5-2020

Applying Imitation and Reinforcement Learning to Sparse Reward Environments

Haven Brown

Applying Imitation and Reinforcement Learning to Sparse Reward Environments

An honors thesis submitted in partial fulfillment
of the requirements for the degree of
Bachelor of Science in Computer Science with Honors

by

Haven Brown
University of Arkansas
Candidate for Bachelors of Science in Computer Science, 2020
Candidate for Bachelors of Science in Mathematics, 2020

May 2020
University of Arkansas

This honors thesis is approved for recommendation to the CSCE thesis defense committee.

David Andrews, PhD
Dissertation Director

John Gauch, PhD
Committee Member

Lora Streeter, PhD
Committee Member

Abstract

The focus of this project was to shorten the time it takes to train reinforcement learning agents to perform better than humans in a sparse reward environment. Finding a general purpose solution to this problem is essential to creating agents in the future capable of managing large systems or performing a series of tasks before receiving feedback. The goal of this project was to create a transition function between an imitation learning algorithm (also referred to as a behavioral cloning algorithm) and a reinforcement learning algorithm. The goal of this approach was to allow an agent to first learn to do a task by mimicking human actions through the imitation learning algorithm and then learn to do the task better or faster than humans by training with the reinforcement learning algorithm. This project utilizes Unity3D to model a sparse reward environment and allow use of the mlagents toolkit provided by Unity3D. The toolkit provided by Unity3D is an open source project that does not maintain documentation for past versions of the software, so recent large changes to the use of the mlagents tools in code caused significant delays in the achievement of the larger goal of this project. Therefore, this paper outlines the theoretical approach to the problem and some of its implementation in Unity3D. This will provide a comprehensive overview of the common tools used to train agents in sparse reward environments particularly in a video game environment.

Contents

Chapter 1

Introduction

1.1 Motivation

The job of a computer scientist is to automate tasks and optimize solutions. Advances in computer hardware that allow users to perform rapid calculations have added another tool to the computer scientist's arsenal: machine learning. Utilizing some mathematical techniques that are centuries old and some developed for the specific use of machine learning with modern computers has allowed the scientific community to create computer agents capable of doing tasks humans would usually perform such as driving cars, solving puzzles, playing games, and improving systems. What some results have shown, and the ultimate goal of most working in machine learning, is that with enough training machine learning agents can even perform these tasks better than humans.

A common approach to training agents is *reinforcement learning* which utilizes a system of rewards to help the agent decide which actions should be repeated. Designing a good rewards based system is a challenging science in and of itself. A cautionary tale in the community that has been repeatedly reported by news sources is that of the self-driving car that hits a pedestrian. An example of this occurred in Arizona with a self-driving Uber. [1] These incidents occur due to a lack of thorough testing of the agents before releasing them onto the streets and because of a poorly defined reward system. For many of these vehicles, the reward system is based on time, so the agent may recognize a negative reward for hitting a pedestrian, but that isn't great enough to counteract the positive reward it will receive for reaching its destination thirty seconds faster. This project's reward system is based on the time it takes to reach the end goal and whether the end goal was reached, since the agent in this project is simply playing a game that doesn't contain any moral obstacles to consider.

A challenge facing every problem reinforcement learning attempts to solve is that it can take days, months or even years to train an agent to do a task as well as a human. However, for such use cases as optimizing a supply chain, the goal of the agent is to do better than humans in order to guide humans to make better decisions for these systems. For environments where agents have to make many decisions before receiving a reward or feedback on their decisions, this training time grows exponentially. So, this is the question facing scientists across the machine learning community right now: How can the training time of reinforcement learning agents be shortened?

1.2 Contributions of this report

The objective of this project was to use pre-existing tools in Unity's package *ml-agents* to train an agent first with *behavioral cloning*, an algorithm that allows an agent to learn to play a game from human demonstrations, then with a reinforcement learning algorithm. [2] The current reinforcement learning algorithm in Unity uses random actions starting out.

1.3 Foreshadow

This project focused on shortening training time of reinforcement learning models within the Unity environment, since this is an environment more organizations are using to model problems. The *Background* section of this paper will discuss the solution that inspired this one and the solution to this problem presented by Unity3D. In the next section, *Creating a sparse reward environment* will further discuss what a sparse environment is, its importance as an area of study within the machine learning community, and how one was modeled for testing the solution this project sought to create. *Current models in ml-agents* elaborates on the imitation learning and reinforcement learning algorithms in theory and their implementations in ml-agents. *Solution* presents a solution that bridges the gap between the behavioral cloning and reinforcement learning algorithms in ml-agents. The following section *Conclusion and Future Works* will review the results of the project and areas within the project where further research would be beneficial.

Chapter Bibliography

[1] Daisuke Wakabayashi. Self-driving uber car kills pedestrian in arizona, where robots roam. `https://www.nytimes.com/2018/03/19/technology/uber-driverless-fatality.html`, March 2018.

[2] A Juliani, V Berges, E Vckay, Y Gao, H Henry, M Mattar, and D Lange. Unity: A

general platform for intelligent agents. https://github.com/Unity-Technologies/ml-agents, 2018.

Chapter 2

Background

Reinforcement learning in sparse reward environments is a prolific research area within machine learning. Many environments researchers, businesses, and technology enthusiasts hope to one day put machines in sparse reward environments. For that reason, researchers are now developing methods of training agents in such environments. These methods are constantly being improved upon to help agents learn faster, become more precise, and develop the ability to multitask. These environments are often modeled in video games.

DeepMind, a company owned by Google, is a leader in this area of machine learning. Though many techniques this team uses to train agents are not novel to the machine learning community, DeepMind has pulled these techniques together to create agents that can beat humans in games and manage complex systems. The team's work on AlphaStar has inspired past projects I have worked on in industry which in turn presented problems I sought to address in this project. AlphaStar is the name of the agent DeepMind created to play the game StarCraft II. StarCraft II is a real-time strategy game that presents the player with a complex system to create and manage, while attempting to complete other various goals on a map, such as defending certain objects or destroying others in the game. This game has a decent learning curve for even human players to manage. There's so much going on in the environment that learning how to set up a supply chain while making sufficient progress toward a goal is quite challenging. For that reason, training agents to play this game is a major feat. However, in December 2018, AlphaStar was trained well enough to beat a professional player at StarCraft II. [1]

Figure 2.1: A snapshot of AlphaStar playing StarCraft II. [1]

Training agents to manage complex systems like those presented in StarCraft II is a challenging and useful technology, so the results of AlphaStar were quite encouraging to the machine learning community. When DeepMind finally published some of their methods used to train AlphaStar, it provided a new model for training agents in sparse reward environments. DeepMind has been working on AlphaStar for years and some of the methods employed to train the agent are specific to StarCraft II, but many of the methods can be generalized. AlphaStar first trained on supervised data of humans playing StarCraft II. The agent then trained on a reinforcement learning algorithm with a complex reward system. A neural network was created that could take in raw data and give out instructions. The details of the architecture of the neural network employ more interesting machine learning techniques, but simply put the agent was trained with a supervised learning algorithm and then an unsupervised learning algorithm.[1] Supervised learning algorithms are algorithms that learn from labeled data. Unsupervised algorithms find patterns in unlabeled data.[2]

Unity3D represented these supervised and unsupervised algorithms in the form of imitation and reinforcement learning algorithms respectively. While the methods employed to train AlphaStar are freely available to the research world, the details are not. To get results like those in AlphaStar, researchers must create their own environments to test in and employ their own algorithms even if they mimic the work of AlphaStar. Unity3D is a great tool to do that with because it is designed for video game development and it has machine learning tools provided in the open source project ml-agents that is managed by engineers at Unity3D. Ml-agents provides imitation and reinforcement learning algorithms that can train agents out of the box. The problem with this implementation is that an agent cannot be trained on an imitation learning algorithm and then a reinforcement learning algorithm as was the case in AlphaStar. This is the case because the neural networks generated by the two algorithms are incompatible.

The ml-agents team at Unity3D set out to solve this problem in response to community feedback. The solution was originally supposed to be integrated into ml-agents in late 2018 or early 2019 according to the ml-agents team on their Github project page. When

I started this project last August, no solution was yet available. However, a solution was released with an update to ml-agents in November 2019. [3]

The ml-agents team utilized an algorithm called Generative Adversarial Imitation Learning (GAIL) to bring together the best from the imitation learning algorithm and the reinforcement learning algorithm. They found the algorithm in a research paper entitled *Generative Adversarial Imitation Learning* by Jonathan Ho and Stefano Ermon. [4] This algorithm works by creating a discriminator to return rewards to the agent. The discriminator takes in observations of the environment from the agent and the agent's action given those observations, it returns a reward to the agent based on how close the agent's actions are to the prerecorded demonstrations of gameplay given to the model until the agent finds an environmental reward higher than the rewards found in the demonstrations. In summary, the GAIL algorithm rewards the agent for choosing actions like those in the demonstrations until the agent can choose better actions. They created an entirely new tool to allow for both imitation and reinforcement learning within their platform. [3]

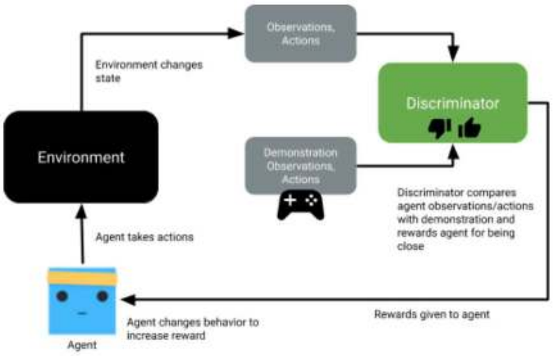

Observations, Actions
Environment changes state
Discriminator
Environment
Demonstration Observations, Actions
Discriminator compares agent observations/actions with demonstration and rewards agent for being close
Agent takes actions
Rewards given to agent
Agent changes behavior to increase reward
Agent

Figure 2.2: Above is a visual representation of the GAIL model. [3]

I have yet to test the GAIL tool in ml-agents since the latest version of ml-agents is incompatible in a lot of features with the version this project has used since the outset of this project, but the tool is at least available now. The great thing about problem solving is that there are many ways to solve a problem. The solution Unity3D put forth is usable within their framework and brings the strengths of imitation and reinforcement learning together, but since ml-agents is open source and built upon the Tensorflow library with python integration, programmers can create countless other solutions to this problem without having to reinvent the wheel.

Chapter Bibliography

[1] Alphastar: Mastering the real-time strategy game starcraft ii. https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii, January 2019.

[2] Jason Brownlee. Supervised and unsupervised machine learning algorithms. https://machinelearningmastery.com/supervised-and-unsupervised-machine-learning-algorithms/, March 2016.

[3] Training your agents 7 times faster with ml-agents. https://blogs.unity3d.com/2019/11/11/training-your-agents-7-times-faster-with-ml-agents/, November 2019.

[4] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. *CoRR*, abs/1606.03476, 2016. URL http://arxiv.org/abs/1606.03476.

Chapter 3

Creating a sparse reward environment

3.1 What is a sparse reward environment?

A sparse reward environment is an environment that requires an agent to make many choices before it receives a reward. [1] Due to the nature of such environments, training agents takes a really long time. Imagine you have just a three by three grid and the goal of the agent is to move from the bottom left square to the top right square with discrete jumps. If you are making only moves toward the goal state, then there are 6 possible direct paths that take you from start to finish. However, there are many indirect paths that you could also take. At any point you could wait at the wrong spot, move away from the goal, or choose an action that is correct, but not the best possible action. These missteps are behaviors some programmers don't allow from agents. This isn't the best approach though since it may force an agent to take a path that is less optimal later on, and it inhibits the learning process of the agents. Humans learn the most from mistakes, so can machine learning agents. The only behaviors that programmers should consider restricting are actions that are against the rules. An example of this in this project where an agent is navigating a maze might be jumping over walls, which may be technically possible, but is not how the environment is intended to be used.

The sheer number of direct and indirect paths an agent could take in even small environments to a sparse reward is so great that training agents to do as well as humans at certain puzzles or tasks can take days or even weeks. To shorten this training time many programmers may be tempted to create a series of small rewards to shape an agent's policy. While this approach may be effective short-term, it is often a slippery slope. Firstly, in the best case scenario, when an agent is fully trained by a programmer's gerrymandered reward system, it is only able to perform a task by the programmer's policy not by the most efficient route. There is a cap on how well the agent can do the task and that cap comes much sooner than it would in an environment where it is only rewarded for completing a task and only punished for making fatal mistakes (such as a car hitting a person). The second big issue with creating many rewards to shape a policy is that there are few policies that are always correct. For example, if an agent is rewarded every time it physically moves closer to the goal in the maze environment then it will run the risk of becoming blocked by a wall. Sometimes agents in these complex environments have to make moves that aren't intuitive to reach the goal most efficiently. Lastly, highly complex reward systems are not robust to changes presented in the environment. In complex environments, agents should be able to handle new challenges and unexpected situations. Agents can't do this if programmers have not designed reward systems that allow for such situations. It's better long term to allow agents to shape their own policy by keeping the simplest possible reward system. There are other ways to shorten training time for agents that don't upset the integrity of the agent's policy. For example, in the maze environment the only reward is the goal at the end. If training yielded an agent that doddled, going down wrong paths and waiting around a lot, then I might make the time it took to finish the maze a factor in the reward at the end.

(a) Mario

(b) Vizdoom

Figure 3.1: Here are two examples of sparse reward environments that other research projects have used to study reinforcement learning. On the left, the video game Mario is depicted. [2] On the right, the platform VizDoom is depicted. Vizdoom is a platform designed for testing reinforcement learning algorithms. It is modeled after the retro video game Doom. [3]

A sparse reward environment can take a long time to train in, but many real world problems are sparse reward environments. Some examples include driving a car from one destination to another, certain video games, and cooking a dish. In the driving scenario the reward will be safely arriving at the final destination. Certain video games that depend on one player beating another such as StarcraftII, Call of Duty, or Mortal Kombat don't yield a reward until the end of the game round when a victor is declared.

Finally, cooking a dish requires getting out ingredients, mixing them in a certain order, and waiting the right amount of time before one discovers whether the food is edible or not. To determine whether a problem could have a sparse reward, consider how many sequential steps an agent would need to take before receiving feedback on their performance. This thought experiment yields that most real world problems are examples of sparse reward environments.

3.2 Maze Environment

Since the focus of this project was to create a generic solution to the given problem, I wanted to create a simple and scalable environment for training agents. The simplest environment I could think of that requires players to make many decisions before reaching a goal was a maze. A maze environment offered several advantages for creating a general purpose solution to the problem this project sought to solve. Since there are already algorithms for creating random mazes, I could create a scalable environment using a maze. This is useful for *curriculum training* which is a technique in machine learning that allows the agent to train in increasingly difficult environments. [4] Curriculum training will be expanded upon in the *Current models in ml-agents* section of this paper.

The other big advantage of the maze environment is that it can be reused for training agents from various perspectives. There are two ways we think about mazes: as a puzzle in the Sunday paper and a life sized space people navigate through for fun in fall festivals and the like. The structure of both these ways we think of mazes are the same, but the players in each of these games must approach their task differently. The maze environment, while very simple, offered a wide variety of possibilities in training agents. It's a good environment not just for this project, but for testing new machine learning brains and agents in the future.

There was more literature than I was expecting on maze generation. It shouldn't have been a surprise though: people have been making mazes as far back as the ancient Egyptians in 500 B.C.E.![5] Maze generation turned out to be a problem solved using graph theory. There were many proposed algorithms such as a recursive division method, randomized walks through a set of walls, and a few greedy algorithms.[6] I used the algorithm I was most familiar with which was a simple depth first search.

In this method, I created data structures called board spaces and linked the north, south, east, and west walls to it along with its position on the board. The algorithm is given starting and ending positions. It marks the starting space as visited and selects a random unvisited neighbor to visit next. The algorithm then deletes the wall between the current cell and the unvisited neighbor, pushes the selected unvisited neighbor on the stack, and marks it as visited. It proceeds by popping the cell off the stack and repeating this process until every cell is visited. Since every cell is visited, the entire maze is connected which is to say that between any two spaces on the board, there is some unblocked path.

The article I read proposed some drawbacks to each algorithm. Some are easier to solve than others because of the nature of the algorithm. For example, depth first search yields long corridors as you can see in Figure 3.2 which may make the puzzle simpler to solve.

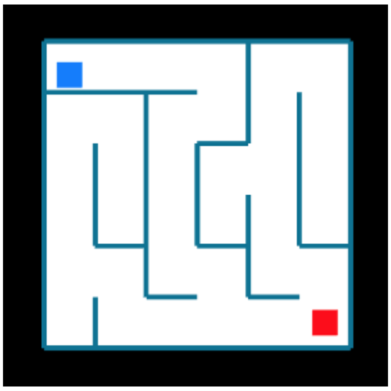


Figure 3.2: A screenshot of a six-by-six maze generated using depth first search in Unity 3D.

Recursive division may prove more difficult to solve since the player essentially has to solve many sub-mazes before entering the final sector that leads to the reward. Figure 3.3 illustrates how recursive division works for this problem. Further research into how agents fare in mazes generated by different algorithms may prove interesting and further prepare an agent to navigate new environments.

Another key decision in training a machine learning agent is what information to give the agent. There are two perspectives games are played in typically: the birds eye view

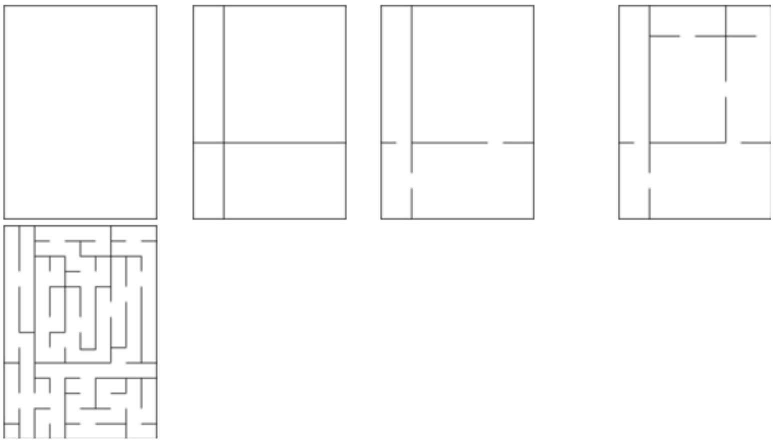


Figure 3.3: A demonstration of maze generation using recursive division. [7]

and the first person perspective. The perspective the player has affects every decision they make, thereby training the player to play the game a certain way. Programmers need to be able to decide which perspective to use and how much information to reasonably give the agent based on the problem the agent is supposed to solve. In general, the agents should have any information a human player might have. This requires some thinking though, since there are assumptions humans make about environments based on prior experience that the agent may not posses. For example, in the *Antigraviator* example put forth by Unity depicted in Figure 3.4, the human understands that along the edges of the road are walls that slow or stop the user in the race. [8]

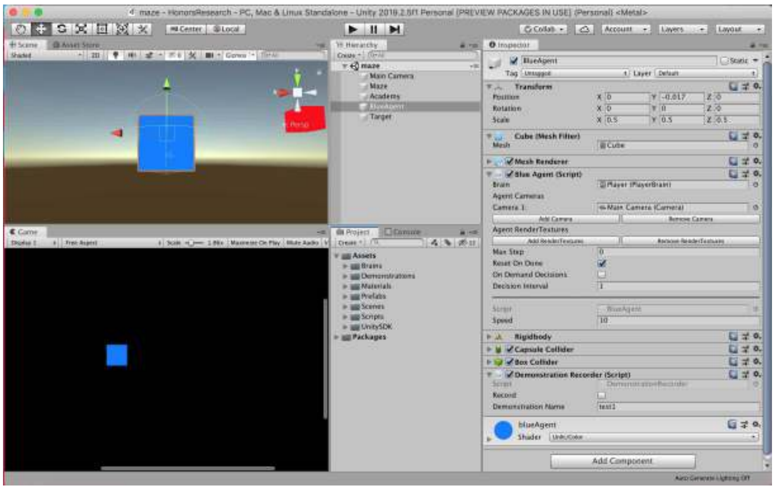
Figure 3.4: This photo was taken to demonstrate Unity's Imitation Learning algorithm. In the article it is presented in it is used to illustrate information humans unconsciously take in that needs to be provided to the machine. [8]

To save the agent countless training hours to teach it what visual information identifies a wall, the programmers gave it several raycasters to allow the agent to gauge the distance between it and the wall. It gives the agent "eyes" like humans have. For those unfamiliar with Unity, *raycaster* is a data structure in Unity that has a starting point, length, and direction. It contains information regarding whether the line hits something, how far out it is, and some other useful information to game programmers. In my summer internship I worked on a project interested in managing supply chains, so I wanted to give my agent in this project a view of the whole board since in a supply chain agents are managing small parts of a system with more information than it may need at each decision interval. However, an agent being trained to do a household task or drive a car, should navigate the environment in first-person like it would in real life. The maze offers a great opportunity to train agents to solve a problem from two very different perspectives, which allows for its reuse in future projects and testing.

3.3 Overview of Unity and ml-agents

Many dismiss video games as simply entertainment or a way for people to blow off steam and waste time. Video games are often overlooked as tools for research in all fields, including machine learning. However in recent years, the machine learning community is seeing how useful video games can be to research and development. Video games allow the real world to be modeled so agents can be trained to do an action without having real world consequences.

Unity3D is a popular game engine available for free for personal and educational use and at a low cost for professional use compared to other software necessary for game design. It allows programmers to model 2D and 3D environments and make those environments interactive. The integration of machine learning into the Unity3D game engine in the form of the open source project ml-agents allows programmers not only the ability to model and play through real world spaces and systems, but also to train agents in those spaces. Within a video game environment, agents can train to do a task while programmers can constantly reevaluate how the environment and reward structure affect the agent's policy. The open source project ml-agents is a comprehensive set of tools to perform machine learning tasks. It integrates Tensorflow with the Unity3D game engine to give developers access to a treasure trove of modern machine learning tools and algorithms. Since this project is entirely open source, programmers can build off that existing toolkit to solve new problems. Some tools available in ml-agents include reinforcement and imitation learning algorithms, curriculum learning, and parallelization of gameplay. These tools can make a diverse set of agents and allow for those agents to be trained faster than using one learning algorithm alone.

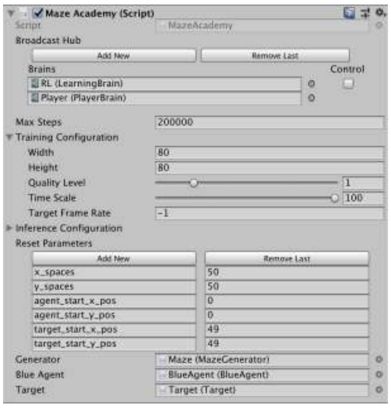


☑ BlueAgent □ Static
Tag Untagged Layer Default
Transform
Position X 0 Y -0.017 Z 0
Rotation X 0 Y 0 Z 0
Scale X 0.5 Y 0.5 Z 0.5
Cube (Mesh Filter)
Mesh Cube
☑ Mesh Renderer
☑ Blue Agent (Script)
Brain Player (PlayerBrain)
Agent Cameras
Camera 1 Main Camera (Camera)
Agent RenderTextures
Max Step 0
Reset On Done ☑
On Demand Decisions □
Decision Interval 1
Script BlueAgent
Speed 10
Rigidbody
☑ Capsule Collider
☑ Box Collider
☑ Demonstration Recorder (Script)
Script DemonstrationRecorder
Record □
Demonstration Name test1
blueAgent
Shader Unlit/Color

Figure 3.5: This is a snapshot of the Unity3D editor itself. It shows the project folder, where all the resources in the project can be found; the inspector, where resources that are in the scene can be manipulated; the scene view which is a 3D rendering of the entire world space; the game view that the user or agent will see; and the hierarchy, where new resources can be added to the scene and arranged.

Ml-agents has a hierarchical structure with the academy at the top, followed by the brain, and lastly the agent. Each scene in Unity3D can have only one academy. The academy manages the environment with reset parameters for when a goal is reached and access to brains being trained in the scene. With an intuitive name, the academy is where brains are trained.

Brains represent the algorithm being used for an agent. The editor representation of the reinforcement learning brain for the maze environment is given in Figure 3.7 to give the reader a simple view of the components of a learning brain in ml-agents. A reinforcement learning brain in a scene is being trained using a PPO model which will be further discussed in *Current models in ml-agents*. The brain requires parameters such as the action space and the information it can expect to receive from the agent. This information can come in the form of a vector observation (such as a position) or a visual



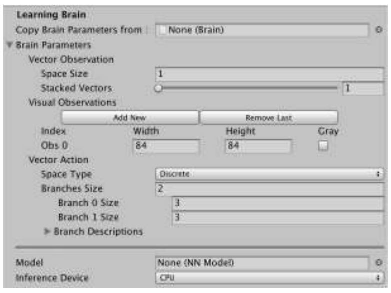
☑ Maze Academy (Script)
Script: MazeAcademy
Broadcast Hub
Brains — Control
RL (LearningBrain) ☐
Player (PlayerBrain)
Max Steps: 200000
Training Configuration
Width: 80
Height: 80
Quality Level: 1
Time Scale: 100
Target Frame Rate: -1
Inference Configuration
Reset Parameters

x_spaces	50
y_spaces	50
agent_start_x_pos	0
agent_start_y_pos	0
target_start_x_pos	49
target_start_y_pos	49

Generator: Maze (MazeGenerator)
Blue Agent: BlueAgent (BlueAgent)
Target: Target (Target)

Figure 3.6: The academy manages the learning environment.

observation rendered from a camera in the scene. It can take longer for brains to learn using visual observations since there is a lot of data for the model to sort out. Whatever information the programmer may choose to give the brain, they must stick to it due to the nature of learning brains in ml-agents. These brains create neural networks and the size of the neural network for each model is determined by a variety of factors including the vector and visual observations. This makes it particularly challenging to add features after training. Of course, one can always retrain when an expanded set of observations is desired, but this is always a time consuming task. Upon completion of training periods, brains create neural networks that can be used again in new brains to perform tasks or continue training.



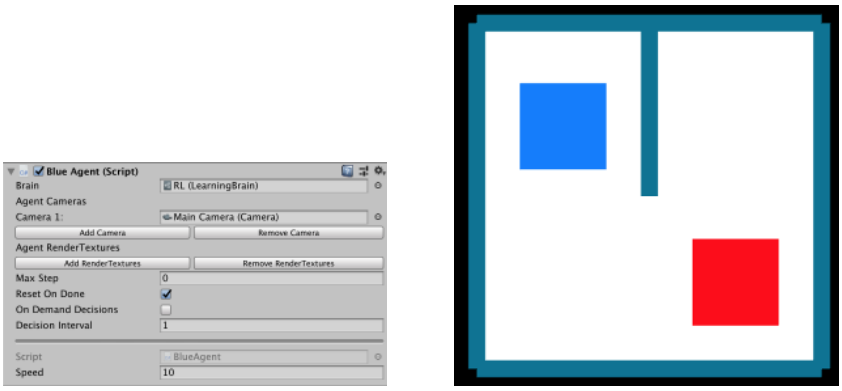
Learning Brain
Copy Brain Parameters from: None (Brain)
Brain Parameters
Vector Observation
Space Size: 1
Stacked Vectors: 1
Visual Observations

Index	Width	Height	Gray
Obs 0	84	84	☐

Vector Action
Space Type: Discrete
Branches Size: 2
Branch 0 Size: 3
Branch 1 Size: 3
Branch Descriptions
Model: None (NN Model)
Inference Device: CPU

Figure 3.7: The brain is the machine learning model being trained.

At the bottom of the hierarchy is the agent. The agent is like a piece in a game. It should have a set of actions it accepts which it then performs and reports the results of for every frame. The agent should be designed so that it can be used by a variety of brains including reinforcement learning, human player, imitation learning, and heuristic brains. The agent is also where the reward structure is defined. The agent can add to a reward or set it at the end of a lesson. In the maze environment, the agent sets a reward of 1f (float) when it collides with the target cube at which point it sends the "done" signal to the academy and the academy resets. It is able to move in the four cardinal directions. All agents have a camera and brain. This agent has a speed since it's movement is continuous. It is also possible to use discrete movements with this particular agent, but continuous movement allowed for faster testing and for this environment one type of movement didn't seem more intuitive than the other.



☑ Blue Agent (Script)
Brain: RL (LearningBrain)
Agent Cameras
Camera 1: Main Camera (Camera)
Agent RenderTextures
Max Step: 0
Reset On Done: ☑
On Demand Decisions: □
Decision Interval: 1
Script: BlueAgent
Speed: 10

(a) Editor (b) Maze

Figure 3.8: Here are two representations of the agent in the maze environment. In the Maze figure, the blue square represents the agent and the red square represents the target cube. The Editor figure is the editor representation of the agent script that manages these variables. Public variables can be altered from the editor.

Tying these pieces together: the academy manages the learning environment, the brains are the models being trained, and the agents are performing the tasks and reporting the outcomes of those tasks. This design allows for easy parallelization of training environments to speed up training times. One need only create a parent object that contains the training area (the maze in this project) and the agent as children. A prefab can be created of this environment by dragging the parent object from the hierarchy of the open scene to the project folder. Prefabs are objects in Unity3D that can be reused in scenes. As many of these prefabs as desired can be instantiated given the computing power to handle it. All of the agents then train on the same brain. The academy must be set up to allow agents to reset their environments since there are many agents and only one academy. Unity3D partnered with Jam City games to use parallelization to train an agent to play Snoopy Pop seven times faster than by training on one environment alone.

[9]

Unity3D is a great and free tool to do some great machine learning research. The biggest drawback of this tool is that ml-agents isn't supported as well as other features in the engine. It is constantly evolving and with each update tools are deprecated that are still in use. It's very easy for ml-agents to become out of sync with its dependencies and even the game engine itself. That makes it difficult to use as a reliable tool in industry. A team might work on a project for months, then running the wrong update could break the project or even their lack of understanding of the finer details of the code. Documentation for past releases is almost impossible to find. Hopefully in coming years ml-agents will gain more official support to make it easier to use for large projects. Ml-agents provides some great benefits though, including a large set of machine learning tools, tutorials that allow new machine learning enthusiasts learn about cutting edge techniques in the field, and the whole thing is open source so programmers can create their own tools to integrate into ml-agents on their own machines. On that note, it also gives a great intro to some commonly used tools in machine learning such as Anaconda and Tensorflow. In the next section, some of ml-agents' tools will be discussed further.

Chapter Bibliography

[1] Martin A. Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. Learning by playing - solving sparse reward tasks from scratch. *CoRR*, abs/1802.10567, 2018. URL http://arxiv.org/abs/1802.10567.

[2] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. *CoRR*, abs/1705.05363, 2017. URL http://arxiv.org/abs/1705.05363.

[3] Marek Wydmuch. Vizdoom: Doom-based ai research platform for reinforcement learning from raw visual information. http://vizdoom.cs.put.edu.pl/.

[4] Guy Hacohen and Daphna Weinshall. On the power of curriculum learning in training deep networks, 2019.

[5] Natasha Geiling. The winding history of the maze, Jul 2014. URL https://www.smithsonianmag.com/travel/winding-history-maze-180951998/.

[6] Maze generation algorithm. https://en.wikipedia.org/wiki/Maze_generation_algorithm, March 2020.

[7] it2051229. Maze generator recursive division.

[8] Imitation learning in unity: The workflow. https://blogs.unity3d.com/2018/05/24/imitation-learning-in-unity-the-workflow/, May 2018.

[9] Training your agents 7 times faster with ml-agents. https://blogs.unity3d.com/2019/11/11/training-your-agents-7-times-faster-with-ml-agents/, November 2019.

Chapter 4

Current models in ml-agents

4.1 Imitation Learning

Imitation learning in ml-agents utilizes a technique called behavioral cloning through observation.[1] This technique was developed to mimic how humans often learn: by watching others perform a task. In ml-agents a student and teacher are designated. The teacher is usually a player brain, but it technically can be a heuristic or reinforcement learning brain. The student is a learning brain trained with the behavioral cloning algorithm designated in a YAML file containing the hyperparameters desired for training. There are two ways to train the student brain. The teacher can train the student in real-time as is the case Unity3D gives in the Antigraviator example project demonstrating its imitation learning algorithm.[2] The teacher can also record demonstrations to train the student with later. Unity3D recommends recording about five minutes of gameplay in a demonstration. Of course this depends on the environment the student is learning in, but for the purposes of the maze environment, five minutes was plenty of time. If the student is going to be required to do something in the last five minutes of gameplay than it did in the first five minutes, of course it will need to see that demonstrated.

Agents trained by an imitation learning algorithm learn much faster than those learning by a reinforcement learning algorithm. This is due to the reward structure of the imitation learning algorithm. In reinforcement learning in a sparse reward environment, the agent receives a reward when it reaches the goal. However, in imitation learning, the agent is constantly receiving feedback on how it's doing. The goal of the student in imitation learning is to take the action that the teacher would have taken given the state. The student can correct bad behaviors as it goes and it receives a higher reward the closer it gets to the behavior of its teacher. This shortens the time it takes to train an agent to play through an environment as well as a human significantly compared to training with a reinforcement learning algorithm. The drawback to an agent trained by imitation learning is that it can only be as good as its teacher. Sometimes this is all that is desired. For example, if a programmer wanted to create a video game non-player character that was as challenging to beat as a person, but not as challenging as a well trained reinforcement learning agent, then an agent trained with imitation learning is perfect.

As mentioned before it is technically feasible to designate a heuristic or reinforcement learning agent as the teacher in ml-agents. On a previous project I worked on using ml-agents in industry, my team thought we could use this feature to train an agent to follow certain policies designated by heuristics algorithms. The result was not an agent that took from the best of all the policies it was trained on, but rather an agent that was perpetually unlearning previous policies to learn new ones. The other flaw in this approach is that the performance of the agent is capped by the success of the heuristic or reinforcement learning brain it was trained to mimic.

Imitation learning is one of the simpler features to use in ml-agents. It doesn't require a lot of hyperparameter tuning as the reinforcement learning brain does. The user simply records demos, which are then fed to the behavioral cloning algorithm to train the agent. Training yields a neural network that can be further trained by the behavioral cloning algorithm with new demonstrations later or can simply be given to an agent's brain to dictate how the agent should behave.

4.2 Reinforcement Learning

Another way we learn as humans is by receiving feedback from our environment. When a child touches a hot pan, they learn that touching hot pans hurts, so they don't do it again. When a girl gives her mom flowers, the mom rewards her daughter with affection. We learn how to behave and the consequences of certain actions over the course of many years of reward-based training. Reinforcement learning algorithms apply this concept to training machine learning agents. These algorithms work by rewarding good behaviors and punishing bad behaviors. They are considered unsupervised algorithms since it is learning from data that is unlabeled. The model is shaped by the feedback actions or a series of actions yield. For that reason these models can't determine whether an action is good or bad until they receive feedback. Think back to the story of the self-driving Uber.

If in training, the model dictating the actions of the vehicle had been given a harsh negative reward for hitting another person or car then the model would have learned immediately that such an action shouldn't be performed again.

Training a reinforcement agent in a sparse reward environment is more like training a person to reach a lifelong goal. Consider an elementary school child that wants to be the president one day. The child takes certain actions and makes choices throughout their life to reach this goal, but they can't determine whether they made the right choices until the goal has been reached. The child may grow up, reach the end of their life having not become president yet, and look back to determine what action was wrong. There's no way for the child to know what step was the wrong one though, because there were so many of them. Now imagine that child is making random choices throughout their life to reach this goal. How many lifetimes would the child have to live before even accidentally succeeding? That's what it's like using reinforcement learning in a sparse reward environment.

There are ways to help the agent learn though. The next section will discuss curriculum learning as a method of shortening the training time of a reinforcement learning algorithm. Training with reinforcement learning is difficult because the actions start out random and it takes a long time for the model to get feedback on the actions it took. Something that often happens in training such an agent is that the agent doesn't move any closer to the goal. It might take no action, or it might take actions that have it going in a circle or moving back and forth. This can be really frustrating which led machine learning enthusiasts to apply another tool pulled from how humans learn: curiosity. This is an intrinsic reward.[3] Humans explore and study new things because we receive some satisfaction from the act of discovery. I've heard many teacher refer to this phenomenon as the "a-ha!" moment in their students. The machine learning community has applied this in the form of curiosity. It is used in reinforcement learning algorithms and mostly in sparse reward environment. A curiosity reward seeks to encourage state exploration by rewarding the agent for taking actions that lead to states the agent had yet to see. The addition of a curiosity reward to reinforcement learning agents can dramatically shorten training time by preventing the agent from continually returning to previous states. There are other forms of intrinsic reward briefly mentioned in *Curiosity-driven Exploration by Self-supervised Prediction* by Pathak, Agrawal, et al, but curiosity is supported by ml-agents and effective on its own.[3]

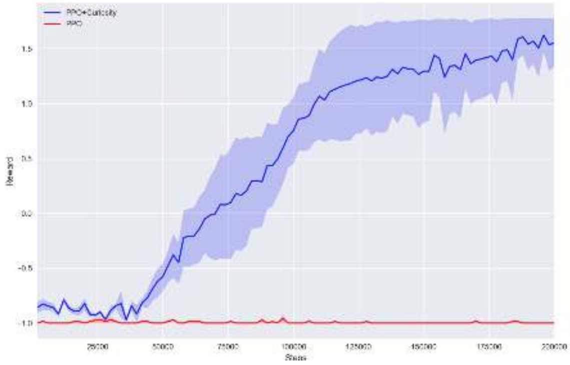


Figure 4.1: I didn't do much testing on the effects of curiosity on reinforcement learning in the maze environment, but here are some results on its effect that Unity found in their own testing environment. [4]

Engineering a good reward system keeps many programmers busy, but it's best to keep reward systems as simple as possible. People often overlook larger consequences of setting policies and it can prevent agents from becoming the best they can be. There is one rule programmers should adhere to in defining a reward structure which is to keep rewards normalized between 0 and 1. Failure to normalize a reward system can result in longer training times or undesirable behaviors in agents.

The reinforcement learning algorithm of ml-agents is a PPO. PPO stands for proximal policy optimization. It utilizes gradient descent and a function that predicts the reward an action might give based on past experiences. This algorithm is the default reinforcement learning algorithm of not only ml-agents but also OpenAI, a prominent AI research laboratory out of San Francisco.[5] There is a ton of research out there on tuning the hyperparameters of algorithms such as this to improve results of training. Hyperparameter tuning gets into the nitty-gritty mathematics of an algorithm. These variables can be adjusted to shorten training time or increase the precision of an algorithm. The process of hyperparameter tuning is tedious and often takes place throughout the training process, being improved upon via trial and error. It may take a long time to train using a PPO, but there are many techniques to shorten that training time and there is no cap on how much the agent can learn.

4.3 Curriculum Learning

Educators use curriculum learning for every class they ever teach. Humans can learn difficult concepts by first learning simpler ones that build to the more complex ideas. Developing a coherent and cohesive curriculum is a constant challenge in education. Lessons should start simple, teaching essential skills in a subject area. They get progressively harder until the student reaches some benchmark or goal. Students aren't thrown straight from the nursery into AP Calculus; they are first taught numbers, then basic arithmetic, and so on until calculus is the next logical step in their education. This approach to learning has been applied to training machine learning agents.

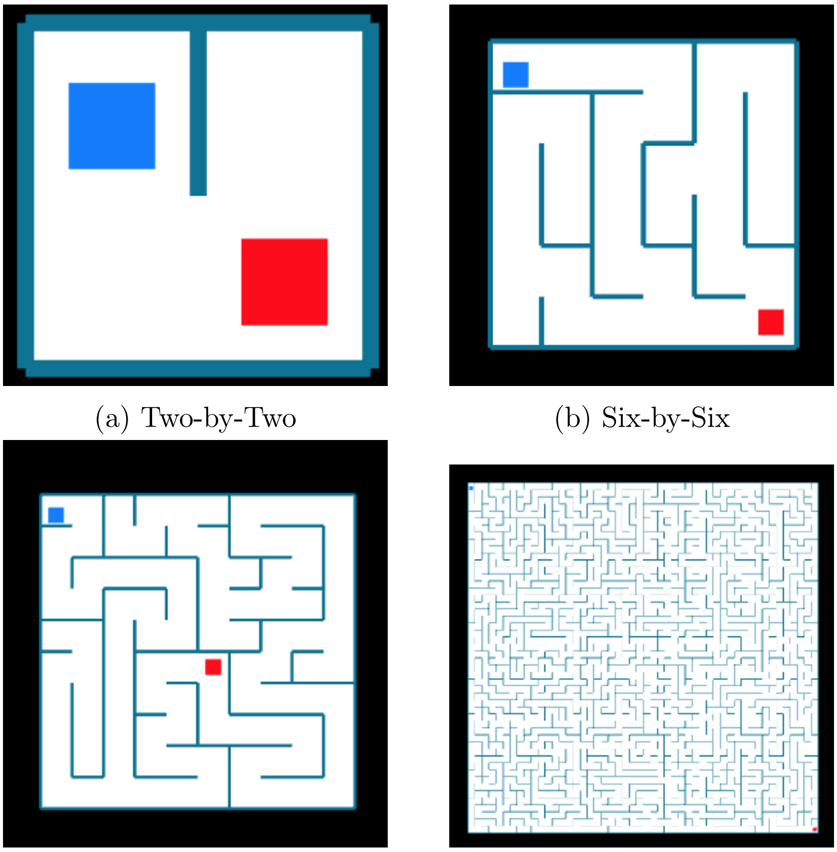
Curriculum learning is not a machine learning algorithm as is the case with the tools described in the previous two sections, but rather a tool to manipulate an environment to help another machine learning algorithm train an agent. As is the case in developing a curriculum for training people, developing a curriculum for training agents can be a challenge. A good curriculum has modules that are progressively more difficult. [6] The term *lesson* indicates a single complete play through of an environment by a machine learning agent in ml-agents, so the term *modules* is used to maintain clarity. The term *module* will be employed here to indicate a change to the environment designed to teach or improve upon a skill or group of skills similar to a chapter of a textbook or a new set of PowerPoint slides in a class. Determining the jump in difficulty between two modules, when to allow an agent to progress to the next module, and how many modules to include in a curriculum is the challenge of developing a good curriculum for an agent. It takes a lot of planning and forethought, but also a lot of trial and error. If an agent moves to a new module and isn't making progress in the changed environment, then at some point it might make sense to add a module between the two modules to ease the transition.

In ml-agents, a curriculum is set forth by a YAML or JSON file.[7] This file indicates how the progress of the agent should be evaluated, when the module should be changed, and how the environment should be changed for each module. The progress of the agent can be evaluated based on a reward threshold or as a percentage of a maximum number of lessons. The reward threshold is set for each module. When the agent's average reward reaches the desired threshold, it transitions to the next module. If progress is measured as a percentage of a maximum number of lessons, then a maximum number of lessons must be set in the academy and the transitions between modules will occur at the threshold percentages given in the curriculum file. Regardless of what determines when a module should change, the process of changing a module is the same. In the academy, a variety of reset parameters are given. Values for these reset parameters are set in the curriculum file and used to shape the environment for the next module. In the maze environment, the reset parameters are the number of squares in the grid, the start position of the agent, and the start position of the target cube. While training, the size of the maze could be increased, or how far the agent starts from the target. This will help the agent to learn the goal is to reach the red cube and the basics of movement before being thrown into a large maze.

Each time the maze environment is created, the maze is generated dynamically using a variable to indicate the number of squares in the x-direction and a variable to indicate the number of squares in the y-direction. The grid is created first and walls are deleted using depth-first search. Generating the maze in code instead of using a predetermined grid size was more challenging than expected, but it allowed for curriculum learning to be easily implemented and it provides an environment with no upper bound on difficulty for an agent to learn in.

This feature has recently undergone some changes in ml-agents.[7] The concept and general structure is the same, but the use of it in code has changed enough that it slowed testing and production since documentation for previous versions of ml-agents is so difficult to find. I have used it in a previous project though, and it does help agents transition from a state of taking random and useless actions to taking actions that help it perform the desired task.

Curriculum learning can be used in conjunction with imitation and reinforcement learning models. Using curriculum learning to record demonstrations for imitation learn-


(a) Two-by-Two

(b) Six-by-Six

(c) Ten-by-Ten

(d) Fifty-by-Fifty

Figure 4.2: Above is an example of how different modules might look in a curriculum for the maze environment. The agent can first train to solve the two-by-two maze, then the six-by-six, ten-by-ten, and finally the fifty-by-fifty maze.

ing models to train from can provide demonstrations containing several environments that look different from each other thereby giving the imitation learning model more action-spaces to learn from. If a curriculum is used to record demonstrations, it should also be used when the imitation learning model is training, to allow the model to learn in similar environments those in which the demonstrations were recorded.

Chapter Bibliography

[1] Faraz Torabi, Garrett Warnell, and Peter Stone. Behavioral cloning from observation. *CoRR*, abs/1805.01954, 2018. URL `http://arxiv.org/abs/1805.01954`.

[2] Imitation learning in unity: The workflow. `https://blogs.unity3d.com/2018/05/24/imitation-learning-in-unity-the-workflow/`, May 2018.

[3] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven exploration by self-supervised prediction. *CoRR*, abs/1705.05363, 2017. URL `http://arxiv.org/abs/1705.05363`.

[4] Arthur Juliani. Solving sparse-reward tasks with curiosity. https://blogs.unity3d.com/2018/06/26/solving-sparse-reward-tasks-with-curiosity/, June 2018.

[5] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

[6] Guy Hacohen and Daphna Weinshall. On the power of curriculum learning in training deep networks. *CoRR*, abs/1904.03626, 2019. URL http://arxiv.org/abs/1904.03626.

[7] Training with curriculum learning. https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-Curriculum-Learning.md, April 2020.

Chapter 5

Solution

The goal of this project was to utilize the short training time of the imitation learning algorithm to create an agent that would have the knowledge to navigate a maze as well as a human player that could then be taught by a reinforcement learning agent to perform better than humans. Since ml-agents is a relatively new project in Unity3D, it changed a lot through the duration of this project and documentation for past versions (including the version I used for this project) was overwritten. The imitation and reinforcement learning algorithms of ml-agents can't be used in series because they have incompatible neural networks. This is because the models have different goals. The neural network developed by the imitation learning algorithm tells the agent what the teacher would have done in a given situation. The neural network developed by the reinforcement learning agent tells the agent which action to take based on the estimated potential reward each action would yield. The agent may not see an extrinsic reward for awhile with a given action, which is part of the calculation in the estimated potential reward. My goal was to create a transition function from the imitation learning algorithm to the reinforcement learning algorithm. Each neural network contains meaningful data about how the agent should behave, which could in theory be interpreted into data the other can understand.

These neural networks are generated using Tensorflow. By studying these two algorithms and the neural networks they produce, I could learn a lot about Tensorflow and how it's integrated with Unity3D. Creating a transition function between these two algorithms would also potentially create a workflow for creating transition functions between other algorithms and even between changing observation sizes. Currently, a developer must decide before training their agent every possible observation and action the agent will ever have available. That means if along the development path, the developer decides to allow the agent another action, then they have to completely retrain their agent from scratch.

Though this solution was not finished for this project, enough development did occur on this project to create and test this solution and others. The results of this project are a sparse reward environment that is expandable with many machine learning tools implemented already. After updating the ml-agents toolkit in my project completely and making some adjustments to the environment to allow for changes to ml-agents, this environment is set up for exploring not only this solution, but many other machine learning techniques. While I would have liked to develop a solid technical solution to the incompatibility between ml-agents' imitation and reinforcement learning algorithms, this project did result in new tools that can be used for the testing and development of machine learning algorithms in a video game environment and in my limited experience in industry, managers and developers tend to underestimate what a large task it is to develop those tools.

Chapter 6

Conclusion and Future Works

6.1 Conclusion

Many real-world problems can be represented as sparse reward environments. Machine learning has the potential to be a powerful tool in solving those problems. The process of applying the power of machine learning to these problems is often trickier to get started on than one might expect. Unity3D is an accessible way to model these sparse reward environments. Environments that mimic those of the real-world can be generated and tested using the Unity3D game engine. Ml-agents helps to enable developers to leverage the power of machine learning within a 3D environment. Developing general purpose tools to shorten the training time of agents in these environments can help to solve many real-world problems with machine learning. Reinforcement learning agents can already learn anything given sufficient training time, but that training time can be so long it isn't a reasonable training algorithm on its own. It could be improved upon by using it in conjunction with imitation learning as shown by the results of AlphaStar. The solution I was seeking in this project was never finished, but luckily ml-agents has put out a new tool to allow developers to leverage both imitation and reinforcement learning in training.

6.2 Future Works

Completing the solution outlined previously would be the next step in this project. Once that solution is complete, it could be compared to the solution put forth by ml-agents to determine which trains faster and which achieves higher accuracy. With some working models complete, it would be interesting to modify the environment and test the agent's performance given changes to the environment. Different maze algorithms could be tested. Enemy agents could be included in the environment. The maze could even change during gameplay. Training agents in a variety of environments creates more robust and intelligent agents. There is a ton of research on improving the performance of reinforcement learning agents. That research can be applied in Unity3D to improve agents, which may further the machine learning and the game development communities.

formstack sign
formerly insuresign

🔒Document Completion Certificate

Document Reference : 4e466749-3e71-4427-9087-a190a9439616
Document Title : Final draft of honors thesis
Document Region : Northern Virginia
Sender Name : Haven Brown
Sender Email : havenbrown1108@gmail.com
Total Document Pages : 32
Secondary Security : Not Required
Participants

1. John Gauch (jgauch@uark.edu)
2. Lora Streeter (lstrothe@uark.edu)
3. David Andrews (dandrews@uark.edu)

Document History

Timestamp	Description
04/30/2020 13:05PM UTC	Document sent by Haven Brown (havenbrown1108@gmail.com).
04/30/2020 13:05PM UTC	Email sent to John Gauch (jgauch@uark.edu).
04/30/2020 13:05PM UTC	Email sent to Haven Brown (havenbrown1108@gmail.com).
04/30/2020 13:41PM UTC	Sender downloaded document.
04/30/2020 14:02PM UTC	Document viewed by John Gauch (jgauch@uark.edu). 72.202.196.21 Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36
04/30/2020 14:02PM UTC	John Gauch (jgauch@uark.edu) has agreed to terms of service and to do business electronically with Haven Brown (havenbrown1108@gmail.com). 72.202.196.21 Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36
04/30/2020 14:02PM UTC	Signed by John Gauch (jgauch@uark.edu). 72.202.196.21 Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36
04/30/2020 14:02PM UTC	Document viewed by John Gauch (jgauch@uark.edu). 34.231.157.157 Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36
04/30/2020 14:02PM UTC	Email sent to Lora Streeter (lstrothe@uark.edu).
04/30/2020 14:02PM UTC	Document viewed by John Gauch (jgauch@uark.edu). 72.202.196.21 Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_6) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36
04/30/2020 14:03PM UTC	Document viewed by John Gauch (jgauch@uark.edu). 108.59.15.70 Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko
04/30/2020 14:06PM UTC	Document viewed by Lora Streeter (lstrothe@uark.edu). 72.204.69.105 Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36

Document History

Timestamp	Description
04/30/2020 14:06PM UTC	Document viewed by Lora Streeter (lstrothe@uark.edu). 34.232.127.140 Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36
04/30/2020 14:08PM UTC	Document viewed by Lora Streeter (lstrothe@uark.edu). 147.255.92.21 Mozilla/5.0 (Windows NT 6.1; WOW64; Trident/7.0; rv:11.0) like Gecko
04/30/2020 15:16PM UTC	Document viewed by Lora Streeter (lstrothe@uark.edu). 72.204.69.105 Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.122 Safari/537.36
04/30/2020 15:16PM UTC	Document viewed by Lora Streeter (lstrothe@uark.edu). 34.231.157.157 Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.129 Safari/537.36
04/30/2020 15:17PM UTC	Lora Streeter (lstrothe@uark.edu) has agreed to terms of service and to do business electronically with Haven Brown (havenbrown1108@gmail.com). 72.204.69.105 Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.122 Safari/537.36
04/30/2020 15:17PM UTC	Signed by Lora Streeter (lstrothe@uark.edu). 72.204.69.105 Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/81.0.4044.122 Safari/537.36
04/30/2020 15:17PM UTC	Email sent to David Andrews (dandrews@uark.edu).
04/30/2020 15:20PM UTC	Sender downloaded document.
04/30/2020 18:18PM UTC	Document viewed by David Andrews (dandrews@uark.edu). 167.224.207.114 Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:65.0) Gecko/20100101 Firefox/65.0
04/30/2020 18:19PM UTC	Document viewed by David Andrews (dandrews@uark.edu). 34.231.157.157 Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:65.0) Gecko/20100101 Firefox/65.0
04/30/2020 18:19PM UTC	David Andrews (dandrews@uark.edu) has agreed to terms of service and to do business electronically with Haven Brown (havenbrown1108@gmail.com). 167.224.207.114 Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:65.0) Gecko/20100101 Firefox/65.0
04/30/2020 18:19PM UTC	Signed by David Andrews (dandrews@uark.edu). 167.224.207.114 Mozilla/5.0 (Macintosh; Intel Mac OS X 10.13; rv:65.0) Gecko/20100101 Firefox/65.0
04/30/2020 18:19PM UTC	Document copy sent to John Gauch (jgauch@uark.edu).
04/30/2020 18:19PM UTC	Document copy sent to Lora Streeter (lstrothe@uark.edu).
04/30/2020 18:19PM UTC	Document copy sent to David Andrews (dandrews@uark.edu).