==

```
+----------------------+
| RL PRACTICE HOMEWORK |
+----------------------+
```

==

GROUP MEMBERS

```
- Matthew Bonnecaze  (bonnem3)
- Justin Hung        (hungj2)
- Phillip Stapleton  (staplp2)
```

==

GITHUB REPOSITORY

- https://github.com/MBon1/GAI-RL-Practice

==

Q-LEARNING TRAINING AND RESULTS

For Q-learning scenes, parameters for training can be changed by modifying the Max_steps, Max_iterations, and Report_on_Iteration variables.

Max_steps is the number of steps the agent is allowed to make in one iteration before a restart is forced. The scene will end if the max number of steps is used or if the agent reaches the goal.

Max_iterations is the number of iterations (episodes) that the agent will carry out before training is complete. For all of our scenes, the default value is around 2000 iterations; though some scenes (like the 10x10 maze) may converge much sooner than that.

Report_on_Iteration is the period over which reports will be recorded into the results CSV file (i.e. every 5 iterations). The values recorded between the first and last iteration in this period will be averaged. For example, if Report_on_Iteration is 5, an entry stating the average score over every period of five iterations will be reported.

The results for Q-learning runs are saved as CSV files and can be found in Assets/Results. These files, which store the iteration number in the A column and the cumulative reward value in the B column, can be opened in an application like Microsoft Excel, Apple Numbers, or Google Sheets, where you will be able to convert the data into a graph.

==

```
+------------+
| MAZE SCENE |
+------------+
```

+ OVERVIEW AND OBJECTIVE

In our maze scene, the agent begins in the bottom-left corner (0,0) of the maze. The goal of the agent is to reach the goal of the maze located in the top corner (nx - 1, nz - 1), where nx and nz are the number of rows and columns in the maze, respectively. Our sample scene is a 10x10 maze with five obstacles.

The agent is represented as a yellow cube. The goal is represented by a tan cube in the top right corner of the maze. Obstacles are represented by black wireframe cubes.

+ SCORING DETAILS

To disincentivize the agent from simply wandering around, we punish the agent with -1 points for moving to another square.

Because of the way that our Q-learning and mlagents (PPO) algorithms work, there are slight differences in punishment for hitting an obstacle. Hitting an obstacle creates a -100 reward for Q-learning implementation Whereas a -1 reward point is given for the mlagents (PPO) implementation.

Reaching the goal is the only way for the agent to actually gain a positive reward. Reaching the goal gives the agent a reward of 100, which is large enough to incentivise the agent to seek out the end of the maze rather than waiting out the clock and using all its given moves.

+ RUNNING THE SCENE

Please note that SimpleMazeMLA.unity is an unused scene.

Q-Learning:

In Assets/SimpleMaze/SimpleMaze.unity (NOT SimpleMazeMLA.unity or SimpleMazeMLA2.unity), you can modify the Alpha and Gamma values for the Q-Learning algorithm as well has how many iterations to run the training for in the QL Maze component on the Player game object. Press the play button and run the scene to begin training. When the scene is run, a CSV is created to report the training results and will be updated with the reported reward values at the specified rate. The CSV will be located in Assets/Results/QLMaze named "QLMaze_" followed by the year, month, day, hour, minute, and second the file was created.

mlagents (PPO):

In a Python/Anaconda terminal, navigate to the MLAgents folder in the GitHub repository. Run the mlagents-learn command using the trainer_config.yaml. The training environment is located in the scene Assets/Results/SimpleMazeMLA2.unity (NOT SimpleMaze.unity or SimpleMazeMLA.unity). Make sure that the Behavior Name in the Behavior Parameter Component on all Players matches the one you want to use in the trainer_config.yaml file. There are three provided behaviors: "QLMaze" (the one used to produce the data seen in Graph 1 of the ML-Agents results subsection of the Findings section of Maze Scene), "QLMaze_1" (the behavior used to produce the data seen in Graph 2, the Alpha test, of the ML-Agents results subsection of the Findings section of Maze Scene), and "QLMaze_2 (the behavior used to produce the data seen in Graph 3, the Gamma test, of the ML-Agents results subsection of the Findings section of Maze Scene). Press the play button and run the scene to begin training.

+ Q-LEARNING IMPLEMENTATION

Q-learning in particular works best with environments where there are a finite number of states. Our Q-table is defined as a Dictionary<Position, Dictionary<Direction, Reward>>. In this case, there are 100 different tiles for a 10x10 grid, and there are four actions per square (up, down, left and right), leading to a total of 400 possible states.

+ MLAGENTS (PPO) IMPLEMENTATION

At the beginning of each episode, the agent's position is set to (0,0), located at the bottom-left corner of the maze. The velocity and angular velocity of the agent are both set to the zero-vector. The observations being collected each step are the position of the target, the position of the agent, and the x and z values of the agent's velocity. Through Ray Perception Sensor 3D, 8 evenly spaced rays are casted checking for the walls and obstacles.

During an action, a force determined by the vectorAction parameter of OnActionReceived() is applied to the agent to move the agent. If the agent collides into the target, a reward of 100 is added and the episode ends. If the agent collides or stays colliding into a wall or an obstacle, a reward of -1 is added. Each step, a small negative reward (about -0.001) is added.

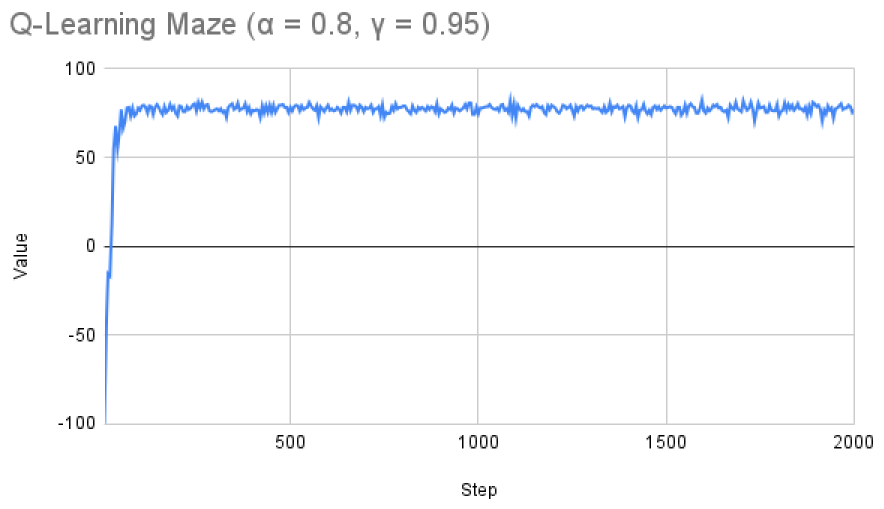
+ FINDINGS

Q-Learning:

At the start, the agent wanders sporadically around the maze in an attempt to explore and gather data. Over time, however, the agent will learn that moving towards the goal and avoiding the top right corner where it has the opportunity to get stuck is good. It should be noted that because the algorithm goes through each possible direction in a set order when exploring, there is a slight bias towards first exploring upwards and then to the left.

Because there are so few possible states, this scene trains incredibly quickly. However, it stands to reason that an agent in a much larger maze would take significantly longer to train. It takes about 30 seconds or so for everything to converge in our scene, and there is a very sharp S curve as shown in our data. The best possible score that this can get is 82, calculated by taking 100 (the goal reward) minus 9 (the number of moves right) minus 9 (the number of moves up).

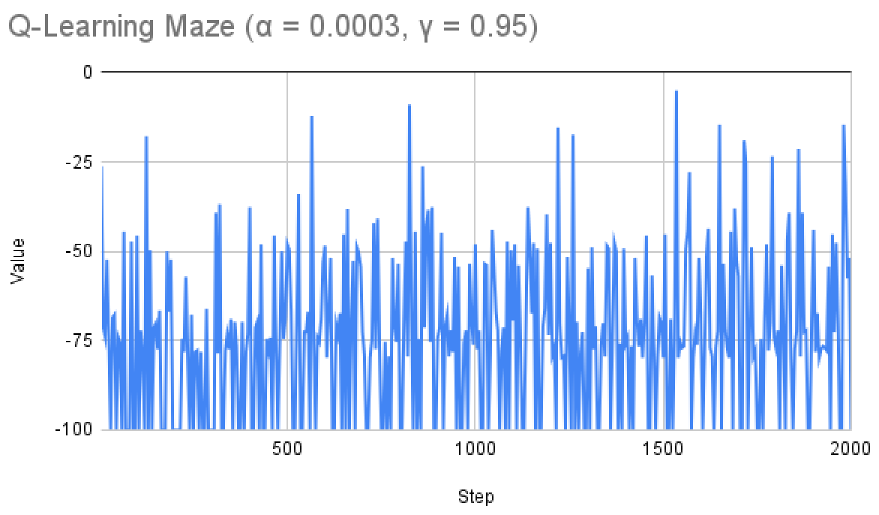
Graph 1 (Base Data)



Learning Rate (Alpha): 0.8
Gamma: 0.95

This is the base metric against which we will compare our Q-learning alpha and gamma tests. For this test, we set the alpha value (learning rate) to 0.8 and the gamma value (discount factor) to 0.95. Our Q-learning implementation for the maze converges very quickly and eventually hovers at the best possible reward value: 82 points.

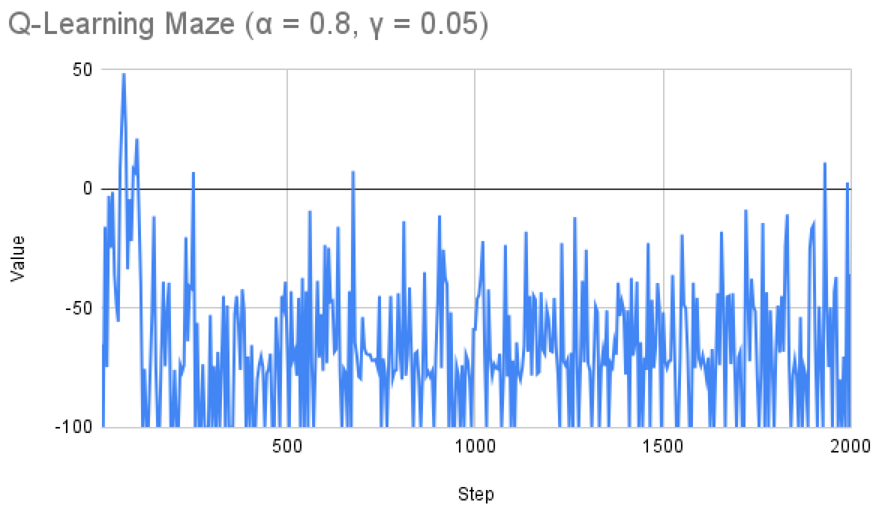
Graph 2 (Alpha Test)



Learning Rate (Alpha): 0.0003
Gamma: 0.95

The Alpha value is the learning rate, ergo how fast the agent learns. The Alpha in this test was almost 0 compared to Base Data for Q-Learning Maze (alpha = 0.8). As such, we expected to see the cumulative reward values to not increase as fast as Graph 1 in this section. However, we did not anticipate for the agent to only have negative reward values.

Graph 3 (Gamma Test)

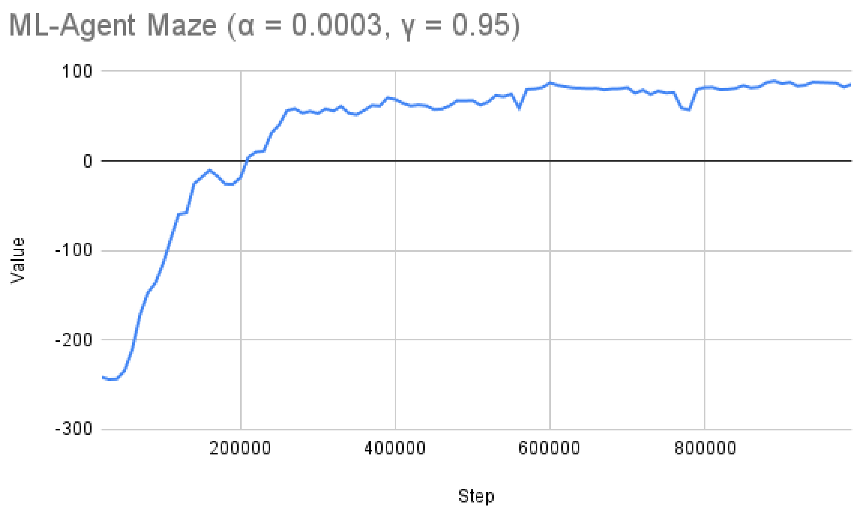


Learning Rate (Alpha): 0.8
Gamma: 0.05

The Gamma value is the discount factor, how much the agent favors future rewards more than immediate rewards. The Gamma in this test was almost 0, a stark contrast to the almost 1 value in our base test for our Q-Learning Maze agent. At first, we anticipated that the agent would reach a similar result to the results shown in Graph 1 of this section, and the agent in this test having a cumulative reward value of almost 50 led us to further expect this. However, we did not anticipate for the reward values to return below -50 on average. This may be because the agent does not value the future rewards as much, so it will not care as much about colliding into obstacles and not arriving at the goal as fast.

mlagents (PPO):

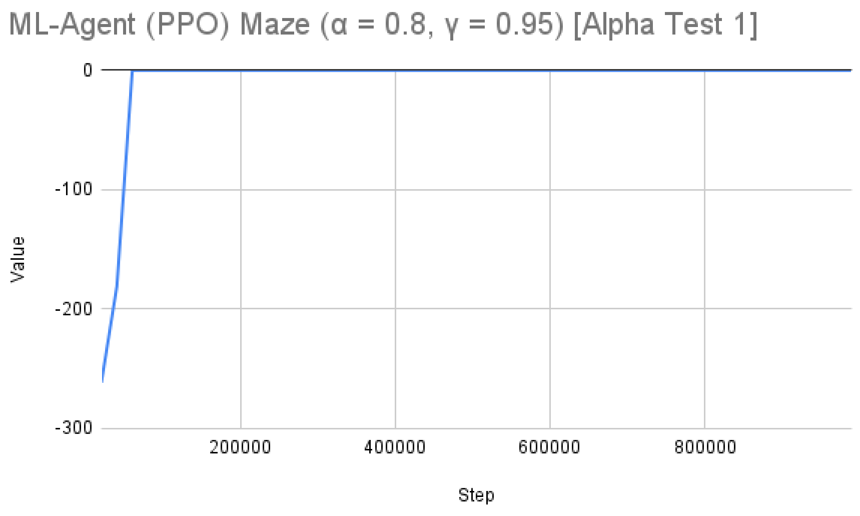
Graph 1 (Base Data)



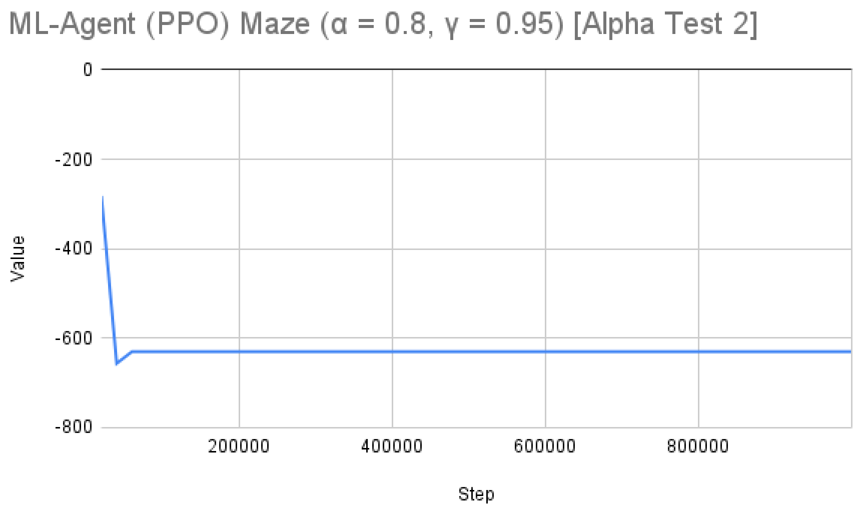
Learning Rate (Alpha): 0.0003
Gamma: 0.95

The biggest thing to note in the initial maze is that our Q-learning seems to be much faster than ML-Agents in this case. As was mentioned in the implementation section, Q-learning is done with a small, finite set of states, so the Q-learning version does not take very long to converge. The ML-Agents implementation, on the other hand, has more degrees of freedom to work with, but still converges with an expected S curve nonetheless.

Graph 2 (Alpha Test)



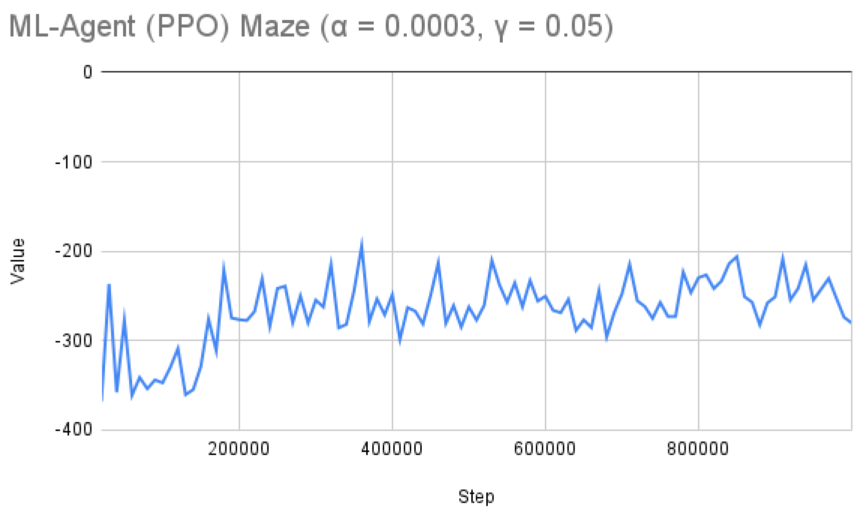
Learning Rate (Alpha): 0.8
Gamma: 0.95



Learning Rate (Alpha): 0.8
Gamma: 0.95

The Alpha value is the learning rate, or in other words how fast the agent can learn. The Alpha in this test was 0.8 compared to Base Data for ML-Agent (PPO) Maze (alpha = 0.0003), a significantly larger learning rate; however, this Alpha value was chosen to match the Alpha value used to produce the Graph 1 (Base Data) for the Q-Learning Maze agent. We expected the agent to converge to learn quicker than in the base example. This is because the agent will value previous information and training over rewards potentially in the future. We were surprised that it only took a couple thousand iterations for the reward values to converge. As seen in both graphs, the reward value plateaus once it reaches a certain amount. What is interesting is that the given reward value does not always converge at the same value. We also learned here that the ML-Agent PPO algorithm likely considers a lower Alpha value with a higher learning rate whereas the inverse occurs in our Q-Learning algorithm.

Graph 3 (Gamma Test)



Learning Rate (Alpha): 0.0003
Gamma: 0.05

Being the discount factor, Gamma determines how much more future rewards are favored by the agent than immediate rewards. The Gamma in this test was almost 0, a stark contrast to the almost 1 value in our base test for our Q-Learning Maze agent. The cumulative reward values were, as we expected based on our results from our equivalent Q-Learning test, in the negatives. With how low the values are, this may further suggest that the closer to 0 the Gamma value is, the less the agent cares about colliding into obstacles and arriving at the goal as fast.

==

```
+--------------------+
| ROLLER BALL SCENE |
+--------------------+
```

+ OVERVIEW AND OBJECTIVE

The Rollerball environment is based off of the Rollerball environment from the tutorial “Making a New Learning Environment” in the ml-agents GitHub documentation (https://github.com/Unity-Technologies/ml-agents/blob/release_1/docs/Learning-Environment-Create-New.md). In the Rollerball environment, there is an agent (a red ball) and a target (a green cube) on a floor. The agent is trying to get to the target without falling off of the floor. While in the original Rollerball the target's position is randomly determined at the start of each iteration, in our episode, the agent's position is randomly determined at the start of each episode.

+ RUNNING THE SCENE

Q-Learning:

In Assets/Rollerball/RollerballQL.unity, you can modify the Alpha and Gamma values for the Q-Learning algorithm as well has how many iterations to run the training for in the QL Roller Agent component on the RollerAgent game object in the training environment. Press the play button and run the scene to begin training. When the scene is run, a CSV is created to report the training results and will be updated with the reported reward values at the specified rate. The CSV will be located in Assets/Results/QLRollerball named "QLRollerball_" followed by the year, month, day, hour, minute, and second the file was created.

mlagents (PPO):

In a Python/Anaconda terminal, navigate to the MLAgents folder in the GitHub repository. Run the mlagents-learn command using the trainer_config.yaml. The training environment is located in the scene Assets/Rollerball/RollerballMLA.unity. Make sure that the Behavior Name in the Behavior Parameter Component on all RollerAgents matches the one you want to use in the trainer_config.yaml file. There are three provided behaviors: "RollerBallMLA" (the one used to produce the data seen in Graph 1 of the ML-Agents results subsection of the Findings section of Roller Ball Scene), "RollerBallMLA_1" (the behavior used to produce the data seen in Graph 2, the Alpha test, of the ML-Agents results subsection of the Findings section of Roller Ball Scene), and "RollerBallMLA_2 (the behavior used to produce the data seen in Graph 3, the Gamma test, of the ML-Agents results subsection of the Findings section of Roller Ball Scene). Press the play button and run the scene to begin training.

+ Q-LEARNING IMPLEMENTATION

Q-learning in particular works best with environments where there are a finite number of states. Our Q-table is defined as a Dictionary<Position, Dictionary<Direction, Reward>>. In this case, there are 100 different tiles for a 10x10 grid, and there are four actions per square (up, down, left and right), leading to a total of 400 possible states. At the beginning of each episode, the agent's position is set to a random tile in the environment.

During each step, an adjacent position in the cardinal directions is determined. If the agent reaches the target, a reward of 100 is given. If the agent falls off of the training area, a reward of -100 is given. Each step, a -1 reward is given if the agent has neither fallen off of the training area or has reached the target.

+ MLAGENTS (PPO) IMPLEMENTATION

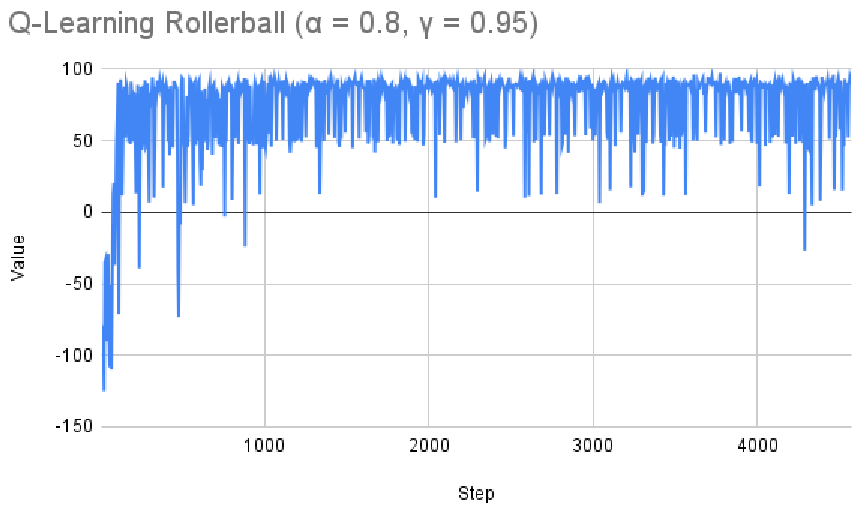
At the beginning of each episode, the velocity and angular velocity of the agent are both set to the zero-vector. The agent's position is set to a random position in the environment. In the original Rollerball, the agent's position would always be set to (0,0) only if the agent fell off of the training area. Further unlike the original Rollerball, the position of the target remains constant. The observations being collected are the position of the target, the position of the agent, and the x and z values of the agent's velocity.

During an action, a force determined by the vectorAction parameter of OnActionReceived() is applied to the agent to move the agent. If the agent collides into the target, the reward is set to 1 and the episode ends. If the agent falls off of the training area, the episode ends.

+ FINDINGS

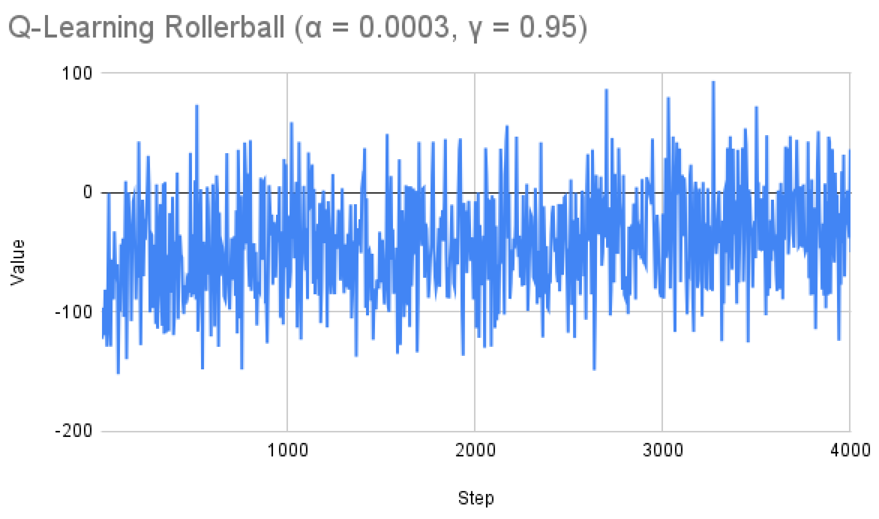
Q-Learning:

Graph 1 (Base Data)



Learning Rate (Alpha): 0.8
Gamma: 0.95

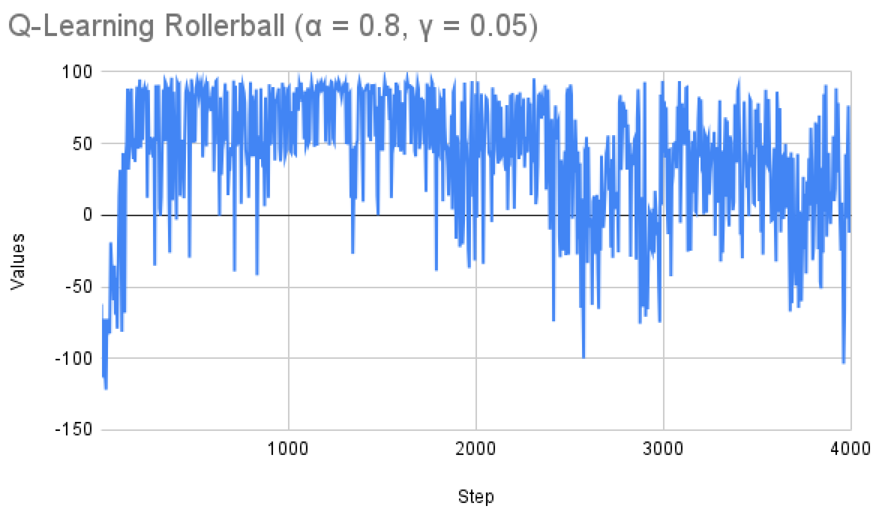
Graph 2 (Alpha Test)



Learning Rate (Alpha): 0.0003
Gamma: 0.95

For this test, we reduced the learning rate from 0.8 to 0.0003. This performed very poorly, which was about as well as we had expected it to, with the trend mostly staying in the negatives. This was likely due to the agent aimlessly wandering around and falling off. We were, however, surprised by the slight upward trend over the 4000 iterations. The increase represents a slow yet arguably steady growth.

Graph 3 (Gamma Test)

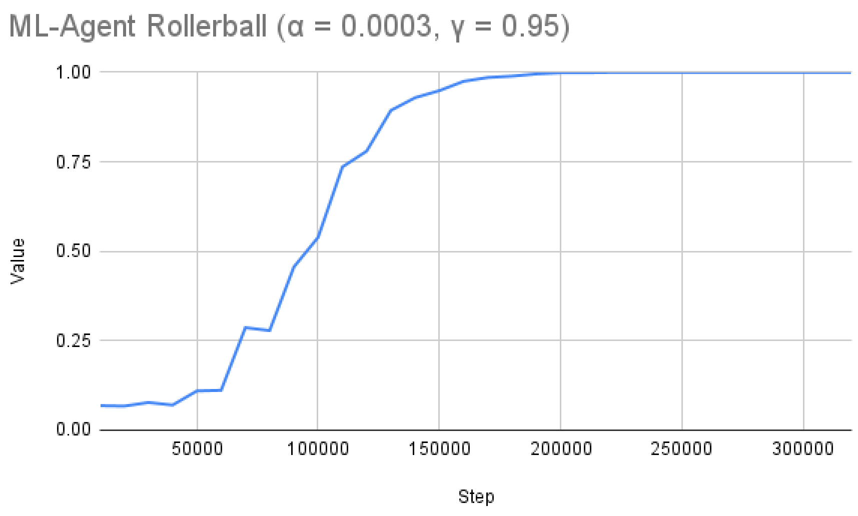


Learning Rate (Alpha): 0.8
Gamma: 0.05

In this test we decreased the gamma value (discount factor) from 0.95, as used in the base training, to 0.05. The reward value for this test appeared to have plateaued, but surprisingly decreased after many iterations. What was especially surprising was that when it plateaued, the graph appeared to be very similar to Graph 1 (Base Graph) in this section. We expected that this test would perform much worse overall, but only began to perform differently from the Base Test near the end of its iterations when the cumulative reward value began to dip. This behavior is possibly influenced by the setup of our scene, but is unlikely that it is due to good spawn points of the agent.

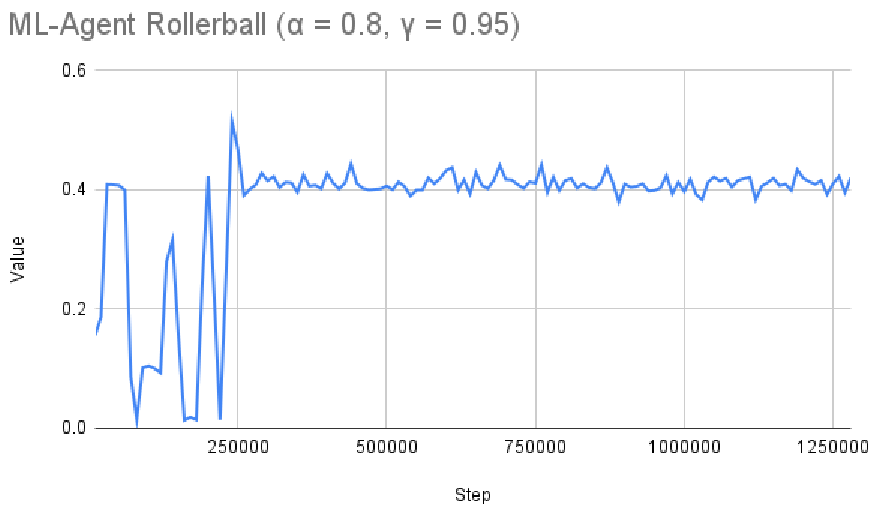
mlagents (PPO):

Graph 1 (Base Data)



Learning Rate (Alpha): 0.0003
Gamma: 0.95

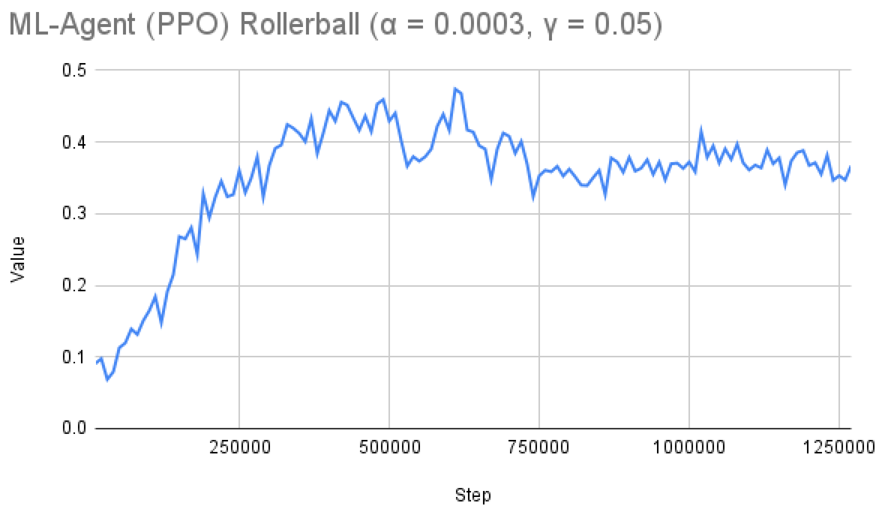
Graph 2 (Alpha Test)



Learning Rate (Alpha): 0.8
Gamma: 0.95

For this test, we increased the alpha value (the learning rate) to 0.8 from the base value of 0.0003. We expected this to perform similarly to the Maze test for Ml-Agents with a larger alpha value. In a way, it did; the value plateaued slightly quicker than in the base test. What was surprising was that the reward converged to a positive value. This is likely due to differences in our implementations. This test run was clearly less efficient than the base test as the reward the agent received by the end of this test was approximately 0.45 compared to the almost 1 reward value in the Base Graph.

Graph 3 (Gamma Test)



Learning Rate (Alpha): 0.0003

    Gamma: 0.05

In this test, we decreased the alpha value (the exploration value) from 0.8 to 0.0003. Similarly to what we observed in the equivalent Q-learning test, the resulting values grew, then the cumulative reward value decreased. As we observed with the Rollerball alpha test, the rewards ended up being less than half of the value that the base test converged to, which was a nearly perfect score. As with the alpha test, the fact that the ML-Agent gamma test has positive values whereas the equivalent Q-learning gamma test in the roller environment is primarily negative is likely due to implementation differences.

General Q-Learning and PPO Discussion

Looking at our base training results in both environments (the maze and Rollerball), our Q-Learning algorithm appears to converge much faster than the PPO algorithm used by ML-Agents. The consistency and accuracy of ML-Agent’s PPO algorithm, however, proved to be more accurate once it converged than our Q-Learning algorithm. Another advantage of ML-Agent’s PPO algorithm that we noticed is that multiple arenas can be used for training whereas our Q-Learning algorithm only supports one instance of the training environment. This potentially is what allowed for PPO to produce much more accurate results in less time. In general, for our environments, the time it took to train an agent using our Q-Learning algorithm was over twice as long as the time it took to train an agent using the ML-Agent’s PPO algorithm for less than half the number of iterations.

```
================================================================

+----------------------+
| BONUS: HALLWAY SCENE |
+----------------------+
```

+ OVERVIEW AND OBJECTIVE

We also created a bonus scene that attempts to emulate a discretized version of the Hallway ML-Agents example scene provided by Unity. Our scene, much like the other Q-Learning scenes, is grid based.

The hallway scene is a smaller 3x10 grid when compared to the original ML-Agents hallway scene. In our Q-Learning hallway scene, the two back corners represent two doors, one of which is red and the other of which is blue at random. These doors are represented by the red and blue cubes.

There is also a sign in the middle of the hallway that is randomly placed that indicates which of the two doors is the correct exit to the hallway. The sign is represented as either a blue or red sphere. The agent, just like with the maze, is represented by a yellow cube. There are no obstacles in this scene.

+ RUNNING THE SCENE

In Assets/SimpleHallway/SimpleHallway.unity, press the play button and run the scene to begin training. When the scene is run, a CSV is created to report the training results and will be updated with the reported reward values at the specified rate. The CSV will be located in Assets/Results/QLHallway named "QLHallway_" followed by the year, month, day, hour, minute, and second the file was created.

+ SCORING DETAILS

In this scene, the agent receives 200 points for making it through the correct door. The agent receives a reward of -1 point for walking around aimlessly, and the agent receives a reward of -100 points for walking through the incorrect door. If the agent walks past or into the sign, it will "tell the agent" what the correct door is, and the agent will have to eventually learn how to make it to the correct gate and learn that the incorrect gate will yield a negative reward over hundreds of iterations.

+ Q-LEARNING IMPLEMENTATION

Unlike with the simple maze, there is a much wider set of states at each square. There are four possible actions (directions that the agent can move in) at a square, but unlike last time, there are six possible states -- condition combinations -- per square.

The states for a square are as follows:

```
- The left goal is red and the target goal is red       // false, 0
- The left goal is red and the target goal is blue      // false, 1
- The left goal is red and the target goal is unknown   // false, 2
- The left goal is blue and the target goal is red      // true, 0
- The left goal is blue and the target goal is blue     // true, 1
- The left goal is blue and the target goal is unknown  // true, 2
```

Whether or not the left goal is blue or red is controlled by a boolean called "blue_is_a" (false if the "A" goal is red and true if the "A" goal is blue). The knowledge of whether or not the target goal is controlled by an integer called "blue_is_goal" (0 if the target goal is red, 1 if the target goal is blue, and 2 if the target goal is unknown).

Therefore, the "state" in our Dictionary<State, Dictionary<Direction, Reward>> Q-table is a string defined as $"{current_position}-{blue_is_a}-{blue_is_goal}", leading to a total of 6 different states per square with four possible actions for each state. This results in 24 * nx * ny different possibilities. Slowly over time, the agent is supposed to learn to associate knowing the correct goal with going to the correct goal. The agent is also eventually supposed to learn that entering the wrong goal is bad.

+ MLAGENTS (PPO) IMPLEMENTATION

For an example of how this scene is supposed to work, see the provided scene located at Assets/Ml-Agents/Examples/Hallway/Scenes/Hallway.

+ FINDINGS

Unlike the maze scene, training takes much longer. There are 720 different possible actions, so training can take upwards of a couple minutes. Unfortunately, while our agent does seem to learn, it appears to be learning incorrect behavior.

With our current implementation, the agent became too afraid of entering the wrong goal, which had the negative consequence of the agent slowly but surely learning to avoid the back end of the hallway altogether. It gets less and less curious and stops exploring the maze and eventually learns that simply staying at the front and not taking the risk of moving up the hallway and potentially entering the wrong door is generally better than getting the answer wrong and being severely punished.

While the raw score in the CSV file may not show any interesting data aside from a convergence at -50, playing the scene in the editor allows us to visualize the trend in learned behavior as it slowly learns to avoid every row starting from the backmost and moving to the front.

==

IMPORTANT SCRIPTS

- QLMaze.cs	(Q-Learning script in SimpleMaze.unity)
- QLMazeAgent3.cs	(ML-Agent script in SimpleMazeMLA2.unity)
- QLRollerAgent.cs	(Q-Learning script in RollerballQL.unity)
- RollerAgent.cs	(ML-Agent script in RollerballMLA.unity)
- QLHallway.cs	(Q-Learning script in SimpleHallway.unity)

==