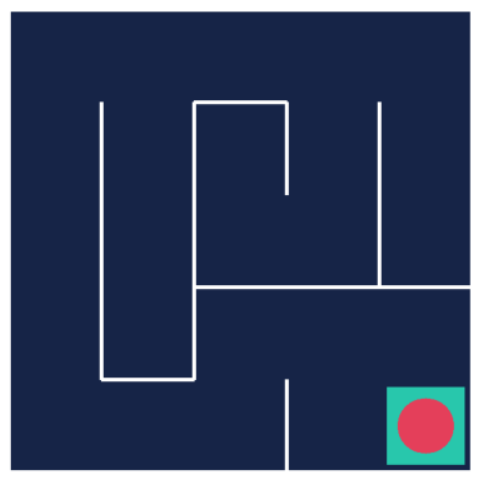
Reinforcement Learning on Maze and Gymnasium Environments

David Macy

Task 1A

Using an epsilon-greedy method to train an agent to solve a predefined maze environment with random start locations.

The challenge here is getting an agent to solve a predefined 5x5 maze as shown in figure 1a when it starts in a different location every episode. The goal here is to understand how many episodes it takes to successfully train an agent to solve the maze. For this particular problem I decided to use an on policy monte carlo algorithm to help train the agent. The on policy monte carlo is designed to train an agent to learn the optimal policy for the maze environment. During the training process, the agent interacts with the maze by taking actions based on its current state and a policy function. The initial policy set is a random one. The method uses episodes, each consisting of a series of steps, to gather experience and learn from it. At each step, the agent selects an action based on its policy, explores the environment, and receives rewards. Rewards are assigned based on how far the agent is from the goal state. The trajectory of the episode is stored, and at the end of each episode, the rewards are calculated. The rewards are then used to update the action values, representing the expected returns for each state-action pair. By iterating through multiple episodes, the agent gradually refines its policy and action values, aiming to maximize the expected reward. Once the agent was trained, I tested the policy 100 times and returned a success rate for how many times it was able to successfully complete the maze.

Figure 1a. The predefined 5x5 maze environment for the first task. The agent starts

at random locations for the beginning of each episode.

When training with random starts, it is difficult to say what the optimal number of episodes is required to guarantee that the agent completes the maze. Due to random chance, even at a high number of episodes trained, it will sometimes result in states it has never encountered before. As one can imagine, higher episodes trained increases the chance that the agent will be successful once it is tested. For the first part of this task I will simply train the agent at 0.99 gamma levels and 0.02 epsilon levels. The alpha level is constant for our on policy monte carlo algorithm. The gamma represents the discount factor used to balance immediate and future rewards. A smaller gamma value makes the agent focus more on immediate rewards, while a larger gamma value makes it consider future rewards to a greater extent. The epsilon controls how much exploring the agent does during its training. A higher epsilon value encourages more exploration while a lower epsilon value promotes the current information it already has.

The way I decided to calculate success was by training the agent with the on policy monte carlo method with a varied number of episodes. After a policy was created, I tested each policy with the agent by running it 100 times in the maze. It had to finish the maze in less than 25 steps or else it would move on to the next episode of testing. The number 25 comes from the 5x5 area. I assumed that if it found the optimal policy it would finish the maze in less than 25 steps and it helped deal with the bug of when the agent did not move because it was given the wrong action in a particular state. For the sake of efficiency, I created 10 policies per increment and tested each individual policy. The increments for training are 1000, 2000, and 3000 episodes. After testing each policy 10 times I determined that about 3000 runs were the most effective. It did not complete the maze every time but it was still substantially better than the rest of the episode increments.

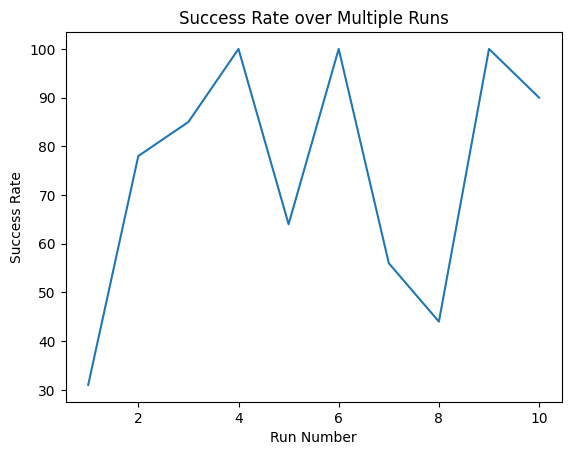


Figure 1b. Success rate of 10 different policies trained each for

1000 episodes. The 100% success rates are likely due to chance.

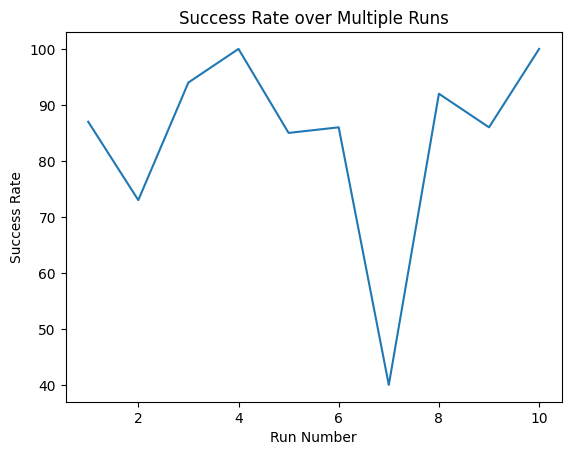


Figure 1c. Success rate of 10 different policies trained each for

2000 episodes.

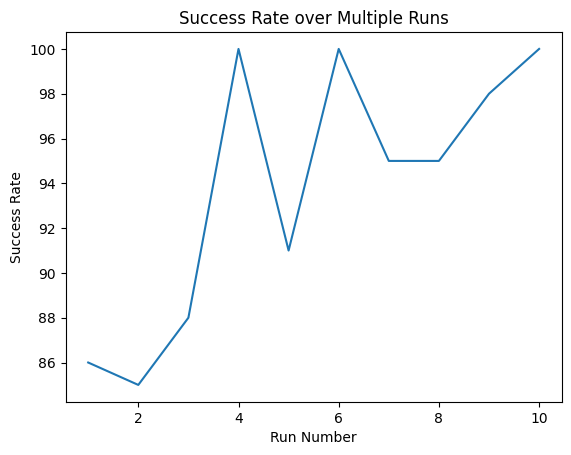


Figure 1c. Success rate of 10 different policies trained each for

3000 episodes.

When using the on-policy monte carlo algorithm, with a discount factor of 0.99 and an exploration factor of 0.2, 3000 episodes is the ideal number of episodes needed to train an agent to successfully complete the maze at least 85% of the time. Even with more episodes trained it would be difficult to guarantee success 100% of the time due to the fact that it starts in random locations at the beginning of every episode. However, it will likely be successful more than 85% of the time as demonstrated by figure 1c.

Task 1B

Modifying the gamma and epsilon values to see how well it performs in the predefined maze.

In this task different values of gamma and epsilon are explored and the success rate varies depending on how these values change. This process is very computationally expensive and one of the reasons why I chose to train with only 2000 episodes and not 3000 episodes. Once again gamma represents the discount factor while epsilon represents the exploration rate. At higher gamma values we should see higher success rates as we would expect since it values the long-term rewards. The higher epsilon values could encourage the agent to explore new areas during training and correlated with the highest success. At 2000 episodes we see that gamma at 0.99 and epsilon and 0.2, gamma at 0.9 and epsilon and 0.3, and gamma at 0.8 and epsilon and 0.3 all had 100%. It is important to note that even the gamma at 0.99 and epsilon at 0.3 had very high success rates. If we look at figure 1d we can see that the bottom left correlates with the lowest success rates while the right correlates with the highest success rates. This shows us that epsilon is very important in training the agent to solve this maze specifically for 2000 episodes.

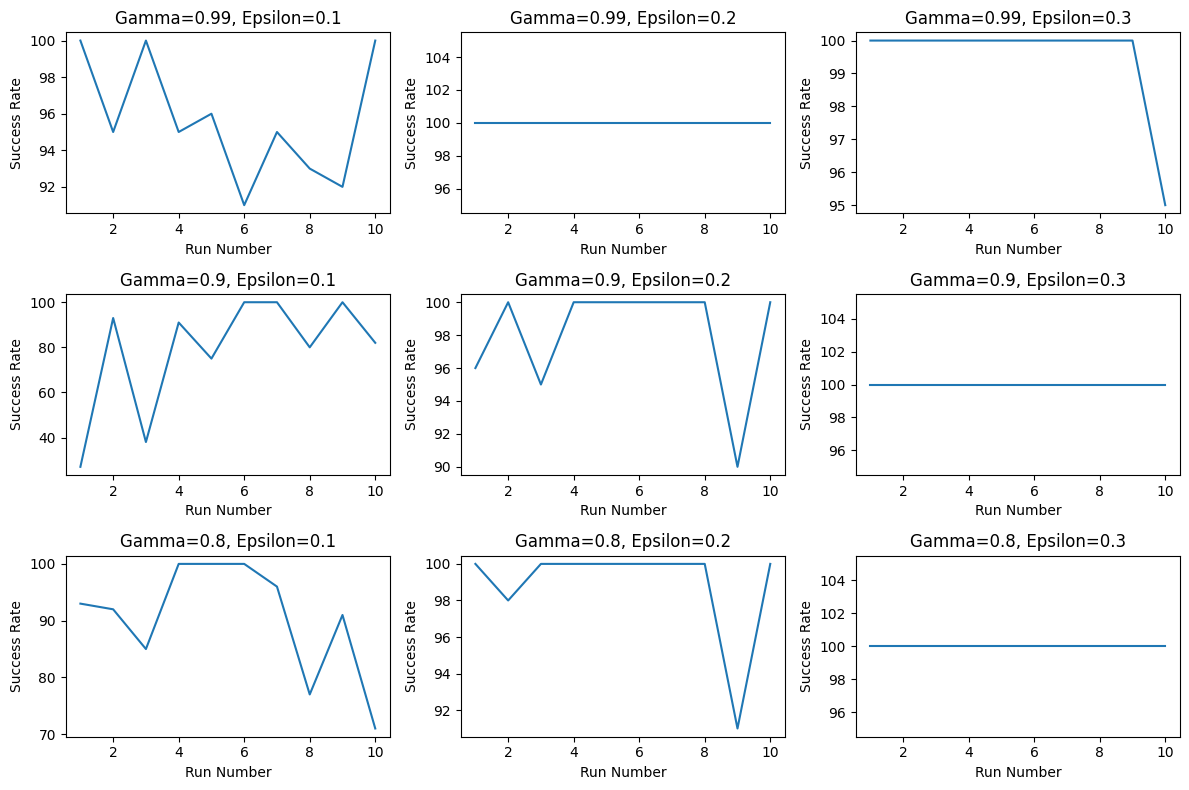


Figure 1d. Policies trained 10 different times for 2000 episodes with varying gamma and epsilon levels. The epsilon values are 0.1, 0.2, and 0.3 and the gamma values are 0.99, 0.90, and 0.8.

Task 1C

Training the discrete mountain car and cart pole environments using a Deep Q-Network algorithm.

The deep SARSA method was used to find the optimal action for a given state. The algorithm updates the Q-network weights using the deep SARSA algorithm. The deep SARSA algorithm is another reinforcement learning algorithm used for training an agent to interact with the environment and learn the optimal policy. The deep SARSA algorithm uses a neural network which I called Q-network to approximate the Q-values, which represent the expected rewards for taking a specific action given a state.

During training, the agent starts by initializing the environment, so in my case either mountain car or cartpole, and setting the initial state. The agent selects an action based on a policy which is defined by the Q-network. Once the agent takes an action it observes the next state and receives a reward. The agent stores its past actions in a replay memory buffer. Once enough experiences are stored in the memory buffer the agent samples a batch of experiences and computes the Q-values of the current states using the Q-network. Target Q-values are calculated by combining the rewards with the discounted Q-values with the rewards.

Overall, the deep SARSA algorithm leverages the neural network and replay memory buffer to learn an optimal policy by iteratively updating the Q-values based on the observed experiences. It aims to maximize the returns for a given environment.

The agent is trained on the MountainCar or CartPole environment, provided by the gymnasium library, for varying episodes, using a batch size of 32, a discount factor of 0.99, a learning rate of 0.001 (alpha), and an epsilon rate of 0.05. Finally, the maximum return achieved during training is reported to get a sense of a successful session.

Before we proceed it is important to understand how the rewards are distributed in the cartpole environment. The goal is to keep the stick upright for as long as possible so a reward of +1 is given for every step taken, including the termination step. According to documentation the threshold is 475 for the version I used. An episode terminates if the pole angle is greater than 12 degrees or the cart position is greater than 2.4.

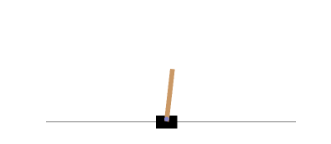


Figure 1e. A visual for the CartPole

environment

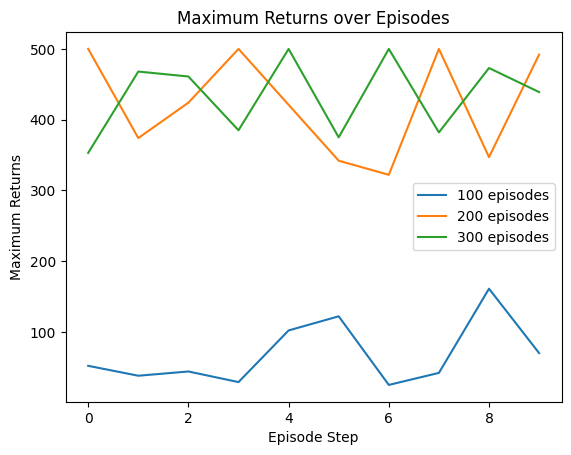


Figure 1f. CartPole took anywhere from 200-300 episodes to

maximize the reward.

The mountain car’s reward system is slightly different. The agent is penalized -1 for each timestep to encourage the agent to reach the flag on top of the right hill as quickly as possible. An episode terminates when the position of the car is greater than or equal to 0.5 or the goal position.



Figure 1g. A visual of the MountainCar Environment.



Figure 1h. Discrete MountainCar took anywhere from 400-500

episodes to effectively train the agent.

Both MountainCar and CartPole took 400-500 and 200-300 episodes to effectively train. This was done using the deep SARSA method as described above in the introduction to this section. In the next challenge we will look into modifying the maze environment and training continuous action spaces for varying gymnasium environments.

Task 2A

Introducing different grid spaces in the maze and randomly generate a maze of specified dimensions

The special grid spaces introduced are a trap grid space, an ice grid space, and a wizard grid space. The trap grid space has a 10% probability of applying a penalty of -10 during the training phase. The ice grid space will allow the user agent to slip through the ice space and into the next space. The wizard grid space will teleport the agent to a random grid in the maze.

All of the new grid spaces are introduced by initializing a list of potential grid states in the constructor. Then the compute reward method checks to see if the next state is one of the trap grid spaces. If it is then there is a 10% chance that the agent will receive a penalty of -10. The ice grid space alters the step method. The method checks to see if the user is in an ice state and if it is it returns the state after the ice state rather than the state it is already on. Essentially it skips itself. Finally the wizard grid space also alters the step method. A get random state method was created which returns a random grid that is not a wizard, trap, ice, or goal state and returns that random state for the agent to teleport to.

In order to randomly generate a maze of specified dimensions the size is predefined in the constructor and it creates the square maze based on the size inserted in the constructor. The maze is represented as a dictionary where each grid cell is a key and the corresponding value is the neighboring grid cell. The algorithm starts at the initial cell(0,0) and explores the maze recursively by visiting the neighboring grid cells. To ensure that each cell is visited a set is created at the beginning. By definition a set can only hold unique values so this helps prevent redundant exploration. The code shuffles the directions to randomize the exploration order allowing for a different maze to be made every time the maze environment is instantiated. This recursive method was built with the help of ChatGPT.

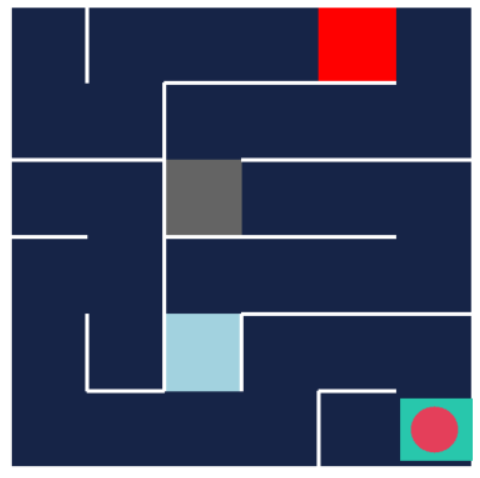


Figure 2a. The maze above is the randomly generated maze. The

red square represents the wizard space, the gray square represents

the trap space and the sky-blue square represents the ice space

For the following tests a 6x6 maze was randomly generated using the method listed above. Reference Figure 2a to see the maze environment. Initially a 10x10 maze but the on policy monte carlo method was extremely computationally expensive and it took a long time to test all parameters. It would make sense that a 10x10 takes so much longer since you are going from 25 possible grid spaces to 100 grid spaces. The 6x6 only increases to 36 grid spaces so it makes it easier to compute the changes in success rate with different parameters.

Furthermore, when training the agent for a 6x6 maze it sometimes appeared to be in an endless loop which never allowed me to train for another episode. I attempted to add a timer to the on policy mc, however that did not seem to work as it was the line

action\_values[s][a] = np.mean(sa\_values[(s, a)])

that seemed to be causing the issue. It would take an extremely long time here and this issue was resolved by changing the line to

action\_values[s][a] = action\_values[s][a] + epsilon \* (G - action\_values[s][a])

This solved the issue and the agent can be trained to solve 6x6 mazes.

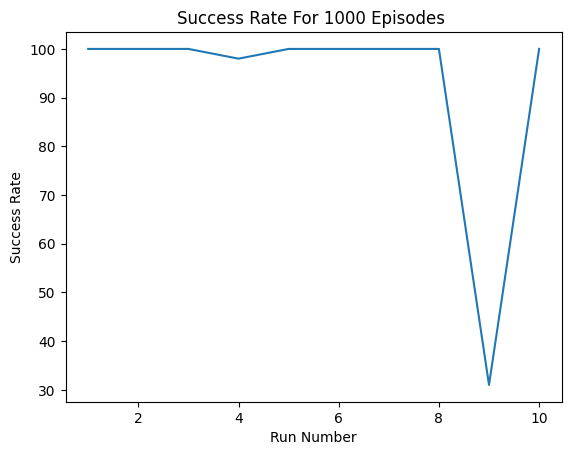


Figure 2b. Success rate of 10 individually trained agents for 1000 episodes.

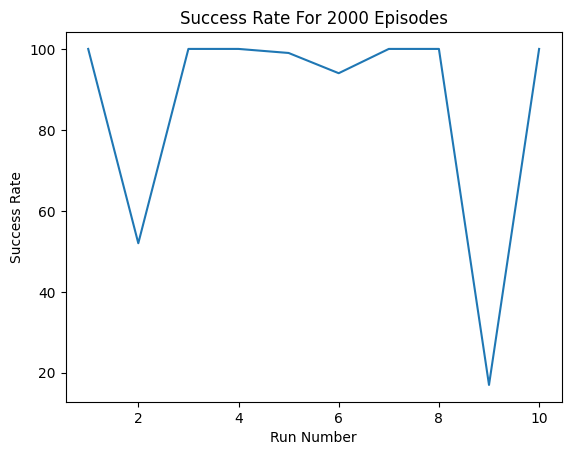


Figure 2c. Success rate of 10 individually trained agents for 2000 episodes.

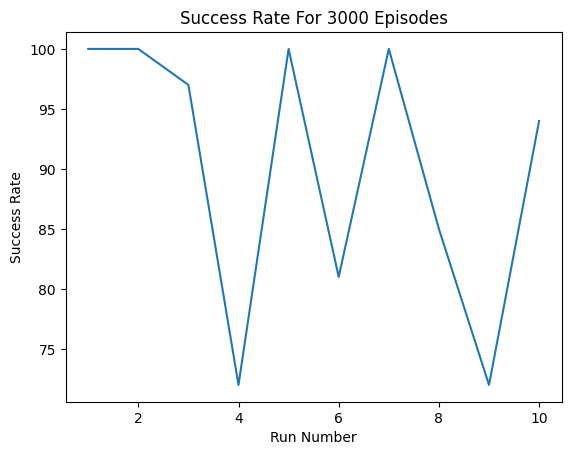


Figure 2d. Success rate of 10 individually trained agents for 3000

episodes.

The optimal number of episodes is around 3000 episodes of training. The lowest success rate obtained after testing 10 different policies that were each trained for 3000 episodes had about 74% success. The other scenarios had several cases where the success rate was below 40%. After testing the success rate for the basic 0.2 epsilon and 0.99 gamma values the next goal is to change the parameters for gamma and epsilon and see how it affects the success rate.

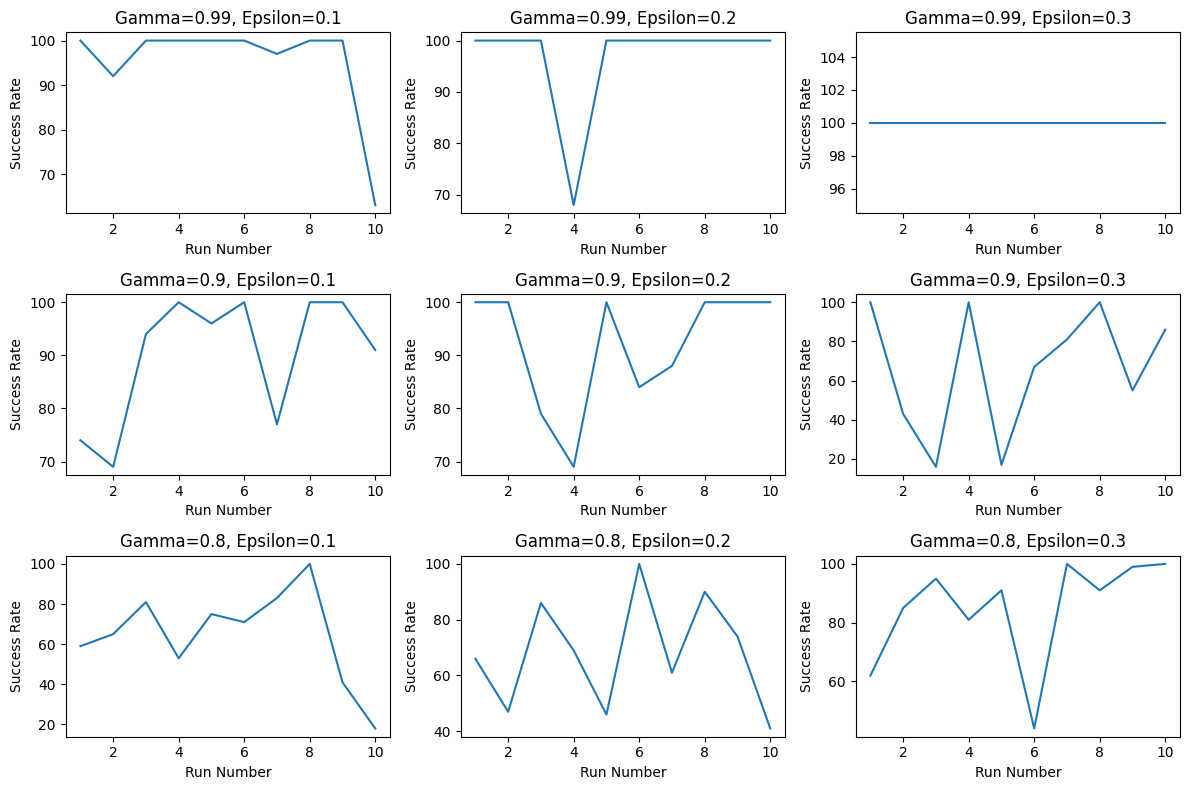


Figure 2e. 10 individually trained agents for 3000 episodes with varying levels

of gamma and epsilon. Gamma had values of 0.99, 0.90, and 0.80. Epsilon

had values of 0.3, 0.2, and 0.1.

In figure 2e it is shown that high gamma has higher success rates overall. This differs from the 5x5 maze where gamma was not as important. This would make sense though since the maze has ground from 25 spaces to 36 spaces and the discount factor would have more of an impact since the agent can start from further away in the maze. The epsilon was not as valued in the 6x6 as it was in the 5x5. This is likely due to the fact that in this maze it was more likely to get lost if it went the wrong direction since the maze is substantially bigger.

Task 2B

Training the discrete lunar landing, discrete acrobot, and continuous mountain car environments using Deep Q-Network algorithms and cross entropy method algorithms.

The LundarLanding and Acrobot use discrete action spaces so the method for training these are the same as the other discrete mountain car and discrete cartpole environments. The LunarLanding environment has a more complex reward system. For each step the reward is increased or decreased the further the lander is to the landing pad. It is also increased or decreased the slower or faster teh lander is moving. It is decreased the more the lander is tilted, so if the angle is not horizontal. It is increased by 10 points for each leg that is in contact with the ground. It is decreased by 0.03 pints for each frame a side engine is fired. Finally it is also decreased by 0.3 points for each frame the main engine is firing. The episode has an additional reward of -100 or +100 for crashing or landing safely. An episode is considered a solution if the agent scores at least 200 points.

The acrobot's reward system is simpler than the lunar landing. The goal is to have the free end reach the target height in as few steps as possible. All steps that do not reach the goal receive a -1 penalty and achieving the target terminates the episode and receives a reward of 0.

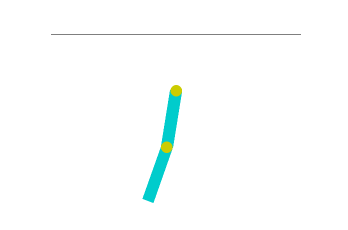
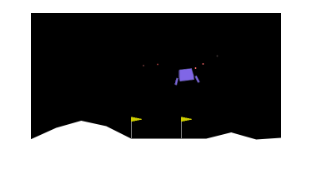


Figure 2f. Visual representation of the Figure 2g. Visual representation of the Lunar

Acrobot environment. Landing environment,



Figure 2h. Lunar landing trained 10 different times with different

intervals of training.

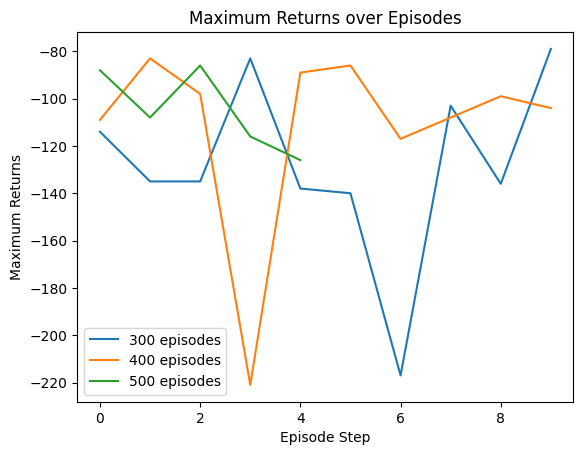


Figure 2i. Acrobot trained 10 different with different intervals

of training.

The algorithm used to train these environments were the same as the other discrete environments. Both of these used a gamma of 0.99, epsilon of 0.2, and a learning rate of 0.001. For the lunar landing environment, it had a successful run 90% of the time at least once after it was trained for 300 episodes. The acrobot solved the environment the fastest when training with 500 episodes.

The continuous mountain car was extremely challenging to find a solution for training the car. I initially had planned to try and turn the continuous action space into a modified discrete action space, however after days of trying this method with no success I decided to try a critic-actor method. This also did not work for me unfortunately. The method I ended up using for solving the continuous mountain car method is the Cross-Entropy Method Algorithm. This is a very neat method that takes in an instance of the agent and essentially duplicates the agent and evaluates which agent performs the best for each episode. So in my case one episode will have 50 different agents and the top 20% are selected based on their rewards and their weights are averaged to obtain the new best weight. This process is repeated for the specified number of iterations so in my case 500 iterations.

I am going to take a bit to explain exactly how this algorithm works since it is also used to solve the bipedal walker. It is also important to note that this is an off policy reinforcement learning algorithm. The agent maintains a distribution over the action space and samples actions from the distribution to interact with the environment. The distribution is then updated based on the performance of the sampled actions (which are the best performers). Epsilon works differently in this algorithm as it decays over time.

The agent class represents the agent that will end up being instantiated in the cross entropy method (CEM) and duplicated for each iteration. The agent is implemented using a neural network to solve the continuous mountain car problem. The important methods in this class include set\_weights, forward, act, and evaluate. The set\_weights method separates the weights for each layer and assigns them to the corresponding layers in the neural network. The forward method allows for the agent to move to the next layer in the neural network. The act method generates an action based on the current state by passing it through the neural network. Finally, the evaluate method evaluates the agent’s performance using the provided weights, discount factor, and maximum time steps, accumulating the episode return over time steps while interacting with the environment.

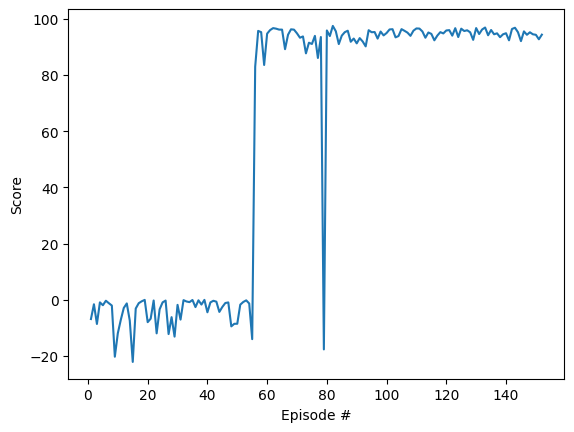


Figure 2j. This agent took about 60 episodes to master the mountain car

continuous environment. Gamma = 1.00 and epsilon = 0.50.

Using the cross-entropy method, a viable solution to the continuous mountain car was achieved. As shown in figure 2j, after about 60 episodes it starts to maximize its rewards. Even though this method is very computationally expensive, since it relies on instantiated 50 different agents per episode each with different weights. However, this turned out to be the only way that I could get the continuous mountain car trained.

Task 3

Training the bipedal walker using a cross entropy method algorithm.

First attempt had a similar outcome as the continuous car did in the first attempt. Here I tried to use a Deep Q-Network to train the walker but it would not move after a certain number of episodes. It tried to learn some random policy due to the high epsilon value I was giving it but then it decided not to move and because gamma was so high it decided to stay at the -100 reward which is the default for falling. This means that it does not even try to move after about 100 episodes. This was evident when we were testing as after 100 episodes the training sped up considerably.

The second attempt looked to change the discount to 0.90 to try and prevent the behavior of sticking with the -100 reward. Furthermore, the learning factor changed to 0.1 and the epsilon to 0.1 as well. It was clear that finding the right parameter would take way too long using this method as it is very time consuming to train the bipedal walker.

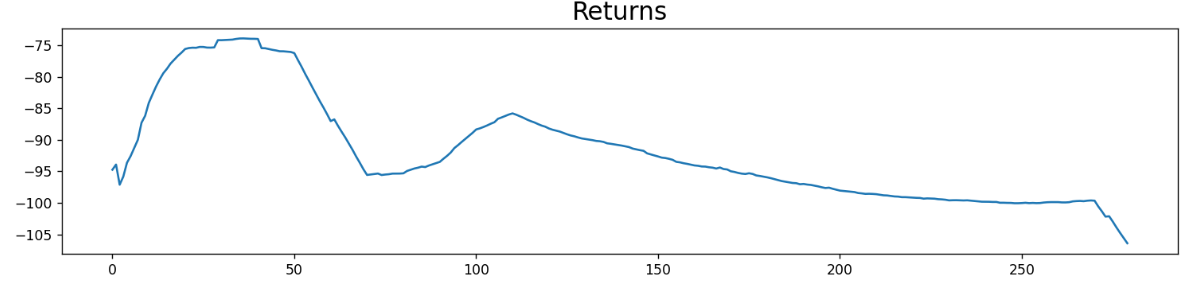


Figure 3a. Returns of the bipedal walker with gamma 0.90, alpha 0.1, and epsilon, 0.1.

This is where I looked for alternatives for the continuous action spaces and came across the cross-entropy method algorithm. The parameters and algorithm are identical to the mountain car continuous problem, however I have it break out of the loop after the average returns are greater than or equal to 300 as that is what is listed as a successful episode in the documentation.

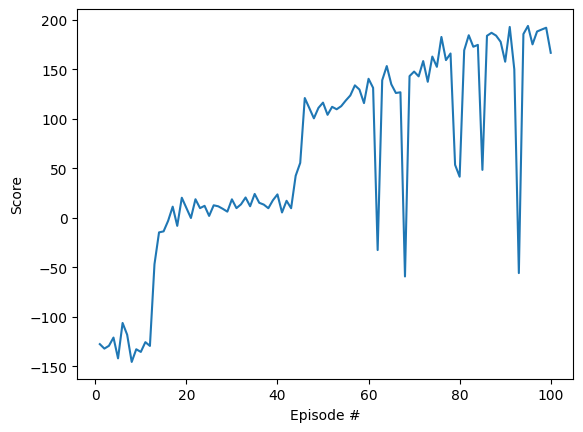


Figure 3b. CEM for training the bipedal work. Gamma is set

to 1.00, epsilon to 0.5 and epsilon decays over time.

Lastly another simulation for 500 loops was run (which takes 9 hours to complete) was conducted to check and see if this could successfully train the agent to solve the bipedal environment. I woke up the next morning and my laptop had restarted. For whatever reason I lost the model and 9 hours of work was just wasted. I could not even plot the max scores as it had lost its data. Luckily it had printed out the episodes until it had crashed so the average score was visible. With this table it is not possible to tell if the bipedal walker was able to solve the environment or not as these are averages although likely it was not.

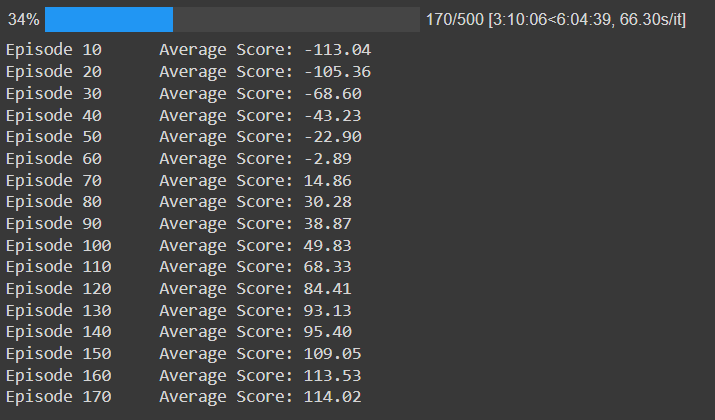
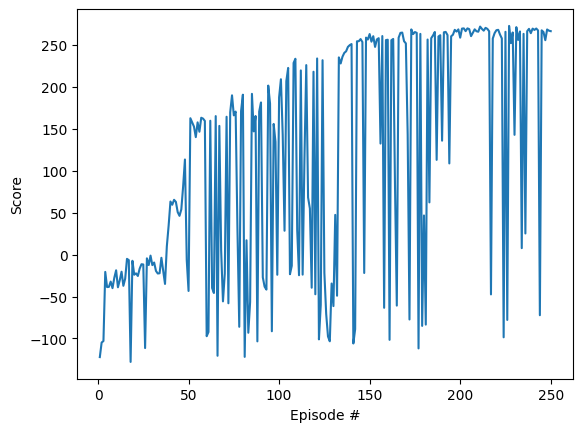


Figure 3c. Second attempt at solving the bipedal walker problem with CEM.



def cem(agent, n\_iterations=250, max\_t=1000, gamma=1.0, print\_every=10, pop\_size=150, elite\_frac=0.15, epsilon=0.52):

Figure 3d. Best attempt to solve the bipedal walker problem with CEM

In conclusion, I was not able to definitively solve the bipedal walker. I was able to get it to walk however a successful episode was listed as having a score of 300 or above. My next steps would have been to decrease the rate at which epsilon decays over time because as we can see above the average score starts to plateau over time. Something else that can be done in the future to attempt to train the bipedal walker is to increase the number of instantiated agents per episode. This would likely take even longer to run then what it already takes to but we would have more chances of getting better results by random chance early on. I am confident that with more time the bipedal walker could have been solved. I will keep working on this problem until I have it solved. For now this is my best attempt.

References

1. <https://colab.research.google.com/github/goodboychan/goodboychan.github.io/blob/main/_notebooks/2021-05-11-CEM-MountainCar.ipynb#scrollTo=abff3eba>

* Without this code I would not have been able to solve the continuous action spaces.

1. ChatGPT

* Assisted in creating the recursive method to randomly build the maze given a specified number of dimensions.

1. <https://gymnasium.farama.org/content/basic_usage/>

* All of the environments except for the maze come from here.