



UNIVERSITY OF PIRAEUS – DEPARTMENT OF INFORMATICS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ – ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

MSc «Advanced Informatics and Computing Systems – Software Development and Artificial Intelligence»

ΠΜΣ «Προηγμένα Συστήματα Πληροφορικής – Ανάπτυξη Λογισμικού και Τεχνητής Νοημοσύνης»

MSc Thesis

Μεταπτυχιακή Διατριβή

Topic: Artificial Intelligence – Machine Learning Agents

Θεματική ενότητα: Τεχνητή Νοημοσύνη – Πράκτορες Μηχανικής Μάθησης

<p>Thesis Title Τίτλος Διατριβής</p>	<p>Behavior of Intelligent Agents using reinforcement learning through Unity's ML-Agent's toolkit Συμπεριφορά Ευφυών Πρακτόρων χρησιμοποιώντας ενισχυτική μάθηση μέσω του εργαλείου ML-Agents της Unity</p>
<p>Student's Name – Surname Όνοματεπώνυμο Φοιτητή</p>	<p>Davillas Christos Δαβίλλας Χρήστος</p>
<p>Father's Name Πατρώνυμο</p>	<p>Dimosthenis Δημοσθένης</p>
<p>Student's ID Number Αριθμός Μητρώου</p>	<p>MPSP 19011 ΜΠΣΠ 19011</p>
<p>Supervising Professor Επιβλέπων Καθηγητής</p>	<p>Panayiotopoulos Themis Παναγιωτόπουλος Θέμης</p>

Νοέμβριος 2022 / November 2022

3 – Member Examination Committee

Τριμελής Εξεταστική Επιτροπή

Themis Panayiotopoulos
Professor

Θέμης Παναγιωτόπουλος
Καθηγητής

Dionisios Sotiropoulos
Assistant Professor

Διονύσιος Σωτηρόπουλος
Επίκουρος Καθηγητής

Ioannis Tasoulas
Assistant Professor

Ιωάννης Τασούλας
Επίκουρος Καθηγητής

Abstract

Game AI is as ancient as AI itself, but with the addition of video games, the discipline has witnessed enormous extension and enrichment in the previous decade. Deep learning development has had a substantial and transformative influence on numerous challenging issues during the last decade, including speech recognition, machine translation, natural language comprehension, and computer vision. Many gaming firms are looking for new solutions and technologies to help their products benefit from AI and Machine Learning. Unity's ML-Agent toolkit, which is also the major component of this thesis, is a very broad and popular tool.

There are seven chapters in the paper. The first two chapters discuss the principles of artificial intelligence, gaming AI technologies, machine learning and its structure, as well as some examples of projects created with mlagent's tools.

From chapter three to six, we follow the project's development step by step, from requirements through design, coding, and then the simulation demo.

Finally, some future considerations and a conclusion manage the thesis's closure.

Acknowledgements

This thesis would not have been possible without the assistance and commitment of my supervisor, Professor Themis Panayiotopoulos, whose helpful advise assisted me in overcoming the challenges and obstacles associated with this subject. I'd also like to thank my girlfriend, who has been there for me from the beginning of my academic adventure to the finish.

List of figures

Figure 1 Battlefield 2042	14
Figure 2 No Man's Sky	15
Figure 3 Cyberpunk 2077	16
Figure 4 The Last of Us Part II.....	17
Figure 5 Project Paidia demo – Learning to collaborate in Bleeding Edge, footage of trained Project Paidia agents. Not representative of final game gameplay or visuals.	19
Figure 6 Honor of kings	22
Figure 7 FSM components.....	24
Figure 8 Transitions between Pac-man Ghost states.....	25
Figure 9 FSM Ghost components	25
Figure 10 Visualization of planning tree	27
Figure 11 Traditional Programming.....	30
Figure 12 Machine Learning	31
Figure 13 Learning Phase.....	31
Figure 14 Inference from Model	32
Figure 15 Machine learning Algorithms	33
Figure 16 Classification - Regression types.....	34
Figure 17 Clustering - Unsupervised Learning	35
Figure 18 RL scenario	36
Figure 19 How Reinforcement Learning works	37
Figure 20 Amazon Lumberyard demo scene.....	40
Figure 21 Sniper Ghost Warrior 3	41
Figure 22 Panda 3D demo scene	42
Figure 23 Unreal Engine demo scene	42
Figure 24 Unity3D demo scene	43
Figure 25 Grid Sensors	48
Figure 26 Unity Inspector tab	49
Figure 27 Dodgeball	50
Figure 28 Illustration of centralized learning with decentralized execution; here, three actors select actions based on the observations of a single agent, while a single critic evaluates the behavior of the whole group for training.	51
Figure 29 Raycast Sensors 3D.....	52
Figure 30 DodgeBallGameController.cs	54
Figure 31 ELO Graph.....	55
Figure 32 ML-Agent's example scene, soccer environment.....	56
Figure 33 Physical robot environment	56
Figure 34 Hardware for robot environment	57
Figure 35 Reinforcement Learning Behavior context.....	58
Figure 36 Computer Vision Sensors	59
Figure 37 Unity Hub – New Project	62
Figure 38 Polygon kids	63
Figure 39 Folders.....	63
Figure 40 GitHub repo, Release 19	64
Figure 41 Package Manager Settings.....	65
Figure 42 Enable Pre-release Packages.....	65
Figure 43 ML-Agents and ML-Agents Extension Packages in the project	66
Figure 44 Playground Area prefab.....	67
Figure 45 Hierarchy window	68
Figure 46 GrassArea inspector	68

Figure 47 Rock inspector	69
Figure 48 TextMeshPro inspector	70
Figure 49 PlaygroundArea Inspector window	71
Figure 50 Big Brother prefab	72
Figure 51 Big Brother Inspector window	73
Figure 52 Little brother prefab	74
Figure 53 Little brother inspector window	74
Figure 54 Boat prefab with inspector window	75
Figure 55 PlaygroundArea.cs (1)	76
Figure 56 PlaygroundArea.cs (2)	77
Figure 57 PlaygroundArea.cs (3)	78
Figure 58 BigBrotherAgents.cs (1)	79
Figure 59 BigBrotherAgents.cs (2)	80
Figure 60 BigBrotherAgents.cs (3)	81
Figure 61 Boat.cs	84
Figure 62 Configuration parameters file	85
Figure 63 mlagents help command	90
Figure 64 mlagents training command	91
Figure 65 Multiple training instances	91
Figure 66 Training phase	92
Figure 67 Command prompt training progress	92
Figure 68 Tensorboard command	93
Figure 69 Cumulative reward graph	93
Figure 70 Episode length graph	93
Figure 71 NN Model export	94
Figure 72 Behavior Parameters inspector tab	94
Figure 73 Training with inference	95

Table of Contents

Abstract	3
Acknowledgements	4
List of figures	5
1. Introduction	9
1.1 Artificial Intelligence	9
1.2 How does AI work?	10
1.3 Where is Artificial Intelligence used?	11
1.4 History of A.I.	12
1.5 Artificial Intelligence in the Gaming Industry	13
1.5.1 Different applications of AI in games	14
1.5.2 Influential and popular projects/games that shook the Gaming Industry with their AI tools	18
1.5.3 AI methods used in games	24
1.5.4 Myths in Video Game AI	28
1.6 Machine Learning	30
1.6.1 How does Machine Learning Work?	31
1.6.2 Machine Learning Algorithms and Where they are Used?	32
1.6.3 Important Components of Deep Reinforcement Learning Method	36
1.6.4 How Reinforcement Learning works?	37
1.6.5 Reinforcement Learning Algorithms	38
1.6.6 Characteristics of Reinforcement Learning	38
1.6.7 Types of Reinforcement Learning	39
2. Approach	40
2.1 A.I. in Game Engines	40
2.2 Unity's M.L. Agents toolkit	44
2.2.1 Introduction notes about Unity's M.L. Agents	44
2.2.2 Projects where M.L. Agents toolkit was used	45
3. Project Specifications	61

3.1	Concept idea	61
4.	Installation	62
4.1	Design	62
4.2	Installation	63
4.3	Environment	67
4.4	Characters and items	71
5.	Implementation	76
5.1	Scripts presentation	76
5.2	Trainer Configuration File	85
6.	Presentation	90
6.1	Training Demo	90
7.	Final Words	96
7.1	Future Work – Conclusion	96
8.	References	97

1. Introduction

1.1 Artificial Intelligence

The short answer to “What is Artificial Intelligence”, is the capability of a digital computer or computer-controlled robot to perform tasks that are generally associated with intelligent beings.

A nonprofessional with a transitory understanding of technology would link it to robots. They'd say Artificial Intelligence is a terminator like-figure that can act and make decisions on its own. However, asking an AI researcher about artificial intelligence, (s)he would say that it's a set of algorithms that are able to produce results without having to be explicitly instructed to do so.

When the machines reveal such intellect, the common terminology is Artificial Intelligence and it has grown to be veritably popular in today's world. It is the simulation of natural intelligence in machines that have been programmed to learn and mimic the actions of human beings. These machines can learn with experience and perform mortal-suchlike tasks.

So, by definition, Artificial Intelligence can be sorted into the following three sections

- An intelligent entity designed by humans
- Able of performing tasks intelligently without being explicitly instructed
- Able of thinking and acting rationally and humanely

To measure if Artificial Intelligence acts more humanly, we need to predicate the human-likeness of an AI entity with four approaches listed below:

- The Turing test
- The Cognitive Modelling approach
- The Law of Thought approach
- The Rational Agent approach

The Turing Test in Artificial Intelligence

The foundation of the Turing Test is that the Artificial Intelligence unit should be capable of holding a discussion with a human agent. The human agent preferably shouldn't be able to determine that they are talking to an Artificial Intelligence. To accomplish these ends, the AI needs to retain the following qualities:

- Natural Language Processing (NLP) in order to communicate successfully
- Knowledge Representation to act out as its memory
- Automated Reasoning to use the stored data to answer questions and draw new assumptions
- Machine learning to detect samples and adapt to new situations

Cognitive Modelling Approach

As the name suggests, this approach tries to develop an Artificial Intelligence model based on Human Cognition. To extract the essence of the human mind, there are 3 methodologies:

- Introspection: studying our thoughts, and constructing a model based on that
- Psychological Experiments: conducting experiments on humans and examining their actions
- Brain Imaging: Using MRI to observe how the brain operates in different situations and reproducing that through code

The Laws of Thought Approach

The Laws of Thought are a significant catalog of logical statements that regulate the operation of our brain. The identical laws can be codified and applied to artificial intelligence algorithms. The concern with this specific approach, is because solving a problem in theory (accurately corresponding to the laws of thought) and solving them in practice can be different, forcing relative nuances to use. Additionally, there are some measures that we take without being completely certain of a result that an algorithm might be unable to reproduce if there are a lot of parameters.

The Rational Agent Approach

A rational agent acts to accomplish the best viable outcome in its current circumstances. Corresponding to the Laws of Thought approach, an entity must act according to the logical statements. However, there are some cases, where there is no logical right thing to do, with various outcomes involving unique outcomes and related compromises. The rational agent approach attempts to make the best possible option in the current situations. It implies that it's a much more dynamic and adaptable agent.

1.2 How does AI work?

When building an AI system, an individual need to carefully handle reverse-engineering human behaviors and capabilities in a machine combined with using its computational talents to surpass what we are capable of.

To understand How Artificial Intelligence works, one needs to deep dive into the numerous sub-domains of Artificial Intelligence and understand how those domains could be utilized to the several spheres of the industry.

First, we have **Machine Learning**:

ML teaches a machine how to make inferences and decisions built on previous experience. It detects samples, analyses past data to understand the meaning of these data points to reach a potential conclusion without having to require human experience. This automation to reach

conclusions by estimating data, saves a human time for businesses and helps them make a better judgment.

Second, there is **Deep Learning**:

Deep Learning is an ML method. It teaches a machine to handle inputs through layers to categorize, infer and forecast the outcome.

Furthermore, there's **Neural Networks**:

Neural Networks or NN for short, works on related principles to Human Neural cells. They are a series of algorithms that portrays the relationship between several essential variables and processes the data as a human brain does.

Natural Language Processing:

NLP is a science of reading, understanding, and interpreting a language by a machine. Once a machine recognizes what the user aims to convey, it reacts accordingly.

Computer Vision:

Computer vision algorithms seek to understand an illustration by breaking it down and examining different parts of the object. This supports the machine to classify and learn from a set of images, to make a better output judgment based on prior observations.

Cognitive Computing:

Cognitive computing algorithms try to imitate a human brain by studying text/speech/images/objects in a way that a human does and tries to give the needed output.

1.3 Where is Artificial Intelligence used?

AI is used in different fields to give insights into user behavior and give suggestions based on the data. For example, Google's predictive search algorithm utilized early user data to foresee what a user would type next in the search bar. Netflix make use of past user data to propose what movie a user might want to see next, making him tied onto the platform and increase watch time. Facebook draw on prior records to automatically provide suggestions to tag your friends, based on their facial features in their images. AI is used everywhere by large businesses to make an end user's life easier.

The uses of Artificial Intelligence would broadly fall under the data processing group, which would include the following:

- Searching within data, and optimizing the search to give the most appropriate results
- Logic-chains for if-then reasoning, that can be applied to achieve a string of commands based on parameters
- Pattern-detection to identify significant patterns in large data set for unique intuitions
- Applied probabilistic models for predicting upcoming results

1.4 History of A.I.

The technology of Artificial Intelligence is a lot older than some would imagine, and the term “AI” is not new for scientists. It was first devised at Dartmouth college in 1956 by a scientist named Marvin Minsky. Also, in Ancient Greek mythology intelligent robots and artificial entities are presented for the first time. The creation of syllogism and its application of deductive reasoning by Aristotle was a defining moment in humanity’s quest to comprehend its own intelligence. Regardless of its long and deep roots, artificial intelligence as we know it these days has only been around for less than a century.

Further down we will dive deep into the important timeline of events related to artificial intelligence:

1943 – Warren McCulloch and Walter Pitts published a paper titled “A Logical Calculus of Ideas Immanent in Nervous Activity” which was the first work on artificial intelligence (AI) in 1943. Their suggestion was an artificial neuron model.

1949 – Donald Hebb proposed the concept for adjusting connection strength between neurons in his book “The Organization of Behavior: A Neuropsychological Theory”.

1950 – Alan Turing, a renowned English mathematician published “Computing Machinery and Intelligence” in which he proposed a test to define if a machine is capable of exhibiting human behavior. This test is famously recognized as the Turing Test. In the same year, two Harvard graduates Marvin Minsky and Dean Edmonds built the first neural network computer known as SNARC.

1956 – The “first artificial intelligence program” named “Logic Theorist” was fabricated by Allen Newell and Herbert A. Simon. This program validated 38 of 52 mathematical theorems, as well as discovering new and more sophisticated proofs for quite a few of them.

For that year also, the word “Artificial Intelligence” was first adopted by John McCarthy, an American scientist at the Dartmouth Conference and was coined for the first time as an academic field.

The enthusiasm towards Artificial Intelligence grew rapidly after this year.

1959 – Arthur Samuel invented the term machine learning while he was working at IBM.

1963 – John McCarthy started an Artificial Intelligence Lab at Stanford.

1966 – Joseph Weizenbaum created the very first chatbot named ELIZA.

1972 – WABOT-1 was the first humanoid robot, built in Japan.

1974 to 1980 – This era is famously known as the first AI winter period. Lot of experts could not pursue/continue their research to the best extent as they fell short of funding from the government and the activity towards AI steadily deteriorated.

1980 – AI returned with a knock! Digital Equipment Corporations developed R1 which was the first thriving commercial professional system and officially closed the AI winter period.

During the same year, the first ever national conference of American Association of Artificial Intelligence was organized at Stanford University.

1987 to 1993 – With evolving computer technology and cheaper alternatives, many investors and the government halted the funding for AI research leading to the second AI Winter period.

1997 – A computer beats human! IBM's computer IBM Deep Blue overcame the greatest world chess champion of that time, Gary Kasparov, and became the first computer/machine to beat a world chess champion.

2002 – The foundation of vacuum cleaners made AI enter residences.

2005 – The American military started investing in autonomous robots such as Boston Dynamics' "Big Dog" and iRobot's "PackBot."

2006 – Companies like Facebook, Google, Twitter, Netflix started utilizing AI.

2008 – Google made a breakthrough in speech recognition and introduced the speech recognition feature in the iPhone app.

2011 – Watson – an IBM computer, won Jeopardy in 2011, a game show in which it had to solve complicated questions and riddles. Watson had demonstrated that it could comprehend plain language and solve complex problems fast.

2012 – Andrew Ng, the Google Brain Deep Learning project's founder, fed 10 million YouTube videos into a neural network using deep learning algorithms. The neural network learnt to recognize a cat without being informed what a cat is, which marked the beginning of a new era in deep learning and neural networks.

2014 – Google made the first self-driving car which passed the driving test.

2014 – Amazon's Alexa was released.

2016 – Hanson Robotics created the first "robot citizen," Sophia, a humanoid robot capable of facial recognition, verbal conversation, and facial emotion.

2020 – During the early phases of the SARS-CoV-2 pandemic, Baidu made its Linear Fold AI algorithm accessible to scientific and medical teams seeking to create a vaccine. The system could anticipate the virus's RNA sequence in just 27 seconds, which was 120 times faster than prior methods. (1)

1.5 Artificial Intelligence in the Gaming Industry

Since the first program that played chess in the '50s, video games have been correlated with artificial intelligence. The attempts to solve the challenge of computers defeating human experts in strategy games such as chess, poker, and GO have drastically advanced AI research. In turn, this has led to enrichments in the design of modern games.

In a broad sense, most video games integrate some form of AI. Nevertheless, modern AI methods can be utilized in distinct areas that support companies to realize business benefits as well. For instance, through graphical upgrades, generating content, balancing in-game complexities and providing intelligence to non-playable characters (NPCs), AI improves the whole user experience whilst saving game companies a lot of budget and time. (2)

The use of AI in games is diverse. New breakthroughs are bringing developers unprecedented with new attitudes to game design, shifting the way games are made and played.

1.5.1 Different applications of AI in games

Player Modeling

Most games, or at least those with high budget, frequently collect data about their players. That could be as simple as when a session starts and ends or be much more detailed, like every action the player took during the game. This data has many uses. Machine Learning can be used for player modelling, which in terms, is the practice of producing computational models centered on players. Expectedly, the most crucial question for numerous developers is when the player will stop or which player and under which circumstances, will pay money for improvements and in-game items. Though, data from players can be used for so much more.

In academic world, player modelling has been researched for 15 years, and study suggests many more things can be modelled like this. We can learn to forecast what players will experience based on their play style, if they think a particular kind of level is entertaining to play, or if a character will be captivating to interact. It is also promising to learn models of a player across multiple games. Hence, platform holders and publishers with numerous games have a unique opportunity to learn a lot about their players and use the models to make predictions in games that have not been published yet. Models like these can help improve games by adjusting them to the player.



Figure 1 Battlefield 2042

Racing and other sports games, already perform an easy form of adaptation to the player in the type of dynamic difficulty adjustment. With better player models, we could take this one step further. For example, developers could select content that fits not just a player's skill level – and normally, skill isn't one-dimensional. Some players might be better at strategy games while others may have rapid reflexes that help them with shooting titles — but they could also select content they know the players will like. So, shaping their experience by proposing them an enjoyable, but not overwhelming, challenge and keeping them playing for longer.

With enough data and Machine Learning methods, we can predict not only what a player will enjoy right now, but even what the player will enjoy in the future. So, the game can decide when to introduce new quests and impasses, or new visual environments, keeping players engaged with new enthralling content. (3)

Procedural Content Generation (PCG)

These models could easily select which content to provide the player from among an arbitrary collection of levels, characters and quests created by the game's designers. But this form of approach becomes exponentially more potent if the game can automatically create new content as well – and that's where Procedural Content Generation or PCG for short, comes in.

PCG is various techniques for automatically generating new content. It's been part of countless games since the early '80s, but its more prominent use is generating levels in roguelike games such as *Diablo* or *Hades*. One remarkably high-profile use of PCG is in the space exploration game *No Man's Sky*. The studio, Hello Games, used PCG to generate literally everything. From distinctive plants and animals to entire worlds. There are more planets than anyone could visit in their lifetime, and every single one of them, has a unique vegetation and wild life.



Figure 2 No Man's Sky

The PCG algorithms that make these games possible depend greatly on randomness, so it is tough to control their output, which of course, is not an issue if it is a design feature — like in *No Man's Sky*. But these limitations mean there are restrictions to what you can do with PCG, right now. However, a series of AI innovations has brought us new PCG methods that could bring revolutionary possibilities for automatically generating content.

Merging PCG with models of what players do and what they enjoy, we could automatically generate totally new and tailored content for players. This entails that not only will players have more personalized experiences, but also will be continuously changing and never running out of new content.

Regardless of some developers' concerns about the technology, AI won't replace human game designers, at least not yet. As an alternative, it'll boost their productivity. A designer can draw up the general shape of a level and place critical features, with the system automatically filling in the blanks. Or they will then be able to adjust the generated level using high-level instructions like 'make it easier for players with slow reflexes' or 'make it less symmetric'. (3)

Game Testing

In the last decade, massive research attempts have gone into creating AI players that can play games. From board games like *Chess* or *GO* and classic Atari games to FPSs like *DOOM* and *Counter-Strike*, we've seen some awe-inspiring work on building or training well-playing agents. Instead of being designed to assist game development, this work, was often carried out by enormous and well-funded teams at major tech companies, primarily serves as a tool to develop and test new AI techniques. Yet, these game-playing agents have a wide range of uses in development. One of the most encouraging is game testing.

The scope and scale of modern games, blended with an arrangement of gaming platforms they must be optimized for, means that testing has become an even more vital stage of development. Games that are shipped packed with bugs and glitches are condemned by fans, as demonstrated by CD Projekt Red's gross miscalculation with the highly awaited yet vilified *Cyberpunk 2077* — the fallout from the launch even led the corporation's valuation to drop by \$1 billion.



Figure 3 *Cyberpunk 2077*

Testing is a very intense procedure and consumes a considerable amount of most game development budgets, which is perhaps why so many get it wrong. Methods that can automate game testing, partially or completely, can rewrite the development process entirely.

These kinds of agents can also help game balancing. By testing a game with many different AI-driven agents, we can see how it would be played by different types of players and adjust the level's boundaries appropriately.

For illustration, we may want a role-playing game to be only somewhat more difficult for players that rush by it than for perfectionists who meticulously scrub every spot, while also being more challenging for trigger-happy players. We can let an algorithm automatically play it through using different playing styles and gradually adjust the parameters of the game to achieve the right balance.

The advantages of this approach only become greater in multiplayer games. The challenge of balancing a Battle Royale game with a hundred people on the same map are immense. Starting with that, you need a hundred committed humans who understand the game every time you want to test a design change. (3)

Improving NPCs

When it comes to computer-controlled players, we're starting to see growth. There have been some unique examples of NPC design in recent years, like *The Last of Us*, *Bioshock Infinite* or *Halo 2*, where developers really pushed the limits of game design. But bots really haven't evolved that much, merely because many of the common approaches to NPC design haven't really changed since they were first created. But that can all change with AI.

One way game-playing AI can improve NPCs is by delivering better partners or team members. It's a popular complaint for gamers as to just how useless NPCs that are supposed to help you can be. With better player modelling, we can create AI agents that can predict with some accuracy how the player will act and plan around that to carry out their objectives.

A breakthrough in gaming will be the use of real-time generated natural language by in-game characters. Almost entirely, game characters "speak" in pre-written sentences, and interaction is usually followed by a 'dialogue tree' where players choose from a limited set of dialogue options.



Figure 4 *The Last of Us Part II*

The immense progress in deep learning-based language models suggests a way of eventually getting around this. By giving language models the proper prompts, it is now feasible to generate

written interactive dialogue on topics and styles. Open AI's GPT-3 language generator has been an enormous step forward in this area of research. The system has given us a glance into the future of AI – writing articles, chatting with humans online and powering the game *AI Dungeon*. The text adventure game allows players to type in commands – anything from 'make tea' to 'attack that dragon with a banana' – and the system does its best to deliver a realistic reply and continue the story. (3)

Graphical upgrades

Since games first appeared, the most common benchmark of technical progress has been graphics. While AI is frequently correlated to behavior, deep learning has made significant inroads along various parts of the graphics pipeline.

Scientists at Intel Labs have been using Machine Learning filters to complete graphical output in older games. They used deep learning, trained on real-world images, to make GTA V's graphics photorealistic, with improved shadows, reflections and textures. But this sort of work isn't just stored for scientists. There's an expanding community of modders that are graphically upscaling old games using related methodologies. The results are far from perfect and from being productized, but it's easy to see where this kind of work will lead us in the future. (3)

Scenarios and stories

Another area where AI is used, is to generate stories and scenarios. Most frequently, is used to create an interactive narrative. In this game, players create or influence a thrilling storyline through actions or what they say. The AI programs use text analysis and generate scenarios based on earlier learned storylines. *AI Dungeon 2* is one of the most famous examples of this application. The game develops a state-of-the-art open-source text generation system built by OpenAI and trained on the Choose Your Adventure books. (2)

(4)

1.5.2 Influential and popular projects/games that shook the Gaming Industry with their AI tools

Quake III: Arena

The PC game industry was flooded with first-person shooters in the late 1990s. DOOM, released by id Software in 1993, popularized the genre, inspiring many imitators and advancements such as Hexen and Duke Nukem 3D. However, as 3D graphics cards became more popular, new titles began to carve out thrilling new worlds and engaging new characters - primarily for you to shoot. Epic Games' Unreal in 1998 helped to provide rich 3D environments for users to explore. Meanwhile, Unreal Tournament from 1999 was part of a movement that redefined the norm for

online multiplayer, which is today widespread. However, it was the followup to DOOM by id Software that really propelled AI advancement.

The Quake franchise began in 1996, as an attempt to carve out new ground following the popularity of DOOM, with players traveling between universes battling off huge hordes. But it wasn't until 1999's Quake III Arena, a multiplayer-focused edition in the series that competed directly with Unreal Tournament, that we became intrigued. While the goal is for users to hop online and battle opponents, you can also play against a variety of simple AI bots, but the true problem is ensuring they know how to maneuver about the combat field swiftly and effectively. To do this, Quake employs a 'area awareness system' (AAS), which computes regions they can traverse and the best way to move across them using both pre-baked knowledge at design time and what's happening right now. It basically serves as a filter through which a bot may decide where to go and how to get there. This was a significant advancement at the time, since most AI bots relied on basic waypoint graphs, which lacked the same level of detail and seldom responded to changes in the environment. (4)

Project Paidia: a Microsoft Research and studio Ninja Theory Collaboration

Project Paidia is a research project in close partnership between Microsoft Research Cambridge and Ninja Theory. Its emphasis is to drive state-of-the-art analysis in reinforcement learning to allow novel applications in modern video games. Particularly: agents that learn to cooperate with human players.

In comparison with traditional approaches to shaping the behavior of bots, non-player characters, or other in-game characters, reinforcement learning does not expect a game developer to predict a broad range of possible game situations, map out and code all required behaviors. As an alternative, with reinforcement learning, game developers control a reward indicator which the game character then learns to modify while reacting fluidly to all aspects of a game's dynamics. The result is a nuanced situation and player-aware emergent behavior that would be demanding or exorbitant to achieve using conventional Game AI.



Figure 5 Project Paidia demo – Learning to collaborate in Bleeding Edge, footage of trained Project Paidia agents. Not representative of final game gameplay or visuals.

Project Paidia aims to learn a very challenging type of behavior: collaboration with human players. Since human players are particularly creative and hard to foresee, creating the experience of a

genuine collaboration towards shared goals has long been vague. Alongside talented colleagues at Ninja Theory, the MSR team discovered a perfect test bed for handling this research which is their latest game *Bleeding Edge*. Bleeding Edge is a team-based game including a range of characters that need work together towards scoring points and triumph over their opponents. In their latest demo, the team showcases how reinforcement learning empowers agents to learn to synchronize their actions.

The Quake III AAS system is essentially a forerunner to what we now refer to as a 'navigation mesh.' A feature in current game engines like Unity and Unreal that allows us to compute navigation data for AI characters in game environments. (5)

Half Life

While Unreal and Quake battled for the title of online shooter king, another FPS by a then-unknown firm would go on to change the industry.

In 1998, Valve released Half-Life, a story-driven first-person shooter that chronicled the story of Dr. Gordon Freeman, a theoretical physicist whose research accidentally opens a doorway to another realm. As mayhem breaks out over the Black Mesa research center, players must assist Freeman and his fellow scientists in escaping, all while avoiding not only hostile aliens pouring through the hole, but also military personnel seeking to cover up the disaster and eliminate all survivors.

Half-concept, Life's inspired by 1997's Goldeneye 007 for the Nintendo 64, was to build characters who lived and breathed their lives apart from the player's involvement. To accomplish this, Half-Life characters employ a simple technique known as Finite State Machines (FSM), in which characters are designed to operate in distinct states (e.g. idle, chasing the player, attacking enemies), and logic is provided to allow them to transition between those states based on what's going on around them. So, if a character is idle and sees an adversary, it enters an attacking condition.

This is used to make foes that go into cover or try to put pressure on the player. But also so they may act in other scenarios, such as working with the player, fleeing from what is going on around them, and contributing to the story's intensity and authenticity when catastrophe begins to unravel. (4)

F.E.A.R.

In the early 2000s, the heyday of Half-Life heralded a new age of First-Person Shooter, resulting in the emergence of new brands that would prove to be stalwarts of the games industry, such as Far Cry, Call of Duty, and Halo. But then there was F.E.A.R., or First Encounter Assault Recon. Monolith Productions, well known for the Lord of the Rings spin-off 'Shadow of Mordor,' wanted to attempt something new after the success of their 1960s-inspired espionage shooter 'No One Lives Forever.'

As the title suggests, F.E.A.R. takes on a more ominous tone, with players battling waves of psychically driven soldiers amid darkly lit docklands and office buildings as 'The Point Man.' While having hallucinations and visions of a frightening young girl destroying everything in her path.

While Alma, the frightening girl, is the marketing emphasis of the FEAR games, it's the opposing troops that left a lasting impact. Monolith has abandoned its earlier pure finite state machine method in favor of a combination of FSMs and another AI technology known as automatic planning.

FEAR uses automated planning, which was inspired by techniques developed in the late 1970s. This is an approach in which you sketch out the high-level goals a character would wish to achieve, and then create actions to assist it reach those goals. A planner then ties together sequences of helpful acts to form - wait for it - a plan! This gave rise to the Goal Oriented Action Planner (or GOAP, as they clearly loved their acronyms), which devised a goal for a character (such as rushing into cover, knocking over a table, or laying down suppressing fire) and then translated the plan actions into specific behaviors for the finite state machine to execute.

GOAP had a significant influence on video games in the following years, with titles such as Tomb Raider and Deus Ex using it for their own opponent AI. Meanwhile, the continued acceptance of GOAP has resulted in other variants of AI planning emerging in other games. Modern franchises like Horizon Zero Dawn and Dying Light employ Hierarchical

Task Network (HTN) planning, a similar method that produces macros of desirable activities that may be performed in sequence. F.E.A.R. is still regarded as one of the best shooters of all time, as well as one of the best instances of AI characters in videogames. (4)

Halo 2

For almost 20 years, the Master Chief's merry space escapades have been a mainstay of the entertainment realm, including video games, books, graphic novels, animation, and, most recently, a TV series on Paramount+. While fans of the franchise may disagree on which edition is their favorite, the most essential release in our opinion is 2004's Halo 2.

With a plethora of archetypes ranging from Grunts to Jackals, Elites, and Brutes, as well as the plethora of combat scenarios players may encounter them in, Halo created a challenging design dilemma for opponent AI characters. Whether it's in a forward operating base, a Forerunner temple, or racing across the Halo in ground or airborne vehicles.

The first Halo used a similar but somewhat different approach to Half-Life, then with Halo 2, they expanded it out much further to create what we now generally refer to as 'Behavior Trees.' As the name implies, a character's behavior is structured less like a state machine, with all the state connections creating a web of possible paths that can occur, and more like a tree shape, where everything starts at the root and is based on key decisions or information about the game state, and it will traverse down a path that eventually leads to a specific set of actions. This is extremely beneficial not just for allowing precise action sequences to be planned, but also for swiftly determining which acts are unnecessary for the character. When Master Chief is driving a Warthog jeep or a Scorpion tank towards an adversary, the NPCs know they just need to focus on the 'vehicular combat' section of their behavior tree. Plus, as Bungie dubbed it, 'Stimulus Behaviors': a procedure in which the tree might be disrupted in order to induce a hard change in behavior. A fantastic example of this is when the player takes out the more advanced Elites, causing the Grunts to fear.

Halo would continue to improve its own AI techniques, but the importance of Behaviour Trees cannot be overstated. In the great majority of AAA releases nowadays, they're used to manipulating non-player character AI. It's also the default AI toolkit in game creation tools like Unreal Engine and CryEngine. (4)

Left 4 Dead

Valve is back on our list thanks to then-subsiary Turtle Rock Studios with Left 4 Dead: a 4-player co-operative online multiplayer game that requires players to work together to survive a zombie

apocalypse. The game reaches our ranking due of its revolutionary 'Director' AI. A gaming system that utilizes some of the proven AI principles from previously in this list in an altogether new way.

The Director employs a finite state machine, similar to the hostile AI in Half-Life (and indeed those in Left 4 Dead as well). However, the activities it may perform this time do not control a single character, but rather influence the tempo of the game by creating zombies, choosing targets for them to focus on, and even activating more deadly boss monsters such as the Tank and Witch.

While the concept of modeling player data and behavior is considerably more common nowadays, the technique in 2008 was quite basic. The game calculates a player's stress level based on the amount of damage they take and how near the zombies are when they are slain. It's not a big issue if you pick them off from distance. But if you're getting harmed and crashing through zombies right in front of you, you're going to be a bit stressed.

Left 4 Dead was a novelty at the time, but the concept of a Director has been adopted in many ways since then. Most open-world games, such as Far Cry and Assassin's Creed, now incorporate some kind of Director to help balance the experience and keep players involved. Meanwhile, the co-operative shooter idea pioneered by Left 4 Dead has had a resurgence in recent years, with titles like Warhammer: Vermintide, World War Z, Deep Rock Galactic, GTFO, and Turtle Rock's own reinvention of the concept, Back 4 Blood. (4)

Mastering Complex Control in MOBA Games with Deep Reinforcement Learning

Tencent, one of the largest companies in the video game industry, recently unveiled an AI system that could defeat professional teams in its MOBA game, *Honor of Kings*. In fact, they published a paper that describes how this technology works with the use of a self-improving actor-critic construction.

Honor of Kings is a good example of a real-time strategy game with dense environments and complex goals. Players learn to plan, attack and defend. Other than that, they also need to control skill combos, induce and mislead opponents.



Figure 6 Honor of kings

The company's architecture entails four key modules:

- Reinforcement Learning (RL) Learner
- Artificial Intelligence (AI) Server
- Dispatch Module
- Memory Pool

Researchers opposed their AI against five professional players as well as other players of various background. In public matches, the system win ratio was 99.81% over 2,100 games, and five of the eight AI-controlled heroes achieved a 100%-win rate.

Honor of Kings provides players only with incomplete information, which means that they can only assume the actions their opponents take. As it is indicated, "the endgame, then, isn't merely AI that achieves Honor of Kings superhero performance, but insights that might be used to develop systems able of solving some of society's roughest challenges." To that end, the Tencent scientists plan to make both their architecture and algorithms open source and free for public use. (6)

Promethean AI. A technology designed to help build virtual worlds

Promethean AI is a brand-new technology created by Naughty Dog's past technical art director Andrew Maximov. It is the world's first artificial intelligence that as mentioned by the creators, "works collectively with artists aiding them in the process of building virtual worlds."

Promethean AI was presented shortly after Andrew left the studio back in 2018. He wishes that this technology will bring the attention back on "imagination, diversity and team encouragement."

Promethean AI can help in producing different environments, as artificial intelligence takes on a lot of tedious and non-creative work. Artists can use relative instructions that suggest assets only fit for a given situation. (7)

The Nemesis System of Shadow of Mordor

Making Up Your Own Story

We've all played a major action-adventure game where the opponent delivers a speech, we go into a big boss fight, take them out, and move on. Sure, there are some stakes involved, and the characters have some backstory, but it's not a tale in the sense that we have any personal engagement. The tale tells us that two people dislike each other, but their reasons do not extend to our gaming. It makes no difference what we do in the game. Things will go as planned as long as we reach the next checkpoint in the plot.

This is what distinguished Middle-earth: Shadow of Mordor. In Monolith's 2014 game, your foes discover who you are. They start taunting you and sometimes even following you around. As you are forced to undergo a torturous onslaught, you can discover their vulnerabilities and exploit them, or you might battle to locate a breach in their defenses. Meanwhile, the adversary mocks your incapacity to defeat them.

What made this game stand out is that when characters in games develop an attachment for players, it's frequently set and predetermined: in role-playing games like Mass Effect, you might irritate or romance individuals, but those plot consequences are already baked into the game. Long before you hit the Start button, the creators have thought of every conceivable outcome for the tale.

Shadow of Mordor, on the other hand, does this procedurally. That is, the orcs you encounter are produced at runtime for your enjoyment. This implies that your interactions with them, as well as the stories you tell about them, are unique to your game. While their experiences may be somewhat similar, no two gamers will ever have the same encounter.

Facing Your Enemy

Shadow of Mordor was among the most elevated games to demonstrate the potential of created narrative: having characters interact with the player and other game components to create their own storylines. An Uruk may beat the player in fight and then insult you when you encounter them again after being reborn. Alternatively, they may depart and remind you of your failure while they do so. They can even resurrect from the dead, bearing wounds and injuries from their previous fight with you and wanting vengeance.

All of this took place within the Nemesis system, which allowed each procedurally created character to develop their attributes and personality and eventually establish them as a senior general officer in this horde of orcs. It implies that an enemy who snuck away during an earlier combat may return minutes later as a tougher, more formidable foe, controlling a larger portion of Sauron's army and desperate to annihilate you. (8)

1.5.3 AI methods used in games

Traditionally, rule-based and finite state machines were used to program NPC behavior. Numerous conditionals were programmed during the development process utilizing these methods to provide NPCs with predictable actions. Developers employed fuzzy logic to cut down on development time while adding some level of unpredictability to games. One of the earliest applications of AI in game programming was through the so-called A* pathfinding algorithm, which set the guidelines for NPC behavior and open-world exploration. Scripting, expert systems, and artificial life (A-life) approaches are further methods. Numerous well-known video games used nondeterministic techniques including decision trees, deep neural networks, genetic AI algorithms for NPC control and reinforcement learning techniques. Below, these strategies are examined. (2)

Finite State Machine (FSM) Algorithm

A limited set of hypothetical states may only have one active state at a time in an FSM, which is a model of computing. An FSM changes from one state to another in response to external factors. Its essential elements consist of:

Component	Description
State	One of a finite set of options indicating the current overall condition of an FSM; any given state includes an associated set of actions
Action	What a state does when the FSM queries it
Decision	The logic establishing when a transition takes place
Transition	The process of changing states

Figure 7 FSM components

Let's use the iconic arcade game Pac-Man as an example. The NPCs are animated ghosts who follow the player and finally catch up to them in the first state of the game (the "chase" state). Every time the player consumes a power pellet and gains the ability to devour the ghosts through a power-

up, the ghosts enter the evade state. The ghosts, now blue in hue, continue to elude the player until the power-up expires, at which point they return to the pursuit state and resume their former behaviors and colors.

The two states that a Pac-Man ghost may be in are pursuit and avoid. It goes without saying that we must offer two transitions, one from chase to evade and the other from evade to pursuit:

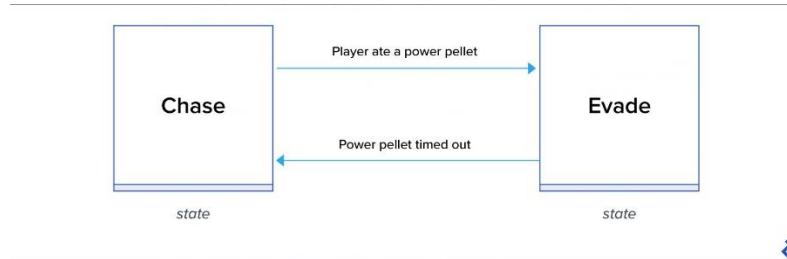


Figure 8 Transitions between Pac-man Ghost states

By design, the finite-state machine probes the present state, which examines the state's choice and action. The option to verify the player's power-up state is shown in the diagram that follows, which illustrates our Pac-Man example. When a power-up starts, NPCs switch from chasing to evading. In the event that a power-up has expired, the NPCs switch from evade to pursuit. Last but not least, if there is no power-up shift, there is no transition. (9)

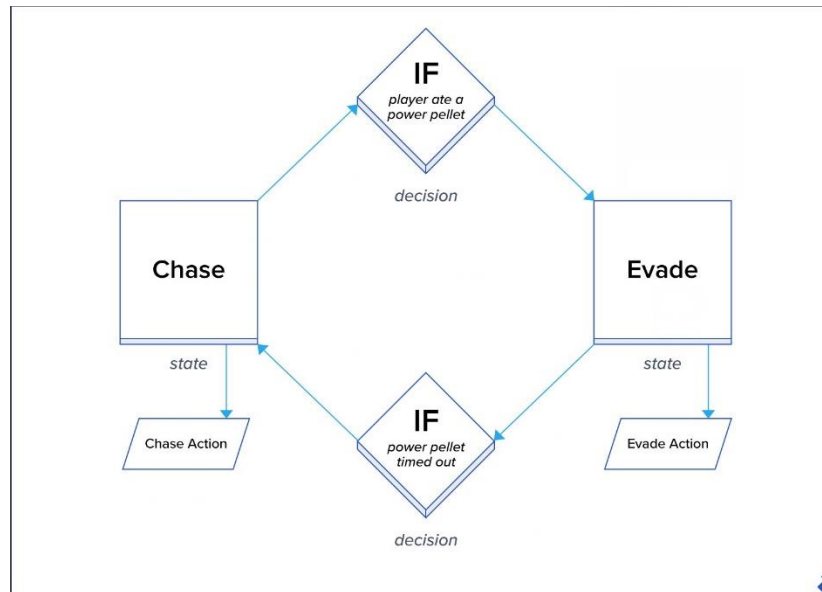


Figure 9 FSM Ghost components

Goal-Oriented Action Planning (GOAP)

Goal-Oriented Action Planning is a simplified STRIPS-like planning architecture created primarily for real-time control of intelligent character action in games. Monolith Productions created it initially for F.E.A.R. This artificial intelligence system also enabled Monolith's *Condemned: Criminal Origins*. Conversations in the AIISC's GOAP Working Group influenced GOAP implementation, as did thoughts from the Synthetic Characters Group's C4 agent framework at the MIT Media Lab and Nils Nilsson's explanation of STRIPS planning in his AI book.

Plan and Goal

A plan is essentially a series of acts that lead to the achievement of a goal, with the actions leading the agent from one state to the other.

Any criteria that an agent wishes to meet is referred to as a goal. Goals in GOAP simply indicate what circumstances must be reached to meet the objective; the GOAP planner determines the procedures necessary to achieve these satisfactory conditions in real time. A goal can define its present importance and when it will be met.

Actions

Every agent is given actions, which are discrete, atomic steps inside a plan that cause an agent to perform something. Playing an animation, playing a sound, changing the state, picking up flowers, and so on are all examples of actions.

Any action is isolated and unaware of the others. Each defined action understands when it is appropriate to perform it and what impacts it will have on the game environment. Each action includes preconditions and consequences characteristics that are used to link actions together to form a valid plan. A precondition is the state that is necessary for an action to perform, while consequences are the changes to the state that occur after the action has been done. For example, if the agent wishes to harvest wheat with a scythe, it needs to first get the instrument and ensure that the prerequisite is satisfied. Otherwise, the agent harvests by hand.

GOAP decides the action to do by calculating the cost of each action. The GOAP planner determines the sequence of activities to utilize by calculating the total cost and picking the sequence with the lowest cost. Actions dictate when to enter and exit a state, as well as what happens in the game world as an outcome of the transition.

The GOAP Planner

An agent creates a plan in real time by conveying a goal to a planner. The GOAP planner examines an action's preconditions and consequences to develop a sequence of activities that will meet the objective. The agent provides the desired aim, as well as the world state and a list of appropriate actions; this process is known as "formulating a plan."

If the planner is successful, a plan is returned for the agent to follow. The plan is carried out by the agent until it is finished, invalidated, or a more relevant aim is discovered. If the objective is reached or another aim becomes more important, the character abandons the present plan and the planner creates a new one.

The planner discovers the solution by constructing a tree. When an action is performed, it is excluded from the list of possible activities.

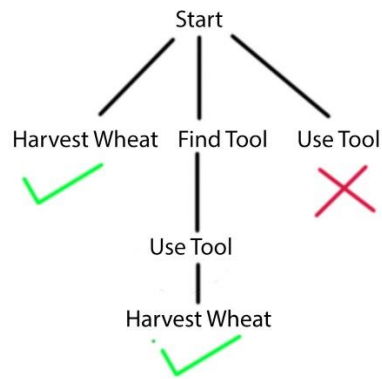


Figure 10 Visualization of planning tree

Furthermore, the planner will go through all potential activities to determine the best answer for the intended goal. Remember that various actions have varying prices, and the planner is constantly seeking for the lowest answer.

The planner aims to perform the Use Tool action on the right, but it confirms as a failed solution due to the prerequisite and effect attributes supplied to all actions.

The Use Tool action cannot be executed since its precondition entails that agent possesses a tool, which it discovered via the locate tool action. However, the agent can collect wheat whether or not they have a tool, but it is considerably cheaper without the tool, thus the agent will prefer getting a tool if one is accessible. (10)

Decision Trees

To accomplish classification and regression, decision trees (DTs) can be trained as supervised learning models. As one of the most fundamental machine learning techniques for game design, they allow for the prediction of a variable's value, through the learning of straightforward decision rules derived from the data characteristics.

Decision trees are rather easy to comprehend, and the outcomes are straightforward to interpret. Techniques for visualizing trees are also highly sophisticated. The created models, sometimes referred to as white box models, may be verified using several statistical tests.

DTs are a tool used in artificial intelligence game design to express decisions and their effects (predictions of actions). DTs are used by the majority of contemporary games, notably those that are narrative-based. Decision trees can help players understand how their decisions will affect the future in one such application. Decision trees, for instance, offer insights about the main character's history and future if specific events were to take place in the famous game: Star Wars Jedi: Fallen Order. (2)

Deep neural networks

Artificial neural networks (NNs) are brain-like structures that are capable of learning different characteristics from training data. NNs are able to simulate extremely complicated real-world and gaming scenarios given a sufficient amount of data. In the creation of gaming agents, NNs improve

on some of the drawbacks of traditional AI methods. Additionally, NNs are self-adaptive and perform well in games with dynamic settings.

NN-based game agents have two methods to learn. They either receive training beforehand (offline) or the learning process can be implemented in real time while playing the game (online). Online learning makes it possible to build interactive game agents that get better over time.

NN-based agents may ensure that the game is difficult even during prolonged engagement by fast adapting to the shifting strategies of human players or other NPCs.

Deep NN (deep learning) has gained popularity as a design option for gaming agents recently. Multiple layers of neural networks are used in deep learning in video games to "gradually" extract features from the input data. Deep NN can perform better when controlling one or more game agents because to its layered approach and greater architectural complexity. The game world itself or NPCs may serve as these agents. (2)

Reinforcement learning

A machine learning technique based on trial and error is reinforcement learning (RL). The model may act out situations while being trained and learn from whether or not they went successfully.

When creating NPCs to make judgments in unpredictable and dynamic contexts, reinforcement learning is beneficial. Since a very long time ago, games have included reinforcement learning. Games are therefore great testing grounds for reinforcement learning systems. At the same time, reinforcement learning is used by some of the top computer gamers (AlphaGo). The fundamental reinforcement learning algorithms, nevertheless, are insufficient for playing high-level games, hence these techniques are frequently combined with other AI techniques like deep learning. (2)

1.5.4 Myths in Video Game AI

Artificial intelligence applications in video games have some restrictions, and there are several AI-related fallacies to be aware of (such as those involving AI in action games). By taking these into account, it will be easier for the designers to remain focused on providing gamers with enjoyable, challenging, and engaging experiences.

Myth 1: Intelligence is everything

It's simple for someone to slip into this trap. He could believe that he has achieved success if he has created the greatest, most intelligent AI. However, engaging gameplay does not naturally come from clever AI. Only intelligence sufficient to enable the intended game experience is required for AI. Beyond that, there is no need (even though devising more powerful AI solutions can be a lot of fun.)

Machine learning is one of several complex and expensive tools used to create AI. For usage in a game that can be shipped, they can take many years to design. The value just isn't there if one buy more than he actually needs for the game.

Player impression is frequently more significant than reality. What good is having the world's most intelligent AI if the player doesn't understand that? There are a lot of comments about popular games and often gamers compliment adversaries' wise judgment and give them special abilities when the AI is actually just somewhat sophisticated. This is partially due to enemy motions and barks: if opponents appear to be acting intelligently and sound like they are, then most people will believe them. This does not mean that having poor artificial intelligence is acceptable, but if an action game is entertaining and rich in satisfying sound, animation, and other feedback, then a respectable amount of intelligence is sufficient.

Myth 2: A squad manager and group tactics are a must

Action games, according to one idea, require a meta-AI to handle group tactics and determine the general strategy of the enemy. Although it appears to be a smart and even reasonable strategy, it won't definitely make the game better. You often can't zoom out and view the entire scene in a first-person or third-person action game. You are unable to foresee the overall adversary assault patterns or obtain a tactical picture. There is virtually no way to tell what's going on beyond the line of foes you are battling because of your extremely narrow range of viewpoints.

A ticketing system is significantly simpler and less expensive, and the player is unlikely to notice the difference. Assume the player is up against a swarm of opponents. When an adversary claims the 'movement ticket,' they will begin moving or pushing towards the player. If you've configured the system such that just one of these tickets is available, the other opponents will notice and remain in place to give cover fire. When done correctly, it is a very effective system.

It's all about perception, and animation and voiceovers may help a lot. When players hear foes yelling and gesticulating at each other, it's just as likely to be a coordinated attack as a genuine group AI manager.

Myth 3: AI needs to respond immediately to player tactics

In most action games, the player should be proactive rather than reactive. If an AI is constantly pinning the player back, it increases the degree of tension and eventually diminishes the level of fun. Sure, there are times when switching things around and putting pressure on the player is appropriate, but in general, proactivity is enjoyable, and we want to keep the player in this condition as much as possible.

AI creates a lot of movement and noise, which might be excessively chaotic, taxing, and annoying. You want players to feel in command, yet quick AI replies might make that difficult.

A delayed AI reaction is frequently advantageous. So, it provides players with windows of opportunity so they may take advantage of that fleeting opportunity to strike at the adversary and feel incredibly smart while doing it. The player's cognitive burden is also much reduced because they aren't required to continually reevaluate their circumstance.

Once more, voiceovers and animation help sell this. They not only make hostile responses plausible, but also exploitable. Players have an opportunity to act if foes hesitate after being hit, say, from behind and appear and sound astonished.

In the end, developing the quickest, smartest, and most complete AI is not a prescription for a successful action game. Your primary objective is to build a playable, pleasant game; therefore you need to make sure that your AI aligns with that. There's always the urge to go large, to keep adding features and bolstering your AI, but this frequently isn't a smart use of resources; at a certain point, it won't enhance the gaming experience, so you're better off concentrating on boosting the effect of your AI. Don't lose sight of your objectives, and I hope I don't sound too much like a shoddy life counselor in saying this. (11)

1.6 Machine Learning

Machine Learning is a collection of computer algorithms that can learn from example and improve themselves without being explicitly programmed by a person. It is an artificial intelligence component that integrates data with statistical methods to anticipate an output that may be utilized to produce actionable insights.

The breakthrough is the concept that a machine may learn from data (for example) to provide reliable results. Data mining and Bayesian predictive modeling are closely connected to machine learning. The machine accepts data as input and formulates replies using an algorithm.

Making recommendations is a common machine learning problem. All Netflix suggestions for users who have an account are based on the user's prior viewing history. Unsupervised learning is being used by IT businesses to enhance user experience with personalized recommendations.

Another use of machine learning is to automate operations like fraud detection, predictive maintenance, portfolio optimization, and so forth.

Machine Learning vs. Traditional Programming

Machine learning differs greatly from traditional programming. A programmer in conventional programming codes all the rules in conjunction with an expert in the industry for which software is being produced. Each rule is built on a logical basis, and the computer will do the output that follows the logical assertion. More rules must be created as the system becomes more complicated. Maintaining it can rapidly become untenable.

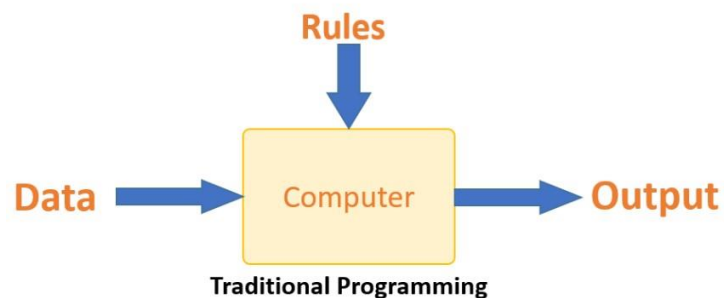


Figure 11 Traditional Programming

This problem is meant to be solved via machine learning. The computer learns how to connect input and output data and then constructs a rule. The programmers do not need to develop new rules every time fresh data is added. To enhance efficacy over time, the algorithms adjust in reaction to new data and experiences.



Figure 12 Machine Learning

1.6.1 How does Machine Learning Work?

Machine learning is the brain in which all learning occurs. The machine learns in the same way that humans do. Experience teaches humans. The more we know, the better we can forecast. By analogy, when we face an unknown circumstance, our chances of success are lower than when we face a known situation. Machines are trained in the same manner and they look at an example to produce an accurate forecast. When given a similar case, the system can predict the outcome. However, the machine, like a person, has difficulty predicting if it is fed a previously unknown case.

The primary goal of machine learning is to learn and infer. First and foremost, the computer learns by discovering patterns. This finding was made possible by data. One critical role of the data scientist is to carefully select the data to offer to the machine. A feature vector is a set of properties used to solve an issue. A feature vector may be thought of as a subset of data that is utilized to solve a problem.

The computer employs sophisticated algorithms to reduce reality and convert this discovery into a model. As a result, the learning step is used to characterize and summarize the data into a model.

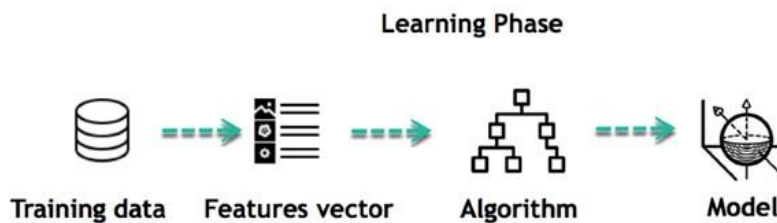


Figure 13 Learning Phase

For instance, the machine is trying to understand the relationship between the wage of an individual and the likelihood to go to a fancy restaurant. It turns out the machine finds a positive relationship between wage and going to a high-end restaurant: This is the model

Inferring

When the model is finished, it may be tested with never-before-seen data to evaluate how strong it is. The new data is turned into a features vector, which is then run through the model to provide a forecast. This is the most wonderful aspect of machine learning. There is no need to retrain the model or alter the rules. You may utilize the previously trained model to infer new data.

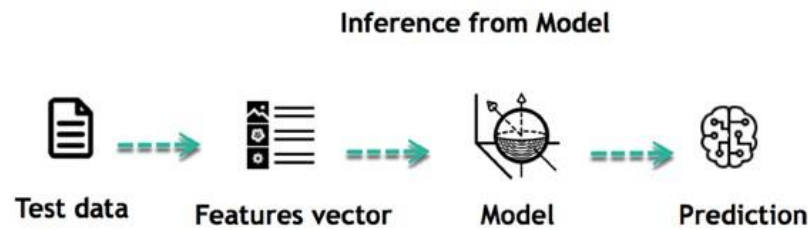


Figure 14 Inference from Model

Machine Learning programs have a simple life cycle that may be stated as follows:

- Create a question
- Gather information
- Make data visual
- Develop an algorithm
- Run the Algorithm
- Gather feedback
- Fine-tune the algorithm
- Repeat steps 4–7 until the results are satisfactory.
- Make a prediction using the model.

Once the algorithm has mastered the ability to make the correct conclusions, it applies that expertise to fresh sets of data.

1.6.2 Machine Learning Algorithms and Where they are Used?

Machine learning is divided into three categories: supervised, unsupervised, and reinforcement learning.

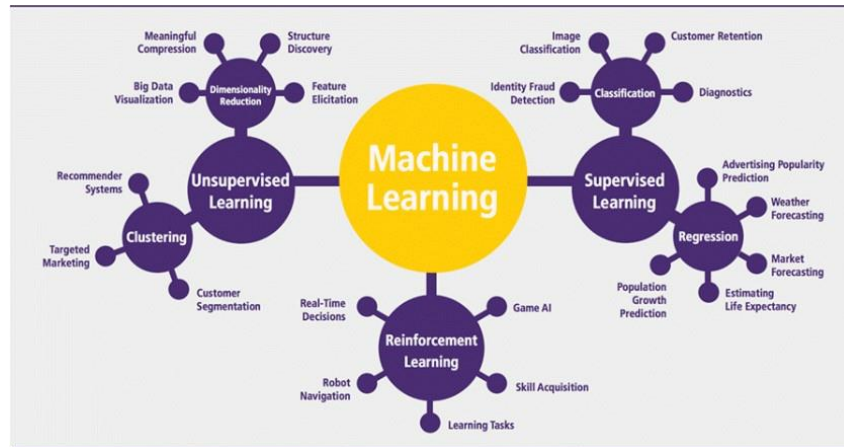


Figure 15 Machine learning Algorithms

Supervised learning

An algorithm learns the link between given inputs and provided outputs by using training data and human feedback. For example, a practitioner might utilize marketing expenses and weather forecasts as input data to forecast can sales.

When the output data is known, supervised learning may be used. New data will be predicted by the program.

There are two types of supervised learning tasks: classification tasks and regression tasks.

Classification

Assume you want to forecast a customer's gender for an ad. You will begin collecting data from your client database on their height, weight, employment, pay, purchase basket, and so on. You know each of your customers' gender; it can only be male or female. The classifier's goal will be to assign a likelihood of being male or female (i.e., the label) based on the information (i.e., features you have collected).

When the model has trained to distinguish whether a person is male or female, you may use new data to generate a forecast. For example, suppose you recently received fresh information from an unknown consumer and want to know whether he or she is male or female. If the classifier predicts male = 70%, the algorithm is certain that this consumer is 70% male and 30% female.

The label might belong to two or more categories. The Machine Learning example above has only two classes, however if a classifier is required to predict an item, it has hundreds of classes (e.g., glass, table, shoes, etc. each object represents a class).

Regression

The job is a regression when the output is a continuous value. For example, a financial analyst may be required to anticipate the value of a stock based on a variety of factors such as equity, prior stock performance, and a macroeconomics index. The algorithm will be taught to estimate stock prices with the least amount of inaccuracy feasible.

Algorithm	Description	Type
Linear regression	Finds a way to correlate each feature to the output to help predict future values.	Regression
Logistic regression	Extension of linear regression that's used for classification tasks. The output variable is binary (e.g., only black or white) rather than continuous (e.g., an infinite list of potential colors)	Classification
Decision tree	Highly interpretable classification or regression model that splits data-feature values into branches at decision nodes (e.g., if a feature is a color, each possible color becomes a new branch) until a final decision output is made	Regression Classification
Naive Bayes	The Bayesian method is a classification method that makes use of the Bayesian theorem. The theorem updates the prior knowledge of an event with the independent probability of each feature that can affect the event.	Regression Classification
Support vector machine	Support Vector Machine, or SVM, is typically used for the classification task. SVM algorithm finds a hyperplane that optimally divided the classes. It is best used with a non-linear solver.	Regression (not very common) Classification
Random forest	The algorithm is built upon a decision tree to improve the accuracy drastically. Random forest generates many times simple decision trees and uses the 'majority vote' method to decide on which label to return. For the classification task, the final prediction will be the one with the most vote; while for the regression task, the average prediction of all the trees is the final prediction.	Regression Classification
AdaBoost	Classification or regression technique that uses a multitude of models to come up with a decision but weighs them based on their accuracy in predicting the outcome	Regression Classification
Gradient-boosting trees	Gradient-boosting trees is a state-of-the-art classification/regression technique. It is focusing on the error committed by the previous trees and tries to correct it.	Regression Classification

Figure 16 Classification - Regression types

Unsupervised learning

A method in unsupervised learning investigates input data without being assigned an explicit output variable (e.g., explores customer demographic data to identify patterns)

When you don't know how to categorize the data and want the algorithm to detect patterns and classify it for you, you may utilize it. (12)

Algorithm Name	Description	Type
K-means clustering	Puts data into some groups (k) that each contains data with similar characteristics (as determined by the model, not in advance by humans)	Clustering
Gaussian mixture model	A generalization of k-means clustering that provides more flexibility in the size and shape of groups (clusters)	Clustering
Hierarchical clustering	Splits clusters along a hierarchical tree to form a classification system. Can be used for Cluster loyalty-card customer	Clustering
Recommender system	Help to define the relevant data for making a recommendation.	Clustering
PCA/T-SNE	Mostly used to decrease the dimensionality of the data. The algorithms reduce the number of features to 3 or 4 vectors with the highest variances.	Dimension Reduction

Figure 17 Clustering - Unsupervised Learning

Reinforcement Learning

Reinforcement Learning is a Machine Learning approach concerned with how software agents should behave in a given environment and it is a deep learning strategy that allows you to maximize a fraction of the total reward.

This neural network learning approach teaches you how to achieve a complicated goal or optimize a given dimension over a number of stages.

1.6.3 Important Components of Deep Reinforcement Learning Method

Typical RL scenario

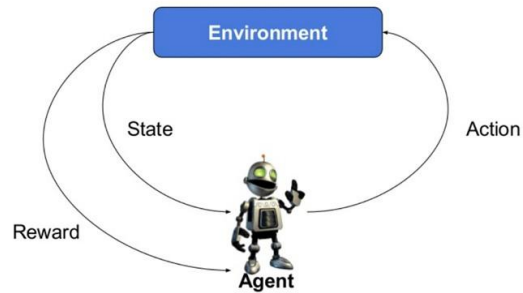


Figure 18 RL scenario

Here are some important terms used in Reinforcement AI:

- Agent: An supposed being that conducts actions in an environment in exchange for a reward.
- Environment (e): A circumstance that an agent must deal with.
- Reward (R): An instant payment made to an agent for completing a specified activity or assignment.
- State (s): A state is the present situation as reported by the environment.
- Policy (π): A technique used by the agent to choose the next action based on the present condition.
- Value (V): This is the predicted long-term return with a discount in comparison to the short-term gain.
- Value Function: It specifies a state's value, which is the entire amount of reward. Beginning with that condition, an agent should be expected.
- Environment model: This simulates the behavior of the environment. It assists you in making conclusions and predicting how the environment will behave.
- Model-based approaches: Model-based methods are used to solve reinforcement learning challenges.
- Q value, also known as action value (Q): Q value and value are quite comparable. The only difference between the two is that the current action accepts an extra argument.

1.6.4 How Reinforcement Learning works?

Let's look at some easy examples to assist us understand the reinforcement learning mechanism.

Consider the following situation for teaching your cat new tricks:

- We can't instruct the cat what to do because she doesn't comprehend English or any other human language. Instead, we use a different approach.
- We simulate a situation, and the cat responds in a variety of ways. If the cat responds in the desired manner, we will reward her with fish.
- Now, whenever the cat is exposed to the same circumstance, it does a similar activity even more excitedly in the hope of receiving greater reward (food).
- It's similar to learning that cats learn "what to do" through pleasant experiences.
- Simultaneously, the cat learns what not to do when confronted with unfavorable events.

Example of Reinforcement Learning

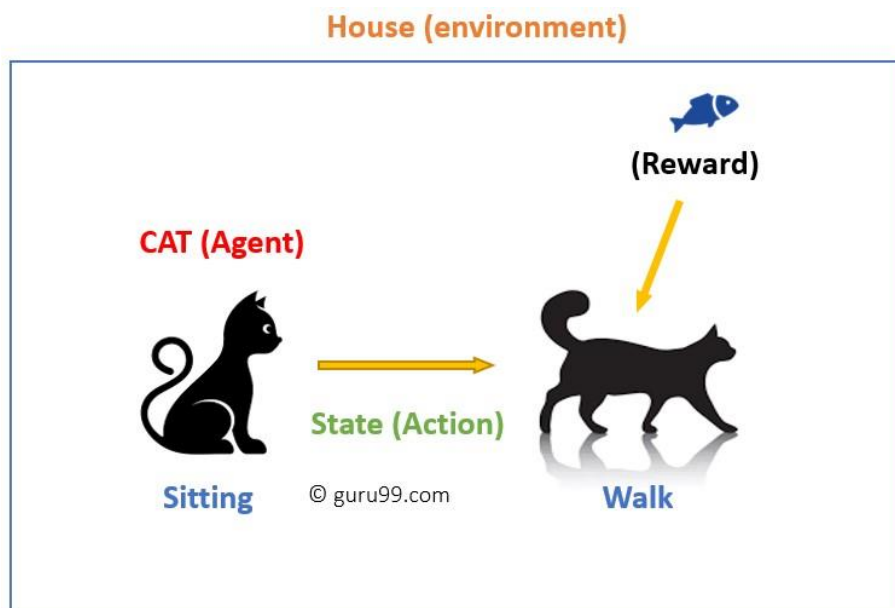


Figure 19 How Reinforcement Learning works

- Your cat is an agent that is exposed to the environment in this scenario. It is your home in this scenario. A state may be your cat sitting while you use a certain term in your cat to walk.
- In response, our agent performs an action transition from one "state" to another "state."
- For example, your cat may go from sitting to walking.
- An agent's reaction is an action, and policy is a technique of choosing an action given a situation with the anticipation of better results.
- They may receive a reward or a punishment as a result of the changeover.

1.6.5 Reinforcement Learning Algorithms

A Reinforcement Learning algorithm may be implemented in three ways.

1. Value-Based:

You should strive to optimize a value function V in a value-based Reinforcement Learning approach (s). In this strategy, the agent anticipates a long-term return of the existing policy states.

2. Policy-based:

In a policy-based RL technique, you strive to devise a policy in which every action taken in each state helps you obtain the most reward in the future.

There are two kinds of policy-based methods:

- Deterministic: The policy produces the same action for each state.
- Stochastic: Every action has a probability, which is defined by the equation below. Policy of Stochasticity:

$$p(a|s) = P[A = a|S = s]$$

3. Model-Based:

You must develop a virtual model for each environment in this Reinforcement Learning approach. The agent learns how to function in that particular setting.

1.6.6 Characteristics of Reinforcement Learning

The following are significant properties of reinforcement learning:

- There is no supervisor, simply a real amount or incentive signal
- Sequential decision making
- Time is critical in Reinforcement issues
- Feedback is always delayed, rather than instantaneous.
- The actions of the agent affect the following data it gets.

1.6.7 Types of Reinforcement Learning

There are two kinds of reinforcement learning methods:

Positive:

It is described as an occurrence that takes place as a result of certain conduct. It improves the power and frequency of the behavior and has a beneficial influence on the agent's activity.

This sort of Reinforcement enables you to maximize performance and sustain change over a longer length of time. However, too much Reinforcement may result in over-optimization of the state, which might have an impact on the outcomes.

Negative:

Adverse Reinforcement is defined as behavior strengthening that happens as a result of a negative circumstance that should have been halted or avoided. It assists you in defining the minimal level of performance. The disadvantage of this strategy is that it just gives enough to fulfill the bare minimum of behavior. (13)

2. Approach

2.1 A.I. in Game Engines

Artificial intelligence is everywhere. It is not uncommon for game makers to use developing technology such as AI and machine learning. However, in recent years, AI has also been utilized to supplement game models in order to develop more intricate games. This section is a list of five game engines that developers may utilize to create AI games.

Amazon Lumberyard or Open 3D Engine (OE3D)

Amazon Lumberyard is an open-source, cross-platform 3D game engine developed by Amazon. This engine, which is based on CryEngine, allows developers to create high-quality games while creating and hosting them on the AWS cloud and attracting players on Twitch. It helps to create high-quality gameplay while decreasing time spent on the hard lifting of designing a game engine, such as server administration.

Some of Lumberyard's key features of AI include:

- Navigation (multi-layer navigation, basic 2D navigation, etc.)
- Individual AI (dynamic tactical points and configurable perception)
- Group and global AI (formations to shift AI characters in some organized way, visual flow graphs of game logic, etc.)
- MMO-capable (support for streaming large maps)
- Simple to use (visual AI decompiler to log signals, behavior change, etc.)



Figure 20 Amazon Lumberyard demo scene

CryEngine

CryEngine, built by German game company Crytek, is one of the most prominent game engines in the gaming business. The CryEngine 3 Sandbox™ allows designers to handle basic AI by using flow graph visual scripting interface, giving them complete control over AI behavior.

This game engine's key AI features include an AI editing system that consists of AI Actions, which allow designers to script AI behaviors without writing new code, AI control objects, which are used to control AI entities and their behaviors in the game world, an AI Debug Recorder, which is a utility designed to document the inputs and decisions AI agents receive/compute in real-time during the course of a play session, and more.



Figure 21 Sniper Ghost Warrior 3

Panda 3D

Panda is an open-source and free engine for creating high-quality, real-time 3D games, infographics, simulations, and other applications. This engine merges the performance of C++ with the simplicity of Python to provide users with a rapid rate of game production. This is a cross-platform engine that contains deployment tools enabling quick application deployment on all supported systems.

Panda3D is now integrated with the PandAI v1.0 AI library, which is a basic AI toolkit that allows you to build artificial intelligence - based behavior in non-playable characters in games. This library includes capabilities for behavior steering and pathfinding.



Figure 22 Panda 3D demo scene

Unreal Engine

Epic Games' Unreal engine is among the most popular gaming engines. The code in this engine is developed in C++, which has a high level of flexibility. The most recent version, Unreal Engine 5, has advanced artificial intelligence technologies.

It contains behavior trees (a mechanism used to establish how an AI should behave or perform) in addition to AI perception, which allows a developer to design an AI character that wanders and attacks foes.



Figure 23 Unreal Engine demo scene

Unity 3D

Another of the most successful game engines is Unity. It is an open-source, cross-platform game engine that can be used to develop 2D, 3D, virtual reality (VR), and augmented reality (AR) games, in addition to simulations or other applications.

The Unity Machine Learning Agents Toolkit (ML-Agents) is a free and open-source Unity plugin that allows games and simulations to be used to train intelligent agents. Through a simple Python API, the agents may be taught utilizing reinforcement learning, imitation learning, neuroevolution, and other methods of machine learning. (14)



Figure 24 Unity3D demo scene

2.2 Unity's M.L. Agents toolkit

As mentioned on the first part of this chapter, we were introduced to the A.I techniques and methods that the most popular game engines use, in summary. In the second part, we'll inspect in more detail Unity's M.L. Agents and examine how teams, companies and indie developers in the gaming industry, use this toolkit for their projects.

2.2.1 Introduction notes about Unity's M.L. Agents

The Unity Machine Learning Agents Toolkit (ML-Agents) is a free and open-source project that allows games and simulations to be used to train intelligent agents. It provides state-of-the-art algorithm implementations (based on PyTorch) to allow game developers and hobbyists to quickly train AI bots for 2D, 3D, and VR/AR games. Engineers may also train Agents using reinforcement learning, imitation learning, neuroevolution, or any other approach utilizing the given easy Python API. These trained agents may be used for a range of tasks, including regulating NPC behaviors (in a variety of situations, including multi-agent and hostile), automating game build testing, and reviewing various game design decisions prior to release. The ML-Agents Toolkit benefits both game developers and AI scientists by providing a common platform for AI developments to be assessed on Unity's rich settings and then made available to the larger research and game developer community.

Features

ML- Agents include several instinctive features. The most interesting of them are listed below:

- ML-Agents provide a variety of environment settings incorporating training situations
- It provides a versatile software development kit that can be integrated into a game or a customized Unity scenario
- The games may be trained using two deep reinforcement learning algorithms: Proximal Policy Optimization (PPO) and Soft Actor-Critic (SAC) (SAC)
- It has built-in support for imitation learning via Behavioral Cloning and Generative Adversarial Imitation Learning
- It has a self-play system for teaching agents in combative situations
- Using environment randomization, train resilient agents
- Agent control that is flexible and allows for on-demand decision making
- Train with several Unity environment instances running at the same time
- Python environment control for Unity
- Wrap Unity learning environments in a gym uniform

How does it work

Developers may use Unity Machine Learning Agents (ML-Agents) to create more intriguing gameplay and a better gaming experience. A developer may use the platform to teach intelligent agents to learn using a blend of deep reinforcement learning and imitation learning.

The steps involved in ML-Agents are:

- 1) First, install the ML-Agents Unity package.

- 2) Next, link the associated Unity project and begin training the agents to learn the proper behavior.
- 3) Once the training is finished, the agent model must be re-entered into the Unity project.

Benefits of ML-Agents

The following are the primary advantages of Unity ML-Agents:

- It is free source and distributed under the Apache 2.0 license. This enables developers to customize and implement ML-Agents to meet their specific requirements.
- The toolkit includes everything required to get started, featuring ready-to-use cutting-edge algorithms, thorough documentation, and example projects.
- It is less difficult to configure the game as an AI training environment. Intelligent characters can be readily created without a lot of code.
- This platform comes with a variety of beginning settings. This implies that if a project is a 2D game, a big game space, or a continuous management system, there are a variety of beginning settings accessible to assist in getting started.
- Unity ML-Agents supports cross-platform inference. It implies that developers may use the Unity Inference Engine to deploy their ML-Agents models on any platform supported by Unity, such as PC, mobile, or console.
- It contains extensible agent training, access to C#, a transmission control protocol, and a low-level Python API, giving you the freedom to experiment with alternative algorithms and approaches for training agents, enriching your advanced AI and research use cases. (15)

2.2.2 Projects where M.L. Agents toolkit was used

How Eidos-Montréal created Grid Sensors to improve observations for training agents

Eidos-Montréal was founded in 2007 with the intention of revitalizing old Eidos Interactive brands like Deus Ex: Human Revolution. Square Enix purchased the studio in 2009. By 2020, Eidos-Montréal employs around 500 people working on both gaming and research projects. The studio recently announced the launch of Eidos-Sherbrooke, a regional branch that incorporates Eidos Labs, a cutting-edge team committed to advancing Eidos-technical Montréal's innovation.

Several initiatives at Eidos Labs make use of machine learning. The Automated Game Evaluating project addresses the issue of testing the functioning of large triple A games by simulating player behavior with reinforcement learning agents (RL). In this section, we'll go through how the Eidos Labs team designed the Grid Sensor inside the Unity Machine Learning Agents Toolkit (ML-Agents) to best reflect the game for machine learning, resulting in faster training times and, eventually, less costly models.

Automated Game Testing using reinforcement learning

Many advances have been made using reinforcement learning to develop AI systems capable of playing games at normal to extraordinary levels, like StarCraft, Dota 2, and the Atari 2600 suite. One of the most difficult difficulties for game producers is the number of compute time and money required to train these models. The sometimes spontaneous nature of triple-A game development exacerbates this difficulty. Developers frequently add features or alter graphics and animations, swiftly and radically changing a game in its early stages when testing is most needed. One of the fundamental aims of the Automated Game Testing project at Eidos Labs is to achieve a happy medium across model realism and training speed while staying independent of ever-changing game aesthetics.

Eidos Labs collaborated with Matsuko, a deep tech startup focused on AI and 3D, on the Automated Game Testing project, which resulted in the creation of the Grid Sensor, to promote innovation and improvement. For prototyping, the team also used the Unity ML-Agents Toolkit. The following people made up the core team:

- Jaden Travnik (Eidos Labs)
- Romain Trachel (Eidos Labs)
- Alexandre Peyrot (Eidos Labs)
- Charles Pearson (Matsuko)
- Martin Čertický (Matsuko)
- Erik Gajdos (Matsuko)

Defining observations in RL and the Unity ML-Agents Toolkit

A important idea in reinforcement learning is an agent's ability to perceive its surroundings (RL). After an agent acts in accordance with its policy (which governs how well the agent must behave at any given moment), the agent examines the various variables of the environment and assesses whether the reward has increased or decreased. Although rewards and actions are powerful levers for refining an RL policy, representation of observations can also have a substantial impact on the agent's behavior, particularly since game engines can take more diverse approaches to observations than the actual world can.

Sensors are the primary mechanism in ML-Agents for representing observations for training and performing models. ML-Agents provides two types of sensors in addition to a generic interface to create observations for the agent which are utilized to train an RL model. The first is the usage of raycasts, which allow agents to watch and collect data about a GameObject from a distance. The developer has the ability to provide not just the distance between the agent and the GameObject, but also a reference, which allows the agent to seek up additional sets of data such as the GameObject's health or whether it is a foe or friend.

The second type makes use of the pixels from an agent's camera. Depending on how the camera is mounted and positioned, the camera presents the agent with either a grayscale or an RGB view of the gaming area. Convolutional neural networks (CNNs) may be trained using these pictures to grasp the nonlinear connection between pixels. The camera sensor, by utilizing CNNs, allows the agent to include high-dimensional pictures as inputs into its observation stream.

Despite the fact that both types of sensors may be utilized in a number of contexts, raycasts and pixel-based cameras have limits when it comes to teaching agents to play games:

- GameObjects can stay concealed from the agent's line of sight. If observing these items is critical to training an agent, this constraint must be accounted for by the agent's network capacity, which is often increased memory.
- Each raycast is autonomous, with no fundamental spatial information shared between them.

- Raycasts are often restricted in length since the agent does not need to know about things on the other side of a scene. This means that an agent may not notice items that are in the path of these beams. The less probable a thing is to be spotted, the smaller it is. In general, fewer raycasts are performed for computational efficiency.
- Using the camera necessitates the rendering of the Environment. When compared to other options, such as operating headlessly only with discrete observations, this can dramatically reduce the number of engine cycles each minute.
- Rendering a Unity Scene headless on a remote server necessitates the use of Xvfb, which might be extremely slow, particularly with higher quality cameras.
- If the texturing of the game's GameObjects are altered, the agent must be retrained if it relies on camera-based observations.
- The camera's RGB offers the agent with a total of three channels. This constraint limits the capability to capture additional object-specific info, like depth.

Using RL agents for Automated Game Testing

The Eidos Labs team aimed to effectively train RL agents in a 3D environment from the standpoint of the player. One of the project's guiding concepts is to guarantee that testing is free of the graphics and attributes of the GameObjects. Most games, especially huge projects, are always evolving. Even minor changes to a GameObject's vertical location, size, color, or material need a developer adjusting the characteristics of the sensors or observations. As a result, testing that is independent of these parameters is critical to ensuring that RL agents can be a viable option in Automated Game Testing.

The team experimented with several raycast implementations in order to keep the observations independent of the graphics and attributes of the GameObjects. However, if the item was too small or too distant, the raycast was unlikely to reach it, and so the object wasn't really noticed during training. This would result in undesirable agent behavior. The team also experimented with several camera systems, however this needed a significant amount of rendering cost and retraining to be strong enough to accept changes in the aesthetics.

Because raycasts and observations have limits, the Eidos Labs team envisioned a new type of sensor.

Grid Sensors

The researchers discovered that the RL agent did not necessarily have to be limited by the kind of data that a human player perceives. Raycasts and cameras are immediately understandable to humans, but they are not necessarily the greatest approach for a machine learning model to describe observations in a game.

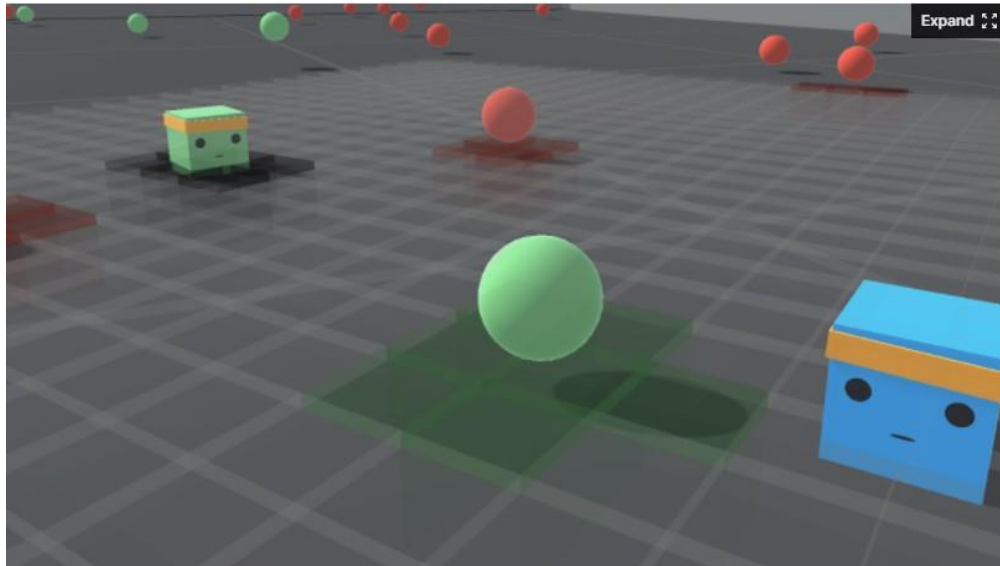


Figure 25 Grid Sensors

The team required a training sensor that could:

- Detect and observe all GameObjects efficiently
- Allow Unity to operate heedlessly for substantially quicker and less expensive observation collecting
- Given the training and rendering limits, it should not be exclusively camera-based
- Make use of pre-existing neural network material and best practices.

Box colliders offer the same methodology as raycasts in terms of data gathering from a GameObject, but they permit the data to be formatted in the same way as pixels on an image do. The MinAtar work, which employed a $10 \times 10 \times N$ binary state representation to express analogs of Atari games to ease the representation challenge for RL agents, also influenced the team. Grid Sensors were eventually developed by Eidos Labs and contributed to ML-Agents.

The Grid Sensor combines the flexibility of raycast data extraction with the scalability of CNNs. The Grid Sensor receives data from GameObjects by checking their physical characteristics and then organizing the information into a "height x width x channel" matrix. This matrix is similar to an orthographic camera picture, but instead of displaying red, green, and blue color values of objects, the "pixels" (or cells) of a Grid Sensor indicate a vector of customizable data of objects relative to the sensor. Another advantage is that the grid may be of lesser resolution, which reduces training time. This matrix may subsequently be put into a CNN for data analysis or training RL agents.



Figure 26 Unity Inspector tab

The arbitrary data acquired from GameObjects inside a grid cell can have features comparable to raycasts. Although one of the initial advantages of employing CNNs on pictures was to prevent feature engineering, the team discovered that this degree of feature engineering is valuable to game designers as a tool of control in practice. It minimizes the representation learning challenge that a CNN faces, as well as the computing resources required to train an agent.

Finally, apart from an in-game camera, the Grid Sensor is entirely dependent on the physics simulation. One of the most significant advantages is that Scene rendering may be deactivated, which significantly improves the amount of engine ticks per minute and, as a result, accelerates training time while still employing high-dimensional data. Furthermore, by detaching rendering and pixels from the machine learning model's input, the visual features of a game, such as textures and lighting, may change without influencing an agent's behavior, allowing for improved generalization of RL-trained models whilst the game is developed. (16)

ML-Agents plays DodgeBall

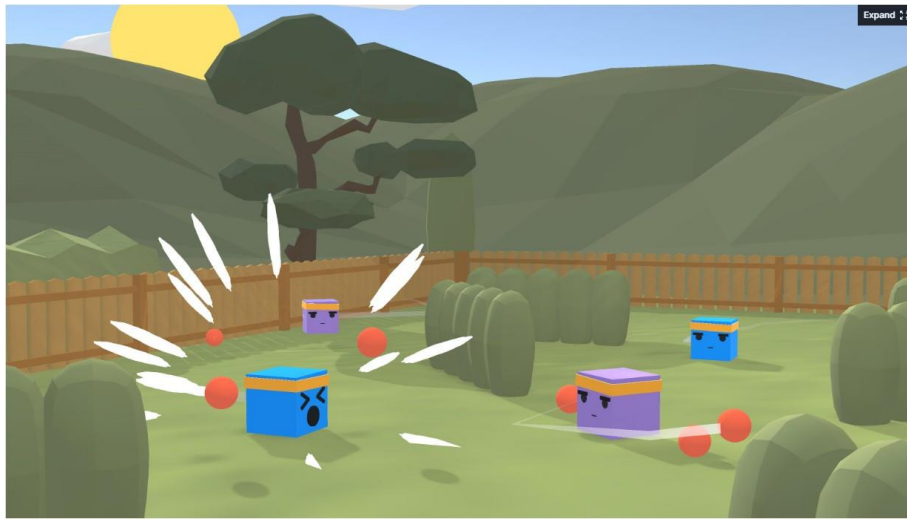


Figure 27 Dodgeball

Unity announced the release of ML-Agents v2.0, which included a tool for writing cooperative behaviors using reinforcement learning. As a result, they collaborated on a new environment that will further illustrate the capabilities of ML-Agents. DodgeBall is a competitive team vs. team shooter-like scenario in which agents engage in Elimination or Capture the Flag matches.

The MA-POCA method, which was recently added to ML-Agents, enables anybody to teach cooperative behaviors for sets of agents. This innovative algorithm combines centrally controlled learning with decentralized implementation. A centralized critic (neural network) analyses the states of all agents in the unit to evaluate how well the agents are performing, whereas the agents are controlled by numerous decentralized actors (one per agent). This enables the agent to take actions based only on what it senses locally while also assessing how excellent its behavior is in the context of the full group. MA-centralized POCA's learning and decentralized execution are depicted in the diagram below.

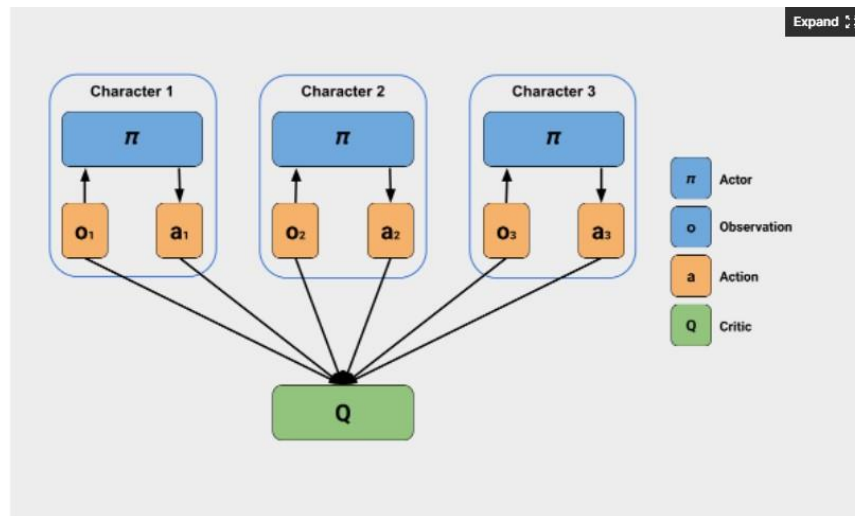


Figure 28 Illustration of centralized learning with decentralized execution; here, three actors select actions based on the observations of a single agent, while a single critic evaluates the behavior of the whole group for training.

The MA-POCA method is unique in that it employs a sort of neural network design known as attention networks, which can handle an infinite amount of inputs. Because the centralized critic may process any number of agents, MA-POCA is especially well enough for cooperative behaviors in games. It enables agents to also be added or withdrawn from a group at any time, similar to how video game characters may be eliminated or spawned in the midst of a team fight. MA-POCA is also built so that agents may make decisions that help the team even if they are detrimental to themselves. This perspective is difficult to create using a hand-coded behavior, but it may be taught based on how valuable an agent's most recent activity was for the group's overall performance. Lastly, many multi-agent reinforcement learning methods assume that all agents pick their next action at the exact same time, but in real-world games with many agents, it is typically preferable for them to make decisions at various times to minimize frame drop. As a result, MA-POCA doesn't really make these assumptions and will continue to function even if the decisions of the agents in a specific group are not in sync. We designed the DodgeBall environment to demonstrate how effectively MA-POCA works in games. It is a fun team against team game with an AI completely trained using ML-Agents.

The DodgeBall environment

The DodgeBall setting is a 3rd person shooter in which players aim to collect as many balls as possible before throwing them at their rivals. There are two game styles available: elimination and capture the flag. In Elimination, each team attempts to remove all players of the opposing team - two hits and they're out. In Capture the Flag, players attempt to grab the opposing team's flag and return it to their base (they can only score when their own flag is still safe at their base). In this mode, being struck by a ball results in the flag being dropped and the player being stunned for 10 seconds before going back to base. Players in both modes can carry a maximum of four balls and rush to avoid approaching balls and get through hedges.

Agents in reinforcement learning examine their surroundings and execute behaviors to maximize a reward. The following describes the observations, actions, and incentives for teaching agents to play DodgeBall.

In DodgeBall, the agents examine their surroundings using the three types of data sources listed below:

- Raycasts: Raycasts allow the agents to see how the environment around them appears. This information is used by agents to locate and capture balls, avoid barriers, and target their foes. To recognize distinct object classes, various kinds of raycasts - shown by multiple colors below - are employed.

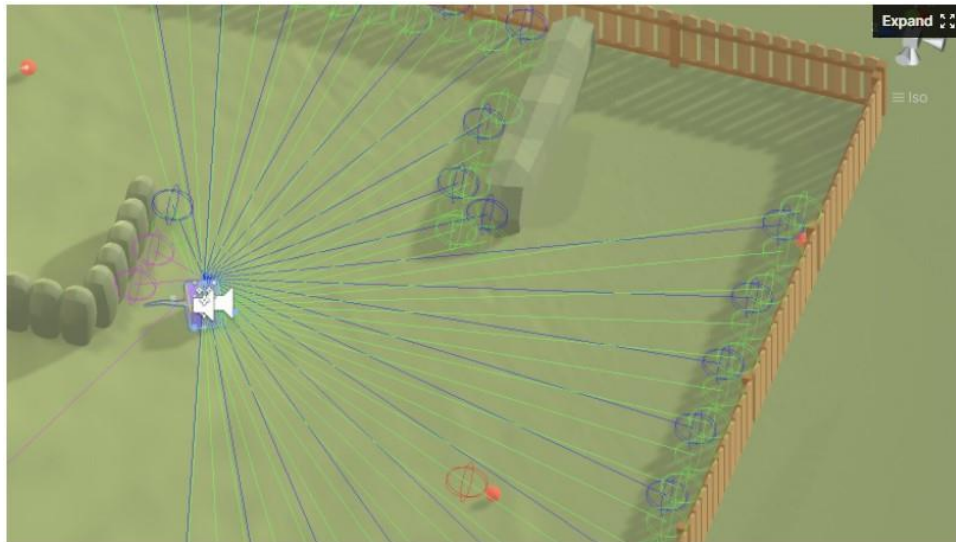


Figure 29 Raycast Sensors 3D

- State data: This comprises the agent's current location, the number of balls it presently has, the amount of hits it can take before being removed, and details about the flags in Capture the Flag game. This information is used by agents to strategize and assess their odds of winning.
- Other agents' status data: This contains the location and healthiness of the agent's colleagues, as well as if any among them is carrying a flag. We utilize a Buffer Sensor for agents to handle a variable amount of observations as this number of agents is not constant (agents can be deleted at any time). The number of observations here corresponds to the quantity of teammates who are still in the game.

In addition, the DodgeBall environment employs hybrid actions, which are a combination of continuous and discrete activities. For movement, the agent has three consecutive actions: move forward, move sideways, and rotate. At the same moment, there are two distinct actions: throwing a ball and sprinting. This action space represents the activities that a human player can do in the Capture the Flag and Elimination scenarios.

Meanwhile, we make an effort to keep the benefits offered to the agents basic. We provide a substantial ultimate prize for winning and losing, as well as a few intermediate prizes for learning how to play the game.

For Elimination:

- Agents are rewarded +0.1 for striking a rival with a ball.
- The team receives a +1 for winning (eliminating all opponents) and a -1 for losing.

- In addition, the victorious team receives a time bonus equal to $1 \frac{\text{(remaining time)}}{\text{(maximum time)}}$.

For Capture the Flag:

- Agents are rewarded +0.02 for striking an opponent with a ball.
- A team receives a +2 for winning (returning the opponent's flag to base) or a -1 for failing.

Although it is tempting to provide agents with several little rewards in order to encourage desired actions, we must avoid too restricting the approach that agents should follow. For example, if we offered a payment for retrieving balls in Elimination, agents could prioritize picking up balls over striking their opponents. By making our incentives as "sparse" as feasible, the agents are allowed to experiment with their own methods in the game, regardless of whether it means extending the training session.

We had to identify what ideal behaviors could look like due to the number of distinct successful techniques that might earn agents these prizes. For example, is it better to retain the balls or scatter them to make them easier to collect later? Is it better to keep together as a squad or spread out in order to discover the adversary faster? The answers to these concerns were influenced by game design decisions we made: If balls were rare, agents would keep them for longer to prevent opponents from obtaining them. Agents would stick with one another as team much as feasible if they were able to know where the adversary was at all times. That said however, when we decided to make modifications to the game, we didn't have to update any AI programming. Simply put, we retrained a new behavior to adapt to a brand-new setting.

Getting agents to cooperate

Teaching a group of agents to collaborate is more difficult than training a single agent to complete a job. The DodgeBallGameController.cs script was written to aid in the management of a group of agents. This script is used to establish and reset the playground (including generating the balls and updating the locations of the agents). It allocates agents to their SimpleMultiAgentGroup and maintains the incentives distributed to each group. For instance, the DodgeBallGameController.cs script controls an agent striking another with a ball in this manner.

```

/// <summary>
/// This method is called when an Agent (hit) got hit by a ball thrown by the
/// Agent thrower.
/// </summary>
/// <param name="hit"> The Agent that was hit</param>
/// <param name="thrower">The Agent that threw the ball</param>
public void PlayerWasHit(DodgeBallAgent hit, DodgeBallAgent thrower)
{
    if (hit.teamID == thrower.teamID)
    {
        // There is no friendly fire
        return;
    }
    var HitAgentGroup = hit.teamID == 1 ? m_Team1AgentGroup : m_Team0AgentGroup;
    var ThrowAgentGroup = hit.teamID == 1 ? m_Team0AgentGroup : m_Team1AgentGroup;
    float hitBonus = 0.1f;

    // If the Agent had the flag, drop it
    dropFlagIfHas(hit, thrower);

    // In the case the hit Agent only has one HP left
    if (hit.HitPointsRemaining == 1)
    {
        // If it was the last Agent alive and the mode is Elimination
        if (HitAgentGroup.GetRegisteredAgents().Count == 1 && GameMode == GameModeType.Elimination)
        {
            // The ThrowAgentGroup Won. Give a reward to this group and a penalty to the other
            ThrowAgentGroup.AddGroupReward(2.0f - m_TimeBonus * (m_ResetTimer / MaxEnvironmentSteps));
            HitAgentGroup.AddGroupReward(-1.0f);
            // The Episode ends for both teams
            ThrowAgentGroup.EndGroupEpisode();
            HitAgentGroup.EndGroupEpisode();
            // Reset the scene for the next game
            ResetScene();
        }

        // The current agent was just killed but there are other agents
        else
        {
            thrower.AddReward(hitBonus);
            if (GameMode == GameModeType.CaptureTheFlag)
            {
                // In CaptureTheFlag mode, the Agent is stuned and then resets
                hit.StunAndReset();
            }
            else if (GameMode == GameModeType.Elimination)
            {
                // In Elimination mode, the agent disappears
                hit.DropAllBalls();
                hit.gameObject.SetActive(false);
            }
        }
    }
    else
    {
        // The Agent still has some HP left
        hit.HitPointsRemaining--;
        thrower.AddReward(hitBonus);
    }
}

```

Figure 30 DodgeBallGameController.cs

Throughout this code, the ball wielder receives a modest reward for striking an opponent; however, the entire group is not awarded until the final opponent is removed.

MA-POCA treats SimpleMultiAgentGroup agents uniquely compared to individual agents. MA-POCA combines their observations to train in a centralized fashion. It also manages group incentives in addition to individual prizes, regardless the number of agents join or leave the team. TensorBoard allows you to track the cumulative rewards that agents get as a group.

Putting it all together

Because Elimination and Capture the Flag are both hostile games, we integrated MA-POCA with self-play to match agents against previous versions of themselves and teach them how to defeat them. As with every self-play run in ML-Agents, we may even track the agents' learning progress by ensuring that the ELO increases. The agents can play just as well as any of us after hundreds of thousands of steps. (17)

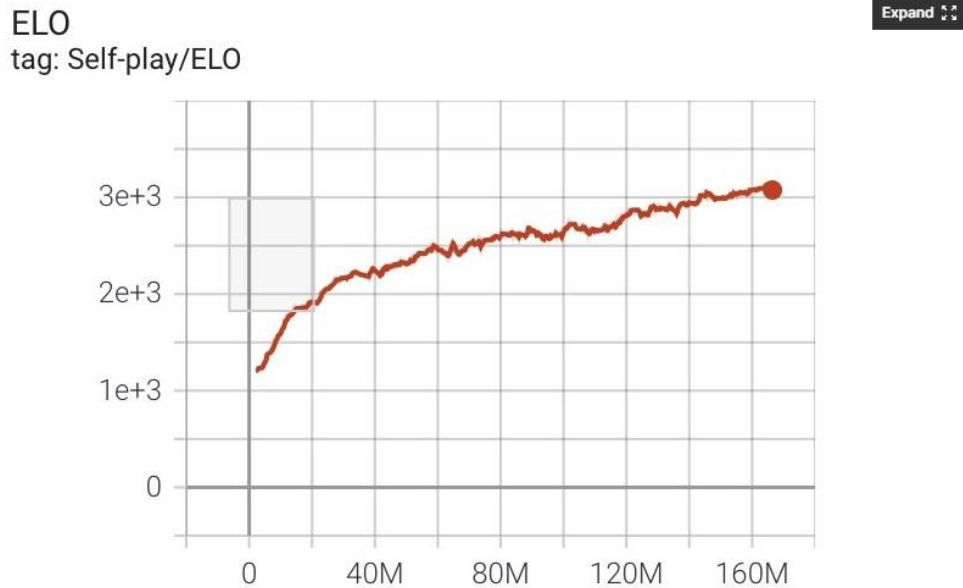


Figure 31 ELO Graph

Robotic soccer using ML-Agents and Sony toio

Inspiration of the project

Ghelia is a firm that specializes in reinforcement learning applications. Hiroaki Kitano, the inventor of Ghelia, founded RobocupSoccer and created the AIBO at Sony. Our team had previously developed an air hockey prototype, but it was not particularly portable due to its numerous various components. When we started talking about making another demo to show consumers what reinforcement learning is, we needed something that would be easier to transport. Because ML-Agents were familiar with a soccer environment, it made reasonable to construct a soccer game using the compact and portable Sony toio robots, which may also lead to viral content.



Figure 32 ML-Agent's example scene, soccer environment

Transferring the ML-Agent model to the real robot

Reinforcement learning must be applied to a real robot in a simulated environment. Fortunately, toio already offers a simulator for Unity named toio SDK for Unity. We were able to utilize it for training immediately after adding the ML-Agents package to it. The toio SDK gave the robot models for Unity, but we still needed to make the ball. We recreated the ball in the simulator using Unity's physics engine and wanted to find a real-world ball that matched the simulation findings. It was discovered that a golf ball gave real-world outcomes that were consistent with the training results.

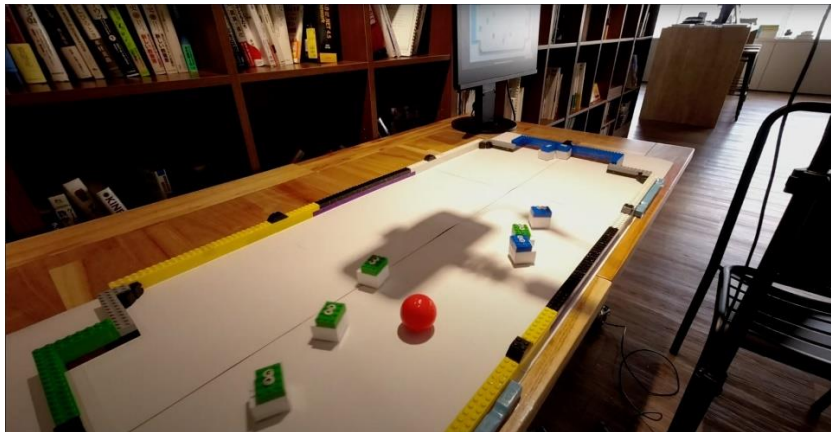


Figure 33 Physical robot environment

Hardware used

We used a golf ball to imitate the soccer ball, but we painted it red to improve identification. We were able to recognize the ball, operate all eight robots (it was a four-on-four soccer game), and do inference using the ML-Agents model using only one iPhone and its camera.



Figure 34 *Hardware for robot environment*

Design of the reward function

Because there were so many own goals at initially, we attempted to create a negative incentive for an own goal. This, however, ended in the goalkeepers failing to defend their goal. When we tried to give a positive incentive for moving the ball, both teams merely went back and forth, not putting the ball in the net and stalling for time. Ultimately, we decided to award one point for scoring on the opponent and deduct one point as being scored on.

Challenges of the project

It was occasionally difficult to determine whether the genuine robots did not perform in accordance with the simulation. For example, the inference didn't always work since we were operating the robot on a slightly inclined floor. At times, the ball bounced differently than in the simulation, and the robots did not react as predicted. The camera's location was also highly delicate, needing millimeter-order accuracy, making it impossible to alter at the event site on a daily basis. We trained for around three days after each significant set of model changes. We needed roughly six training sessions to reach our goals.

Addressing the problem of robot collision

Following a recent goal, the agents in the ML-Agents sample square up in their initial position, but this is not the case with real robots. Some issues, such as avoiding toio collisions, proved difficult to solve with only reinforcement learning. While we attempted to create a reward function for this circumstance at first, we finally solved it heuristically. (18)

Unity AI 2021 Interns projects

Unity's interns worked hard over the summer of 2021 to make meaningful accomplishments to their work at the firm. The parts that follow are a showcase of their work.

Discoverable Behavior Diversity (Kolby Nottingham, Computer Science, University of California Irvine)

Users frequently desire influence on how an agent accomplishes a task, whether in the actual or virtual environment. They aren't simply concerned with work completion; they want it handled right. Obtaining a desired behavior can be tough and frequently requires retraining an agent while modifying environment conditions. In this project, we investigate a way for learning a range of behaviors during training and give a means for the end user to pick the preferred behavior

Maximum Entropy Diverse Exploration (MEDE) is implemented and applied to tasks in the ML-Agents toolset. This strategy promotes an agent to tackle a problem in several manners during training so that the user may choose the learned behavior to utilize after training. Rather of being specified, behaviors are discovered naturally, and MEDE promotes behaviors to be as diverse as possible while still completing the original objective. An agent learns behaviors by training a discriminator to estimate the likelihood that an agent's experience was caused by a certain activity. MEDE then tends to encourage the agent to operate in such a way that the discriminator can quickly determine which behavior the agent is employing.

Only agents that pick actions by a continuous space were examined by the initial MEDE algorithm. MEDE was modified as part of this research to operate with agents that have a discrete or mixture set of actions. MEDE has also been updated with parameter scheduling and extra normalization on the discriminator network to increase training resilience. Originally, MEDE required users to describe a set variety of actions for the agent to learn; however, we expanded MEDE to deal with behaviors having a continuous range of potential values. MEDE with continuous behaviors enables the agent to learn an endless number of behaviors that all transition seamlessly. In practice, smoothly shifting actions are heavily dependent on the present activity.

Simplifying Hyperparameters in ML-Agents (Scott Jordan, Computer Science, University of Massachusetts)

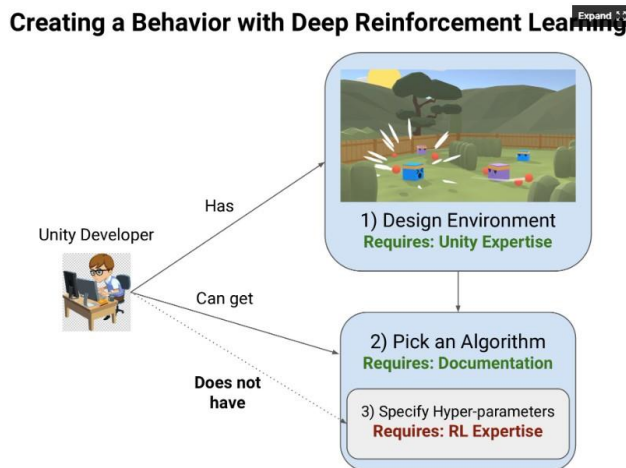


Figure 35 Reinforcement Learning Behavior context

Reinforcement Learning (RL) may learn complicated behaviors and assume the characteristics of Non-Playable Characters (NPCs) or bots in a game with minimum coding when applied successfully. Specifying an algorithm's hyperparameters, like learning rate and network size, is an essential and difficult component of utilizing RL to learn agent behaviors. These settings can be unintuitive and fragile, preventing the learning of desirable behaviors if they are set incorrectly. As a result, trial and error, together with RL knowledge, are frequently required for effective RL implementation. We concentrated on identifying the correlations between various hyperparameters for this project in order to reduce the amount of adjustments necessary to maximize performance.

The fundamental contribution of the project is to alter Proximal Policy Optimization (PPO), a popular and successful RL method, such that the user only needs to tune two of the most regularly modified hyper-parameters instead of five. We decrease the number of hyperparameters whilst also categorizing five parameters as frequency and work. The frequency parameter governs how frequently the agent's behavior may change, whereas the work parameter governs how much the agent's behavior can change each step of the learning process. Furthermore, because the correlations across variables are better known, some hyperparameters may be automatically modified during training to enhance computing efficiency. We created unique environments (Unity games that can be utilized to train RL agents) that can have their properties, like the reward function and the number of simultaneous simulations, altered without having to rebuild the Unity game each time. Users will be able to train a respectful behavior with significantly fewer trials and errors and tweaking of obtuse settings if this work is implemented into the ML-Agents Toolkit.

Computer Vision Sensor for ML-Agents (Grant Bowlds, Computer Science and Mathematics, Vanderbilt University)



Figure 36 Computer Vision Sensors

I was tasked with developing a Computer Vision (CV) sensor for Unity's ML-Agents Toolkit. The ML-Agents Toolkit enables researchers, game developers, and researchers to create and train agents in Unity settings using Reinforcement Learning (RL), especially visual or numerical observations. While numerical observations such as direction and distance to a target help the agent to learn rapidly, they are constrained and do not support a wide range of issue areas. Visual observations are more appropriate when the agent's status and environment are difficult to evaluate. Visual observations, on the other hand, need more training at a slightly slower pace compared to numerical measurements. I developed a sensor that enables creators to utilize the output of their own pre-trained models CV models as observations, resulting in some intuitive bias

but allowing training to be faster and provide better results. Customers seeking to utilize their own CV models for reinforcement learning will benefit from the addition of this sensor to the ML-Agents Toolkit.

The most difficult aspect of my research was developing and implementing a complicated machine learning methodology. Errors may appear at any stage in the process, making debugging tough, but I learned a lot on working consciously and solving issues effectively. The project's next stages involve developing a demonstration and adding it into a version of ML Agents extensions. The variety of Unity tools which I was able to employ was my favorite aspect of this project. I used Computer Vision Perception package to produce synthetic data for my sensor's CV model, the Barracuda package to assist my model, and ML-Agents toolkit and ML-Agents Cloud to develop my sensor and train an agent. (19)

3. Project Specifications

3.1 Concept idea

The project was inspired by a tutorial discovered on "Immersive Limit", a website which is about 3D development and Deep Learning. (20) The purpose here is to illustrate the usage of reinforcement learning and proximal policy optimization (PPO) algorithm, in a small setting using Unity's ML-Agent toolkit.

The project's purpose is to teach the agent (in our example, the big brother) how to navigate about a modest environment, pick up boats, and present them to his younger brother.

When all boats are collected from the scene, the characters are respawned and placed randomly at the environment. This scene resets on every round.

During each round, the older brother learns how to collect the boats one by one, while analyzing the surroundings and using continual penalties and incentives to achieve his aim of giving the boats to his younger brother.

Requirements

The following are the tools and applications that were utilized in the construction of the demo:

- Unity Hub ver. 3.3.0 and editor ver. 2021.3.3f1
- Polygon Kids (low poly asset pack from Synty Studios)
- Mixamo.com (for handling the animations of the characters)
- Microsoft Visual Studio Community 2019 (for coding the scripts)
- Unity ML-Agents (Release_19)
- Python ver. 3.8.10
- TensorBoard (visualization of the training process)

Packages used in Project:

- ML-Agents ver. 2.2.1-exp.1
- ML-Agents Extensions ver. 0.6.1-preview
- TextMeshPro ver. 3.0.6

PC Specifications

The simulation was run on a computer using the following specs:

- CPU: AMD Ryzen 5 2600X
- GPU: EVGA RTX 2080 Super
- RAM: 32GB
- OS: Windows 10 ver. 21H2

4. Installation

4.1 Design

First, a project with the necessary editor version and name was established from the Unity hub. When the editor is opened, assets are imported, as well as the installation of the necessary packages and folder construction.

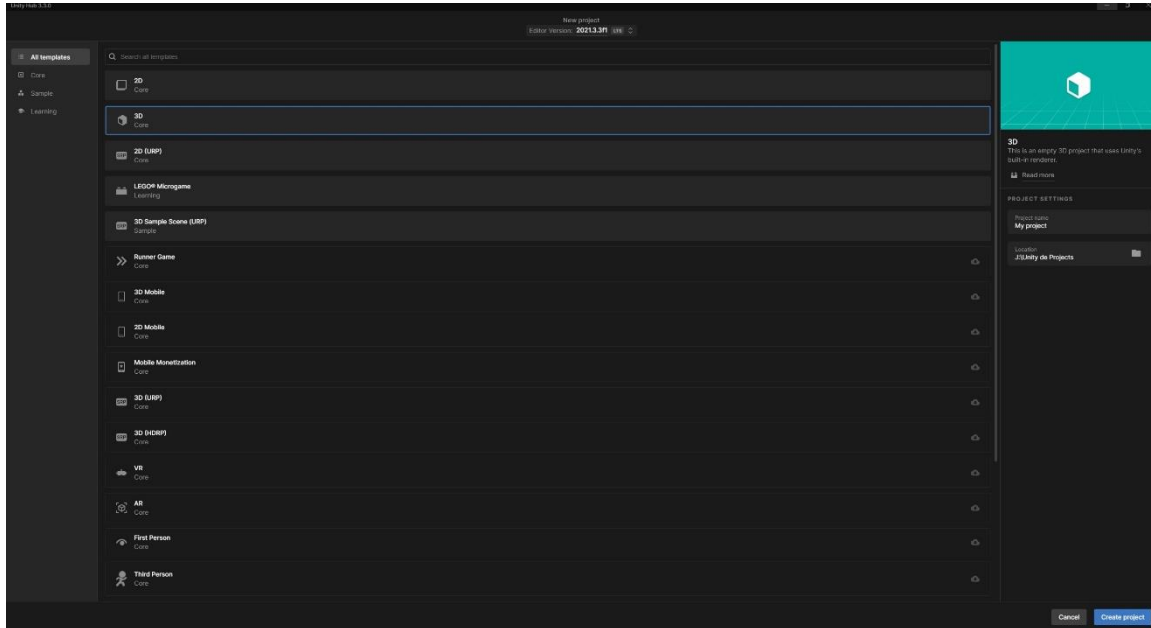


Figure 37 Unity Hub – New Project

As mentioned previously, the packages that need to be added to the project, are:

- ML-Agents 2.2.1-exp.1
- ML-Agents Extensions 0.6.1
- TextMeshPro

The assets for this project were imported from the computer and are presented here. Following that, five folders are established in order to arrange the entire asset tab:

- Animations (containing the animations for each character)
- ML-Agents (containing examples from Release 19 of the cloned GitHub repository)
- NNModels (containing the neural network brains from training)
- Prefabs (containing the prefabs that were made for the simulation)
- Scripts (containing the scripts)

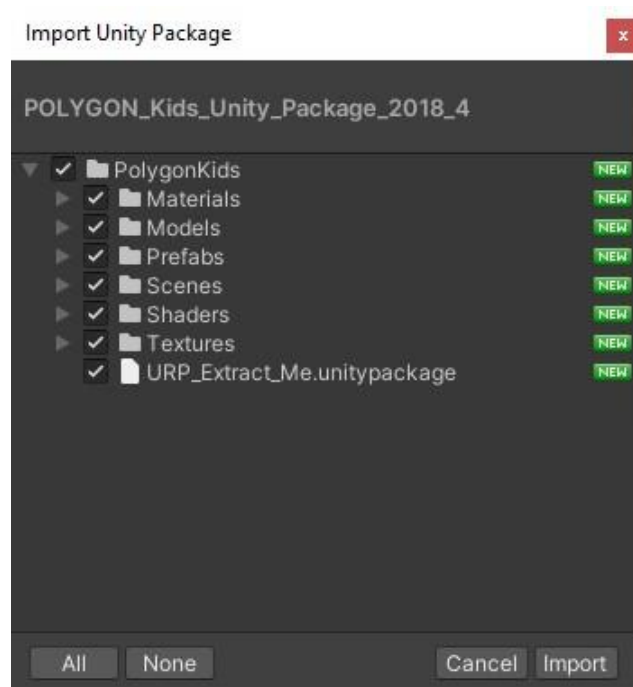


Figure 38 Polygon kids

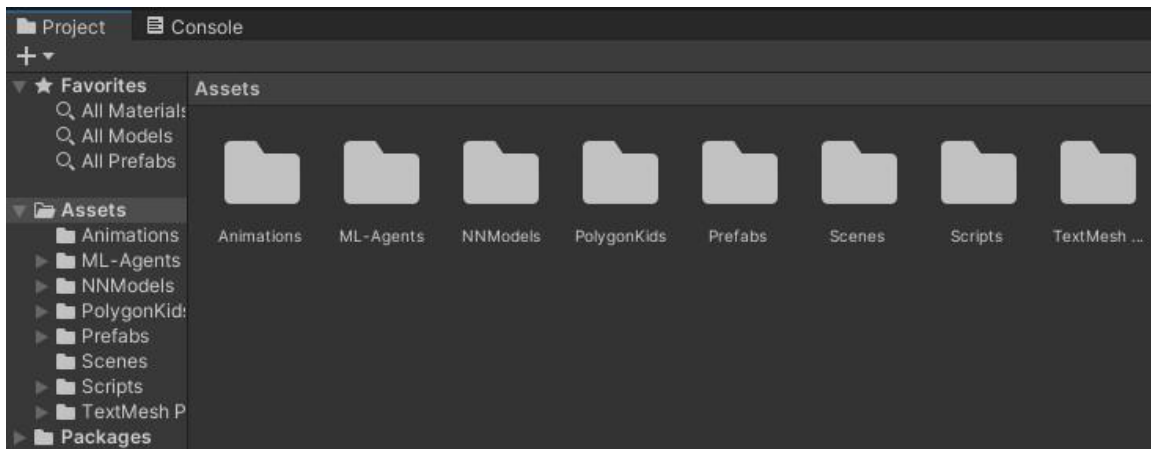


Figure 39 Folders

4.2 Installation

Finally, as a last step, ml agents must be deployed within the project. Specifically, by using a virtual environment to keep all Python project requirements isolated from other projects' dependencies.

This approach has the following advantages:

- Easy project management of the dependencies
- It facilitates the use and testing of several library versions by immediately spinning up a new environment and confirming the code's compliance with the various versions.

The following actions must be taken first to ensure proper installation:

- 1) Navigate to the project's folder directory path, via the command prompt
- 2) Download the `get-pip.py` file using the command `curl https://bootstrap.pypa.io/get-pip.py -o get-pip.py`
- 3) Run the following `python3 get-pip.py`
- 4) Then check pip version using `pip3 -V` (in this case it is pip 22.2.2)

Virtual Environment setup:

- 1) To create a new environment with the name `sample-env` execute `python -m venv python-envs\sample-env`
- 2) Activating this environment requires the execution of a file, which is contained inside the `Scripts` folder of the environment. The command is `python-envs\sample-env\Scripts\activate`
- 3) Then, pip must be upgraded to the latest version using `pip install --upgrade pip`

ML-Agent's packages setup:

Step 1:

The GitHub branch of the corresponding release version must be either downloaded manually or cloned via git.

Version	Release Date	Source	Documentation	Download	Python Package	Unity Package
main (unstable)	--	source	docs	download	--	--
Release 19	January 14, 2022	source	docs	download	0.28.0	2.2.1
Verified Package 1.0.8	May 26, 2021	source	docs	download	0.16.1	1.0.8

Figure 40 GitHub repo, Release 19

Step 2:

The importing of the ml agents and extensions packages into the project was done manually.

Navigate to Package Manager by choosing Windows Package Manager after Unity is open. The packages button in the upper left corner must be changed to Unity Registry (rather than In Project).

The package that has to be installed is in preview, thus enable preview packages in the advanced options by hitting the cogwheel icon in the top right corner.

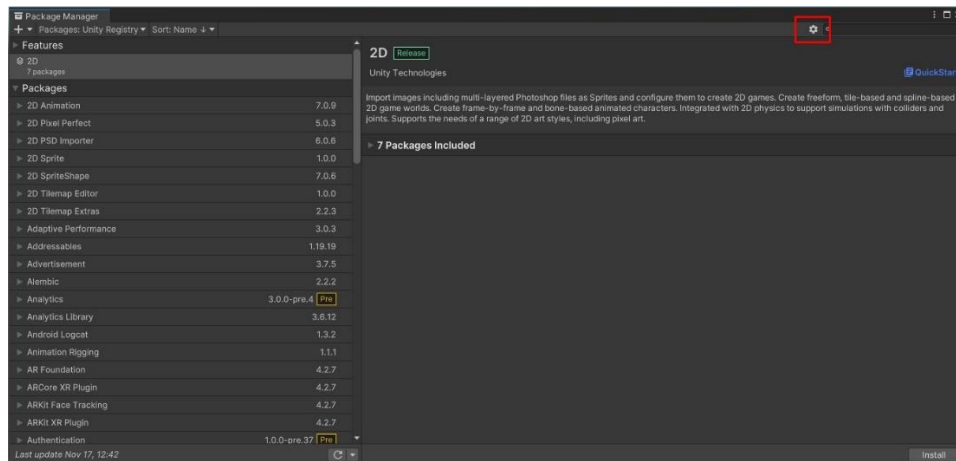


Figure 41 Package Manager Settings

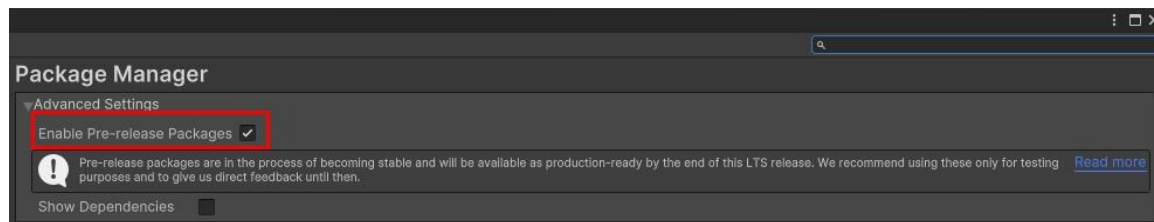


Figure 42 Enable Pre-release Packages

Step 3:

On the top left corner of the package manager window, click the plus icon and choose "add package from disk."

From the cloned repository of the release 19, navigate to that folder and search for com.unity.ml-agents. Select the package.json file and wait for the installation. After completion, it should be shown under Packages: In Project.

Step 4:

The same approach must be done to install the ml agents' extensions package and use all of the toolkit's functionality.

Select "add package from disk" once more, browse to the com.unity.ml-agents.extensions folder, and install the package.json file. Once installed, it should be visible in the Project's packages.

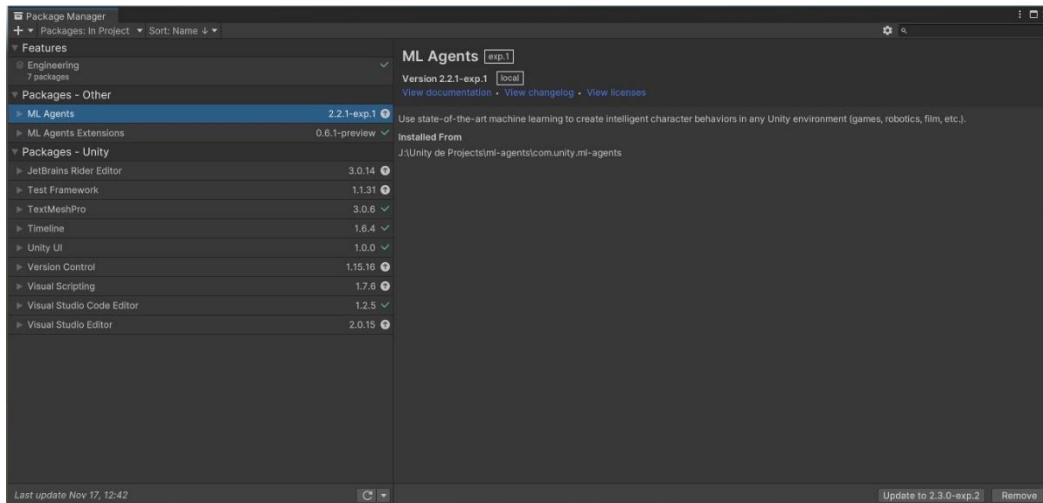


Figure 43 ML-Agents and ML-Agents Extension Packages in the project

Step 5:

Once both packages have been installed, there should be a compiler error at the bottom of the editor that matches to a specific example setting. Deleting it resolves the issue while causing no disruptions to the main project. Simply browse to the examples folder within the project window, select and remove the PushBlockWithInput folder.

Step 6:

To ensure that everything is working properly, choose an environment, launch a scene, and click play to watch the AI play the exact game.

ML-Agent's Python package setup:

Step 1: Installing PyTorch

The PyTorch package must be installed prior to ML-Agents. Activate the virtual environment and perform the following command from the command prompt: `pip3 install torch==1.7.1 -f https://download.pytorch.org/whl/torch_stable.html`

Step 2: Installing ml-agents

For the installation of the mlagents Python package, from the virtual environment run the following command: `python -m pip install mlagents==0.28.0`

If everything is installed correctly, running the command `mlagents-learn --help`, will list all the command line parameters that can be used with `mlagents-learn` command. The next step is to begin training. (21)

4.3 Environment

Synty Studios' "Polygon Kids" low poly asset pack was used to construct the setting. A basic test playground was created for the agent to train on by combining a few prefabs.

The boats drift around the lake and the little brother waits expectantly in the grassy area, watching his bigger brother recover his boats for him.

To boost the aesthetic appeal of the setting, some clouds, trees and seats were added.

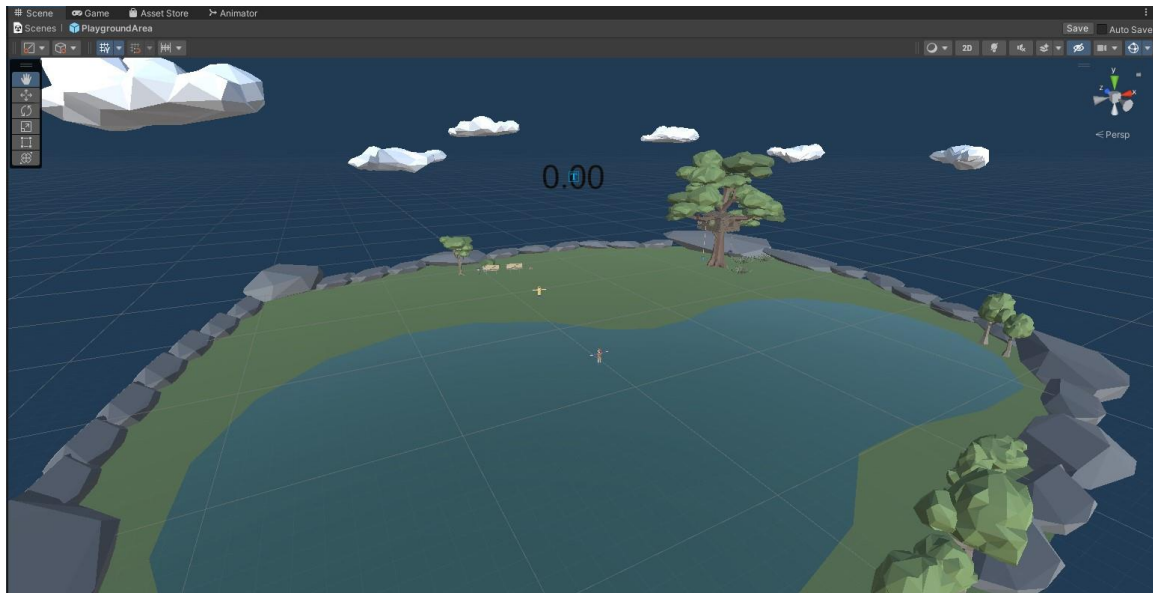


Figure 44 Playground Area prefab

The hole environment is a child object to a parent empty object called "PlaygroundArea" in the hierarchy window. Furthermore, to improve organization, the prefabs were separated into distinct folders.

There isn't much information about the prefabs' components. Almost all of them have the same components. To be more realistic, the ground and perimeter include materials dependent on the kind. The only difference is the mesh collider added to the ground, rocks and the trees to keep the agent within the borders of the training ground.

Lastly, a UI was imported, called "TextMeshPro" which shows the cumulative reward of the agent as a timer.

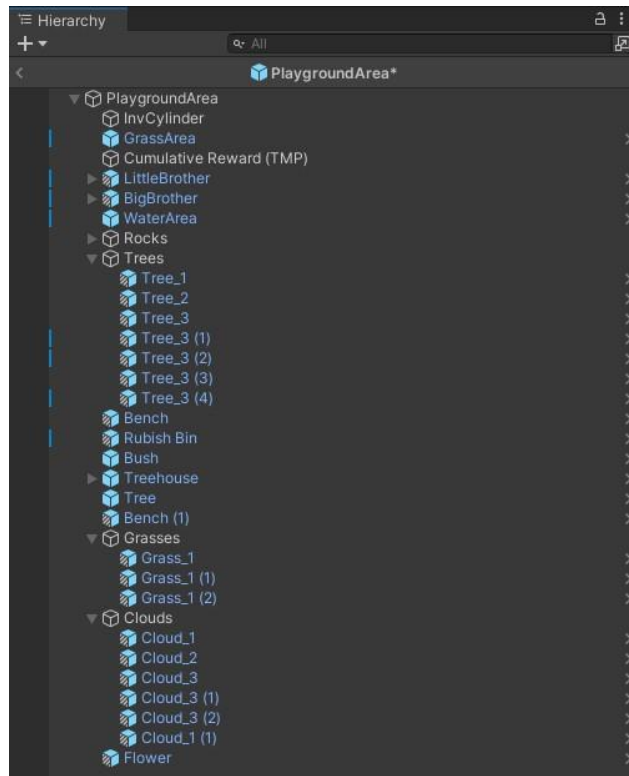


Figure 45 Hierarchy window

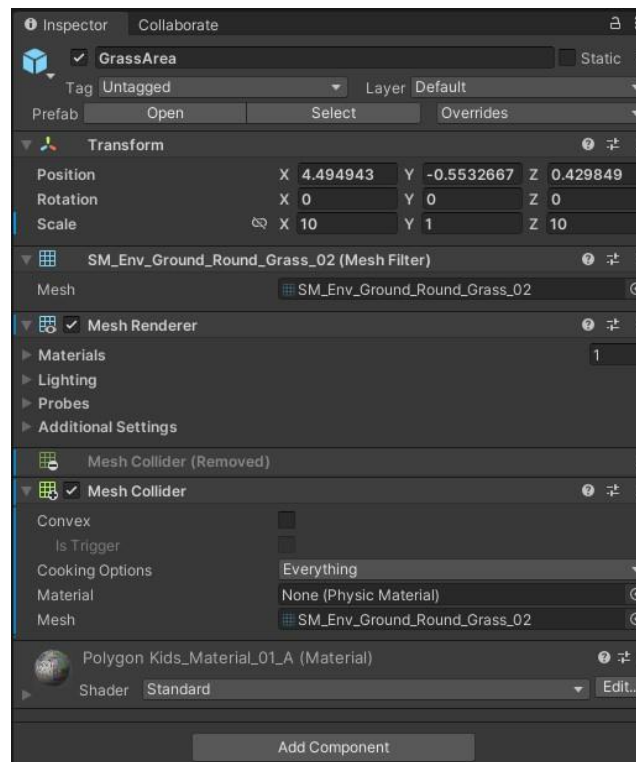


Figure 46 GrassArea inspector

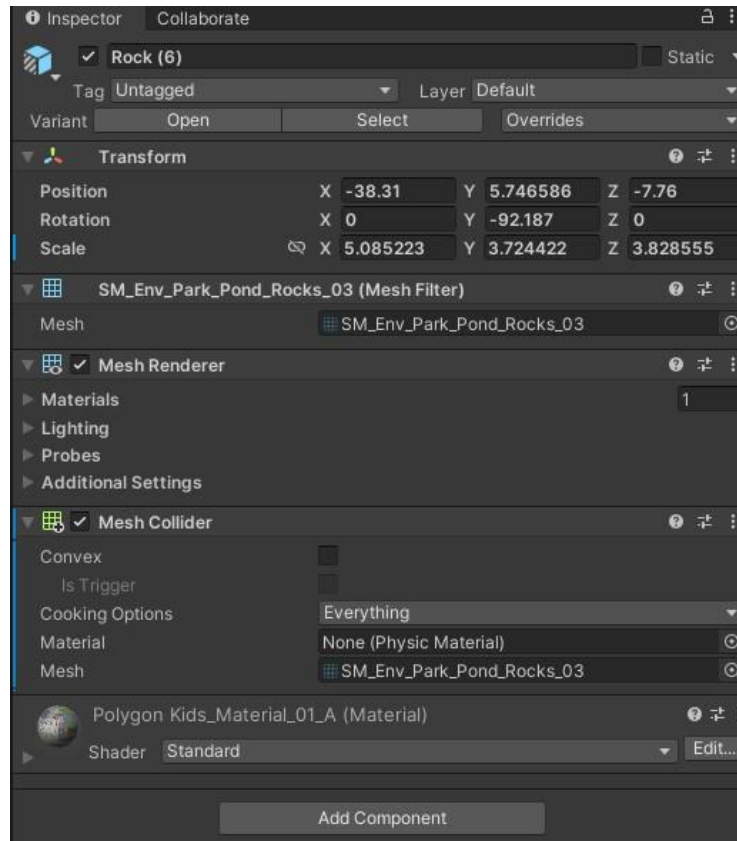


Figure 47 Rock inspector

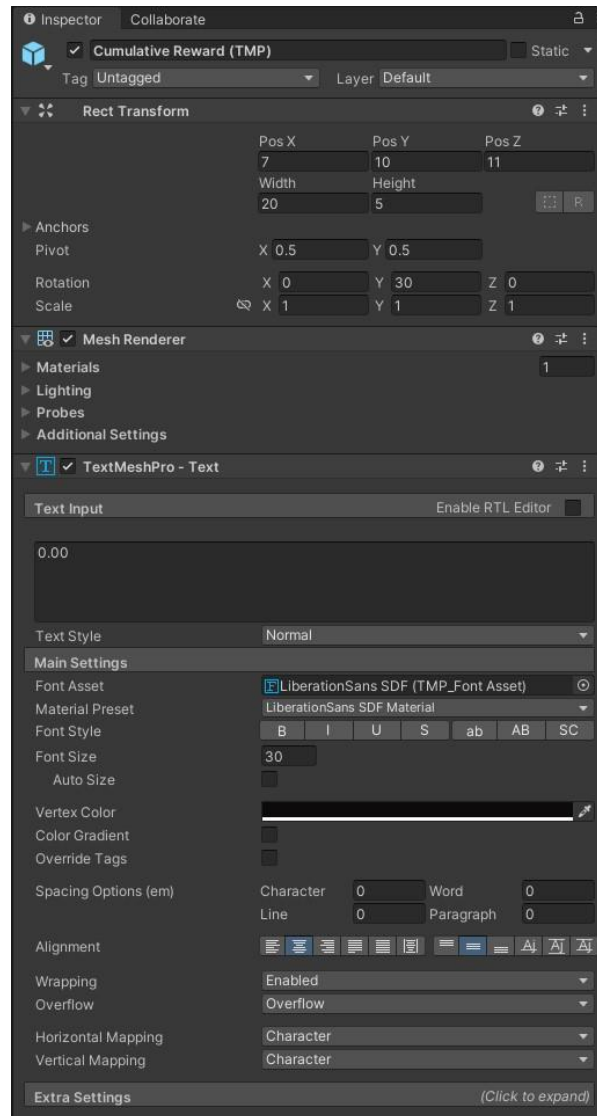


Figure 48 TextMeshPro inspector

The parent object "PlaygroundArea" has an extra script in the inspector window that maintains the training area with the little brother, the older brother and several boats. It is in charge of removing the boats, respawning them and putting the two brothers at random positions. More information on the scripts will be provided in the next chapter.

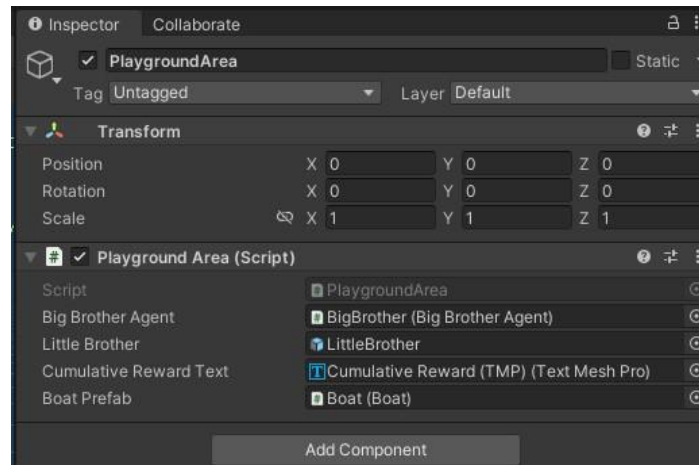


Figure 49 PlaygroundArea Inspector window

4.4 Characters and items

As previously stated, the project comprises of two characters. In particular, two brothers. These two prefabs are given in this section, along with their accompanying inspector tabs.

Big Brother

The machine learning agent that will be taught in the project is big brother. The prefab was chosen from a large selection of characters in the asset pack, and the inspector tab is made up of several distinct components.

The big brother agent is created in a way that observes the environment and take actions using deep learning.

The key points worth mentioning for each component are listed below:

- A rigidbody with locked rotation constrains on the X, Y, Z axis so that it doesn't flip over
- A capsule collider with the direction to Y-axis
- Animator
- Big Brother Agent script
- Behavior parameters script
- Ray Perceptron Sensor 3D script
- Decision Requester script

The above scripts will be showcased in detail in the following chapter.

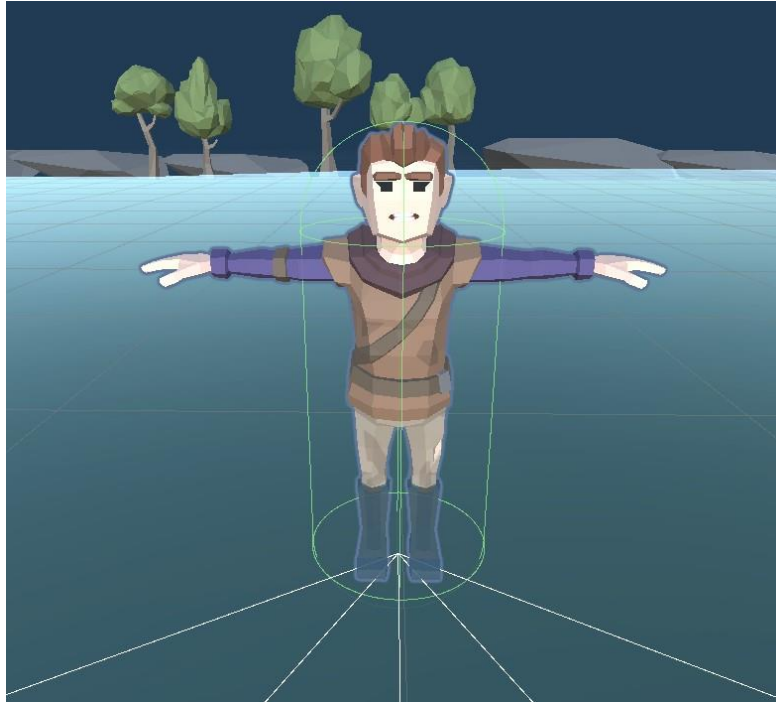


Figure 50 Big Brother prefab

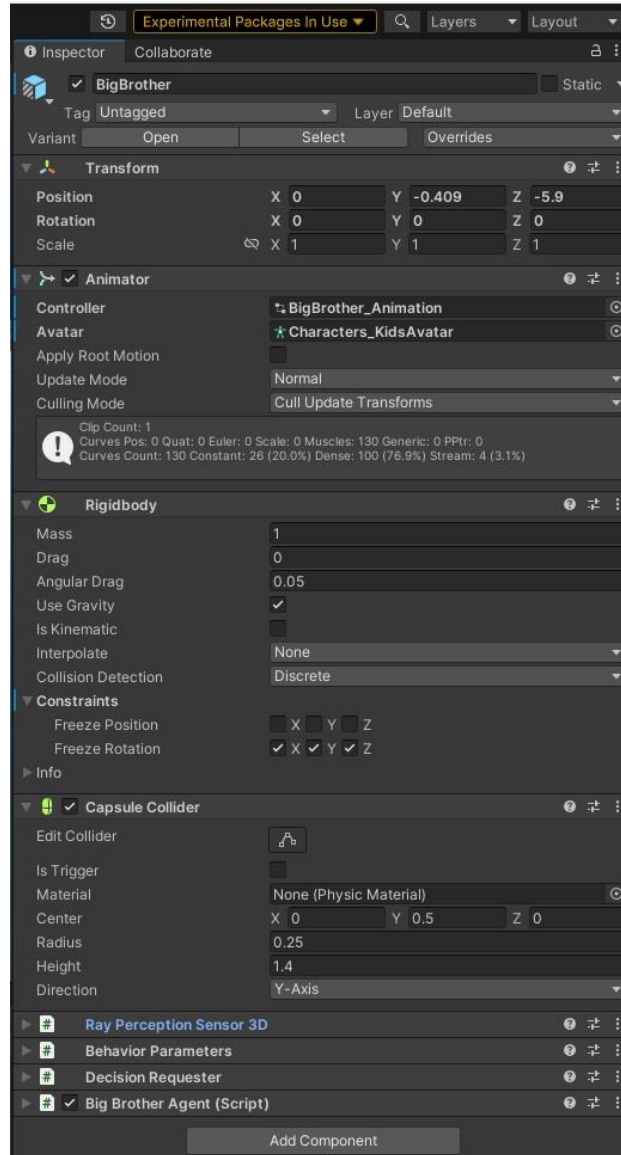


Figure 51 Big Brother Inspector window

Little brother

For the younger brother the inspector tab is simpler with fewer components.

- A Rigidbody constrained to lock position in the X and Z axis and lock rotation in X,Y,Z axis.
- A capsule collider that serves as the main collider of the prefab.
- Animator
- A tag with a certain id “littleBrother” that communicates with the agent through a script



Figure 52 Little brother prefab

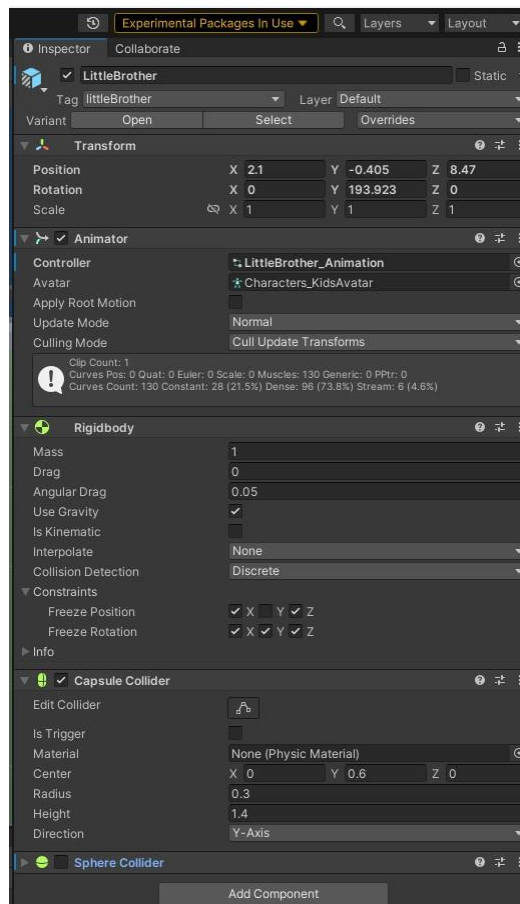


Figure 53 Little brother inspector window

Boat

The boat serves as a target for the agent, with the goal of giving it to his brother. Also in this case, the inspector tab is not complex at all.

- Rigidbody component with locked rotation X and Z
- A capsule collider as the main collider
- A boat script

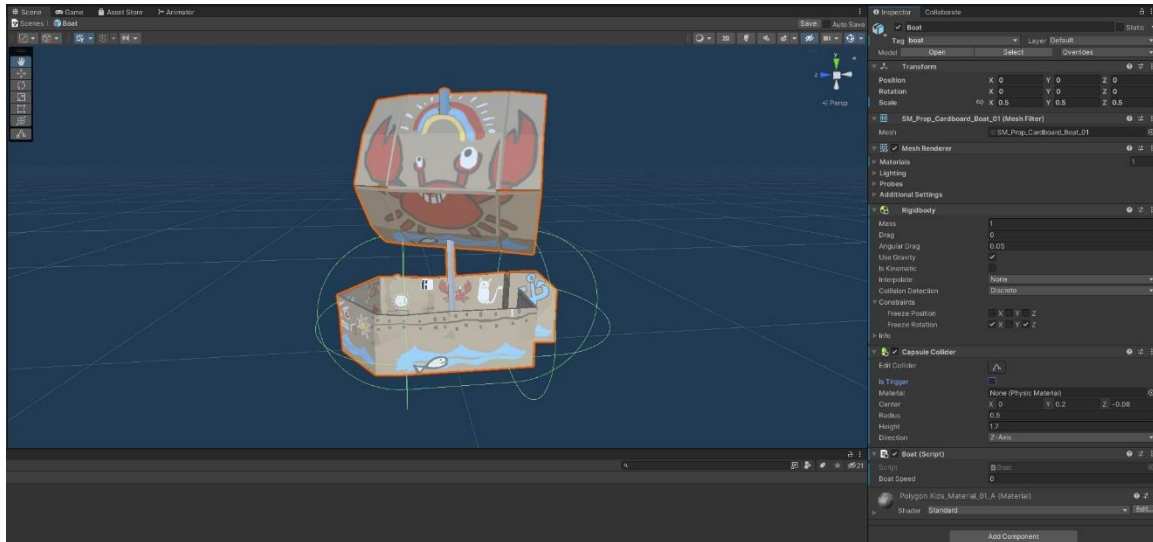


Figure 54 Boat prefab with inspector window

5. Implementation

5.1 Scripts presentation

The scripts developed for the project are detailed in this chapter, with comments throughout the code. These scripts handle scene preparation (such as randomized placement of the big brother agent), big brother decision making, boat movement and agent-scene interaction. The three scripts that were created for the project are:

- PlaygroundArea.cs
- BigBrotherAgent.cs
- Boat.cs

PlaygroundArea.cs

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using Unity.MLAgents;
5  using Unity.MLAgents.Examples;
6  using TMPro;
7
8  [UnityScript(2 asset references)] 3 references
9  public class PlaygroundArea : Area
10 {
11     [Tooltip("The agent inside the area")]
12     public BigBrotherAgent bigBrotherAgent;
13
14     [Tooltip("The little brother inside the area")]
15     public GameObject littleBrother;
16
17     [Tooltip("The TextMeshPro text that shows the cumulative reward of the agent")]
18     public TextMeshPro cumulativeRewardText;
19
20     [Tooltip("Prefab of a boat")]
21     public Boat boatPrefab;
22
23     private List<GameObject> boatList;
24
25     /// <summary>
26     /// Reset the area, including boat and big brother placement
27     /// </summary>
28     2 references
29     public void ResetArea()
30     {
31         RemoveAllBoat();
32         PlaceBigBrother();
33         PlaceLittleBrother();
34         SpawnBoat(6, .5f);
35     }
36
37     /// <summary>
38     /// Remove a specific boat from the area when it is taken
39     /// </summary>
40     /// <param name="boatObject">The boat to remove</param>
41     1 reference
42     public void RemoveSpecificBoat(GameObject boatObject)
43     {
44         boatList.Remove(boatObject);
45         Destroy(boatObject);
46     }
47
48     /// <summary>
49     /// The number of boats remaining
50     /// </summary>
51     1 reference
52     public int BoatRemaining
53     {
54         get { return boatList.Count; }
55     }
56
57     /// <summary>
58     /// Choose a random position on the X-Z plane within a partial donut shape
59     /// </summary>
60     /// <param name="center">The center of the donut</param>
61     /// <param name="minAngle">Minimum angle of the wedge</param>
62     /// <param name="maxAngle">Maximum angle of the wedge</param>
63     /// <param name="minRadius">Minimum distance from the center</param>
64     /// <param name="maxRadius">Maximum distance from the center</param>
65     /// <returns>A position falling within the specified region</returns>
66     4 references

```

Figure 55 PlaygroundArea.cs (1)

```

62 public static Vector3 ChooseRandomPosition(Vector3 center, float minAngle, float maxAngle, float minRadius, float maxRadius)
63 {
64     float radius = minRadius;
65     float angle = minAngle;
66
67     if (maxRadius > minRadius)
68     {
69         // Pick a random radius
70         radius = UnityEngine.Random.Range(minRadius, maxRadius);
71     }
72
73     if (maxAngle > minAngle)
74     {
75         // Pick a random angle
76         angle = UnityEngine.Random.Range(minAngle, maxAngle);
77     }
78
79     // Center position + forward vector rotated around the Y axis by "angle" degrees, multiplies by "radius"
80     return center + Quaternion.Euler(0f, angle, 0f) * Vector3.forward * radius;
81 }
82
83 /// <summary>
84 /// Remove all boats from the area
85 /// </summary>
86 private void RemoveAllBoat()
87 {
88     if (boatList != null)
89     {
90         for (int i = 0; i < boatList.Count; i++)
91         {
92             if (boatList[i] != null)
93             {
94                 Destroy(boatList[i]);
95             }
96         }
97     }
98
99     boatList = new List<GameObject>();
100 }
101
102 /// <summary>
103 /// Place the big brother in the area
104 /// </summary>
105 private void PlaceBigBrother()
106 {
107     Rigidbody rigidbody = bigBrotherAgent.GetComponent<Rigidbody>();
108     rigidbody.velocity = Vector3.zero;
109     rigidbody.angularVelocity = Vector3.zero;
110     bigBrotherAgent.transform.position = ChooseRandomPosition(transform.position, 0f, 360f, 0f, 9f) + Vector3.up * .5f;
111     bigBrotherAgent.transform.rotation = Quaternion.Euler(0f, UnityEngine.Random.Range(0f, 360f), 0f);
112 }
113
114 /// <summary>
115 /// Place the little brother in the area
116 /// </summary>
117 private void PlaceLittleBrother()

```

Figure 56 PlaygroundArea.cs (2)

```

117 private void PlaceLittleBrother()
118 {
119     Rigidbody rigidbody = littleBrother.GetComponent<Rigidbody>();
120     rigidbody.velocity = Vector3.zero;
121     rigidbody.angularVelocity = Vector3.zero;
122     littleBrother.transform.position = ChooseRandomPosition(transform.position, -45f, 45f, 4f, 9f) + Vector3.up * .5f;
123     littleBrother.transform.rotation = Quaternion.Euler(0f, 180f, 0f);
124 }
125
126 /// <summary>
127 /// Spawn some number of boats in the area and set their speed
128 /// </summary>
129 /// <param name="count">The number to spawn</param>
130 /// <param name="boatSpeed">The speed</param>
131 private void SpawnBoat(int count, float boatSpeed)
132 {
133     for (int i = 0; i < count; i++)
134     {
135         // Spawn and place the boat
136         GameObject boatObject = Instantiate<GameObject>(boatPrefab.gameObject);
137         boatObject.transform.position = ChooseRandomPosition(transform.position, 100f, 260f, 2f, 13f) + Vector3.up * .5f;
138         boatObject.transform.rotation = Quaternion.Euler(0f, UnityEngine.Random.Range(0f, 360f), 0f);
139
140         // Set the boat's parent to this area's transform
141         boatObject.transform.SetParent(transform);
142
143         // Keep track of the boat
144         boatList.Add(boatObject);
145
146         // Set the boat speed
147         boatObject.GetComponent<Boat>().boatSpeed = boatSpeed;
148     }
149 }
150
151 /// <summary>
152 /// Called when the game starts
153 /// </summary>
154 @ UnityMessage | 0 references
155 private void Start()
156 {
157     ResetArea();
158 }
159
160 /// <summary>
161 /// Called every frame
162 /// </summary>
163 @ UnityMessage | 0 references
164 private void Update()
165 {
166     // Update the cumulative reward text
167     cumulativeRewardText.text = bigBrotherAgent.GetCumulativeReward().ToString("0.00");
168 }

```

Figure 57 PlaygroundArea.cs (3)

Details about the script

The PlaygroundArea script will handle a training area with one older brother, one little brother, and a number of boats. It is in charge of eliminating boats, generating boats, and randomly placing the two siblings. There may be numerous PlaygroundAreas in the Scene for more effective training.

In order to use the mlagents features, the script needs to have the statements for *MLAgents* and *TMPPro* and the class inherits from *Agent* instead of *Monobehaviour*. The variables of *bigBrotherAgent*, *littleBrother*, *cumulativeRewardText* and *boatPrefab*, keep track of important objects in the scene and are hooked up to objects that are public variables.

When a boat is collected by the big brother, the bigBrotherAgent script runs RemoveSpecificBoat() to remove it from the water. The two siblings will be placed in the area in the following duties. It is best to spawn boats in the sea and place the little brother on land. Because the big brother can roam between land and sea, he can be assigned to either.

ChooseRandomPosition() selects a random spot inside wedges surrounding the area's central point using specified radius and angle limitations. More information may also be found in the code's comments.

The *ResetArea()* function calls a new function called *RemoveAllBoat()*, to make sure no boats are in the area before spawning new boats.

Following that, two additional functions that put the two brothers are added. *PlaceLittleBrother()* and *PlaceBigBrother()*. They set rigidbody velocities to zero in both situations since unexpected things might happen while practicing for lengthy periods of time at 100x speed. Big brother, for example, may fall through the floor and then speed downhill. The location would be restored when the area resets, but if the downward velocity is not reset, big brother might rip through the earth.

After that, a new *SpawnBoat()* function is introduced, which deploys a given number of boats in the environment and determines their default sailing speed. The comments in the code give further information.

Finally, a new *Start()* method is established from whence *ResetArea()* is called, as well as a new *Update()* function that changes the cumulative prize display text on the area's back wall every frame. It is not required for training, but it does aid in visualizing how well the agents are functioning.

BigBrotherAgents.cs

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4  using Unity.MLAgents;
5  using Unity.MLAgents.Actuators;
6  using Unity.MLAgents.Sensors;
7
8  [UnityScript(1 asset reference)] 1 reference
9  public class BigBrotherAgent : Agent
10 {
11     [Tooltip("How fast the agent moves forward")]
12     public float moveSpeed = 5f;
13
14     [Tooltip("How fast the agent turns")]
15     public float turnSpeed = 180f;
16
17     [Tooltip("Prefab of the heart that appears when the young brother gets it's boat")]
18     public GameObject heartPrefab;
19
20     private PlaygroundArea playgroundArea;
21     new private Rigidbody rigidbody;
22     private GameObject littleBrother;
23     private bool isFull; // If true, big brother cannot carry more boats
24
25     /// <summary>
26     /// Initial setup, called when the agent is enabled
27     /// </summary>
28     6 references
29     public override void Initialize()
30     {
31         base.Initialize();
32         playgroundArea = GetComponentInParent<PlaygroundArea>();
33         littleBrother = playgroundArea.littleBrother;
34         rigidbody = GetComponent<Rigidbody>();
35     }
36
37     /// <summary>
38     /// Perform actions based on a vector of numbers
39     /// </summary>
40     /// <param name="actionBuffers">The struct of actions to take</param>
41     15 references
42     public override void OnActionReceived(ActionBuffers actionBuffers)
43     {
44         // Convert the first action to forward movement
45         float forwardAmount = actionBuffers.DiscreteActions[0];
46
47         // Convert the second action to turning left or right
48         float turnAmount = 0f;
49         if (actionBuffers.DiscreteActions[1] == 1f)
50         {
51             turnAmount = -1f;
52         }
53         else if (actionBuffers.DiscreteActions[1] == 2f)
54         {
55             turnAmount = 1f;
56         }
57
58         // Apply movement
59         rigidbody.MovePosition(transform.position + transform.forward * forwardAmount * moveSpeed * Time.fixedDeltaTime);
60         transform.Rotate(transform.up * turnAmount * turnSpeed * Time.fixedDeltaTime);
61
62         // Apply a tiny negative reward every step to encourage action
63         if (MaxStep > 0) AddReward(-1f / MaxStep);
64     }
65 }

```

Figure 58 BigBrotherAgents.cs (1)

```

63     /// <summary>
64     /// Read inputs from the keyboard and convert them to a list of actions.
65     /// This is called only when the player wants to control the agent and has set
66     /// Behavior Type to "Heuristic Only" in the Behavior Parameters inspector.
67     /// </summary>
68     /// <returns>A vectorAction array of floats that will be passed into <see cref="AgentAction(float[])"></returns>
69     13 references
70     public override void Heuristic(in ActionBuffers actionsOut)
71     {
72         int forwardAction = 0;
73         int turnAction = 0;
74         if (Input.GetKey(KeyCode.W))
75         {
76             // move forward
77             forwardAction = 1;
78         }
79         if (Input.GetKey(KeyCode.A))
80         {
81             // turn left
82             turnAction = 1;
83         }
84         else if (Input.GetKey(KeyCode.D))
85         {
86             // turn right
87             turnAction = 2;
88         }
89
90         // Put the actions into the array
91         actionsOut.DiscreteActions.Array[0] = forwardAction;
92         actionsOut.DiscreteActions.Array[1] = turnAction;
93     }
94
95     /// <summary>
96     /// When a new episode begins, reset the agent and area
97     /// </summary>
98     11 references
99     public override void OnEpisodeBegin()
100    {
101        isFull = false;
102        playgroundArea.ResetArea();
103    }
104
105     /// <summary>
106     /// Collect all non-Raycast observations
107     /// </summary>
108     /// <param name="sensor">The vector sensor to add observations to</param>
109     9 references
110     public override void CollectObservations(VectorSensor sensor)
111     {
112         // Whether the big brother has acquired a boat (1 float = 1 value)
113         sensor.AddObservation(isFull);
114
115         // Distance to young brother (1 float = 1 value)
116         sensor.AddObservation(Vector3.Distance(littleBrother.transform.position, transform.position));
117
118         // Direction to young brother (1 Vector3 = 3 values)
119         sensor.AddObservation((littleBrother.transform.position - transform.position).normalized);
120
121         // Direction big brother is facing (1 Vector3 = 3 values)
122         sensor.AddObservation(transform.forward);
123
124         // 1 + 1 + 3 + 3 = 8 total values
125     }

```

Figure 59 BigBrotherAgents.cs (2)


```

124     /// <summary>
125     /// When the agent collides with something, take action
126     /// </summary>
127     /// <param name="collision">The collision info</param>
128     private void OnCollisionEnter(Collision collision)
129     {
130         if (collision.transform.CompareTag("boat"))
131         {
132             // Try to collect the boat
133             TakeBoat(collision.gameObject);
134         }
135         else if (collision.transform.CompareTag("littleBrother"))
136         {
137             // Try to give the boat to the little brother
138             GiveBoat();
139         }
140     }
141
142     /// <summary>
143     /// Check if agent is full, if not, collect the boat and get a reward
144     /// </summary>
145     /// <param name="boatObject">The boat to collect</param>
146     private void TakeBoat(GameObject boatObject)
147     {
148         if (isFull) return; // Can't hold another boat while full
149         isFull = true;
150
151         playgroundArea.RemoveSpecificBoat(boatObject);
152
153         AddReward(1f);
154     }
155
156     /// <summary>
157     /// Check if agent is full, if yes, give the boat to little brother
158     /// </summary>
159     private void GiveBoat()
160     {
161         if (!isFull) return;
162         isFull = false;
163
164         // Spawn heart
165         GameObject heart = Instantiate<GameObject>(heartPrefab);
166         heart.transform.parent = transform.parent;
167         heart.transform.position = littleBrother.transform.position + new Vector3(0, 2, 0);
168         Destroy(heart, 4f);
169
170         AddReward(1f);
171
172         if (playgroundArea.BoatRemaining <= 0)
173         {
174             EndEpisode();
175         }
176     }
177 }
178

```

Figure 60 BigBrotherAgents.cs (3)

Details about the script

The `BigBrotherAgent` class, which inherits from the `Agent` class, handles observing the environment, taking actions, interacting, and accepting player input.

First, `MLAgents` statement is used once again alongside the inheritance from `Agent` instead of `Monobehaviour`. Public variables are introduced to keep track of the big brother agent's motion and turn speed, as well as the heart prefab. To keep track of things, private variables are also introduced.

When the agent wakes up, the method `InitializeAgent()` is called once. Because it is not called every time the agent is reset, there is a separate `ResetAgent()` method.

The agent receives and responds to orders in `OnActionReceived()`. These commands might come from a neural network or a human player, but this function treats them equally.

The parameter of `actionBuffers` argument is a struct that includes an array of numerical values that correspond to the agent's activities. This project employs "discrete" actions, which implies that each integer value (e.g., 0, 1, 2,...) corresponds to an option. The alternative is to use "continuous" actions, which enable you to enter any fractional number between -1 and +1 (e.g., -1, 0.23, 4,...). Discrete actions provide only one option at a time, with no in-between options.

In this case:

- `actionBuffers.DiscreteActions[0]` can either be 0 or 1, indicating whether to remain in place (0) or move forward at full speed (1).
- `actionBuffers.DiscreteActions[1]` can either be 0, 1, or 2, indicating whether to not turn (0), turn in the negative direction (1), or turn in the positive direction (2).

When trained, the neural network has no idea what these activities do. It simply understands that when it perceives the world in a specific manner, certain activities result in more reward points. This is why it will be critical later in this script to establish an effective observation of the surroundings.

The `OnActionReceived()` function applies the movement and rotation after understanding the vector actions and then adds a little negative reward. This minor negative reinforcement pushes the agent to accomplish its task as soon as feasible.

In this example, a $-1 / 5000$ reward is provided for each of the 5,000 steps (this is comes from the variable `MaxStep`, which is set later). If the big brother finishes early — say, in 3,000 steps — the negative reward generated by this line of code is $-3000 / 5000 = -0.6$. If the agent completes all 5,000 steps, the overall negative payoff is $-5000 / 5000 = -1$.

The `Heuristic()` function allows the agent to be controlled without the need of a neural network. This method will accept keyboard inputs from the human player, translate them into actions, and store those actions in an array named `DiscreteActions`. When a person is playing, the `OnActionReceived` method reads the same array (rather than an AI). In this project:

- The default `forwardAction` will be 0, but if the player presses 'W' on the keyboard, this value will be set to 1.
- The default `turnAction` will be 0, but if the player presses 'A' or 'D' on the keyboard, the value will be set to 1 or 2 respectively to turn left or right.

- Override the *Heuristic()* function.

When the agent has finished handing all of the boats to his little brother or has reached the maximum number of steps, the main Agent class automatically invokes the *OnEpisodeBegin()* method. This function will be used to reset the area.

The big brother agent monitors his surroundings in two ways. The first method is to use raycasts. This is equivalent to the agent shooting a number of laser pointers and checking if they strike anything. It is comparable to LIDAR, which is used by self-driving vehicles and robotics. Raycast observations are added using a RayPerceptionSensor component that was previously added to the Unity Editor.

The agent also observes the environment using numerical values. An observation can be converted into a list of integers and added as an observation for the agent, whether it be a true/false value, a distance, an XYZ location in space, or a quaternion rotation. The comments in the code provide further information on what was added.

When deciding what to observe, one must use extreme caution. The agent will be unable to fulfill his mission if he lacks sufficient knowledge about its surroundings. For example, suppose the agent is blinded and floating in space. What information about his surroundings would he need to make an informed decision?

This big brother agent, as it is currently implemented, has no memory. It is vital to assist him by informing him of where things are at each update phase so that he may make a decision.

Next, *OnCollisionEnter()* is used to detect and respond to collisions with objects having the tags "boat" or "littleBrother."

If the big brother doesn't already have one, a new *TakeBoat()* function for collecting boats is being introduced. It will remove the boat from the area and receive a reward for doing so.

Finally, a function for giving the boat to the younger sibling is being developed. A floating heart appears in the air, indicating how much the younger brother appreciates his sibling for recovering his boats. There is also a timer set for auto-destroy. The agent is rewarded, and if there are no boats left, *Done()* is called, which automatically calls *AgentReset ()*.

Boat.cs

```

1  using System.Collections;
2  using System.Collections.Generic;
3  using UnityEngine;
4
5  [UnityScript(1 asset reference) | 2 references]
6  public class Boat : MonoBehaviour
7  {
8      [Tooltip("The sailing speed")]
9      public float boatSpeed;
10
11     private float randomizedSpeed = 0f;
12     private float nextActionTime = -1f;
13     private Vector3 targetPosition;
14
15     /// <summary>
16     /// Called every timestep
17     /// </summary>
18     [Unity Message | 0 references]
19     private void FixedUpdate()
20     {
21         if (boatSpeed > 0f)
22         {
23             Sail();
24         }
25
26     }
27     /// <summary>
28     /// Sail between random positions
29     /// </summary>
30     private void Sail()
31     {
32         // If it's time for the next action, pick a new speed and destination
33         // Else, sail toward the destination
34         if (Time.fixedTime >= nextActionTime)
35         {
36             // Randomize the speed
37             randomizedSpeed = boatSpeed * UnityEngine.Random.Range(.5f, 1.5f);
38
39             // Pick a random target
40             targetPosition = PlaygroundArea.ChooseRandomPosition(transform.parent.position, 100f, 260f, 2f, 13f);
41
42             // Rotate toward the target
43             transform.rotation = Quaternion.LookRotation(targetPosition - transform.position, Vector3.up);
44
45             // Calculate the time to get there
46             float timeToGetThere = Vector3.Distance(transform.position, targetPosition) / randomizedSpeed;
47             nextActionTime = Time.fixedTime + timeToGetThere;
48         }
49         else
50         {
51             // Make sure that the boat does not sail past the target
52             Vector3 moveVector = randomizedSpeed * transform.forward * Time.fixedDeltaTime;
53             if (moveVector.magnitude <= Vector3.Distance(transform.position, targetPosition))
54             {
55                 transform.position += moveVector;
56             }
57             else
58             {
59                 transform.position = targetPosition;
60                 nextActionTime = Time.fixedTime;
61             }
62         }
63     }
64 }

```

Figure 61 Boat.cs

Details about the script

The Boat class will be attached to each boat and will cause it to sail. Because Unity lacks water physics, they just flow in a straight path toward a specific location with this code to make things simple. Below is an overview of the variables:

- boatSpeed controls the average speed of the boats
- randomizedSpeed is a slightly altered speed that will change randomly each time a new sail destination is picked.
- nextActionTime is used to trigger the selection of a new sail destination
- targetPosition is the position of the destination the boat is sailing toward

FixedUpdate is called at regular intervals of 0.02 seconds (it is not affected by frame rate) and allows us to communicate even while the agent is training at a higher game pace, which is frequent when training ML-Agents. In it, we determine if the boat should sail and, if so, we invoke the *Sail()* method.

The sail functionality will be introduced next. At each update, the boat will either choose a new speed and destination or return to its existing location.

When it comes time to change course, the boat will:

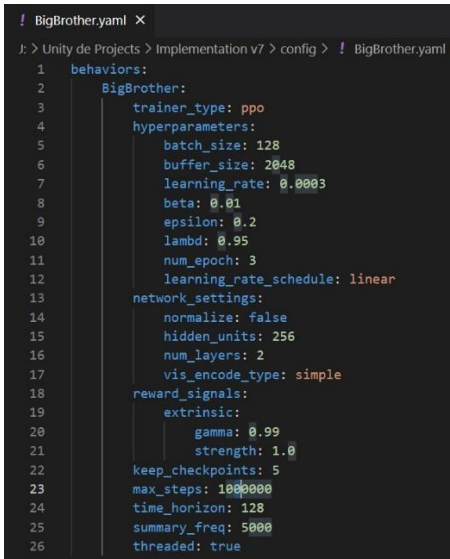
- Select a new randomized speed ranging from 50% to 150% of the average boat speed.
- Choose a new goal place (in the water) to sail toward.
- Turn the boat around to face the objective.
- Determine the amount of time required to get there.
- Otherwise, the boat will move toward the target to avoid sailing past it.

5.2 Trainer Configuration File

A configuration file must be created before the siblings may be trained.

This section will provide a full summary of the setup information that the ML-Agents training program will need, with the configuration file. Some of these are hyperparameters — a phrase that deep learning practitioners may be acquainted with — while others are ML-Agent-specific settings.

On the computer, we create a new folder named config (for consistency), and within it, we create the config file called "BigBrother.yaml."



```

! BigBrother.yaml x
J: > Unity de Projects > Implementation v7 > config > ! BigBrother.yaml
1 behaviors:
2   BigBrother:
3     trainer_type: ppo
4     hyperparameters:
5       batch_size: 128
6       buffer_size: 2048
7       learning_rate: 0.0003
8       beta: 0.01
9       epsilon: 0.2
10      lambda: 0.95
11      num_epoch: 3
12      learning_rate_schedule: linear
13     network_settings:
14       normalize: false
15       hidden_units: 256
16       num_layers: 2
17       vis_encode_type: simple
18     reward_signals:
19       extrinsic:
20         gamma: 0.99
21         strength: 1.0
22     keep_checkpoints: 5
23     max_steps: 100000
24     time_horizon: 128
25     summary_freq: 5000
26     threaded: true

```

Figure 62 Configuration parameters file

It is important to note that the name of the .yaml file must exactly match the Behavior Name that was set within Unity in the bigBrother's Behavior Parameters. Otherwise, the agent will not properly train. These configuration parameters govern how training will take place. Details regarding the hyperparameters are discussed further down.

Common Trainer Configurations

One of the first considerations to be taken about the training run is whether to utilize PPO or SAC trainers. Some training setups are shared by both trainers, while others are dependent on the trainer's preference. PPO trainer was utilized for the project and will be discussed further below.

trainer_type

(default = ppo) The type of trainer to use: ppo (Proximal Policy Optimization) or sac (Soft Actor Critic)

summary_freq

(The default value is 50000) The number of experiences required before producing and displaying training data. The granularity of the graphs in Tensorboard is determined by this.

time_horizon

(The default value is 64) How many experience steps should be collected per agent before putting it to the experience buffer? When this limit is reached before the completion of an episode, a value estimate is utilized to anticipate the agent's overall expected reward based on its present status. As a result, this parameter trades off between a less biased but greater variance estimate (long time horizon) and a more biased but less varied estimate (short time horizon) (short time horizon). When there are numerous rewards within an episode, or when episodes are too long, a lesser number may be preferable. This number should be large enough to capture all of the significant activity inside an agent's activities sequence.

32 - 2048 is a typical range.

max_steps

(default = 500000) The total number of steps (i.e., observations gathered and actions made) that must be completed in the environment (or across all environments if many are used in parallel) before the training process is terminated. If your environment has several agents with the same behavior name, all steps made by those agents will contribute to the same max steps count.

5e5 - 1e7 is a typical range.

keep_checkpoints

The maximum number of model checkpoints to maintain (default = 5) After the number of steps indicated by the checkpoint interval parameter, checkpoints are saved. When the maximum number of checkpoints is reached, the oldest checkpoint is erased when a new checkpoint is saved.

threaded

(yes by default) Model changes are permitted by default while the environment is being stepped. This marginally violates the PPO on-policy premise in exchange for a training speedup. You can prevent parallel updates by setting threaded to false to ensure PPO's stringent on-policy nature.

hyperparameters -> learning_rate

(default = $3e-4$) Gradient descent initial learning rate. Corresponds to the degree of difficulty of each gradient descent update step. If training is inconsistent and the reward does not consistently grow, this should be reduced.

$1e-5$ to $1e-3$ is a typical range.

hyperparameters -> batch_size

The number of experiences in each gradient descent iteration. This should always be several times less than buffer size. This value should be huge if you are utilizing a continuous action space (in the order of 1000s). If you're working with a discrete action space, this number should be lower (in order of 10s).

(Continuous - PPO): 512 - 5120

hyperparameters -> buffer_size

(PPO default = 10240)

PPO: The number of experiences needed to update the policy model. Corresponds to the number of experiences that should be gathered before learning or upgrading the model. This should be several times the size of batch size. A bigger buffer size usually equates to more reliable training updates.

Typical price range: PPO: 2048 – 409600

hyperparameters -> learning_rate_schedule

(The default for PPO is linear) Determines how the rate of learning varies over time. We advocate declining the learning rate until max steps for PPO so that learning converges more steadily. However, under some circumstances (for example, training for an indeterminate period of time), this capability can be deactivated.

The learning rate decays linearly, reaching 0 at max steps, whereas constant maintains the learning rate constant during the training cycle.

network_settings -> hidden_units

(default = 128) The number of units in the neural network's hidden layers. The number of units in each fully linked layer of the neural network. This should be minimal for basic cases when the proper action is a simple combination of the observation inputs. This should be bigger for issues where the action involves a very complicated interplay between the observation variables.

32 - 512 is a typical range.

network_settings -> num_layers

The number of hidden layers in the neural network (default = 2) The number of hidden layers existing after the observation input or after CNN encoding of the visual observation. Fewer layers are likely to train quicker and more effectively for basic situations. For more sophisticated control problems, additional layers may be required.

1 - 3 is a typical range.

network_settings -> normalize

(false by default) Whether or not the vector observation inputs are normalized. This normalization is based on the vector observation's running average and variance. Normalization can be beneficial in circumstances of sophisticated continuous control issues, but it can be detrimental in cases of simpler discrete control problems.

network_settings -> vis_encoder_type

Encoder type for encoding visual observations (default = basic).

Simple (default) employs a basic encoder with two convolutional layers, nature cnn employs Mnih et al's implementation with three convolutional layers, and resnet employs the IMPALA. Resnet is a considerably larger network than the other two since it is made up of three stacked layers, each having two residual blocks. match3 (Gudmundsoon et al.) is a smaller CNN tailored for board games that may be utilized with visual observation sizes as small as 5x5.

Trainer-specific Configurations (PPO-specific Configurations)

There are more trainer-specific settings available depending on the trainer selected and they are listed below.

hyperparameters -> beta

(standard = $5.0e-3$) The entropy regularization's strength, which makes the policy "more random." This guarantees that agents explore the action space adequately throughout training. Increasing this will result in more random acts. This should be tweaked so that the entropy (as measured by TensorBoard) gradually declines as the reward increases. Increase beta if entropy falls too rapidly. Reduce beta if entropy reduces too slowly. $1e-4$ to $1e-2$ is a typical range.

hyperparameters -> epsilon

(default = 0.2) Determines how quickly the policy can change during training. This value corresponds to the allowable level of divergence between old and new policies during gradient descent updating. Setting this parameter to a low value will produce more consistent updates but will also slow down the training process.

0.1- 0.3 is a typical range.

hyperparameters -> lambda

(standard = 0.95) When computing the Generalized Advantage Estimate, the regularization parameter (lambda) is employed (GAE). When computing an updated value estimate, this is how much the agent depends on its existing value estimate. Low values equate to a greater reliance on the current value estimate (which might be very biased), whereas high values correspond to a greater reliance on the actual rewards obtained in the environment (which can be high variance). The parameter represents a trade-off between the two, and the appropriate value might result in a more stable training process.

0.9 - 0.95 is a typical range.

hyperparameters -> num_epoch

Number of passes through the experience buffer when doing gradient descent optimization (default = 3). The greater the batch size, the larger this is permissible. Reduced this will result in more reliable updates at the expense of slower learning.

3 - 10 is a typical range.

Reward Signals (Extrinsic Rewards)

The reward signals section allows you to configure parameters for both extrinsic (environment-based) and intrinsic reward signals (e.g. curiosity and GAIL). In addition to any class-specific hyperparameters, each reward signal should define at least two parameters, intensity and gamma. It should be noted that in order to remove a reward signal, its record in reward signals must be deleted fully. At all times, at least one reward signal should be defined.

extrinsic -> strength

(default value = 1.0) Factor used to multiply the reward provided by the environment. Typical ranges will differ according on the reward signal. 1.00 is a typical range.

extrinsic -> gamma

(default = 0.99) Discount factor for future environmental rewards. This is how far into the future the agent should be concerned about potential rewards. This number should be high in instances when the agent should be active in the present to prepare for benefits in the far future. It might be smaller when the rewards are more immediate. Must be less than or equal to 1.

0.8 - 0.995 is a typical range. (22)

6. Presentation

6.1 Training Demo

In this chapter, we will show you how to train the agent. Unity's ML-Agents manage training using Python's API, and as described in Chapter 4, we utilize commands to start the training.

By opening the command prompt we navigate to the project's folder and specifically to the virtual environment that was created. Then, we run a quick check to see if everything went correct regarding the installation of ml agents and PyTorch. The command is *mlagents-learn --help*. The following figure indicates that everything works correct.

```

[thesis env] J:\Unity de Projects\Implementation v7>mlagents-learn --help
usage: mlagents-learn.exe [-h] [--env ENV_PATH] [--resume] [--deterministic] [--force] [--run-id RUN_ID] [--initialize-from RUN_ID] [--seed SEED] [--inference]
                        [--base-port BASE_PORT] [--num-envs NUM_ENVS] [--num-areas NUM_AREAS] [--debug] [--env-args ...] [--max-lifetime-restarts MAX_LIFETIME_RESTARTS]
                        [--restarts-rate-limit-n RESTARTS_RATE_LIMIT_N] [--restarts-rate-limit-period-s RESTARTS_RATE_LIMIT_PERIOD_S] [--torch] [--tensorflow]
                        [--results-dir RESULTS_DIR] [--width WIDTH] [--height HEIGHT] [--quality-level QUALITY_LEVEL] [--time-scale TIME_SCALE]
                        [--target-frame-rate TARGET_FRAME_RATE] [--capture-frame-rate CAPTURE_FRAME_RATE] [--no-graphics] [--torch-device DEVICE]
                        [trainer_config_path]

positional arguments:
  trainer_config_path

optional arguments:
  -h, --help            show this help message and exit
  --env ENV_PATH        Path to the Unity executable to train (default: None)
  --resume              Whether to resume training from a checkpoint. Specify a --run-id to use this option. If set, the training code loads an already trained model to
                        initialize the neural network before resuming training. This option is only valid when the models exist, and have the same behavior names as the current
                        agents in your scene. (default: False)
  --deterministic       Whether to select actions deterministically in policy. "dist.mean" for continuous action space, and "dist.argmax" for deterministic action space
                        (default: False)
  --force               Whether to force-overwrite this run-id's existing summary and model data. (Without this flag, attempting to train a model with a run-id that has been
                        used before will throw an error. (default: False)
  --run-id RUN_ID       The identifier for the training run. This identifier is used to name the subdirectories in which the trained model and summary statistics are saved as
                        well as the saved model itself. If you use TensorBoard to view the training statistics, always set a unique run-id for each training run. (The
                        statistics for all runs with the same id are combined as if they were produced by a the same session.) (default: ppo)
  --initialize-from RUN_ID
                        Specify a previously saved run ID from which to initialize the model from. This can be used, for instance, to fine-tune an existing model on a new
                        environment. Note that the previously saved models must have the same behavior parameters as your current environment. (default: None)
  --seed SEED           A number to use as a seed for the random number generator used by the training code (default: -1)
  --inference           Whether to run in Python inference mode (i.e. no training). Use with --resume to load a model trained with an existing run ID. (default: False)
  --base-port BASE_PORT
                        The starting port for environment communication. Each concurrent Unity environment instance will get assigned a port sequentially, starting from the
                        base-port. Each instance will use the port (base_port + worker_id), where the worker_id is sequential IDs given to each instance from 0 to (num_envs -
                        1). Note that when training using the Editor rather than an executable, the base port will be ignored. (default: 5005)
  --num-envs NUM_ENVS  The number of concurrent Unity environment instances to collect experiences from when training (default: 1)
  --num-areas NUM_AREAS
                        The number of parallel training areas in each Unity environment instance. (default: 1)
  --debug              Whether to enable debug-level logging for some parts of the code (default: False)
  --env-args ...       Arguments passed to the Unity executable. Be aware that the standalone build will also process these as Unity Command Line Arguments. You should choose
                        different argument names if you want to create environment-specific arguments. All arguments after this flag will be passed to the executable. (default:
                        None)
  --max-lifetime-restarts
                        MAX_LIFETIME_RESTARTS
                        The max number of times a single Unity executable can crash over its lifetime before ml-agents exits. Can be set to -1 if no limit is desired. (default:
                        10)
  --restarts-rate-limit-n
                        RESTARTS_RATE_LIMIT_N
                        The maximum number of times a single Unity executable can crash over a period of time (period set in restarts-rate-limit-period-s). Can be set to -1 to
                        not use rate limiting with restarts. (default: 1)
  --restarts-rate-limit-period-s
                        RESTARTS_RATE_LIMIT_PERIOD_S
                        The period of time --restarts-rate-limit-n applies to. (default: 60)
  --torch              (Removed) Use the PyTorch framework. (default: False)
  --tensorflow         (Removed) Use the TensorFlow framework. (default: False)
  --results-dir RESULTS_DIR
                        Results base directory (default: results)

```

Figure 63 *mlagents help* command

Afterwards we type *mlagents-learn config/BigBrother.yaml --run-id=Thesis_training_2* to start training. The *config/BigBrother.yaml*, indicates that the agents, uses the custom training configuration parameters from the relative path and the run-id indicates a unique name for that round.

```
(thesis_env) J:\Unity de Projects\Implementation v7>mlagents-learn config/BigBrother.yaml1 --run-id=Thesis_training_2
```



```
Version information:  
ml-agents: 0.28.0,  
ml-agents-envs: 0.28.0,  
Communicator API: 1.5.0,  
PyTorch: 1.7.1+cu110  
[INFO] Listening on port 5004. Start training by pressing the Play button in the Unity Editor.
```

Figure 64 mlagents training command

By pressing the play button in the Unity Editor our agent starts the training process. We notice that the movement is very fast because its running at 20x speed. Because the first time the relies on Python and the inputs and data that were given from the script BigBrotherAgent.cs. It takes some time to achieve its goal of collecting the boats and giving them to his younger sibling. For that case, we duplicated the PlaygroundArea 7 times, for a total of 8 instances that train simultaneously and speed up the training by a large amount.

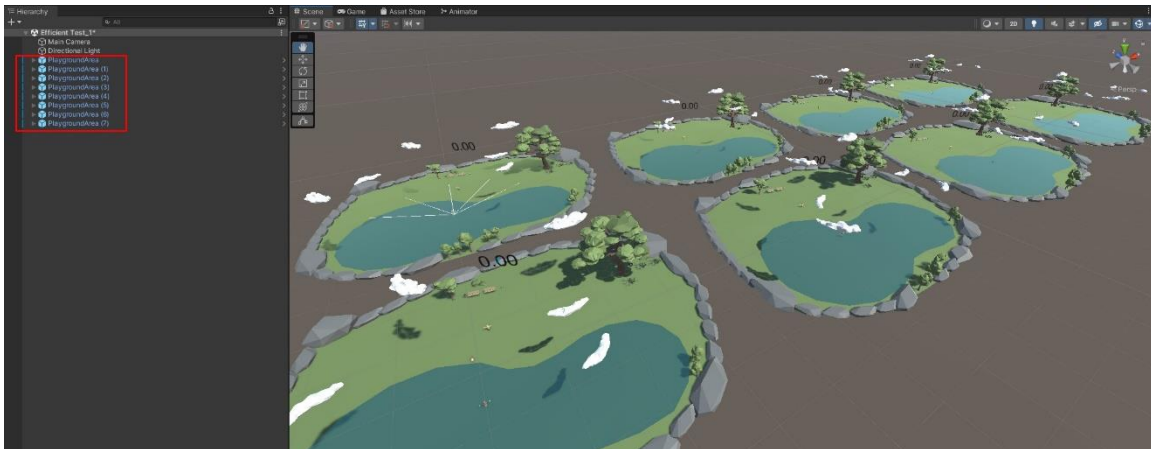


Figure 65 Multiple training instances

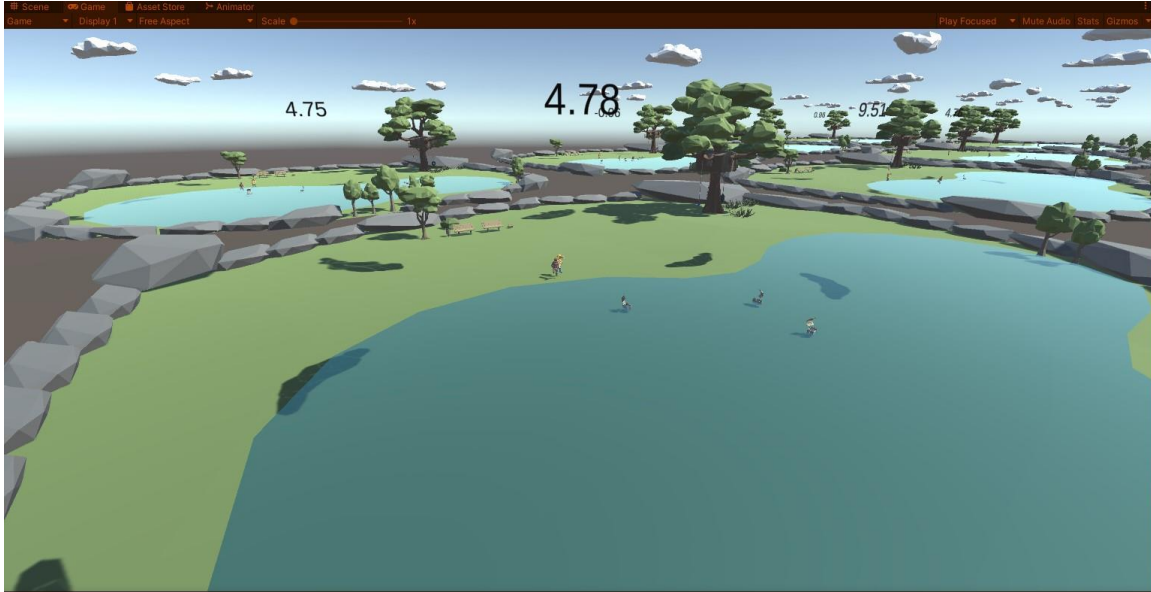


Figure 66 Training phase

As the training proceeds, we get periodic updates which include:

- Step: The number of timesteps that have elapsed
- Time Elapsed: How much time the training has been running (in real-world time)
- Mean Reward: The average reward (since the last update)
- Std of Reward: The standard deviation of the reward (since the last update)

```
[INFO] BigBrother. Step: 140000. Time Elapsed: 270.301 s. Mean Reward: -0.125. Std of Reward: 0.331. Training.
[INFO] BigBrother. Step: 145000. Time Elapsed: 279.326 s. Mean Reward: 0.000. Std of Reward: 0.000. Training.
[INFO] BigBrother. Step: 150000. Time Elapsed: 288.036 s. No episode was completed since last summary. Training.
[INFO] BigBrother. Step: 155000. Time Elapsed: 297.338 s. Mean Reward: 0.250. Std of Reward: 0.661. Training.
[INFO] BigBrother. Step: 160000. Time Elapsed: 306.528 s. Mean Reward: 0.571. Std of Reward: 0.728. Training.
[INFO] BigBrother. Step: 165000. Time Elapsed: 315.178 s. Mean Reward: 2.000. Std of Reward: 0.000. Training.
[INFO] BigBrother. Step: 170000. Time Elapsed: 323.955 s. Mean Reward: 0.750. Std of Reward: 0.968. Training.
[INFO] BigBrother. Step: 175000. Time Elapsed: 333.013 s. No episode was completed since last summary. Training.
[INFO] BigBrother. Step: 180000. Time Elapsed: 341.201 s. Mean Reward: 1.250. Std of Reward: 1.479. Training.
```

Figure 67 Command prompt training progress

Your agents should eventually become quite adept at gathering boats. When the agents can't pick up boats any quicker, the Mean Reward will stop growing (about 7.5 points). We could terminate training at this point by hitting the Play button in the Editor, but we let it train until it reached the "max steps: 1000000" value set in the yaml config file.

TensorBoard is a program that provides a more complete display of the results, including metrics, model graphs, and histograms, rather than merely the command prompt. We navigate to the project's virtual environment by opening another command terminal and running the command `tensorboard -logdir results`, and we receive a message with a specific url that tells where to access the visuals of the training data.

```
Administrator: Command Prompt - tensorboard --logdir results
Microsoft Windows [Version 10.0.19044.2251]
(c) Microsoft Corporation. All rights reserved.

C:\WINDOWS\system32>cd J:\Unity de Projects\Implementation v7

C:\WINDOWS\system32>j: cd

J:\Unity de Projects\Implementation v7>python-envs\thesis_env\Scripts\activate

(thesis_env) J:\Unity de Projects\Implementation v7>tensorboard --logdir results
TensorFlow installation not found - running with reduced feature set.
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all
TensorBoard 2.11.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

Figure 68 Tensorboard command

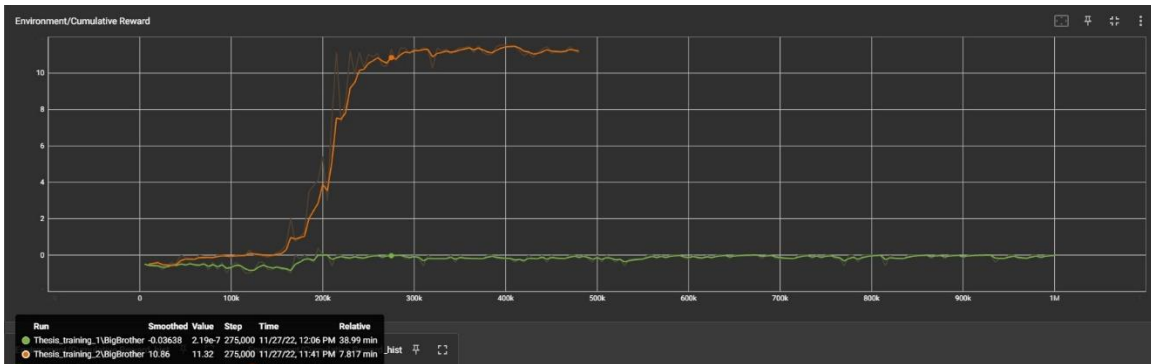


Figure 69 Cumulative reward graph

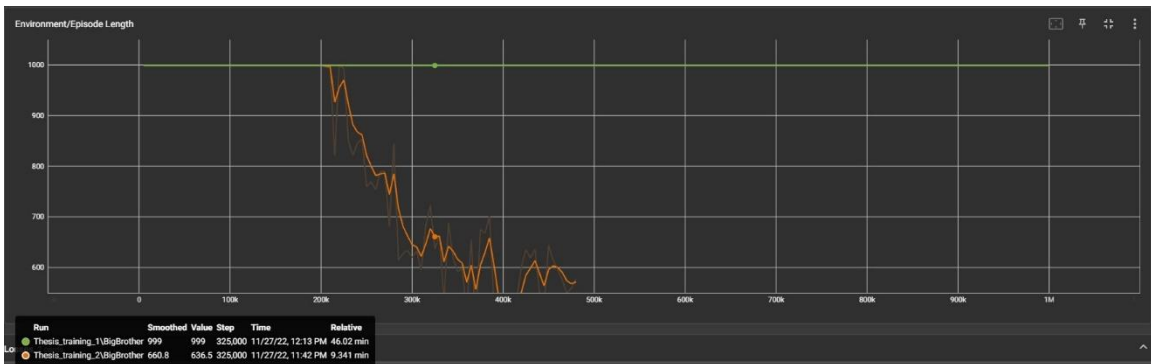


Figure 70 Episode length graph

The cumulative reward is the benefit provided to the agent each time he collects a boat or gives one to his brother, therefore the graph should always grow or be at a maximum level, indicating that the brain operates much better.

The episode duration decreases in the graph, indicating that the agent learns to attain his goal faster and more precisely.

As the round ended, the training has exported a BigBrother.onnx file that represents the trained neural network for our BigBrother. It should be mentioned that this neural network is only applicable to the present agents. The neural network will not operate if we modify the observations in the BigBrotherAgent script or RayPerceptionSensorComponent3D's CollectObservations() method. The neural network outputs match to the actionBuffers argument of the BigBrotherAgent script's OnActionReceived() method.

```
[INFO] BigBrother. Step: 995000. Time Elapsed: 1819.547 s. Mean Reward: 11.532. Std of Reward: 0.079. Training.
[INFO] BigBrother. Step: 1000000. Time Elapsed: 1829.051 s. Mean Reward: 11.560. Std of Reward: 0.044. Training.
[INFO] Exported results\Thesis_training_2\BigBrother\BigBrother-999962.onnx
[INFO] Exported results\Thesis_training_2\BigBrother\BigBrother-1000523.onnx
[INFO] Copied results\Thesis_training_2\BigBrother\BigBrother-1000523.onnx to results\Thesis_training_2\BigBrother.onnx.
```

Figure 71 NN Model export

The next stage is to give the agent a brain and have it begin to function and make decisions using the neural network that we've trained. As described in Chapter 4, we built a folder to hold neural networks exported from training sequences. To import the brain into the BigBrother, we must first copy and paste the.onnx file from the results folder into the asset NNModels folder. The file is then dragged and dropped into the Model area of BigBrother's inspector tab.

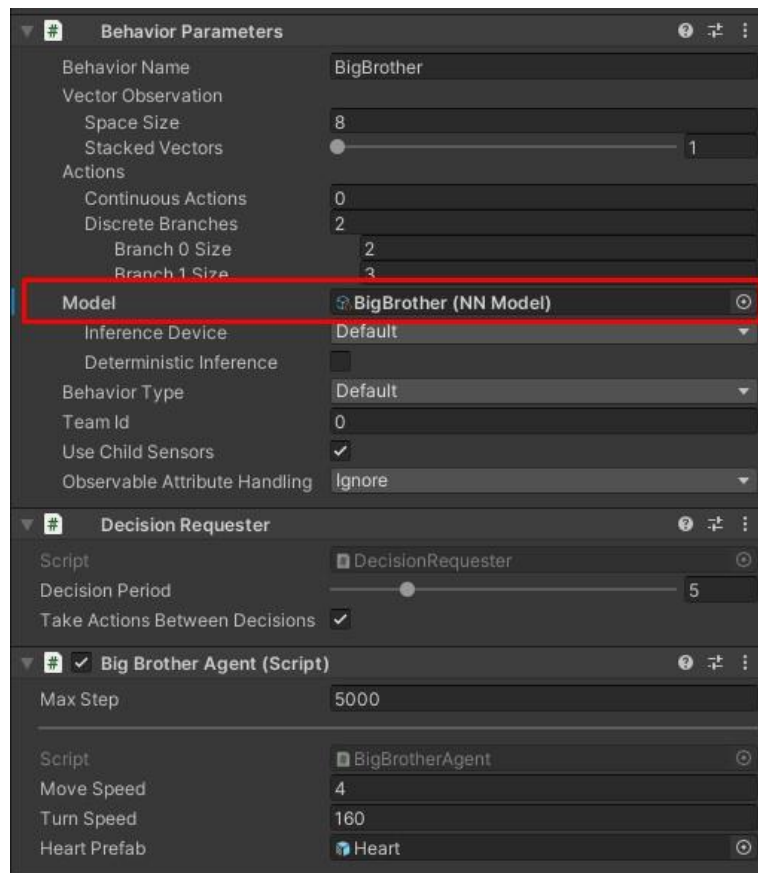


Figure 72 Behavior Parameters inspector tab

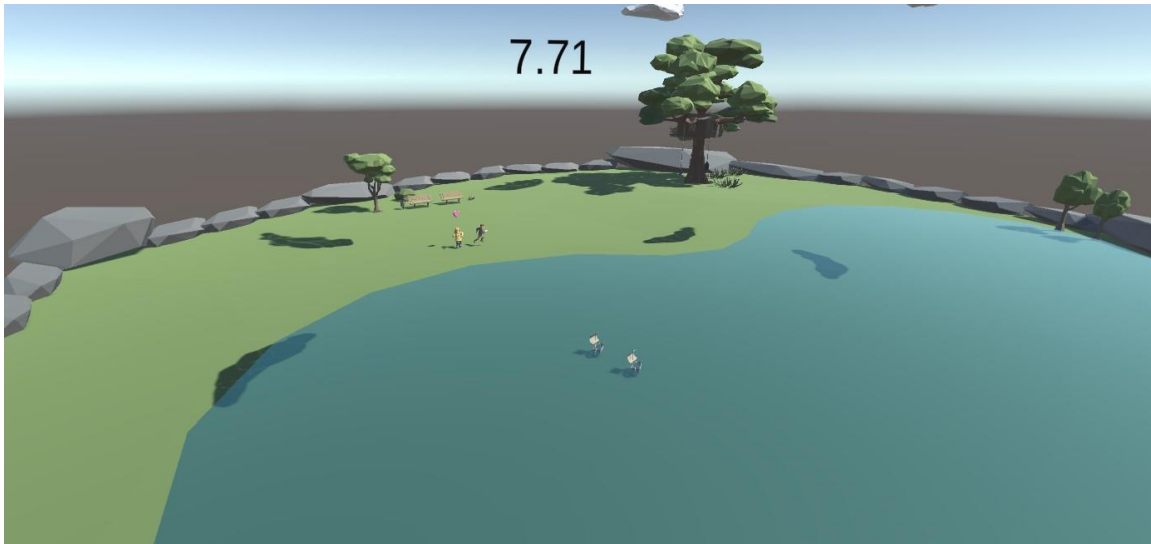


Figure 73 Training with inference

The agent is now gathering boats and giving them to his younger brother. This process is known as "inference," and it denotes that the neural network makes the judgments. The agent makes observations and responds to decisions by doing actions.

As a result, we have successfully thoroughly taught ML-sibling Agent's to collect boats and deliver them to his younger brother.

7. Final Words

7.1 Future Work – Conclusion

This project provided an introduction to the possibilities of Unity's ML-Agents. To say the least, it's fantastic to investigate how to design a simple model, train it with certain setups, and see it learn on its own how to interact with the given environment. Future work has a lot of potential, especially for upgrades, however the following areas can make this project stand out.

Curriculum learning

Curriculum learning is a method of training a machine learning model that gradually introduces more challenging components of a problem such that the model is constantly optimally challenged.

Cooperative behaviors

ML-Agents has capabilities for training cooperative behaviors, which are groups of agents working together to achieve a shared objective, with the success of the individual being related to the success of the entire group. In such cases, agents are often rewarded as a group. For example, if an agent team defeats a rival team, everyone gets awarded, including agents who did not actively contribute to the victory. This makes figuring out what to do as an individual challenging because you can win for doing nothing and lose for doing your best.

We have always been captivated by technical developments and fiction as humans, and we are now living in the middle of the biggest breakthroughs in our history. Artificial intelligence has surfaced as the next big thing in technology. Organizations all across the world are developing game-changing advances in artificial intelligence and machine learning. Artificial intelligence is not only influencing the future of every sector and human being, but it is also the primary driver of developing technologies such as big data, robots, and IoT. Given its rate of expansion, it will remain to be a technical pioneer for the years ahead. As a result, there are several chances for qualified and certified individuals to pursue a satisfying career. (1)

With new capabilities such as autonomous character growth, learning, and adaptability, the effect of AI in the game business is projected to rise even further. The fundamental concept is to create games with agents who are not static but grow as the game is played. Future NPCs will be able to grow throughout gameplay, making it more difficult to predict their actions. AI-powered games will grow more complex and difficult to anticipate as played duration increases. As a result, the games' play-life will be greatly prolonged and AI techniques that enable these opportunities will likewise get more sophisticated. (2)

8. References

1. <https://www.mygreatlearning.com/blog/what-is-artificial-intelligence/>. November 2022. Blog Article. Last visited Friday 25th November 2022.
10. <https://medium.com/@vedantchaudhari/goal-oriented-action-planning-34035ed40d0b>. 12 December 2017. Blog post. Last visited 25th November 2022.
11. <https://www.massive.se/blog/inside-massive/gamedev/myths-in-video-game-ai/>. 30 November 2021. Blog post. Last visited 25th November 2022.
12. <https://www.guru99.com/machine-learning-tutorial.html>. Updated 19 November 2022. Blog post. Last visited 25th November 2022.
13. <https://www.guru99.com/reinforcement-learning-tutorial.html#reinforcement-learning-algorithms>. Updated 19 November 2022. Blog post. Last visited 25th November 2022.
14. <https://analyticsindiamag.com/5-best-game-engines-for-developers-to-build-ai-games/>. 7 August 2019. Blog post. Last visited 25th November 2022.
15. <https://analyticsindiamag.com/everything-you-need-to-know-about-machine-learning-in-unity-3d/>. 15 August 2020. Blog post. Last visited 25th November 2022.
16. https://blog.unity.com/technology/how-eidos-montreal-created-grid-sensors-to-improve-observations-for-training-agents?utm_source=pocket_mylist. 20 November 2020. Blog post. Last visited 25th November 2022.
17. https://blog.unity.com/technology/ml-agents-plays-dodgeball?utm_source=linkedin&utm_medium=social&utm_campaign=ml_global_generalpromo_2021-07-12_ml-agents-dodgeball. 12 July 2021. Blog post. Last visited 12 November 2022.
18. <https://blog.unity.com/games/made-with-unity-soccer-robots-with-ml-agents>. 6 August 2021. Blog post. Last visited 6th November 2022.
19. <https://blog.unity.com/technology/unity-ai-2021-interns-accelerating-learning-with-the-ml-agents-toolkit>. 30 September 2021. Blog post. Last visited 25th November 2022.
2. <https://pixelplex.io/blog/how-ai-enhances-game-development/>. 2 November 2021. Blog post. Last visited 25th November 2022.
20. <https://www.immersivelimit.com/tutorials/reinforcement-learning-penguins-part-1-unity-ml-agents>. 11 December 2020. Tutorial. Last visited 25th November 2022.
21. https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/Installation.md. 14 January 2022. GitHub Repository. Last visited 25th November 2022.

22. https://github.com/Unity-Technologies/ml-agents/blob/release_19_docs/docs/Training-Configuration-File.md. 15 December 2021. GitHub Repository. Last visited 25th November 2022.
3. https://gameworldobserver.com/2022/02/02/how-ai-will-revolutionize-video-games?utm_campaign=Feb_Daily%20Posts&utm_content=197774373&utm_medium=social&utm_source=linkedin&hss_channel=lcp-19153113. 2nd February 2022. Blog post. Last visited 25th November 2022.
4. <https://modl.ai/top-5-most-influential-fps-games-for-ai/>. 4 July 2022. Blog post. Last visited 25th November 2022.
5. <https://www.microsoft.com/en-us/research/project/project-paidia/>. September 2020. Research article . Last visited 25th November 2022.
6. <https://gameworldobserver.com/2019/12/25/ai-honor-kings>. 25 December 2019. Blog post. Last visited 25th November 2022.
7. <https://gameworldobserver.com/2021/03/30/naughty-dogs-former-tech-art-director-discusses-his-promethean-ai-technology-designed-to-help-build-virtual-worlds>. 30 March 2021. Blog post. Last visited 25th November 2022.
8. <https://modl.ai/the-nemesis-system-of-shadow-of-mordor/>. 14 October 2022. Blog post. Last visited 25th November 2022.
9. <https://www.toptal.com/unity-unity3d/unity-ai-development-finite-state-machine-tutorial>. n.d. Blog post. Last visited 25th November 2022.