most enjoyable part of the project as it sparked a genuine interest in AI and machine learning that I hope to keep working on in the future. The second chapter looked at Unity’s ML-Agents toolkit documentation and helped identify the technique for implementing the AI which marked the completion of the first objective.

1. Establish a set of game rules and agent behaviours to achieve the desired ‘hide and seek’ behaviour.

The agents were set up to play a variation of the game of hide and seek with the following rules:  
  
The players were split into two teams – the hiders and seekers. The goal of the hider was to reach a randomly positioned target while avoiding the seeker. The goal of the seeker was to patrol the environment and chase the hider when spotted. It was decided that only the seeker would know the location of the target to prevent the seekers from guarding it.

1. Build at least 3 game level prototypes on which the training will take place.

Three training environments of different sizes were built on which the training and evaluation process took place.

1. Define a set of observable parameters for the hider and seeker agents that will be used to compare their performance to a fully scripted agent.

A total of seven parameters were defined and all of them were discussed in the evaluation chapter. These parameters are as follows.

* Hider agent success rate in reaching target uncaught vs FSM agent (% from 100 game rounds)
* Hider agent success rate in reaching target uncaught vs machine learning agent (% from 100 game rounds)
* Hider agent average time to reach target vs FSM agent (seconds across 100 game rounds)
* Hider agent average time to reach target vs machine learning agent (seconds across 100 game rounds)
* FSM agent proficiency in performing its tasks (Observed visually)
* Machine learning agent proficiency in performing its tasks (Observed visually)
* Computational resource usage of traditional agent and trained agent

1. Build a FSM seeker, implement and train the hider and seeker agents using the rules and behaviours defined in objective 2.

A total of three agents were implemented in this project. First, a fully scripted seeker agent was implemented using a Finite State Machine design which would play the game using the rules established previously. Then, the same FSM behaviour design was reimplemented in a machine learning agent using sensors and a reward system. Lastly, a hider agent was also implemented using simple sensor observations and a reward system.

1. Evaluate if, and to what extent the behaviours of FSM AI can be reproduced using Machine Learning agents using the observable parameters defined in objective 4.

The evaluation chapter took a look at how the trained hider agent performs against both the scripted AI and a trained AI, compared their movement patterns and computational resource usage. While the performance of the trained agents was not as perfect when compared to a scripted agent, they could still perform their tasks well enough at no extra cost of performance. The next chapter elaborates on this further.

* 1. What Has Been Learned

At the beginning of the project, I was curious to find out why, despite all of the advances in the field, has machine learning not overtaken the video game industry in things such as NPC character control. I wanted to know what are the challenges that machine learning faces in video games and decided to investigate this by attempting to train autonomous agents that could replace traditional Finite State Machine agents in a video game environment. Having completed this project, I am now confident enough to discuss my findings on these challenges.

* + 1. Can Machine Learning Agents Replace Traditional AI ?

Based on the results of this project, I can confidently argue that the behaviours of Finite State Machine based AI can be successfully reproduced using machine learning.

In the end, the final trained models for the hider and seeker agents managed to play a simple game of hide and seek using a set of established rules. The behaviours of the scripted agent were clearly defined and implemented using the Finite State Machine model. The same FSM model was replicated in the machine learning agent using a set of sensor observations and rewards. By comparing their movement path trails, it was obvious that the trained agent could patrol the environment and chase the hider much like the scripted agent.

* + 1. To What Extent Can Machine Learning Agents Replace Traditional AI?

While this project has shown that it is possible to replace traditional AI using autonomous agents, the extent of this depends on several factors: behaviour complexity, environment scale and complexity, training method and training time.

The biggest concern for autonomous agents is the complexity of their behaviour. For this project, the agent behaviour consisted of only three states: patroling, chasing and idle. This was very simple to reimplement with the machine learning agent using sensor observations and rewards. Even with only two sources of rewards (checkpoints and caching agent), it took some time to balance the reward system for the seeker agent. Introducing more states, or behaviours would require more balancing and training time.

Another thing to consider is the scale and complexity of the environment. The training results show a significant difference in training time in static and dynamic environments. Teaching the hider agent to reach a non-moving target without a seeker took several minutes of training in all environments. Introducing a seeker agent increased the training time to several hours. The success of the training also heavily depends on the rewards, as the training agents thrive in reward-rich environments. Larger environments often have sparse rewards which can slows down the training speed significantly.

Lastly, the trained agent must be convincing enough to be a suitable replacement for a scripted agent by replicating its movement as accurately as possible. For example, with only several minutes of training the seeker agent‘s movements were not as accurate and precise when patrolling the environment when compared to a scripted agent. At this stage the agent would not be a convincing replacement, however could be improved with more training time. A good example of this is the hider agent. At first, the agent would get stuck on walls, making it unrealistic. Given enough training time, the agent became much more proficient and collect the targets while avoiding obstacles much like a human player would expect an NPC to do.

* + 1. The Pros and Cons of Training Machine Learning Agents

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | C:\Users\upsid\AppData\Local\Microsoft\Windows\INetCache\Content.Word\Screenshot_28.png  Trained AI | | C:\Users\upsid\Desktop\Screenshot_27.png  Traditional AI | |
|  | Pros | Cons | Pros | Cons |
| Training | Fast training for simple tasks | Long training | No training needed | - |
| Scripting | Very little time to script. Less code and easier to maintain. | - | - | Long time to script. More code and difficult to maintain. |
| Task proficiency | Can handle tasks in static and dynamic environments | Needs a lot of training to get to a convincing proficiency level | Perfect at performing tasks | - |
| Movement | - | Messy and unpredictable movement | Precise and predictable movement | - |
| Adapting to new environments | Can adapt to new environments | Needs extra training to re-adapt | Works perfectly in all environments | - |
| Computational resource usage | Identical performance with the traditional AI | | Identical performance with the trained AI | |

Table 4. Comparing the pros and cons of trained and scripted agents

The traditional AI is more favourable in almost every category except the scripting time. Programming a Finite State AI took more lines of code that were split across multiple script files which made it more difficult to find bugs and implement changes. There is no one correct way to script a Finite State AI which gives the programmer a lot of flexibility. On the other hand, this increases the odds of writing error-prone code and makes it more difficult to find resources and get feedback on it. With the ML-Agents documentation, the programming took very little time as both the hider and seeker agents use the same template and share a lot of code that fits in a single script file. Every aspect of the ML-Agents toolkit is extensively documented and has its forums where the users can get feedback from other users and the toolkit moderators.

* + 1. Why Are We Not Using Machine Learning Agents In Video Games?

Machine Learning is already used in video game companies for many aspects of a game. I have reached out to Michele Condò, a Lead Generalist Programmer at Ubisoft Reflections about how Ubisoft uses machine learning. As it turns out, Ubisoft is using machine learning for autonomous navigation, cloth simulation, learned matching etc. These machine learning algorithms are self-contained to improve some aspects of the game during the development process and are applied in the editor/engine. When asked about why machine learning agents aren‘t seen that often in video games, Michele pointed out several interesting things.

**„There is no such problem that cannot be solved in a classic way“**

The main problem with machine learning comes from a design point of view. For example, it is difficult to explain to a production team why it is necessary to spend days to make an NPC go from point A to B when it is already possible using an A\* pathfinding algorithm on a navmesh, which we already have a solution for. Autonomous vehicles are an actual topic in machine learning and video games. Implementing traffic is already possible using splines, physics simulations as well as decision-making algorithms for wandering behaviours, complex patterns for chasing and so on.

This does not mean that machine learning has no uses. Machine learning can drastically improve what we can achieve. In terms of autonomous vehicles, it is true that we already have existing algorithms for them, but what would happen if we want a vehicle that drivers in a rural environment with no roads where there are no splines or any constraints defined? This problem is not yet completely solved in a classic way because using a navmesh and pathfinding in this scenario is much more complicated and expensive. Instead, using machine learning to train vehicles to drive in an open environment can achieve a very good and straightforward result, at the cost of training them for several days.

**„Huge companies can access more resources“**

When starting this project, I believed was that we don‘t see machine learning that often in most video games because smaller studios or solo developers don‘t have access to the same tools as big companies such as Ubisoft that are using closed source software. Michele argues that the problem per se is not in the closed source software, as Ubisoft is using PyTorch and other open-source frameworks to train the networks. He emphasizes the difference between high-level tools such as Unity‘s ML-Agents and frameworks such as PyTorch and Tensorflow. The tool that I believed was missing for small developer teams is a high-level tool – something you can add to your game and use. Underneath the ML-Agents toolkit is using the same machine learning algorithms that can be found online in many different languages and packages such as PyTorch. Ubisoft, of course, has made its own private tools at the same ML-Agents toolkit level that they use in their games. While it is possible for any smaller company to build their own tools on top of the open-source frameworks, it requires a lot of work and people which most cannot afford.

This means that the main difference between big and small companies is related to computational power and human power; huge companies have server farms and hundreds of employees. For a single developer or researcher, the process of training networks can take months or years of work. Clustering huge amounts of data can take 3-4 days which then has to be analyzed. If a bug is discovered or a wrong estimation was made, everything needs to be fixed, analyzed again, reiterated, etc. Having access to more resources helps to speed up the process.

Fortunately, with Unity constantly updating and developing its machine learning toolkit, these issues may be solved soon. Partnering with game companies such as JamCity, Unity introduced various new features in ML-Agents like asynchronous environments, generative adversarial imitation learning (GAIL), and more to speed up training [16]. More so, an ML-Agents cloud offering is planned to be launched in 2021 that will enable users to train networks on a scalable cloud infrastructure.

6.3 What Has Been Learned

Overall, during the implementation of the project, there have been many issues that slowed down the progress. The main problem I encountered was the training of the agents. I had inconsistent luck in training as the agents sometimes would refuse to explore the area and get stuck in a corner or glitch through the environment and fall off. I am unsure on whether it was my error or the fault of the toolkit, but this lead to a lot of time being spent fine-tuning the agent hyperparameters to improve the training consistency.

Another issue is that I couldn‘t find a way to successfully implement the raycast observations. Unity provides several example projects with raycast observations that I have explored and yet could not make work in my project. The agents ended up learning to avoid walls by a long trial-and-error process which could have been sped up using these observations.

6.4 Personal Development

Having completed this project I feel much more confident in the skills and knowledge that I have developed throughout. Below are some areas in which I believe I have developed the most:

* Artificial Intelligence and Machine Learning – Learning about AI and ML has been the most enjoyable part of the project. Having never studied either field before, I have expanded my knowledge on both topics greatly and developed a keen interest to continue learning about them further.
* AI in video games – During the project I learned a lot about how AI NPCs are implemented in video games, both traditionally and using machine learning. I was also able to apply that knowledge practically by scripting my Finite State-based AI and train an autonomous agent.
* Problem-solving – Throughout the project, I encountered many roadblocks in training, programming and learning the toolkit itself. Solving these issues on my own has developed my problem-solving skills greatly.
* Working with toolkits and libraries – I have never worked with such large toolkits before this project and had to overcome a steep learning curve. Working with the ML-Agents toolkit has made me more comfortable with understanding and making use of documentation written by other people which is an important skill for any developer to have.

6.5 Future Work

This project has only scratched the surface of what Unity‘s ML-Agents toolkit has to offer and there are many ways that the agents could be improved further:

* Use a different training algorithm

Training the final models for the agents took a total of 23 hours. Even with this amount of training, the results are not entirely convincing and need more training time. The training algorithm used in this project is PPO, which is more general-purpose and stable than many other reinforcement learning algorithms. In contrast with PPO, a Soft Actor-Critic (SAC) algorithm uses a slightly different approach in training and requires 5-10 times fewer samples to learn the same task as PPO. Switching to the SAC algorithm could yield better training results with less time.

* Add sensory observations to the agents for obstacle detection

During the training process, there was no difference in agent performance in navigating the environment with and without raycast observations. I believe that this could have been caused by a mistake in the implementation of raycast observations and these observations should be included in further versions for more experimentation.

* Introduce more agents in a single environment

The computational resource usage test suggests that it would take roughly the same amount of environments to start pushing the system to its limit. Instead, a more interesting observation would be to see how many agents can be fitted in a single environment and still perform their tasks successfully. Due to the way my agents are implemented, adding each additional hider or seeker agent would require a significant amount of changes to the code with each addition and such testing could not be done without additional time.

* Experiment with imitation learning

It is often more intuitive to simply demonstrate the behaviour that the agent needs to perform, rather than attempting to have it learn via trial-and-error methods. Imitation learning uses pairs of observations and actions from a demonstration to learn a policy. It can be used alone to learn a specific type of behaviour (i.e. a specific style of solving a task) or in conjunction with reinforcement learning to dramatically reduce the time the agent takes to solve the environment. I believe this method has a lot of potential for further development of this project. Imitation learning trains agent to behave as closely as possible as its demonstrations. Using demonstrations directly from a FSM agent could result in much more accurate behaviour.

References

1. Michele Condo [2021, February 26th] "*Applications of Neural Networks in Computer Game Development"*[Online Industry Seminar]
2. Authors, M., 2019. *Emergent Tool Use from Multi-Agent Interaction*. [online] OpenAI. Available at: <https://openai.com/blog/emergent-tool-use/> [Accessed 10 April 2021].
3. elementsofai. 2018. *A Free Online Introduction To Artificial Intelligence For Non-Experts*. [online] Available at: <https://www.elementsofai.com> [Accessed 7 December 2020].
4. Anderson, M., 2017. *Twenty years on from Deep Blue vs Kasparov: how a chess match started the big data revolution*. [online] The Conversation. Available at: <https://theconversation.com/twenty-years-on-from-deep-blue-vs-kasparov-how-a-chess-match-started-the-big-data-revolution-76882> [Accessed 17 March 2021].
5. Ackerman, E., 2016. *How Google Wants to Solve Robotic Grasping by Letting Robots Learn for Themselves*. [online] IEEE Spectrum: Technology, Engineering, and Science News. Available at: <https://spectrum.ieee.org/automaton/robotics/artificial-intelligence/google-large-scale-robotic-grasping-project> [Accessed 17 March 2021].
6. Ibm.com. 2020. *What is Machine Learning?*. [online] Available at: <https://www.ibm.com/cloud/learn/machine-learning> [Accessed 19 March 2021].
7. Guo, Y., 2017. *The 7 steps of machine learning*. [online] Youtube.com. Available at: <https://www.youtube.com/watch?v=nKW8Ndu7Mjw> [Accessed 19 March 2021].
8. Hao, K., 2018. *What is machine learning?*. [online] MIT Technology Review. Available at: <https://www.technologyreview.com/2018/11/17/103781/what-is-machine-learning-we-drew-you-another-flowchart/> [Accessed 9 April 2021].
9. Lou, H., 2017. *AI in Video Games: Toward a More Intelligent Game*. [online] Science in the News. Available at: <https://sitn.hms.harvard.edu/flash/2017/ai-video-games-toward-intelligent-game/> [Accessed 21 March 2021].
10. Lara-Cabrera, R., Nogueira-Collazo, M., Cotta, C., & Fernández-Leiva, A. J. 2015. *Game artificial intelligence: challenges for the scientific community* [online] Available at: <http://ceur-ws.org/Vol-1394/paper\_1.pdf> [Accessed 20 March 2021].
11. Nodet, A., 2019. *Human-Like Playtesting with Deep Learning*. [online] Medium. Available at: <https://medium.com/techking/human-like-playtesting-with-deep-learning-92adafffe921> [Accessed 10 April 2021].
12. OpenAI. 2017. *OpenAI Five*. [online] Available at: <https://openai.com/blog/openai-five/> [Accessed 23 January 2021].
13. Bauckhage, C., 2014. *Clustering Game Behavior Data*. [online] Andersdrachen.files.wordpress.com. Available at: <https://andersdrachen.files.wordpress.com/2014/07/clustering\_game\_behavior\_data\_tciaig2014.pdf> [Accessed 9 April 2021].
14. Dillet, R., 2018. *Unity CEO says half of all games are built on Unity*. [online] Techcrunch.com. Available at: <https://techcrunch.com/2018/09/05/unity-ceo-says-half-of-all-games-are-built-on-unity/> [Accessed 10 April 2021].
15. Unity Technologies, 2017. *Unity ML-Agents Toolkit*. [online] GitHub. Available at: <https://github.com/Unity-Technologies/ml-agents> [Accessed 10 April 2021].
16. Unity Technologies, n.d. *Machine Learning Agents | Unity*. [online] Unity. Available at: <https://unity.com/products/machine-learning-agents> [Accessed 11 April 2021].