Agent-Based Machine Learning Analysis & Virtual Development

Lorenzo Ramirez, Jr.

Kettering University

Department of Computer Science

Dr. Michael Farmer

Author Note

This senior thesis is submitted as a partial fulfillment of the graduation requirements of Kettering University, which is needed to obtain a Bachelor of Science in Computer Science. The conclusions and opinions expressed in this thesis are from myself and do not necessarily represent the position of Kettering University or anyone else affiliated with this culminating undergraduate experience.

Although this thesis represents the compilation of my own efforts, I would like to acknowledge and extend my sincere gratitude to the following individuals for their valuable time and assistance, without whom the completion of this thesis would not have been possible: Dr. Michael Farmer, Faculty Thesis Advisor and Professor Guiseppe Turini, Committee Member.

Abstract

Machine Learning (ML), a significant subfield of Artificial Intelligence, encompasses various statistical algorithms designed to learn from data, such as reinforcement learning and deep learning. One modeling approach that incorporates ML models is Agent-Based Modeling (ABM), where an ‘Agent’ A.I. learns to perform tasks by interacting with a virtual or physical environment. These agents can master a diverse range of tasks, from baking the perfect cake to competing in a soccer match against other agents, depending on the complexity of the task, the quality of training, and the sophistication of the model used. This research paper delves into the fundamentals of Machine Learning, explores the applications of ABM, elucidates the learning process of an agent starting from a state of zero knowledge, and provides an overview of developing a custom ABM using the Unity game engine. This paper, written in a simple and accessible manner, aims to fuel the curiosity of those at the beginning of their journey into machine learning and ABM technology and hopefully inspire them to develop their own agents to perform interesting tasks.

Keywords: *A*gent-Based Models, Artificial Intelligence, Machine Learning, Deep Learning, Reinforcement Learning, ML-Agents, Unity

Table of Contents

Agent-Based Machine Learning Analysis & Virtual Development

Machine Learning is a fascinating aspect of the Computer Science field that encompasses various methods through which a computer can learn to have remarkably accurate and precise abilities. Machine Learning should be explained in an easy-to-understand manner so that individuals with basic computer literacy can comprehend just how exciting and valuable Machine Learning is to humanity. Given the vast and constantly updating information in the field of Machine Learning, it is more than what a single paper can cover. Therefore, this paper broadly explains Machine Learning and delves deeper into reinforcement learning models, focusing on Agent-Based Modeling.

Machine Learning

In the digital, data-driven era, machine learning (ML) has become a staple in creating highly advanced and accurate pieces of software. ML-enabled products are often used without understanding the complexity and limitations of the system. From a non-technical perspective, ML can be seen as a miracle to humanity that can automatically solve millions of previously unsolvable issues. While properly trained ML models can accomplish extraordinary feats such as accurately predicting the weather, simulating organic behaviors, and generating impressive imagery, these abilities are not developed overnight using one algorithm or model. Each problem must be carefully analyzed to understand which ML algorithm is appropriate to solve the problem effectively. The number of algorithms used in ML models is growing every year. However, there are a few important ones to note within the deep learning section of ML: supervised, unsupervised, and reinforcement learning.

Forms of Machine Learning

Algorithms, such as supervised learning, unsupervised learning, and reinforcement learning, are among the many different forms a machine learning model can take to accomplish its tasks. Developers must choose the most relevant model for the problem they are facing. A model designed to classify items is not well-suited to interact with environments, so understanding what the algorithms do under the hood can increase the likelihood of success. Each algorithm category is highly complex and has enough content to fill multiple papers, so each will be explained briefly, providing sufficient detail to understand their capabilities and use cases.

Supervised Learning

Supervised learning can recognize relationships within a collection of items, predicting which category an item should be in based on past patterns. A supervised learning model must be trained on an already categorized dataset so the algorithm can learn which patterns relate to which categories through either classification or regression algorithms (IBM Technology, 2022). After training, the algorithm would be tested with a different data set without knowing the answers to evaluate its percentage of success.

Classification algorithms, such as Logistic Regression, categorize data into discrete groups of items (Sarangam, 2021). A classic binary classification problem typically taught to beginners is the classification of cats and dogs. The algorithm is given images of dogs and cats, and its objective is to sort those images into their respective group by analyzing pixel patterns within the images and making predictions on their correct classification.

The Logistic Regression algorithm is suited for classification problems, especially binary classification, to separate the data points into distinct groups linearly. Remember that the logistic regression algorithm is most suited for linear datasets; using it on non-linear datasets will produce poor performance due to the need to separate the data well. The sigmoid function $y = \frac{1}{1+e^{-x}}$, where $x = b_0 + b_1x$ is used to separate the linear data into groups, as illustrated in Figure 1 (Liu, 2019).

Regression algorithms, such as linear regression, predict continuous values like grocery prices or weather forecasts. Regression is used to find the best matching rows to make predictions as close to actual values as possible (Sarangam, 2021). The linear regression algorithm is a simpler version of the logistic regression algorithm. The linear regression equation is $y = b_0 + b_1x$, where b_0 is the intercept and b_1 is the slope coefficient, which is to "predict the value of a variable based on the value of another equation," according to IBM (2021).

Unsupervised Learning

Unsupervised learning is a machine learning algorithm that groups data into classes without human supervision, allowing it to discover hidden patterns within the data (IBM, 2023). The two main methods in unsupervised learning are Association and Clustering. Association is when the algorithm identifies relationships between similar items. For instance, recommendation systems in streaming services use associations to suggest shows and movies to users based on their viewing history (IBM, 2023). Conversely, Clustering is when the algorithm groups similar items together (Sharma, 2024). For example, among millions of images, those containing animals would be grouped separately from those containing people. An example of an unsupervised learning algorithm is the K-Means Algorithm, which groups observations into clusters with the nearest mean.

Reinforcement Learning

Reinforcement Learning uses algorithms that most closely emulate how a human learns through trial and error. It is commonly used to train Agent-Based Modeling to learn through environmental interaction, whether with the physical or digital world. Consider how a baby learns how to walk. At first, they cannot move beyond wiggling and randomly flailing their limbs. As they fail different ways to move more efficiently, they slowly learn how to roll over, crawl, stand on two legs, and finally move their legs one at a time to walk forward. All this while an adult is rewarding the baby for attempting to walk. A baby would typically learn to walk within 10 – 18 months, according to the Pregnancy Birth & Baby website (2022).

Now, teaching an agent-based model to walk on two legs within a virtual world would take a fraction of the time it takes a baby to learn to walk. Like the baby, the agent learns through trial and error, guided by rewards and punishments through a closed-loop system, as illustrated in Figure 3 (Morales, 2020). The first step of the closed-loop system involves the agent perceiving its environment via sensors, including touch, hearing, or vision. The agent can be in different forms of an environment, whether it be a graph, 3D simulations, or in an environment without graphics. For example, the game engine Unity can provide perceptions through the agent's sensors through collision detection, audio listeners, and ray casts for 'vision.' The data that the agent perceives from the environment is input into the deep reinforcement learning policy algorithm to map the perceptions to the agent's actions that are required to achieve those perceived states. Then, the perceived states (“state-action pair”) are mapped to the reward values to estimate the cumulative reward through the value function, such as the Bellman Equation (Ved, 2018). The Bellman Equation is $V_\pi(s) = E_\pi[R_{t+1} + \gamma * V_\pi(S_{t+1})]$, where $E_\pi[R_{t+1}]$ is the expected value of the immediate reward and $\gamma * V_\pi(S_{t+1})$ is the discounted value of the following state (Hugging Face, 2024). Once the algorithm decides on an action, the agent executes that action within the virtual environment. These actions could encompass a variety of behaviors, such as moving, approaching objects, or attacking objects within the environment. Following this, the agent is rewarded or punished based on the established reward system. The agent then repeats all of these steps throughout the training process.

Reinforcement learning problems adhere to a specific structure to ensure the algorithm receives all the necessary data for effective learning. Firstly, the goal of the algorithms must be clearly defined for the developer; they need to understand what the model aims to achieve precisely. The agent(s) must possess all the necessary properties to perceive the environment and take action. Their perceptions must be finely tuned and thoroughly tested to guarantee they capture all necessary information. The agents need to understand what exact actions are possible within the boundaries of their object space. The environment should be clearly defined as everything that is not an agent, as it can only assist the agent in decision-making through rewards. Finally, the reward system must be rigorously tested to provide the optimal reward count to the agent, fostering the learning process; the rewards must be tailored to guide the agent toward its goal.

Proximal Policy Optimization (PPO), the default for Unity's ML-Agents library, is a reinforcement learning algorithm developed by OpenAI (Achiam, 2018). It can be applied to environments with continuous (floating-point numbers) or discrete (between -1 and 1) action spaces. Continuous actions are better suited for situations where infinite actions can occur, like moving from point A to point B. In contrast, discrete actions are for finite decisions like playing a chess game. (Bourne, Gallimard, & Tunnicliffe, 2006) After each trial, the algorithm's policy is updated to minimize the cost function as much as possible using this algorithm: $\theta_{k+1} = argmax\ E\ [L(s, a, \theta_k, \theta)]$ (Achiam, 2018).

Agent-Based Models

Agent-Based Modeling (ABM) is a Machine Learning model that autonomously interacts with environments to learn how to complete tasks within a predetermined ruleset. Through a stochastic model, agents learn from a blank slate to become masters of whatever task they are designed to perform through millions of trials and errors. Agents can be virtual objects that interact with a virtual environment or physical robots that interact with physical objects. To learn, the agents must go through a processing loop: firstly, the agent observes the world, then updates their internal model using the ML algorithm it is given, then the agent takes action, and the whole cycle repeats while the agent is building on top of what they already know (Explorer, 2019).

Agent models use reinforcement learning algorithms as they correlate with the trial-and-error learning method. Depending on the model, ABMs can be helpful for researchers to study the behavior of agents within specific environments, such as pitting two agents against each other in a hide-and-seek game, like OpenAI did in their *"Emergent Tool Use from Multi-Agent Interaction"* study (Baker, et al., 2019). As the agents trained over millions of trials, they learned to exploit the physics engine to move objects in unintended ways to scale high walls to reach their adversaries.

Design

Designing the agent can be tricky because the developer will need to design multiple iterations of the learning model to adjust what the agent can interact with and how it is rewarded/punished. Choosing a suitable learning algorithm is also essential, as it can improve the results tremendously. Penalizing or rewarding the wrong actions can result in the agent failing to grasp their task. For example, if the agent gets penalized too often without that many rewards, they can learn to refrain from interacting with anything to avoid getting penalized during their trials.

Agents and Their Attributes

Agents are autonomous computers that observe their environment and make decisions based on their current situation, role, and abilities (Crooks, 2017). Attributes describe the basic information about the agent(s) that drive their behavior within the environment (Struthers, 2021). Along with dictating the agent’s behaviors, their set attributes would drive their decision-making. They must adhere to the roles and attributes they were given. For example, an agent in the role of a seeker in hide and seek must seek out other agents, specifically hiders.

Reward System

The reward system is a vital aspect of the agent's learning process, helping it determine if its actions are right or wrong. Rewards add points to the agent's counter, while punishments involve deducting points. Continuing with the OpenAI hide-and-seek example (Baker, et al., 2019), the seeker-agent will be given one point when it finds a seeker but will lose one point if it does not find the seeker within the time limit. Through further experimentation over millions of trials, more rewards and punishments may be added to determine the most effective reward system.

Environment and Agent Interactions

A well-constructed environment is essential for teaching an agent because the agents gain rewards and observations from it as they learn to interact with it. The agent needs to receive consequences every time it interacts with the environment, whether it results in a punishment or a reward. A simplified example would be an agent in the form of a plane being heavily penalized for not staying in the sky. Along with the penalty, the agent can be rewarded for maintaining level flight, reinforcing the goal of staying in the sky. After many trials of being penalized for not avoiding objects and rewarded for staying in the sky, the agent will learn to remain airborne.

Evaluation and Adjustment

Frequent evaluation is necessary to ensure that the agents learn from their past actions during training. The amount of time given to training is managed by the number of decisions, also known as steps, the agents make. The developer should evaluate the agent's reward statistics and watch its behavior often to ensure it is learning correctly. The cumulative mean of the total number of points is an important statistic to monitor proper training. Logging the cumulative mean of rewards after a specified number of steps is one way to illustrate how well the agent learns from its rewards and punishments intake. If the agent's behavior is not doing what the developer intended, adjustments to the learning algorithm, reward system, and observations must be made.

Cumulative Mean of Rewards

Some statistics to monitor for the agent and trials include the rewards earned or lost. A simple indication that the agent is learning is an increase in the mean number of rewards after thousands of trials. The cumulative mean is expected to be negative for the first few hundred trials because the agent is still learning what actions to take and what to avoid. However, if the mean number remains negative or continues to decrease, adjustments may be needed in the learning algorithm and reward system.

Learning Algorithm Adjustments

Every model requires a specially tuned algorithm to learn at its highest capacity, so further experimentation is necessary to find the optimal combination of hyperparameters and reward systems. The algorithm's hyperparameters are typically the first place to look when seeking to improve the agent's abilities. Adjusting specific parameters of the learning algorithm, such as the learning rate, number of layers, number of epochs, and more can significantly impact training performance. While tuning the algorithm, adjustments are recommended to carefully evaluate how the modification affected the agent's learning ability.

Reward System Adjustments

Adjusting the reward system can also help performance, and this is where most of the experimenting will take place. The developer can experiment with different observational methods, such as collision and raycasting, to nudge the agent to what they want to learn. Another aspect of the reward system to remember is the value each reward/punishment gives the agent. Taking away too much of a reward can result in the agent staying in place to avoid those punishments, or giving too much of a reward can teach the agent to always do that action and never try new ones. Evaluation and adjustments can be tedious, but seeing computers learn to do things independently is worth the hassle.

Deep Reinforcement Learning

Reinforcement learning has already been explained, but Deep Reinforcement Learning is a more advanced version of this type of learning. Some problems are too complex for simple reinforcement learning algorithms, but deep reinforcement learning can help fill those gaps as it changes how it handles information. Deep RL utilizes the same closed-loop system illustrated in Figure 3 that the simplified version of R.L. uses (Morales, 2020). Instead of using mapping functionality for the state-value pairs in a table during training, Deep RL uses function approximation, which allows the “agent to generalize value of states it has never seen before, or has partial information about, by using the values of similar states” (Williams, 2022). Deep RL thrives in open-ended scenarios, such as teaching an agent to play soccer with continuous action space values.

Agent-Based Model Research Studies

Like all Machine Learning models, Agent-Based Modeling (ABM) is still developing, so there are numerous research gaps and untapped potential. Despite these limitations, ABM is still widely used across multiple industries and research studies. Over the past decade, agent models have been used to simulate the behavior of leadership and followers within flocks of birds (Cristiani, Menci, Papi, & Brafman, 2021) and the transport patterns of entire cities like Paris (Hörl & Balac, 2021).

Research – Flocking Behavior of Birds

Agent-based modeling can be used in various industries, such as simulating interactions within different roles in specific environments, such as a simulation of how leadership works within a flock of birds, according to ornithologists' understanding of bird behavior.

Emiliano Cristiani and his team trained a multi-agent model to simulate the flocking behavior of birds within different bird roles (2021). They state that the model is based on two seemingly contradictory ideas: there is no designated flock leader, so local interactions between group members determine the flock's direction, and leadership does exist because birds assume different roles based on their behaviors (2021, p. 3). However, Cristiani and his team found that these ideas are also contradicted by the fact that:

> All birds occasionally try to change direction and act equally as group controllers. If others follow them, they keep moving in a new direction; otherwise, they cease moving solo and return to the group. (pp. 4-5)

With those flock behaviors in mind, they created a multi-agent-based model where birds assume the roles of leaders or followers. Leaders are solely responsible for changing flocking directions, while followers are primarily meant to follow the leader and the flock. Role changes were simulated through two mathematical processes: stochastic for the follower-to-leader change and deterministic for the leader-to-follower change.

The interactions between the bird agents are simplified to avoid colliding with each other, to stay together, and to exhibit flock behavior through these interactions. The Nearest Neighbor algorithm is used to change leadership. According to Cristiani, Menci, Papi, & Brafman (2021), ‘If either an agent has been a leader for p time units or the distance from its nearest neighbor is over d space units, then it returns to being a follower” (pp. 4-5), so the agents can switch roles mid-simulation to emulate the role reversal in natural flocks.

While their primary goal was to achieve realistic flocking behavior within a 3D environment, they also conducted experiments with 2D flocks consisting of 200 agents to showcase the model's main features (p. 10). Their principal 3D simulation included 400 models, demonstrating that an entire flock of birds can only change directions when a critical mass of leaders is reached (Cristiani, Menci, Papi, & Brafman, p. 16).

Research – City-Wide Simulation of Paris

Agent-based modeling can be used at a macro level for complex simulations of city activities. In a research study conducted by Sebastion Hörl and Milos Balac (2021), they simulated the “travel demand with individual households, persons, and their daily activity chains" within Paris, France (Hörl & Balac, p. 1). They aimed to demonstrate that it is possible to use public datasets of cities to simulate travel demand successfully. By utilizing datasets of census data, commuting relations, aggregated zonal information, and a micro-sample of around 30% of households within France to obtain income information, they could simulate transport demand within Paris using Agent-Based Modeling (Hörl & Balac, p. 3). The agents' attributes were designed to calculate commute distances between agents and to generate daily plans for each agent. The simulation results concluded that it is possible to simulate travel demand using publicly available city datasets (Hörl & Balac, p. 15). However, it is essential to note that the simulation only represents the travel demand of residents and does not consider tourists, as that data is not available on a large scale.

Unity Machine Learning Agents Library

The Unity game engine is software for developing video games, simulations, virtual environments, and more. Libraries within Unity are used to add prebuilt functionalities to developers' projects, and one notable library for developing artificial intelligence is the ML-Agents library (Unity, 2024). This library equips developers with user-friendly tools to train their agents within a virtual environment. Agents can learn to handle properties unavailable in graph environments, such as momentum, gravity, and collision detection while interacting with the environments (Unity, 2017). With sufficient training, they can master any task they are trained to do. For instance, if they were trained to navigate through a busy metropolitan area, they could learn when it is appropriate to move and how to avoid collisions with other objects. The ML-Agents library is a handy toolkit that creates artificially intelligent agents for behavior research purposes, for NPC (non-player characters) behaviors within video games, and for developing robotics to an extent without needing physical machinery besides a capable computer.

Key Features of Unity's ML-Agents Library

Unity's ML-Agents library provides developers with a toolkit to create intelligent and engaging A.I. for their projects using deep reinforcement learning. Fortunately, developers do not need to be experts in Machine Learning to leverage the benefits of this library. ML experts, particularly in agent-based modeling, can access the library through Unity's open-source policy and modify the code to suit their needs (Unity, 2017). For beginners or casual learners, the library and its documentation are straightforward in creating intelligent agents. The library is compatible with C# and low-level Python APIs, enabling agents to be trained using learning algorithms other than the default Proximal Policy Optimization (PPO) algorithm. The Bellman Equation ($V_\pi(s) = E_\pi[R_{t+1} + \gamma * V_\pi(S_{t+1})]$) is used during the training process to guide the decision-making process. In addition to its ease of use and inclusivity, Unity provides developers with access to starter environments to gain hands-on experience with the ML-Agents library (Unity, 2024). Overall, this library is a great starting point for any aspiring ML developer or those who want to experiment with teaching ML agents to do exciting actions.

Application – Controlling NPC Behavior

A possible application for trained agents that was previously mentioned was an implementation into an NPC system. Typically, NPC behavior is conveyed through animations and A* Pathfinding to allow the NPC to navigate across a NavMesh to its designation. A NavMesh is a generated overlay that informs A.I. NPCs where they can navigate, preventing them from venturing into areas that are off-limits or out of bounds. It is designed as a simple method to incorporate NPC behaviors into a virtual setting. Dijkstra shortest path algorithm is used to navigate across a NavMesh or a directed graph by finding the shortest path to the target. This algorithm is used to find the shortest path within mazes, as illustrated in Figure 2, where the agent determines which direction to move based on the cost amount of moving (Zafrany, 2016). Each direction has a number, called a “cost,” that will add or subtract from the agent’s cumulative reward, where the highest cost is not the direction the agent should move, such as a boundary or moving away from the target. It will generally choose the less costly direction to maximize its cumulative reward. Dijkstra’s algorithm of A.I. pathfinding can be replaced with the ML-Agents library to train agents to adapt to new environments automatically after training.

Through the ML-Agents library, NPCs, or “agents," can be taught many behaviors, such as self-animation without pre-rendered animations, behaving in specific ways to fulfill a given role, and automatically adapting to new environments after training in other similar environments. For example, a humanoid agent can be trained to walk upright by learning to manipulate all of its joints and limbs to stay balanced while navigating (Warehouse, 2023). Like the baby analogy previously mentioned, the agent would begin the training by only being able to flail around. However, after millions of trials and errors of walking on a flat plane, the agent could walk smoothly like a capable adult human. The agent must still be trained on other terrains, such as stairs or declines. Once the agent is trained to a satisfied state, the model can be deployed to every NPC within a video game to give the game a more natural atmosphere that older games may lack.

Custom-Built ML Navigation Agent

Developing a custom agent-based model with unique problems to solve can lead to a deeper understanding of Agent-Based Modeling. Experimenting with adjusting the reward system, observation methods, and algorithm settings can help learn how agents adapt to their environment based on how those factors are configured. Additionally, experiencing agent-specific issues throughout development can teach more about the inner workings of the algorithms and learning processes. This section is a first-hand experience of the development process of training an agent to navigate three-dimensionally through virtual maze environments to reach a pressure plate target. A basic understanding of reinforcement learning algorithms and Unity is required.

Model Design

The agent is trained solely to navigate through multiple versions of a maze to reach its randomly spawned target. The model will utilize the Unity Editor and Unity's ML-Agents library to train the agents since the editor can allow quicker training times and adds 3D visuals to see how the agent behaves. The model will use Proximal Policy Optimization as its learning algorithm. It will record training metrics of cumulative means of rewards, policy changes, and learning rate losses to evaluate how successful the model is after training.

Agent's Goal

The agent's goal is straightforward: navigate to the target plate by any means necessary without going out of bounds or exceeding the 45-second time limit. The ideal agent will avoid getting stuck on obstacles and quickly reach the target within a third of the established time limit. The agent should consistently reach the target in new environments and adapt to unpredictable spawning patterns for itself and the target.

Agent and Its Attributes

Design. The agent is designed as a red cube with a face, as shown in Figure 4. The agent can be in most other shapes, but the size should be within reason to fit within the environment. Also, invisible ray casts are stationed on the front of the agent with a 30-degree radius facing outward to gather visual data for the observations and reward systems.

Observations. The model requires capturing observations to help the agent analyze its surroundings and make decisions. The ray casts stationed on the agent are automatically collected as visual observations to give the agent a form of sight. Figure 18 shows the CollectObservations method, which collects the agent's relative positions and velocity observations. These observations are passed to the algorithm's policy to request actions from the model that the agent uses to navigate through and interact with the environment.

Navigation. To give the agent the ability to navigate independently, it must receive the values of continuous actions from the model. In Figure 19, in the OnActionReceived method, the actions for the agent to manipulate its positional axes, y-rotational axis, and velocity are received. Those values change the agent's force of direction, rotation, and orientation, as shown in the latter half of Figure 19. Since the agent can manipulate all positional axes, including the y-axis, it can fly to give the agent more control over its navigation through the environment.

Learning Algorithm. The ML Agent library uses a configuration file to modify how the model learns and how the environment treats itself as a training environment. The learning algorithm's hyperparameters are listed here to change to find the best configuration for the connected model. In this model's configuration file, the most significant hyperparameters of the PPO learning algorithm are the learning rate that's set to 0.001, the number of epochs that's set to 3, the beta value that's set to 0.005, and the batch size that's set to 1024. Also, the learning rate, beta, and epsilon's schedules are all linear. The file offers other settings, such as hidden layers, deterministic settings, number of trials to run, frame rates, quality levels, how often the results are logged at checkpoints, and more. This model is set to run 20 times faster than real-time for a million steps to fast-track the training. For every 50,000th step, a report of how well the training has been saved to .pt and .onnx files that can be viewed through a TensorBoard GUI.

Target Pressure Plate

The target is a square pressure plate stationed on the ground of the environment that the agent must collide with to gain the highest reward and end the current episode. Figure 5 shows the target in its default blue color, while Figure 6 shows it as green when the agent reaches it. The target is stationary throughout the entire episode; it is relocated once the episode concludes, whether from success or failure.

Model Training

Training the model well requires experimentation and randomness to encourage adaptability in future testing. Early iterations of the environment and movement systems did not encourage adaptability to new environments. As seen in Figure 10 of the older environments, the platforms were straightforward, and the fact that the agents and targets respawned in the exact locations added little to no randomness between each episode. After experimenting with different environment layouts, nine environments with all different layouts and walls, as shown in Figure 7, and random spawning were created. The model's flexibility improved exponentially after incorporating the uniqueness and unpredictability of the training.

The model has been trained to three steps to measure how the agents behave and how successful the training is at each stage: 500,000, 750,000, and 1 million steps. Each stage provides insights into how well the model can predict its actions, how random their decisions are, and the cumulative mean of many other metrics. These will demonstrate how much training is required for this model to reach its target in time effectively.

Initialization

At the beginning of every simulation, the agent and environment properties must be initialized to their default state to ensure the model is calibrated correctly for training, as shown in Figure 17 in the Initialize method. Firstly, the base class's Initialize method is called to set up the default parameters, and the default environment parameters are set by the Unity ML Agent Academy, which is the Unity singleton class that manages the training and the agent's decision-making abilities. Then, the agent's orientation, game object, and previous position properties are initialized to their default values of zero for the orientation and references to the agent's game object's rigid body and starting positional axes.

Spawning

Nine agents are trained simultaneously in nine unique environments to accelerate training time. Random spawn points and unique environments avoid overfitting the model to one specific situation, such as the original environments and spawn locations. The agent and target have four and five different spawn points, respectively. The spawn points are laid out like a tic-tac-toe board, as illustrated in Figure 8: the target can spawn on any of the corners and the middle, while the agent can spawn in the remaining open spots between the target spawn points. Each maze layout is designed to accommodate the spawn points by keeping those spawn areas free of obstructions. The spawn functionalities are used in multiple areas of the source code, such as the OnEpisodeBegin method in Figure 25 and the OnTriggerEnter and Respawn methods in Figure 26.

Reward System

The reward system has been carefully crafted, including how often the agents receive rewards and punishments and how many points are given and taken. There are three categories of rewards that the agent can receive: positional rewards, observational rewards, and time-related rewards. By all means, this reward system does not guarantee producing the most intelligent agent after training. Constant experimentation with the reward system is vital to ensuring the agent can learn as the developer intends. The agent will inhibit interesting behaviors if it receives a reward too often, not often enough, too little, or too much, so it is best to keep track of each version of the reward system and how it produces agents.

Positional Rewards. Positional-related rewards are to track where the agent is within the environment and to encourage the agent to stay within boundaries and move toward the target's location. The agent must find out where the target is when it begins training. The single hint of its location is from the rewards given when it moves closer to the target and is punished for moving away. In Figure 23, in the FixedUpdate method, the positions of the agent and target are captured in every frame and used to determine if the agent is closer to the target than in the previous frame. As the agent moves closer to the target, it is rewarded with 0.001, but 0.001 points are removed from the agent as it moves further away. The rewards are minuscule to avoid over-rewarding and over-punishing the agent because of how quickly the points can add up after each frame of an episode.

Additionally, the agent is encouraged to stay within the environment's boundaries by removing half a point from the agent's total rewards, as seen in the latter half of the FixedUpdate method of Figure 23. The current episode also ends when the agent is removed from the boundary, which is heavily discouraged. The length and width boundaries are set at the edges of the map, but the agent is given a semi-large height boundary to allow floating above walls to reach targets quickly.

Observational Rewards. The most significant contributors to the total reward count are collision and ray cast sensors, giving the agent senses of sight and touch. After learning the target's direction, the agent would constantly bump into walls blocking the target. So, with the ability to detect obstructions and the target itself, it can learn to avoid the walls when a quarter point is deducted every time the agent collides with game objects labeled with the "Wall" tag, as seen in Figure 24 in the OnCollisionEnter method. This punishment, however, only teaches the agent to move away from the wall after it collides with it, essentially bouncing off every wall. Raycasts may solve that issue; they can detect objects a few meters away from it in whatever direction they face. These ray casts are stationed on the front of the agent with a 30-degree radius to give it some limitation to its sight. As shown in Figure 21 in the OnActionReceived method, the ray casts detect walls and deduct 0.0001 points from the agent but add 0.0001 points when the ray casts detect the target.

Time Rewards. The agent is encouraged to find the most efficient and quickest way to reach the target by giving it a time limit and giving rewards based on its velocity. The agent's current velocity is multiplied by 0.01 and given as a reward, as shown in Figure 20 in the OnActionReceived method, to encourage moving quickly through the environment. However, if the agent moves too fast, it most likely moves outside the environment boundary. The punishment administered via the boundary limits keeps the agent from moving too quickly. In conjunction with the velocity reward, the agent is punished for taking too long to reach the target. Each episode has a 45-second time limit, and when that limit is reached, the agent is punished with a 0.05-point reduction, as shown in Figure 22 in the FixedUpdate method.

Model Observations and Results

Three model versions have been trained at different steps to compare the model's efficiency and behavior differences after the different levels of training: 500K, 750K, and 1 million steps. Most observations will center around the 500,000-step model since it is the most consistent at reaching the target. Each version of the model developed interesting strategies to navigate through the environment to reach the target, such as floating across the maze or bouncing off walls like a pinball machine. However, as the model trained further, the agent's decisions became more reckless and sporadic. This decrease in quality can be explained through a metric evaluation of the policy, learning loss rates, and environment cumulative reward of all training steps.

Along with the automatic metrics, the results of a few hundred episodes have been captured to evaluate how often the agents reached the target, fell out of the maze boundary, and failed to reach the target within the 45-second time limit. Overfitting is a common way a model worsens the more it trains. However, other factors to consider when evaluating a model, such as algorithm configurations and the reward system balance, can be modified to improve behaviors beyond 500,000 steps.

Efficiency Evaluation

Each version of the model was tested for five minutes to observe how often the agent reached the target successfully. The results in Table 1 show how often each version succeeds by reaching the target, fails due to exiting the maze boundary, and fails due to time running out before reaching the target. The best version in all areas is the 500,000-step version, which completed the most trials with 366 targets reached, four time limit fails, and 93 out of boundary fails. Each subsequent version completes fewer trials and fails more trials more often than the previous version. The more trained agents tend to take more time to find the target, resulting in the time limit being reached, and tend to fly higher and quicker, resulting in going out of bounds. The increase in time limits reached is shown in Figure 12, as the mean length of each episode increases after step 500,000.

A vital statistic captured during training is the cumulative mean of rewards throughout all training steps. The cumulative mean of rewards is displayed as a table in Table 2 and as a chart in Figure 11. Table 2 only shows cumulative rewards up to 500,000 steps, while Figure 11 shows the cumulative means up to 1 million steps. In order to indicate a successful model, the cumulative mean should increase as the training session continues. As expected, the cumulative mean begins as a negative number because the agent loses points from erratically moving around, which leads to bumping into walls and leaving the boundary. The cumulative mean increases once the agent moves closer to the target more often and accidentally collides with the target. Accidental target collisions are imperative for early learning because losing rewards due to constant wall collision punishments and moving away from the target can hinder training. After the initial learning curve, the agent minimizes the number of times it collides with the walls and consistently reaches the target to increase its cumulative mean of rewards gradually and consistently.

Behavior Observations

Behavioral observations can provide developers with a more visual representation of how well the model is performing. The following observations will be from a human observer watching all three model versions test themselves for an hour each. As shown in Table 1, the version that reached the most targets within five minutes was the 500,000-step version, consistent with how the model behaved during testing.

Since the 500,000-step model could adjust its height, it learned to float above walls to reach its target quicker by avoiding the walls entirely. Sometimes, it would get stuck on the walls as it floated upwards, but it would correct itself by moving backward and launching itself from that spot to gain more space between itself and the wall to scale it better. However, that strategy is not 100% effective as the agent sometimes overshoots the target entirely and goes out of the boundary. However, it does work enough times for the agent to keep that strategy in its memory.

The agent was trained to avoid colliding with the walls through its collision and ray casts-detection sensors, but the agent still collides with walls in the 500K-step model and subsequent versions. Fortunately, it was learned not to bump into walls repeatedly but instead bounce off walls to navigate around the maze. Once it touches a wall and realizes it is not the target, it immediately redirects to a different wall to bounce off and get around it, blocking the target. Figure 9 demonstrates how the agent would typically navigate an environment to reach a target if it did not float above walls. The agent is not restricted to navigating in straight lines but can also curve around walls.

The subsequent 750K and one million-step versions behave more recklessly while moving toward the targets. Compared to the 500K-step model, the agents with more training tend to fly higher and move quicker, which limits their precision in reaching targets. With the more erratic movement strategies, it tends to overshoot its targets and fly out of all boundary lines. This behavior hinders the efficacy of reaching the targets, as shown in Table 1, where the number of fails due to exceeding the time limit or crossing boundary lines increases.

Learning Loss Results

The Unity ML-Agents library automatically provides metrics in histograms and graphs to measure how successful the model's training sessions are (Unity, 2017). Those charts can be viewed through a TensorBoard GUI after the result reports are saved and the designated checkpoint has been reached. Some metrics are used to evaluate the loss functions, quantifying how close the model's predictions are to the actual results. After training, this custom model provided two learning loss metrics for the policy loss, which measures the mean magnitude of the policy loss function and correlates to how much the policy for deciding future actions is changing, and value loss, which measures the mean loss of the value function update and correlates to how well the model can predict (Unity, 2017).

The policy loss is illustrated in Figure 13 as a line graph that fluctuates throughout training. The magnitude should decrease during training to indicate a successful training session, but there are many highs and lows throughout the graph to conclude otherwise. The lowest point is 0.02068 at step 600,000, which rises again as training goes to step 1 million. These fluctuations indicate instability within the decision-making process because the chart should stabilize after decreases. The learning rate or batch size can be adjusted to remedy this instability. On the other hand, additional training may be required because spikes in policy loss are expected when the model explores other strategies, leading to temporary instability.

The value loss is illustrated in Figure 14 as a line graph that significantly increases from 0.02686 at step 50,000 to 0.9539 at step 700,000 and then slightly decreases after step 700,000. While the agent is learning, this mean of loss should tread exactly how it does in Figure 14 to indicate success: increasing and decreasing once the reward stabilizes. However, it decreases as it reaches 1 million, so additional training may be needed to stabilize the reward.

Policy Statistics

Along with the learning loss functions, the policy performance is recorded. Policy performance statistics evaluate how well the agent can determine their following action through the policy. The two charts for the policy statistic measurements are the changes in entropy, which measure how random the decisions of the model are, and the changes in the learning rate, which measure how large a step the training algorithm takes as it searches for the best policy (Unity, 2017).

The entropy changes are illustrated in Figure 15, where they gradually decrease to indicate successful training. If it decreased too quickly, the beta hyperparameter would need to be increased to achieve that slow, gradual decrease trendline. No adjustments are necessary since it has gradually decreased throughout its training sessions.

The learning rate changes are illustrated in Figure 16, where it starts at 0.0000948 and decreases down to 0.0000197 by step 500,000. The learning rate is supposed to decrease over time to indicate successful training. However, when it reaches step 1 million, it increases and decreases twice, resulting in two gradual spikes in the changes. These fluctuations are visually apparent when the agent behaves more poorly beyond step 500,000.

Future Adjustments and Additions

The model can be significantly improved to help it train beyond 500,000 steps since it gets worse at reaching the target the more it trains. A camera-based lidar sensor can add more visual observations that help see objects before colliding with them, thus remedying the agents bouncing off walls to navigate. One thing to keep in mind with camera sensors is noise. Too much noise from the sensor could hinder its ability to observe using the camera, so limiting as much noise as possible is critical to getting the most out of a camera sensor. Some adjustments to the learning algorithm's hyperparameters can also help, but that requires significant testing to find a better combination of values, as tuning can be tedious.

Additional unpredictability and interactions can add new behaviors to how the agents reach their targets. Currently, the targets are static in one place for the agent to reach for the entirety of the episode. The agent's goal can become more challenging if the target can also move, producing different strategies beyond floating above walls. The target can also become an agent, changing the model into a multi-agent model that pits two agents against each other in a cat-and-mouse game. This target-to-agent suggestion is a significant change that can lead to new behaviors and challenges with refining the hyperparameters, observations, and reward system. New interactions, such as moveable objects, can be implemented, so the agent can use mini-walls, stairs, or simple cubes to reach their target.

The following suggestion is not necessarily made to improve the learning model but to improve how the model is presented. Currently, the model, environment, and game models are straightforward, using cubes and planes. The current context of the environment could be more compelling to a human observer with some slight modifications. To align with the current behaviors of consistently floating above walls and to add better context to the model, change the appearance of cubes to birds attempting to land in nests instead of target pressure plates. This context would raise the environment's height significantly, add trees and tree branches for the nests to spawn on, and add the nests themselves. While this suggestion is primarily for aesthetic purposes, the environment and models' appearance can influence expectations for how the model should behave. Observers can now expect the agents to fly, while that behavior was not an expectation in the current maze-like environments.

Conclusion

Machine Learning

As an essential aspect of modern computing, machine learning is implemented in many consumer products, but knowing how the algorithms that power them work can give further insight into the capabilities of ML. Supervised, unsupervised, and reinforcement types of machine learning can only tackle problems catered to how their algorithms learn. Developers must understand how each type of learning works and the problems at which they excel. For example, reinforcement learning excels immensely in agent-based modeling.

Agent-Based Modeling

Through reinforcement learning, agent-based modeling teaches A.I. how to interact with environments to accomplish their task. As explained previously, the applications of agent-based modeling are vast and used in many exciting studies to observe how a machine imitates the behaviors of organisms in situations and simulates systems in large environments. For example, the flocking behavior of birds and the demand for travel within a large city like Paris. These studies merely scratch the surface of ABM, as agents can be taught to precisely manipulate parts of themselves, such as joints or entire limbs, to navigate independently.

The design of an agent sets the stage for how well their training can go. They must be aware of their environments, what actions they are permitted, what limitations they are held under, and how the reward system reinforces their behavior. Designing the agent requires experimenting with the aspects mentioned to create the perfect system to accomplish learning.

Additionally, evaluating and adjusting the model is just as imperative to improving the model. The Unity ML-Agents library provides metrics of how well the agents are training through multiple charts and graphs. These charts and graphs illustrate the changes in the algorithm’s policy, the model’s cumulative rewards, and the learning loss functions. Paying attention to these metrics can help the developer to know what to change to improve the model significantly.

ML Agents on Navigation

The agent model explained in detail in a previous section is a custom-built model to describe the first-hand experience of designing, evaluating, and adjusting an agent-based model. The model's goal is to navigate through a maze to locate and touch their targets by any means necessary. The quickest strategies learned to reach the target were fascinating to watch develop in real time. While evaluating the model's metrics, there is a clear indication that the hyperparameters and some aspects of the model's design can be modified to improve the results. As seen in Figure 13, the fluctuations within the policy loss suggest that the learning rate and batch size can be tuned further to help stabilize the mean magnitude of policy loss. Despite the improvements that can be made, the custom-built model learned to take advantage of its observations, actions, and abilities to reach its target successfully.

References

Achiam, J. (2018). *Proximal Policy Optimization*. Retrieved from OpenAI Spinning Up: https://spinningup.openai.com/en/latest/algorithms/ppo.html.

Baker, B., Kanitscheider, I., Markov, T., Wu, Y., Powell, G., McGrew, B., & Mordatch, I. (2019). Emergent Tool Use from Multi-Agent Autocurricula. *ICLR 2020.* Virtual.

Bonzon, T. (2023, November 10). *How to Make AIs Target Objects with Unity ML Agents*. Retrieved from Zenva: https://gamedevacademy.org/unity-ml-agents-object-target-tutorial/

Bourne, W., Gallimard, R., & Tunnicliffe, J. (2006). *Environments.* Retrieved from Multi-Agent Systems: https://www.doc.ic.ac.uk/project/examples/2005/163/g0516302/index.html

Cristiani, E., Menci, M., Papi, M., & Brafman, L. (2021). An All-Leader Agent-Based Model for Turning and Flocking Birds. *Journal of Mathematical Biology*, 4-5.

Crooks, A. (2017, February 22). *Applications of Agent-Based Models*. Retrieved from GIS and Agent-Based Modeling: https://www.gisagents.org/2017/02/applications-of-agent-based-models.html

Explorer, C. (2019, March 7). *Agent-Based Modeling: Machine Learning and Agent-Based Modeling*. Retrieved from YouTube: https://www.youtube.com/watch?v=BS20rrrBDiI

Hörl, S., & Balac, M. (2021). Synthetic population and travel demand for Paris and Île-de-France based on open and publicly available data. *Transportation Research Part C: Emerging Technologies, Volume 130*, 1.

Hugging Face. (2024). *The Bellman Equation: simplify our value estimation*. Retrieved from Hugging Face: https://huggingface.co/learn/deep-rl-course/en/unit2/bellman-equation

IBM. (2021, October). *What is linear regression?* Retrieved from IBM: https://www.ibm.com/topics/linear-regression

IBM. (2023). *What is Unsupervised Learning?* Retrieved from IBM: https://www.ibm.com/topics/unsupervised-learning

IBM Technology. (2022, July 27). *Supervised vs Unsupervised Learning*. Retrieved from YouTube: https://www.youtube.com/watch?v=W01tIRP_Rqs

Liu, D. (2019, October 30). *Logistic Regression*. Retrieved from Keji Tech: https://techs0uls.wordpress.com/2019/10/30/logistic-regression/

Morales, M. (2020). *Grokking Deep Reinforcement Learning.* Manning.

Pregnancy Birth & Baby. (2022, June). *Learning to Walk*. Retrieved from Pregnancy Birth & Baby: https://www.pregnancybirthbaby.org.au/learning-to-walk

Sarangam, A. (2021, March 4). *Classification vs Regression: An Easy Guide in 6 Points*. Retrieved from U Next: https://u-next.com/blogs/artificial-intelligence/classification-vs-regression/

Sharma, P. (2024, February 20). *The Ultimate Guide to K-Means Clustering: Definition, Methods and Applications*. Retrieved from Analytics Vidhya: https://www.analyticsvidhya.com/blog/2019/08/comprehensive-guide-k-means-clustering/

Struthers, E. (2021, February 19). *An Introduction to Agent-Based Modelling*. Retrieved from 4CDA: https://4cda.com/an-introduction-to-agent-based-modelling/#:~:text=Rules%20in%20the%20model%20determine%20how%20an,is%20a%20zombie%20then%20behave%20like%20a

Unity. (2017). *ML-Agents Overview*. Retrieved from Unity Technologies GitHub: https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/#additional-features

Unity. (2017). *Unity ML-Agents Toolkit*. Retrieved from Using Tensorboard to Observe Training: https://unity-technologies.github.io/ml-agents/ML-Agents-Overview/#training-modes

Unity. (2017). *Unity-Technologies / ml-agents*. Retrieved from GitHub: https://github.com/Unity-Technologies/ml-agents

Unity. (2024). *Unity Machine Learning Agents*. Retrieved from Unity: https://unity.com/products/machine-learning-agents

Ved, M. (2018, July 13). *Reinforcement Learning*. Retrieved from Medium: https://medium.com/@mehulved1503/reinforcement-learning-e743bcd00962

Warehouse, A. (2023, April 23). *AI Learns to Walk (deep reinforcement learning)*. Retrieved from YouTube: https://www.youtube.com/watch?v=L_4BPjLBF4E&t=29s

Williams, T. (2022, August 8). *Reinforcement Learning Vs. Deep Reinforcement Learning: What's the Difference?* Retrieved from Techopedia: https://www.techopedia.com/reinforcement-learning-vs-deep-reinforcement-learning-whats-the-difference/2/34039

Zafrany, S. (2016, September). *Deep Reinforcement Learning for Maze Solving*¶. Retrieved from Samy Zafrany: https://www.samyzaf.com/ML/rl/qmaze.html

Zhang, J. (2021, September 22). *Proximal Policy Optimization (PPO) with Unity ML-Agents*. Retrieved from Coder One: https://www.gocoder.one/blog/training-agents-using-ppo-with-unity-ml-agents/

Tables

Table 1

Agent Trial-Ending Events within Five Minutes

Event	500K Steps	750K Steps	1Mil Steps
Out of Bounds	93	31	23
Out of Time	4	16	21
Reached Target	366	347	249
Total Trials	462	394	293

Note: This table presents the results of testing three different model intelligences for five consecutive minutes using identical hardware. Given the current training configuration of the model, training with 500,000 steps yields the best results.

Table 2

Cumulative Mean Reward Over Time

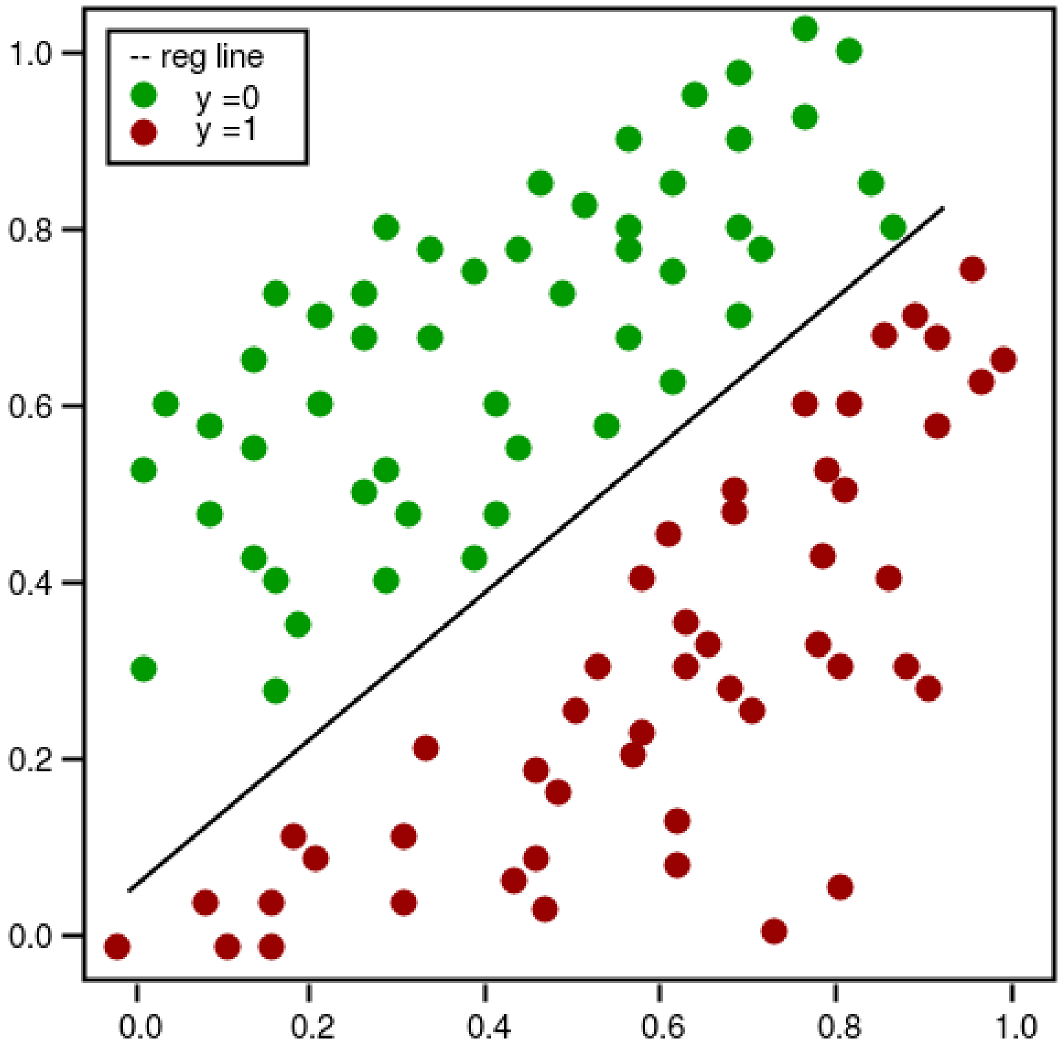
Step Counter	Mean Value of Rewards
50,000 Steps	-5.86
100,000 Steps	9.78
150,000 Steps	8.594
200,000 Steps	13.53
250,000 Steps	21.36
300,000 Steps	33.12
350,000 Steps	40.49
400,000 Steps	53.77
450,000 Steps	82.78
500,000 Steps	114.6

Note: Correlates to Environment/Cumulative Reward Chart in Figure 11.

Figures

Figure 1

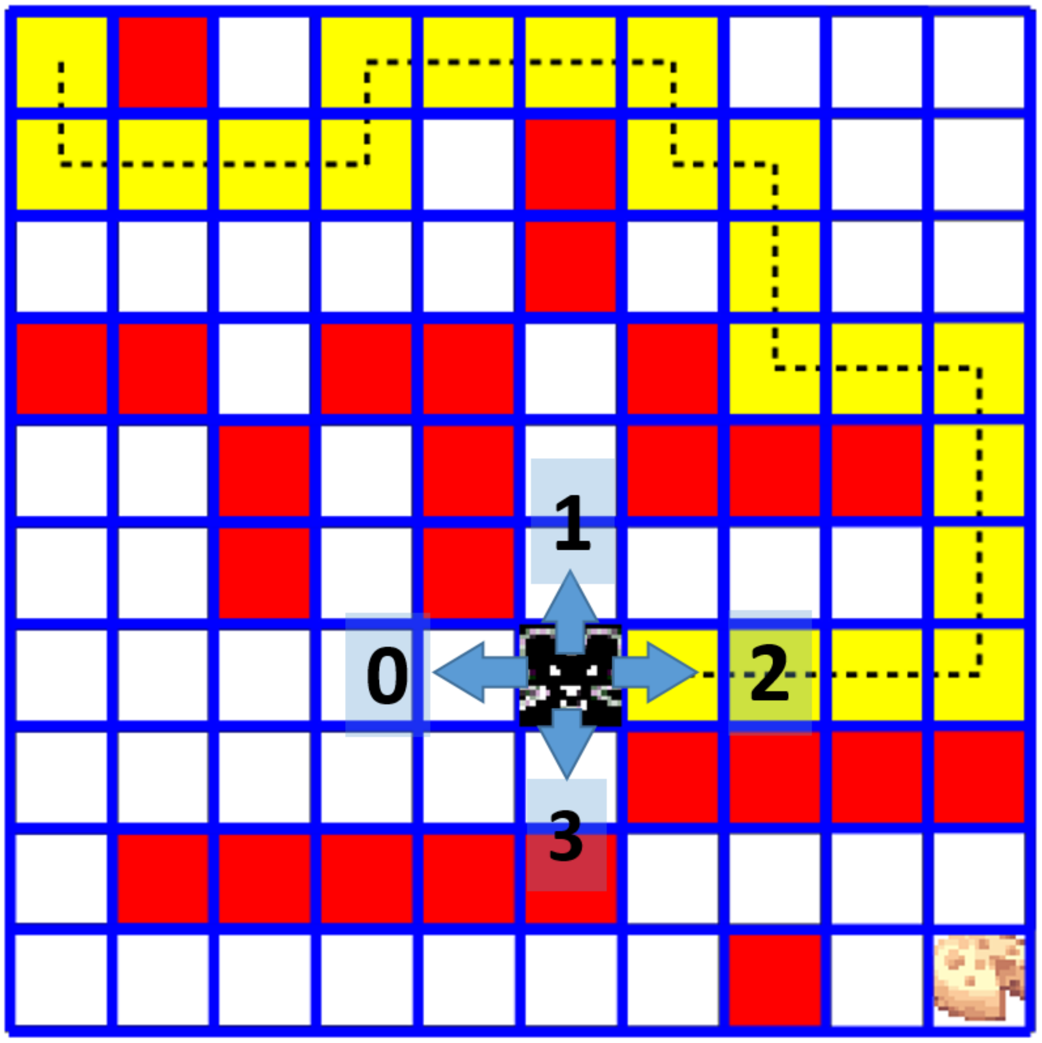
Logistic Regression Example



Note: Separates the linear data into groups, provided by Davide Liu (2019).

Figure 2

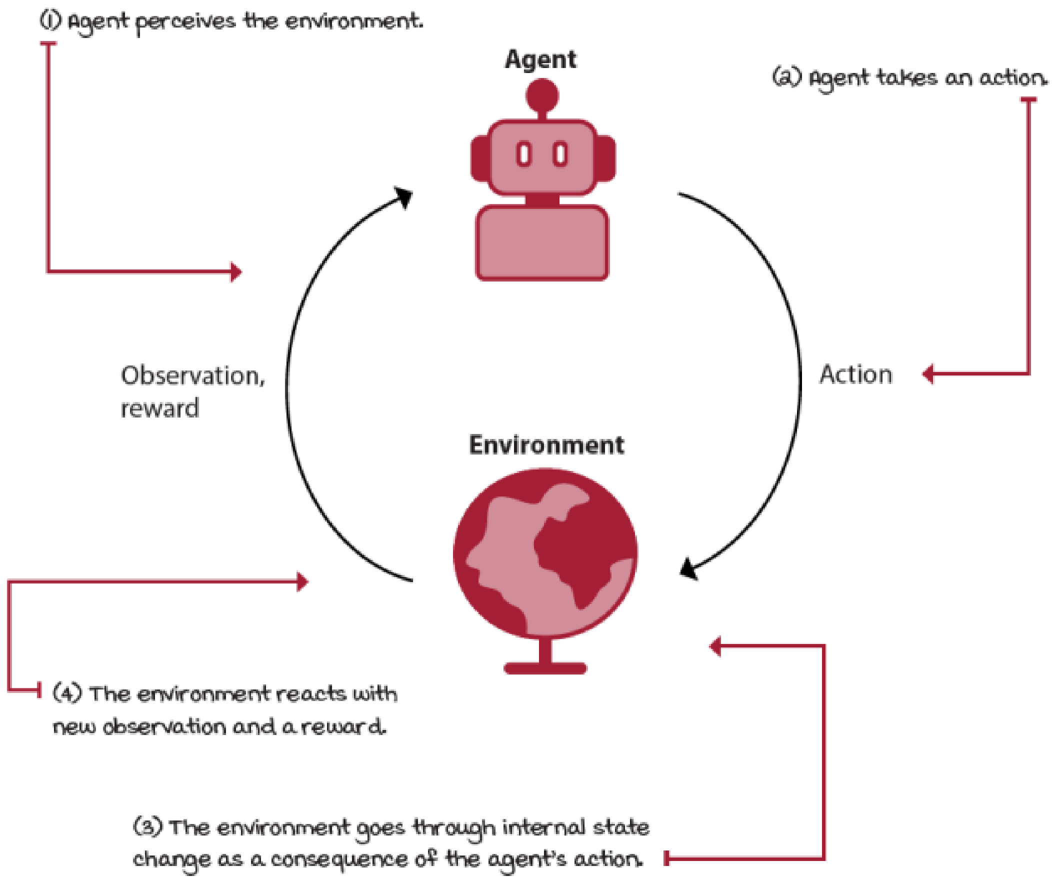
Dijkstra's Algorithm Movement Cost Visualization



Note: The maze illustrates the movement cost from Dijkstra's algorithm, provided by Samy Zafrany (2016)

Figure 3

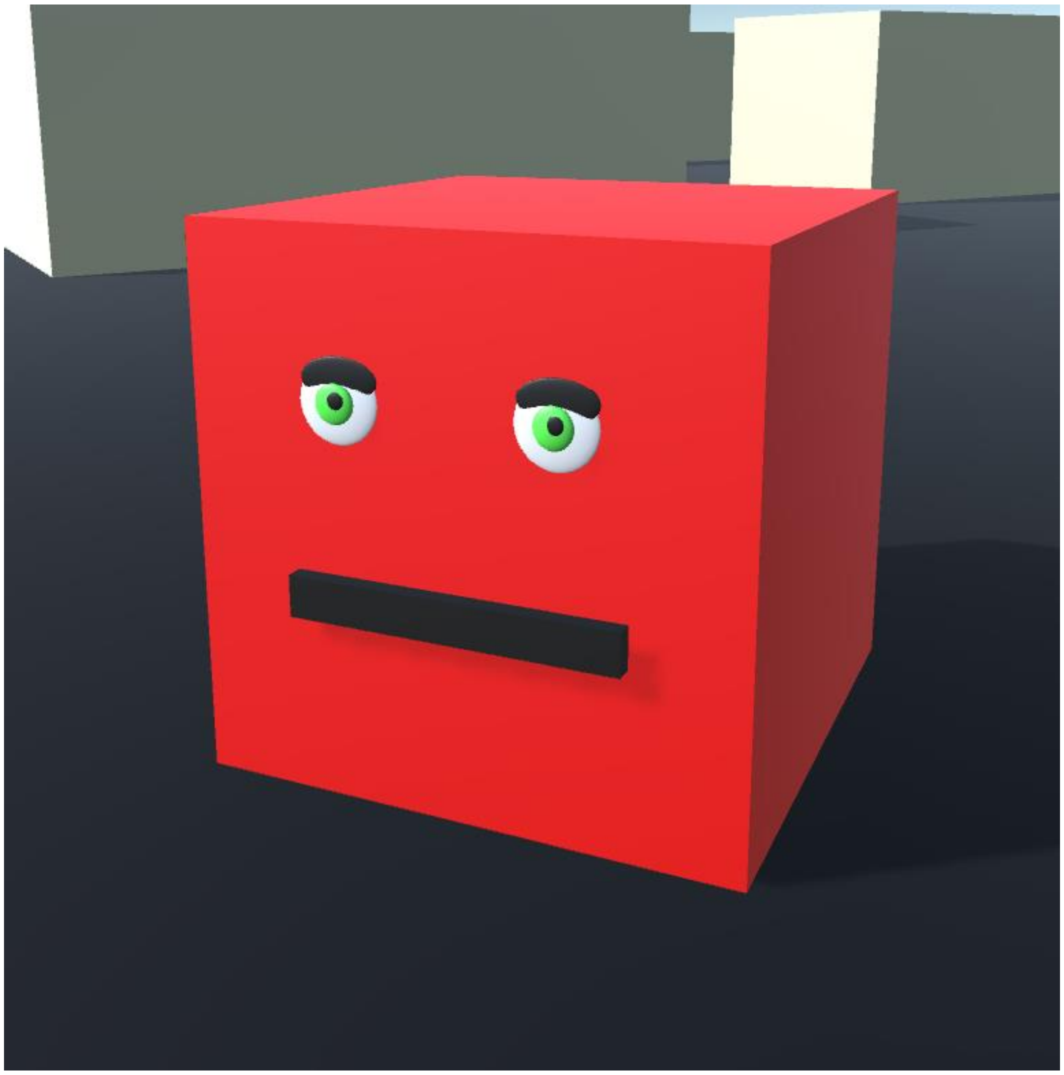
Deep Reinforcement Agent-Based Modeling System Loop


(1) Agent perceives the environment.
Agent
(2) Agent takes an action.
Observation, reward
Action
Environment
(4) The environment reacts with new observation and a reward.
(3) The environment goes through internal state change as a consequence of the agent's action.

Note: System loop for an agent receiving observations from the environment, responding with action, and receiving a reward/punishment in return, provided by Miguel Morales (2020).

Figure 4

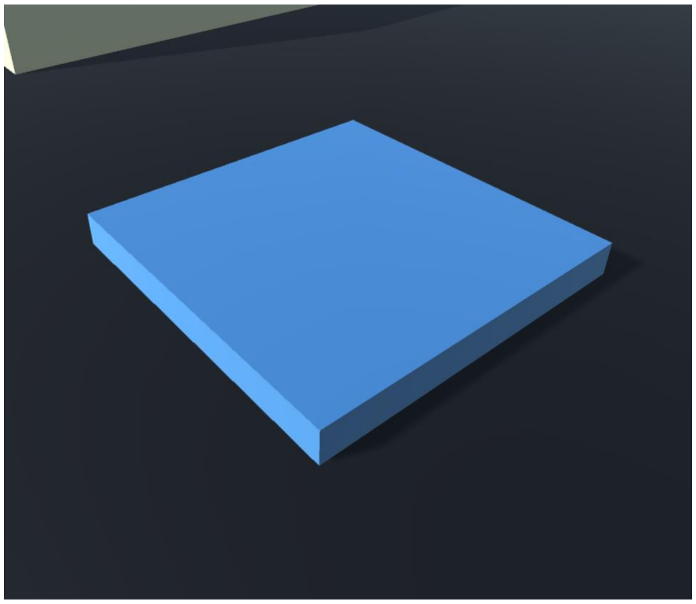
Agent Model



Note: There are also invisible ray-cast lines emanating from the target’s forehead, but the face is added as a visual component of direction and intent.

Figure 5

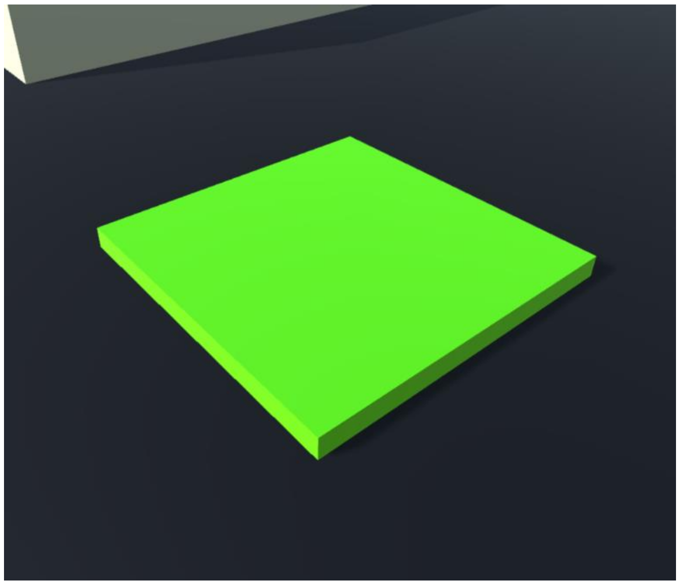
Target Pressure Plate Default State



Note: Returns to this state when an episode indicates the agent has not met its goal.

Figure 6

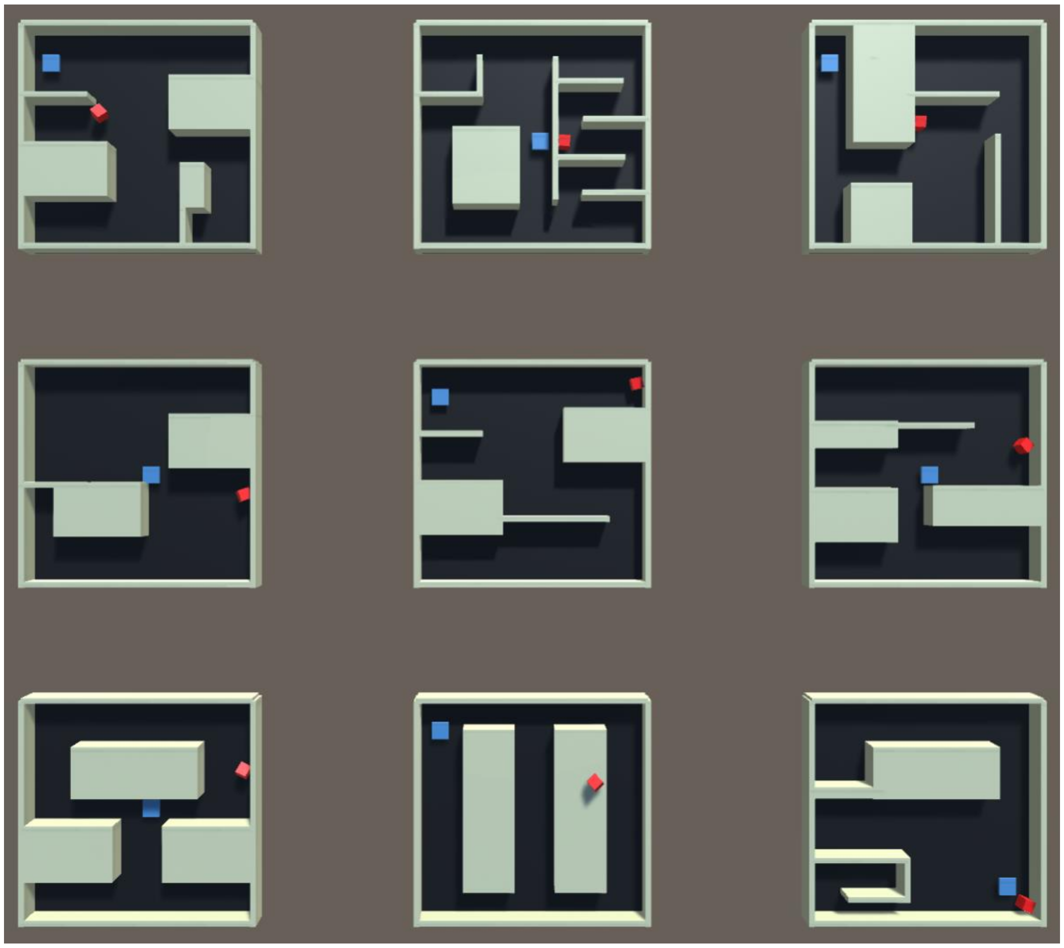
Target Pressure Plate Success State



Note: Material turns green and decreases in the y-axis to indicate success.

Figure 7

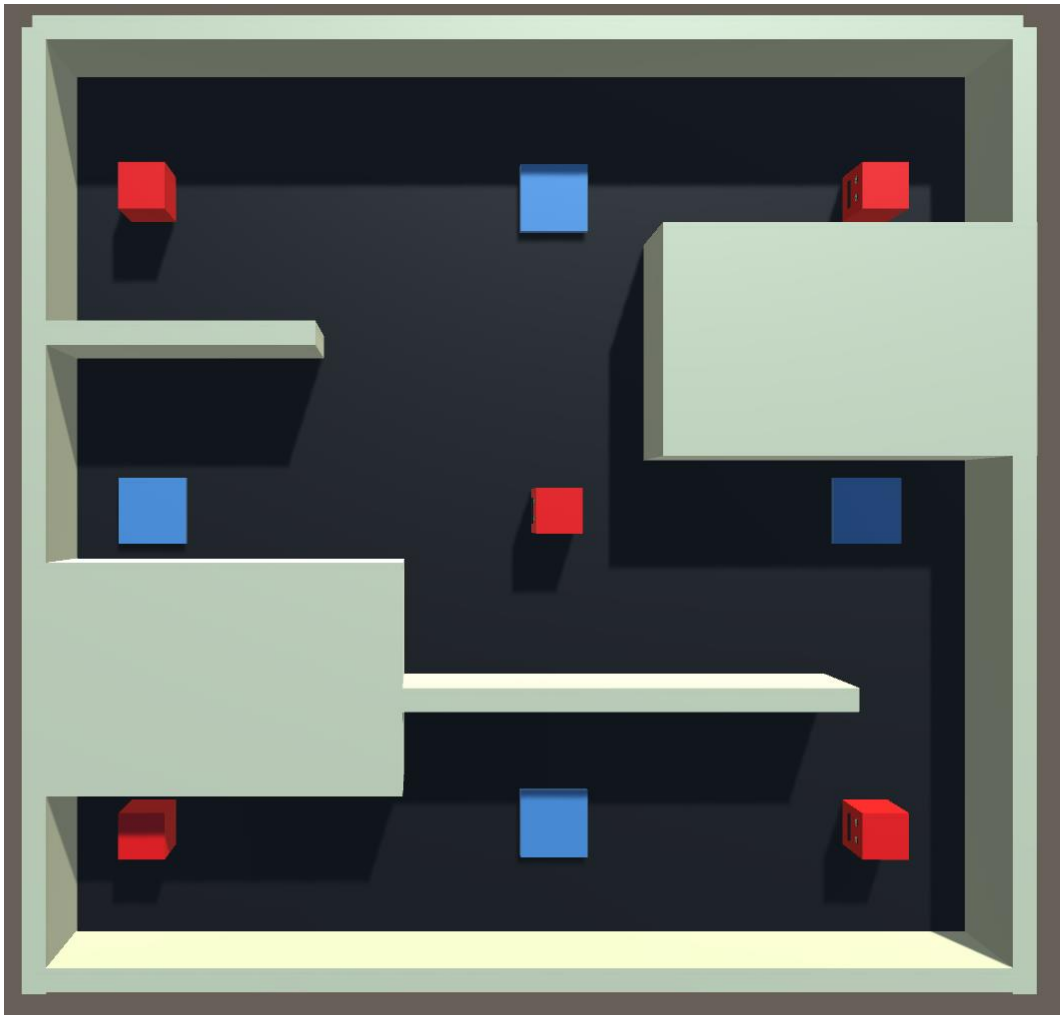
Maze Environments in Progress of Testing



Note: 9 unique mazes for the agents to train in. Agents can fly over walls or through corridors to reach their targets.

Figure 8

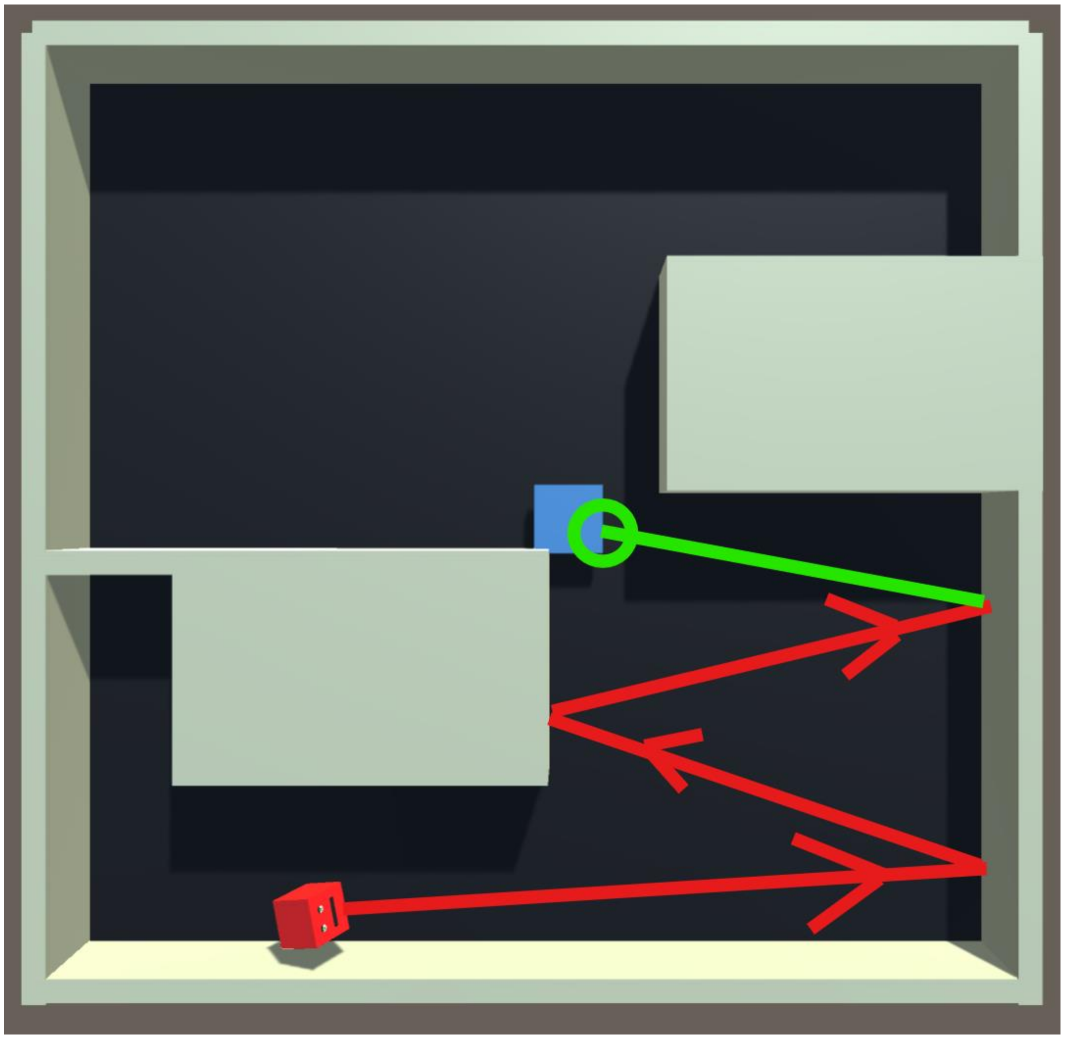
Agent and Target Spawn Locations



Note: At the start of each trial, the agent and target are randomly spawned in these locations.

Figure 9

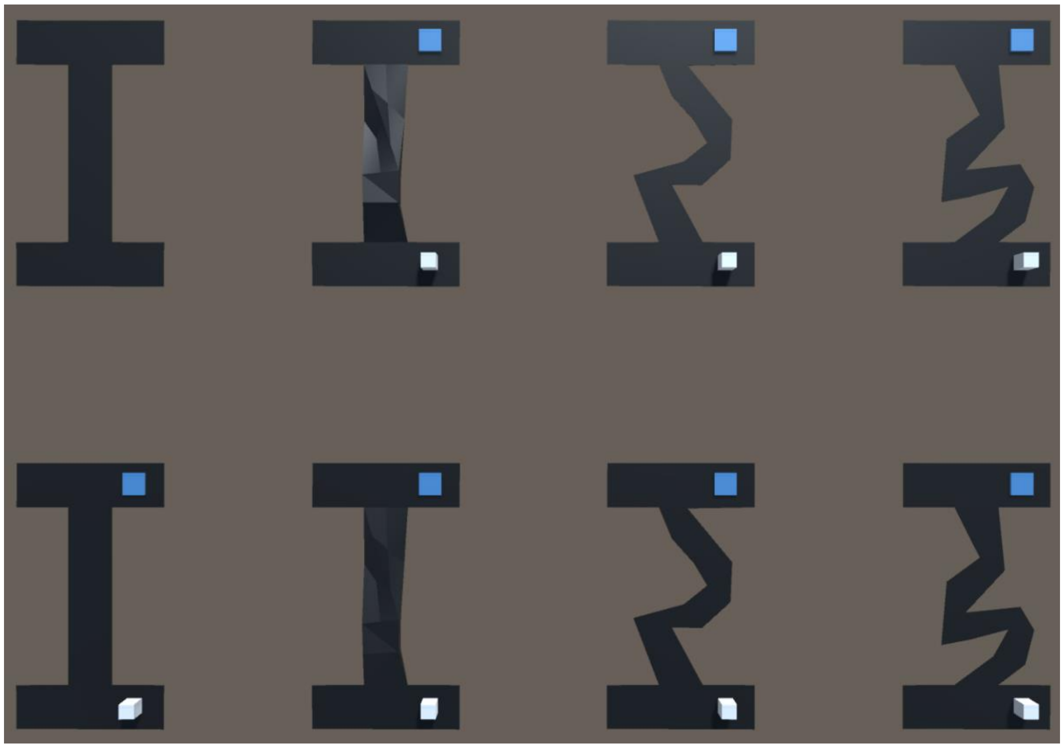
Agent Path Track to Target Example



Note: This is only one example of a common path an agent would take to get to its target since it tends to bounce from wall to wall. It is worth noting that the agent can also curve around corners and jump over walls to reach their target.

Figure 10

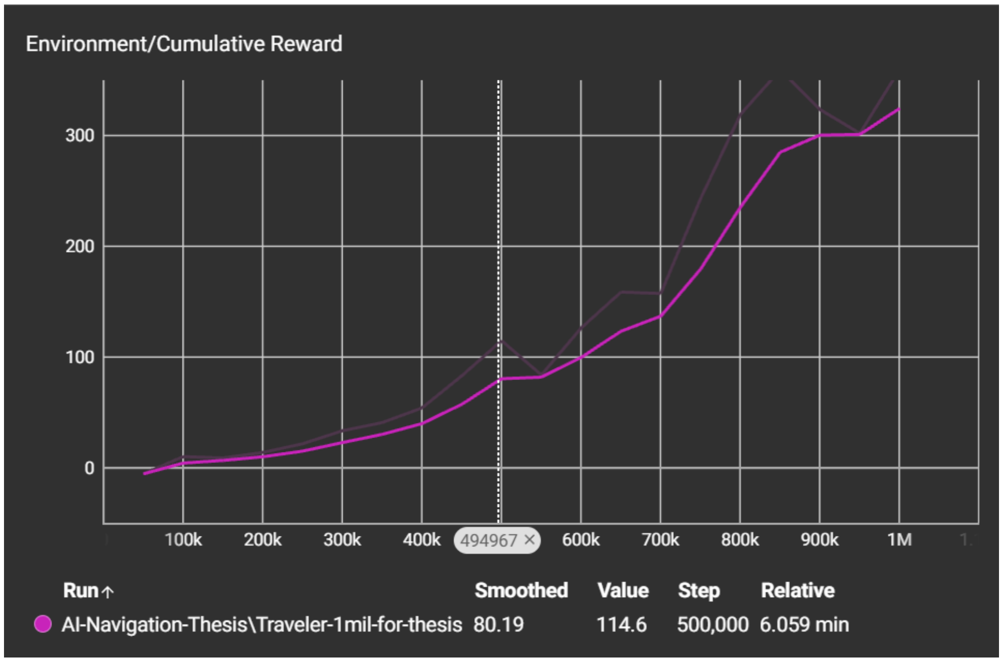
Original Designs of Environments



Note: These environments did not encourage adaptability because of the lack of randomness and similar environment structure besides the bridge layout.

Figure 11

Environment/Cumulative Reward Chart

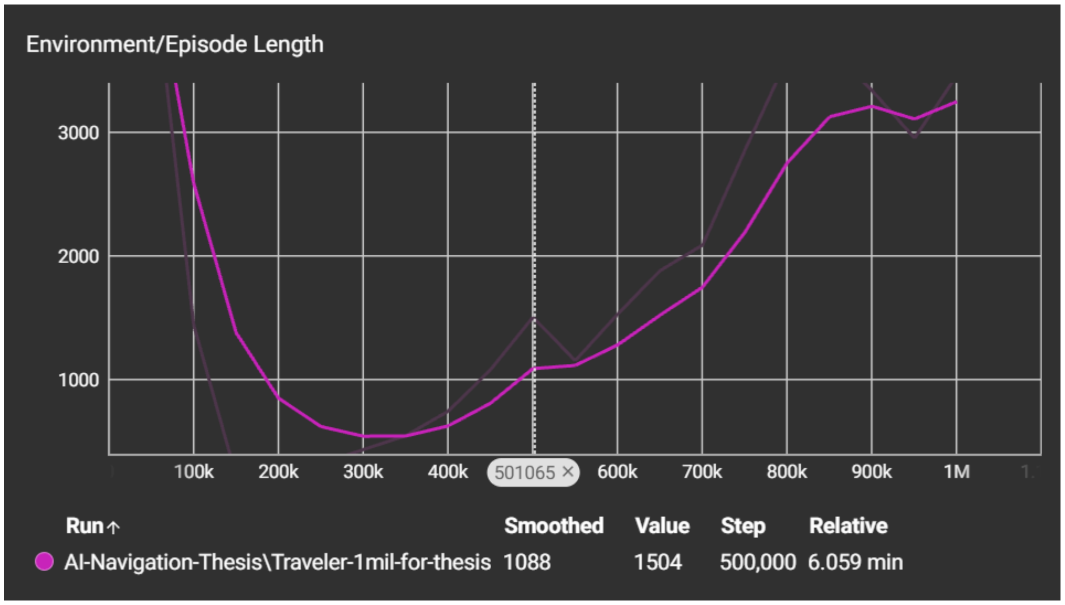


Run↑	Smoothed	Value	Step	Relative
● AI-Navigation-Thesis\Traveler-1mil-for-thesis	80.19	114.6	500,000	6.059 min

Note: This describes the cumulative mean, labeled as "value," of the reward of all agents.

Figure 12

Environment/Episode Length Chart

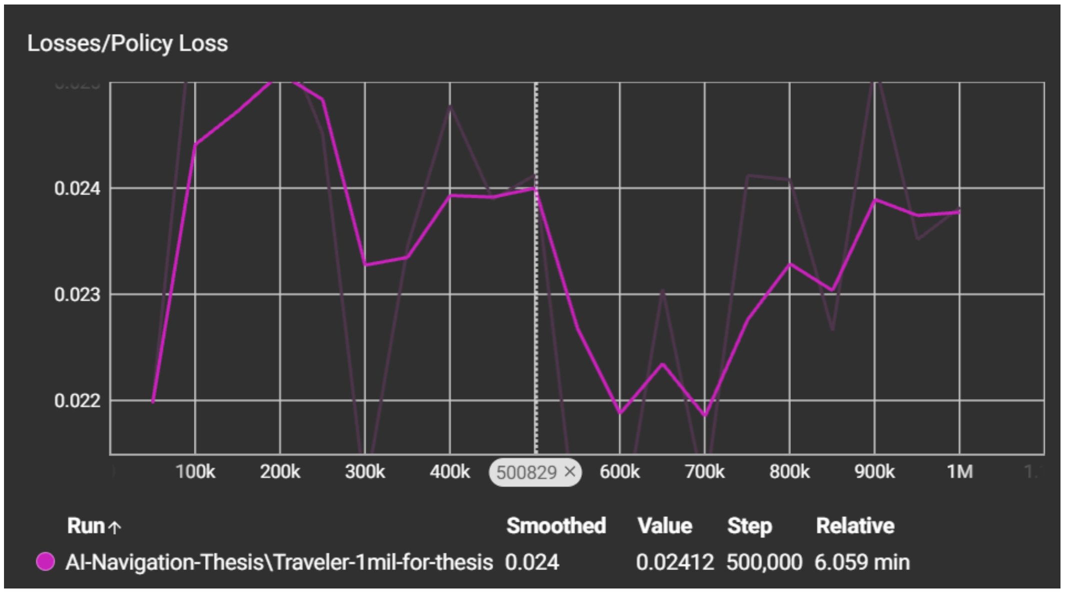


Run↑	Smoothed	Value	Step	Relative
● AI-Navigation-Thesis\Traveler-1mil-for-thesis	1088	1504	500,000	6.059 min

Note: This describes the mean length of each episode of all agents.

Figure 13

Losses/Policy Loss Chart

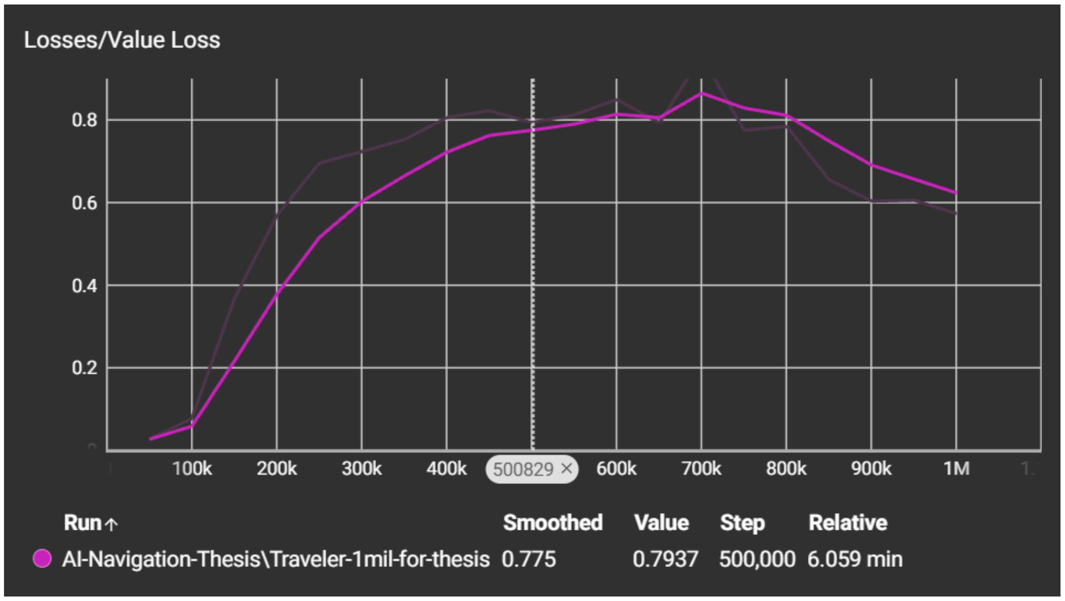


Run ↑ | Smoothed | Value | Step | Relative
● AI-Navigation-Thesis\Traveler-1mil-for-thesis 0.024 0.02412 500,000 6.059 min

Note: This describes the mean magnitude of the policy loss function.

Figure 14

Losses/Value Loss Chart

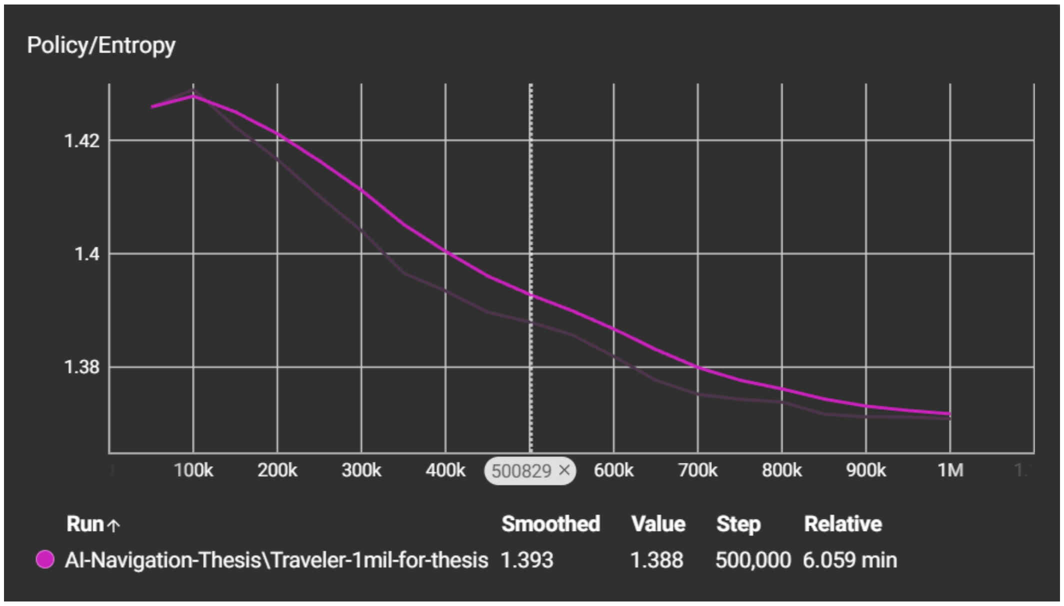


Run ↑ | Smoothed | Value | Step | Relative
● AI-Navigation-Thesis\Traveler-1mil-for-thesis 0.775 0.7937 500,000 6.059 min

Note: This describes the mean loss of the value function update.

Figure 15

Policy/Entropy Chart

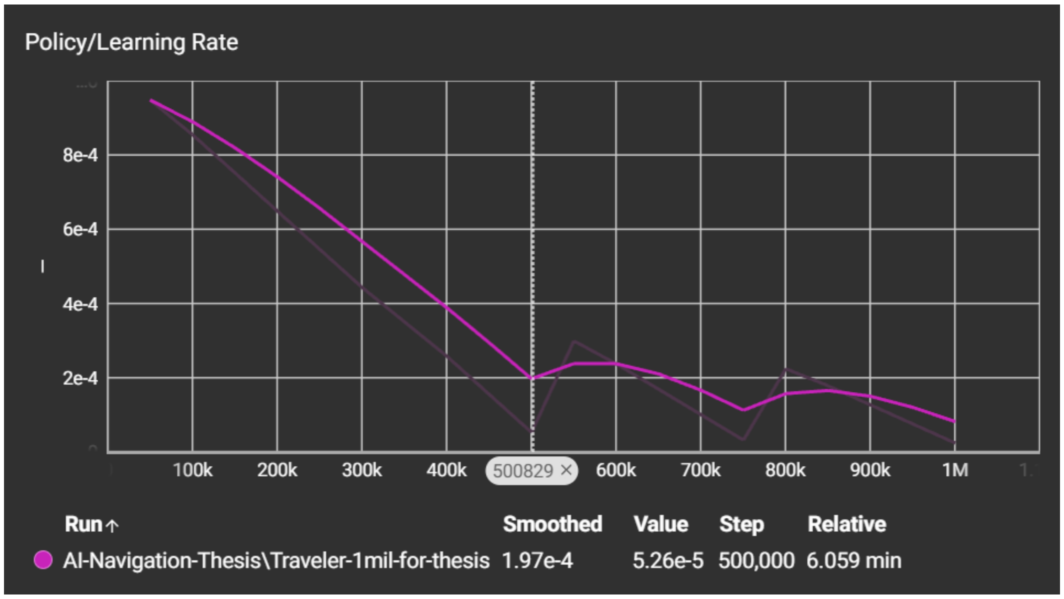


Run ↑	Smoothed	Value	Step	Relative
● AI-Navigation-Thesis\Traveler-1mil-for-thesis	1.393	1.388	500,000	6.059 min

Note: This chart describes how random the decisions of the model are.

Figure 16

Policy/Learning Rate Chart

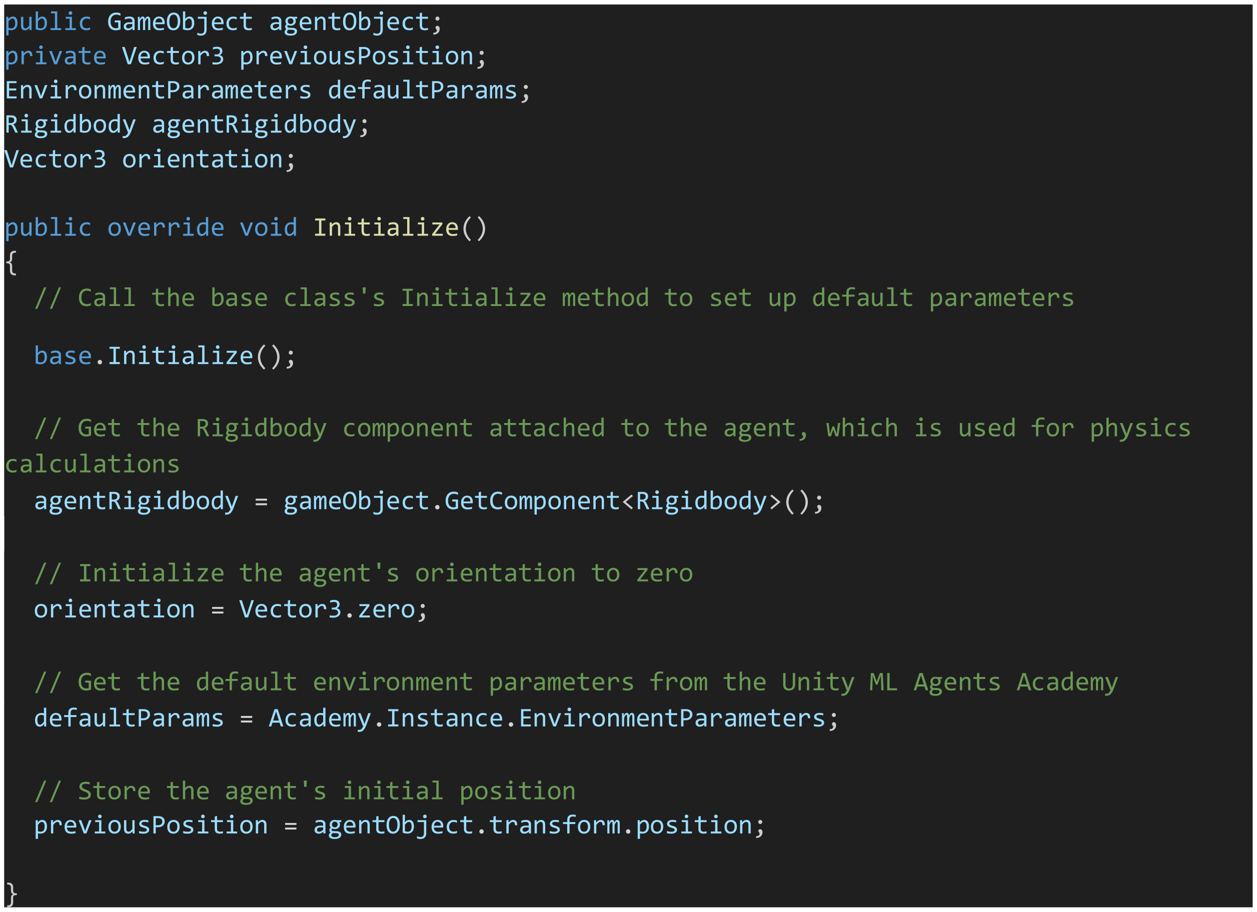


Run ↑	Smoothed	Value	Step	Relative
● AI-Navigation-Thesis\Traveler-1mil-for-thesis	1.97e-4	5.26e-5	500,000	6.059 min

Note: This describes how large of a step the algorithm takes as it searches for the best policy.

Figure 17

Initialize Method in MovingAgent C# Class



```csharp
public GameObject agentObject;
private Vector3 previousPosition;
EnvironmentParameters defaultParams;
Rigidbody agentRigidbody;
Vector3 orientation;

public override void Initialize()
{
  // Call the base class's Initialize method to set up default parameters

  base.Initialize();

  // Get the Rigidbody component attached to the agent, which is used for physics
calculations
  agentRigidbody = gameObject.GetComponent<Rigidbody>();

  // Initialize the agent's orientation to zero
  orientation = Vector3.zero;

  // Get the default environment parameters from the Unity ML Agents Academy
  defaultParams = Academy.Instance.EnvironmentParameters;

  // Store the agent's initial position
  previousPosition = agentObject.transform.position;

}
```

Note: Initializes the agent's physical properties, orientation, position, and environment parameters at simulation start.

Figure 18

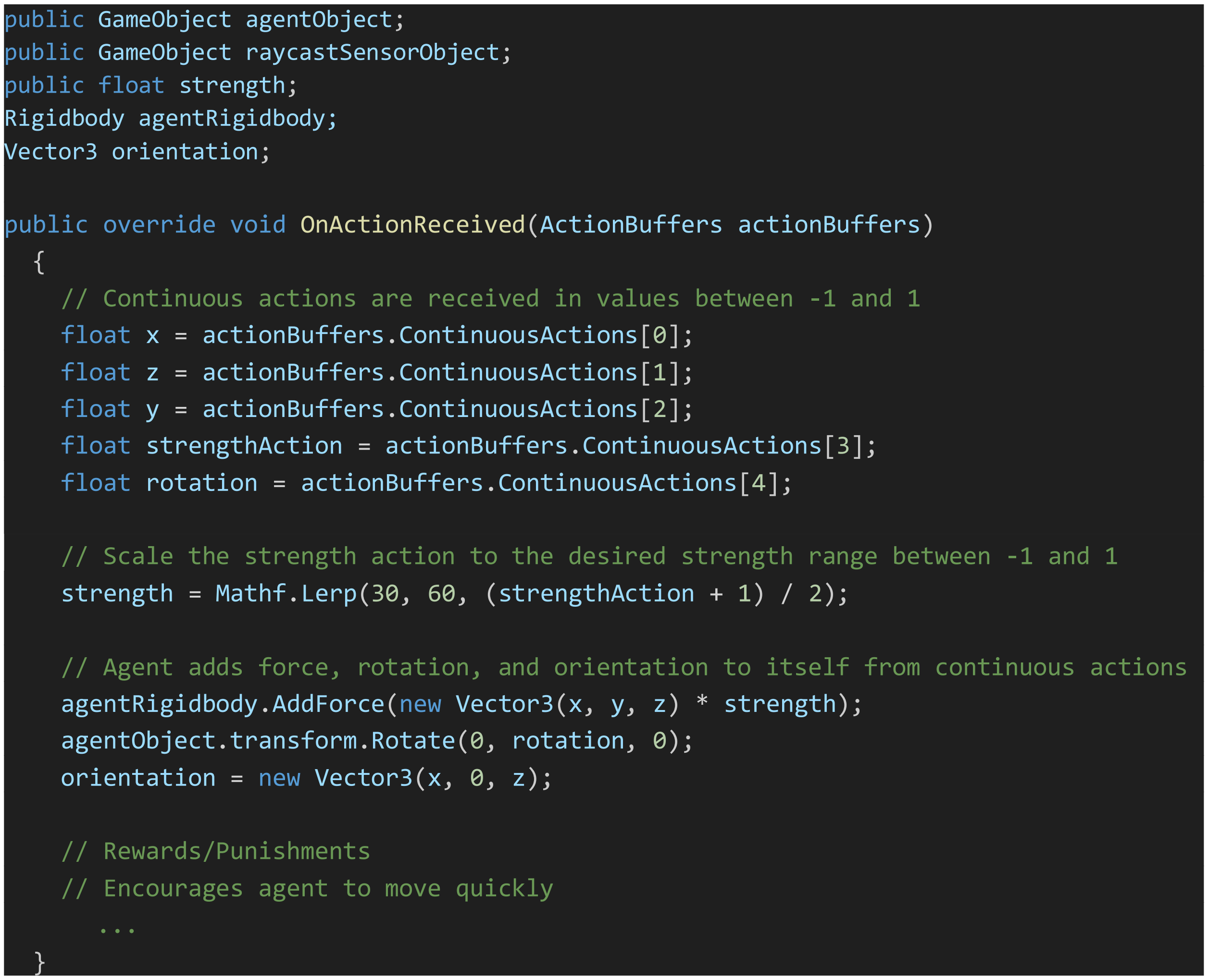
CollectObservations Method in MovingAgent C# Class

```
public GameObject target;
public GameObject agentObject;
Rigidbody agentRigidbody;

public override void CollectObservations(VectorSensor sensor)
  {
    Vector3 relativePosition = target.transform.position - agentObject.trans-
form.position;
    sensor.AddObservation(relativePosition);
    sensor.AddObservation(agentRigidbody.velocity);
  }
```

Note: Collects the current position and velocity of the agent to pass them to the decision-making policy to request an action as a response.

Figure 19

OnActionReceived Method in MovingAgent C# Class

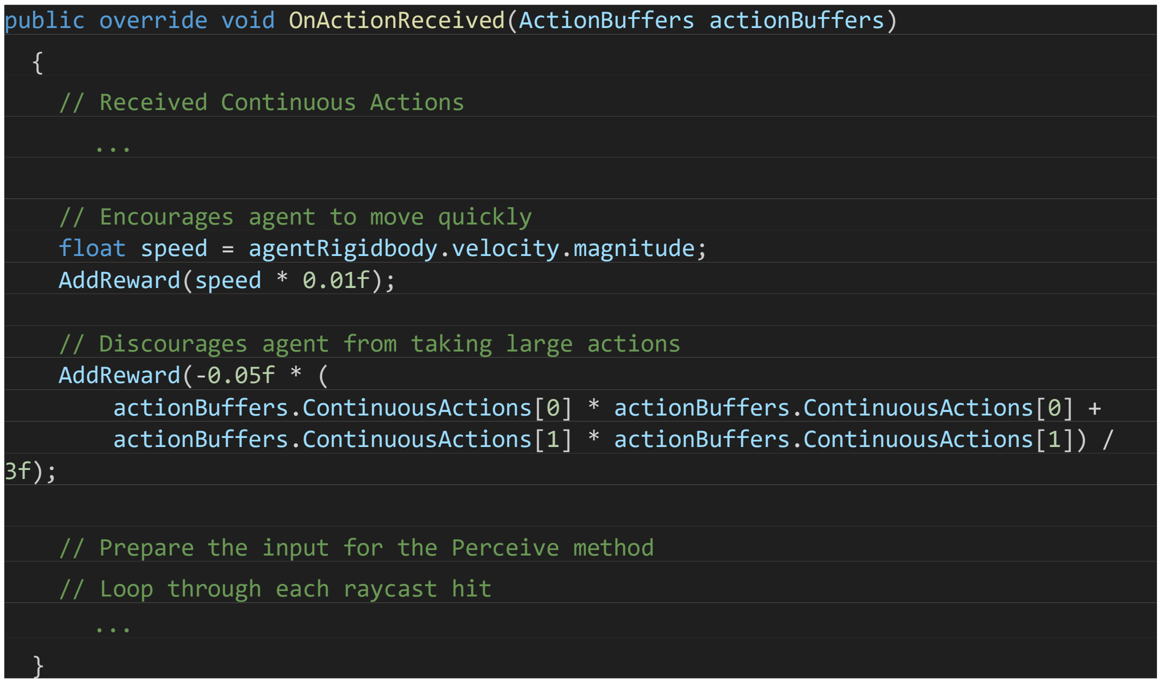


```csharp
public GameObject agentObject;
public GameObject raycastSensorObject;
public float strength;
Rigidbody agentRigidbody;
Vector3 orientation;

public override void OnActionReceived(ActionBuffers actionBuffers)
  {
    // Continuous actions are received in values between -1 and 1
    float x = actionBuffers.ContinuousActions[0];
    float z = actionBuffers.ContinuousActions[1];
    float y = actionBuffers.ContinuousActions[2];
    float strengthAction = actionBuffers.ContinuousActions[3];
    float rotation = actionBuffers.ContinuousActions[4];

    // Scale the strength action to the desired strength range between -1 and 1
    strength = Mathf.Lerp(30, 60, (strengthAction + 1) / 2);

    // Agent adds force, rotation, and orientation to itself from continuous actions
    agentRigidbody.AddForce(new Vector3(x, y, z) * strength);
    agentObject.transform.Rotate(0, rotation, 0);
    orientation = new Vector3(x, 0, z);

    // Rewards/Punishments
    // Encourages agent to move quickly
      ...
  }
```

Note: Receives five continuous actions from the model: x-axis, y-axis, z-axis, rotation, movement strength, and ray casts perception sensors. Then, the agent uses those actions to change its rotation and orientation and adds force to itself to move its position.

Figure 20

OnActionsReceived Method in MovingAgent C# Class



```
public override void OnActionReceived(ActionBuffers actionBuffers)
  {
    // Received Continuous Actions
       ...

    // Encourages agent to move quickly
    float speed = agentRigidbody.velocity.magnitude;
    AddReward(speed * 0.01f);

    // Discourages agent from taking large actions
    AddReward(-0.05f * (
        actionBuffers.ContinuousActions[0] * actionBuffers.ContinuousActions[0] +
        actionBuffers.ContinuousActions[1] * actionBuffers.ContinuousActions[1]) /
3f);

    // Prepare the input for the Perceive method
    // Loop through each raycast hit
       ...
  }
```

Note: Allocates rewards and punishments for speed-related actions.

Figure 21

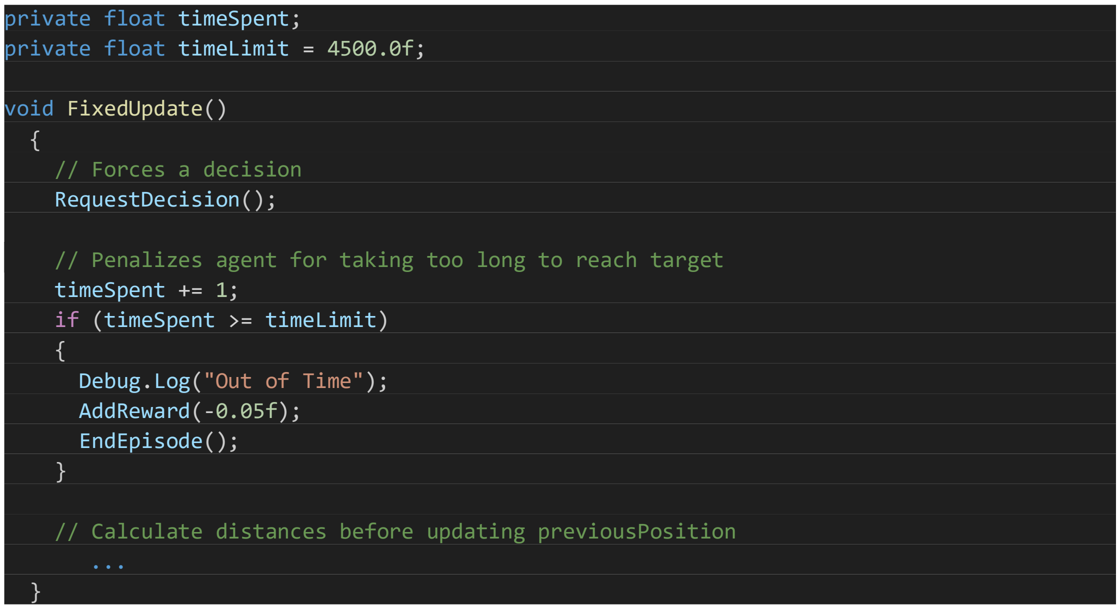
OnActionsReceived Method in MovingAgent C# Class

```
public override void OnActionReceived(ActionBuffers actionBuffers)
  {
    // Received Continuous Actions
       ...

    // Encourages agent to move quickly
       ...

    // Prepare the input for the Perceive method
    var rayPerceptionSensorComponent =
raycastSensorObject.GetComponent<RayPerceptionSensorComponent3D>();
    var rayInput = rayPerceptionSensorComponent.GetRayPerceptionInput();
    var perceptionOutput = RayPerceptionSensor.Perceive(rayInput);

    // Loop through each raycast hit
    foreach (var raycastHit in perceptionOutput.RayOutputs)
    {
      if (raycastHit.HitTagIndex >= 0 && raycastHit.HitTagIndex <
rayPerceptionSensorComponent.DetectableTags.Count)
      {
        string hitTag =
rayPerceptionSensorComponent.DetectableTags[raycastHit.HitTagIndex];
        // Check if the raycast hit a GameObject with a specific tag
        if (raycastHit.HitTagIndex != -1)
        {
          // Encourages agent to go towards a target
           if (hitTag == "Target"){ AddReward(0.0001f); }
          // Discourages agent from hitting/staying on walls
           else if (hitTag == "Wall") { AddReward(-0.0001f); }
        }
        else { Debug.Log("HitTagIndex is out of range."); }
      }
    }
  }
```

Note: Allocates rewards and punishments for Raycasts-related observations.

Figure 22

FixedUpdate Method in MovingAgent C# Class

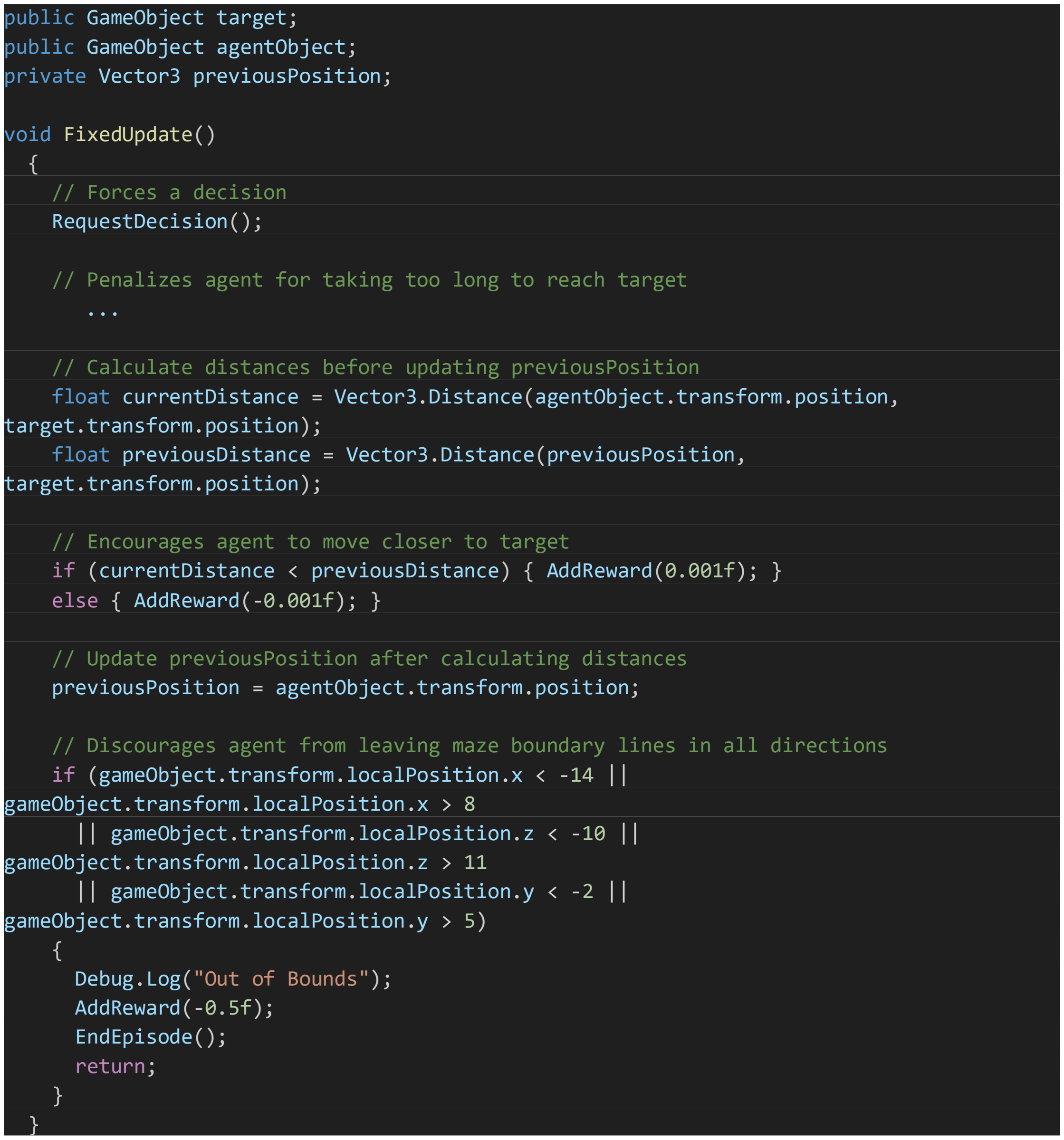


```
private float timeSpent;
private float timeLimit = 4500.0f;

void FixedUpdate()
  {
    // Forces a decision
    RequestDecision();

    // Penalizes agent for taking too long to reach target
    timeSpent += 1;
    if (timeSpent >= timeLimit)
    {
      Debug.Log("Out of Time");
      AddReward(-0.05f);
      EndEpisode();
    }

    // Calculate distances before updating previousPosition
       ...
  }
```

Note: Allocates punishment for agents exceeding the time limit.

Figure 23

FixedUpdate Method in MovingAgent C# Class

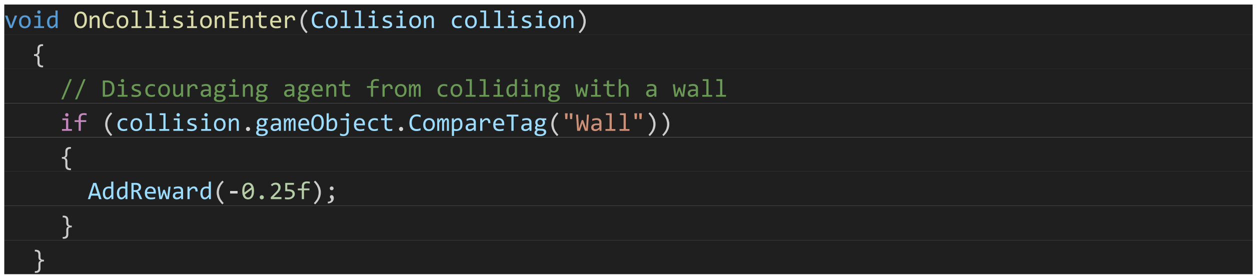


```
public GameObject target;
public GameObject agentObject;
private Vector3 previousPosition;

void FixedUpdate()
  {
    // Forces a decision
    RequestDecision();

    // Penalizes agent for taking too long to reach target
       ...

    // Calculate distances before updating previousPosition
    float currentDistance = Vector3.Distance(agentObject.transform.position,
target.transform.position);
    float previousDistance = Vector3.Distance(previousPosition,
target.transform.position);

    // Encourages agent to move closer to target
    if (currentDistance < previousDistance) { AddReward(0.001f); }
    else { AddReward(-0.001f); }

    // Update previousPosition after calculating distances
    previousPosition = agentObject.transform.position;

    // Discourages agent from leaving maze boundary lines in all directions
    if (gameObject.transform.localPosition.x < -14 ||
gameObject.transform.localPosition.x > 8
      || gameObject.transform.localPosition.z < -10 ||
gameObject.transform.localPosition.z > 11
      || gameObject.transform.localPosition.y < -2 ||
gameObject.transform.localPosition.y > 5)
    {
      Debug.Log("Out of Bounds");
      AddReward(-0.5f);
      EndEpisode();
      return;
    }
  }
```

Note: Allocates rewards and punishments for the agent moving closer to the target, moving away from the target, and moving out of bounds.

Figure 24

OnCollisionEnter Method in MovingAgent C# Class



```
void OnCollisionEnter(Collision collision)
  {
    // Discouraging agent from colliding with a wall
    if (collision.gameObject.CompareTag("Wall"))
    {
      AddReward(-0.25f);
    }
  }
```

Note: Allocates punishment for colliding with objects with the “Wall” tag.

Figure 25

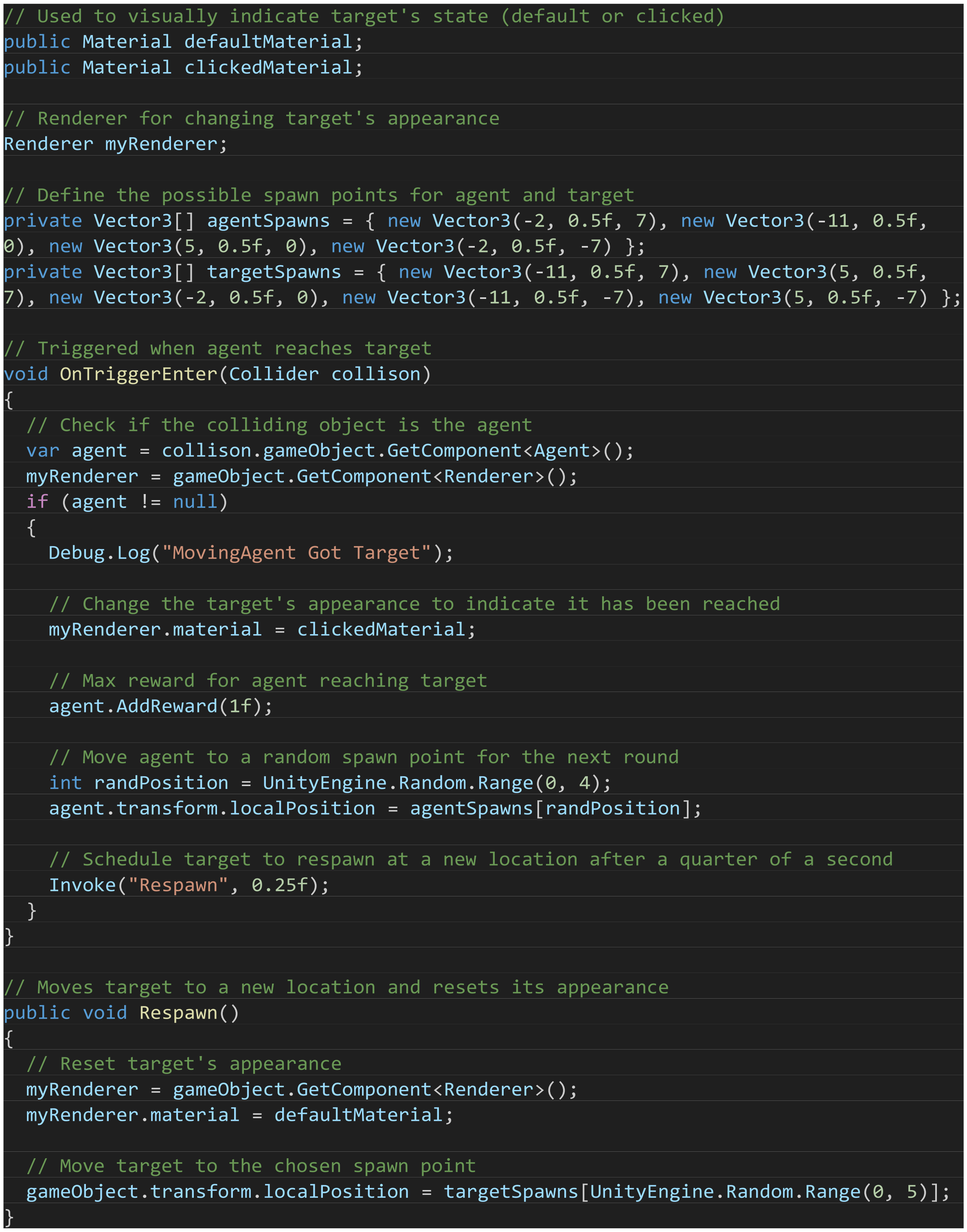
OnEpisodeBegin Method in MovingAgent C# Class

```
private float timeSpent;
private Vector3[] agentSpawns = { new Vector3(-2, 0.5f, 7), new Vector3(-11, 0.5f, 0),
new Vector3(5, 0.5f, 0), new Vector3(-2, 0.5f, -7) };

Rigidbody agentRigidbody;

public override void OnEpisodeBegin()
  {
    // Select a random spawn point for the agent at the start of each episode
    int randPosition = UnityEngine.Random.Range(0, 4);
    gameObject.transform.localPosition = agentSpawns[randPosition];

    // Reset the agent's velocity to ensure it doesn't carry over speed from the
previous episode
    agentRigidbody.velocity = Vector3.zero;

    // A reference to the maze environment
    var environment = gameObject.transform.parent.gameObject;

    // Get all the targets in the environment
    var targets = environment.GetComponentsInChildren<StationaryTarget>();

    // Respawn each target at the start of the episode
    foreach (var t in targets)
    {
      t.Respawn();
    }

    // Reset the time counter tracking how long the agent has been trying to reach
the target
    timeSpent = 0.0f;
  }
```

Note: Prepares environments for a new episode by spawning agents and targets in randomly pre-selected positions across the environment and resetting velocities and elapsed time.

Figure 26

OnTriggerEnter and Respawn Methods in StationaryTarget C# Class



```
// Used to visually indicate target's state (default or clicked)
public Material defaultMaterial;
public Material clickedMaterial;

// Renderer for changing target's appearance
Renderer myRenderer;

// Define the possible spawn points for agent and target
private Vector3[] agentSpawns = { new Vector3(-2, 0.5f, 7), new Vector3(-11, 0.5f,
0), new Vector3(5, 0.5f, 0), new Vector3(-2, 0.5f, -7) };
private Vector3[] targetSpawns = { new Vector3(-11, 0.5f, 7), new Vector3(5, 0.5f,
7), new Vector3(-2, 0.5f, 0), new Vector3(-11, 0.5f, -7), new Vector3(5, 0.5f, -7) };

// Triggered when agent reaches target
void OnTriggerEnter(Collider collison)
{
  // Check if the colliding object is the agent
  var agent = collison.gameObject.GetComponent<Agent>();
  myRenderer = gameObject.GetComponent<Renderer>();
  if (agent != null)
  {
    Debug.Log("MovingAgent Got Target");

    // Change the target's appearance to indicate it has been reached
    myRenderer.material = clickedMaterial;

    // Max reward for agent reaching target
    agent.AddReward(1f);

    // Move agent to a random spawn point for the next round
    int randPosition = UnityEngine.Random.Range(0, 4);
    agent.transform.localPosition = agentSpawns[randPosition];

    // Schedule target to respawn at a new location after a quarter of a second
    Invoke("Respawn", 0.25f);
  }
}

// Moves target to a new location and resets its appearance
public void Respawn()
{
  // Reset target's appearance
  myRenderer = gameObject.GetComponent<Renderer>();
  myRenderer.material = defaultMaterial;

  // Move target to the chosen spawn point
  gameObject.transform.localPosition = targetSpawns[UnityEngine.Random.Range(0, 5)];
}
```

Note: Allocates reward for agent reaching target and resets environment for next episode.