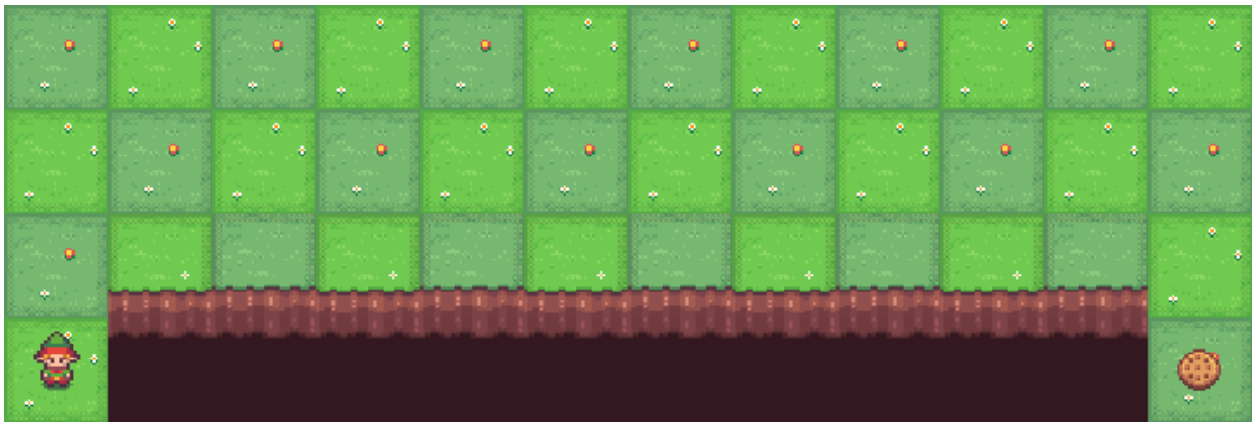
Project Documentation: Cliff_Walking

Introduction:

There are many top-down games in the gaming industry such as Pac-Man, Pokemon, and etc. These top-down games require the player to move around a set grid to complete a task. There are often many obstacles that hinder the player from completing their task, for example, environmental obstacles like trees and non-playable enemies like monsters. In this project, we analyze a similar scenario where the player experiences environmental obstacles.

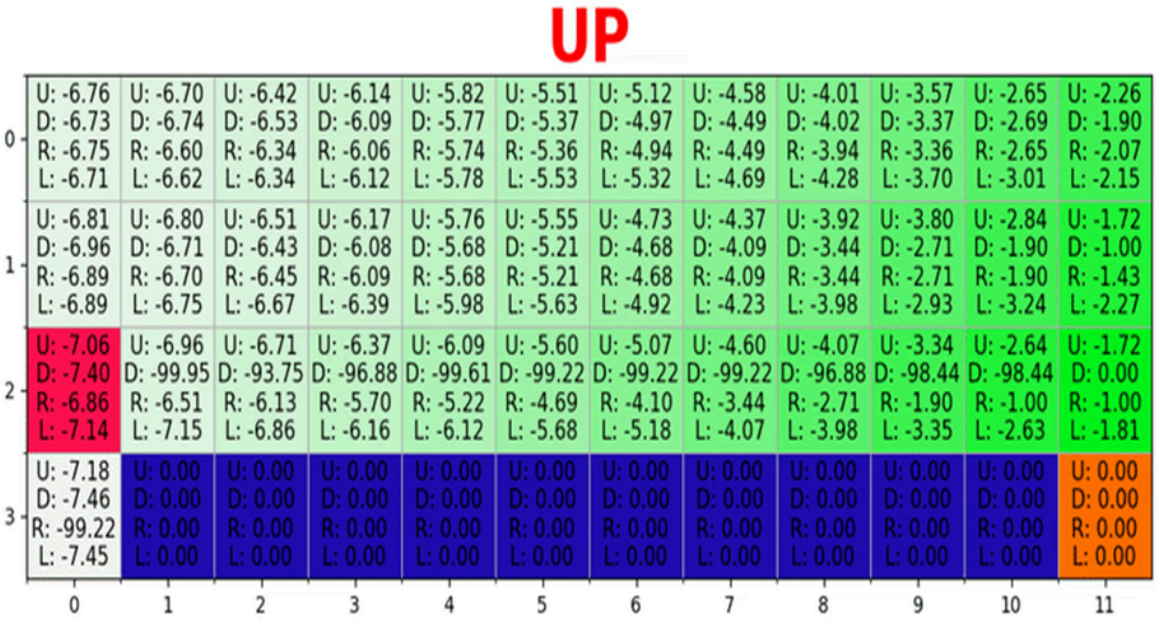
Cliff walking is a scenario in which the objective of the agent is to traverse a 4x12 grid to reach an end goal as displayed in the image below. The environment of this game is the area in which the agent can traverse. The agent traverses the environment through 4 actions: up, down, left, and right. The agent starts at the bottom left([3,0]) and traverses the grid to reach the bottom right([3,11]). In grids, ([3,1]) to ([3,10]), there is a hole where if the agent falls into the hole, the game resets and places the agent back at the bottom left square.



Cliff walking has a unique reward system compared to other problems because in cliff walking each movement the character takes results in a reward of -1. The goal of this problem is to minimize the amount of steps needed to reach the end tile. When the agent falls into the hole, the reward for the episode doesn't reset and the agent is placed at the start again. This reward system punishes the agent for falling off the cliff and rewards staying on the green tiles in the environment. Episodes are used to run and train the agent to make the best decisions within the environment. Each episode lasts until the agent has successfully reached the end state. If the agent falls off the cliff, the agent resets to the start position, but the total reward doesn't reset, so we punish the agent for not staying on the correct path.

Using Deep Q Learning:

Since the agent's objective is to traverse the environment to reach the end tile, Deep Q Learning or DQN is used to teach the agent how to reach the end tile as well as minimize the amount of negative reward accumulated to reach the end state. A two-dimensional Q-table of size 4x12 is used, so we can keep track of the transition probabilities in which the agent wants to make. On the table, the area in which the agent would fall and reset to the start position has 0 probability because the objective is to reach the end. The hole where the agent falls isn't a valid area the agent can take action from.



The Q-table is initially constructed with all 0s as we have not discovered and updated any probabilities. Each Q value is updated through the Q-equation, with the learning rate initially set at 0.1 and the discount factor set to 0.9.

$$Q^{new}(S_t, A_t) \leftarrow (1 - \underbrace{\alpha}_{\text{learning rate}}) \cdot \underbrace{Q(S_t, A_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \underbrace{\left(\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(S_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}_{\text{new value (temporal difference target)}}$$

As we run through many episodes, the Q-values get updated as more episodes are completed.

Adding Epsilon-Greedy

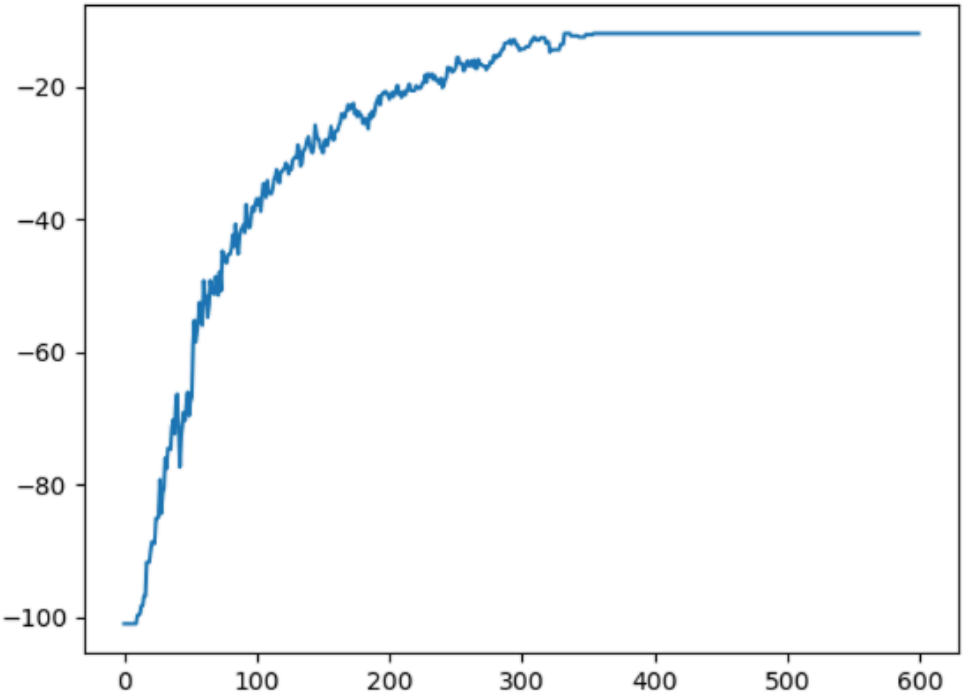
After implementing a DQN, an epsilon was created to help ensure a balance between exploration and exploitation. This is done to prevent the agent from exploring other possible routes that is restricted from the high Q-value. An initial value of 0.001 is set for epsilon. The agent 99.9% of the time will take the action in which the Q-table yields the highest. The 0.1% chance allows for the agent to take a different action that isn't suggested by the Q-table. The epsilon helps ensure that even if the agent finds a solution/path to the end state, the agent will still make efforts to deviate from the Q-table to see if there are more optimal solutions.

Running Through the Environment With No Reinforcement Learning

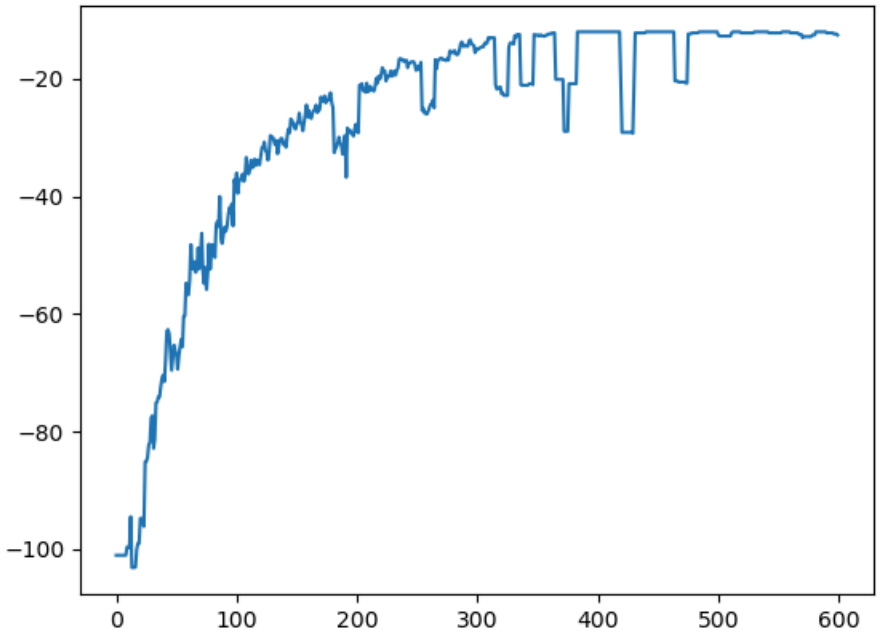
Initially, running through the environment with the application of reinforcement learning algorithms results in the model randomly selecting an action at each state of equal probability. The amount of time required for the model to complete a successful episode is largely dependent on chance and takes much longer time compared to using reinforcement learning. With the addition of resetting the agent's position to the start position if the agent falls into the hole, this adds extra time and more negative reward. The idea of not using any memory that helps the model make more correct actions punishes the model even more in the event the agent falls off the cliff.

Finetuning the Knobs For the Fastest Optimal Solution

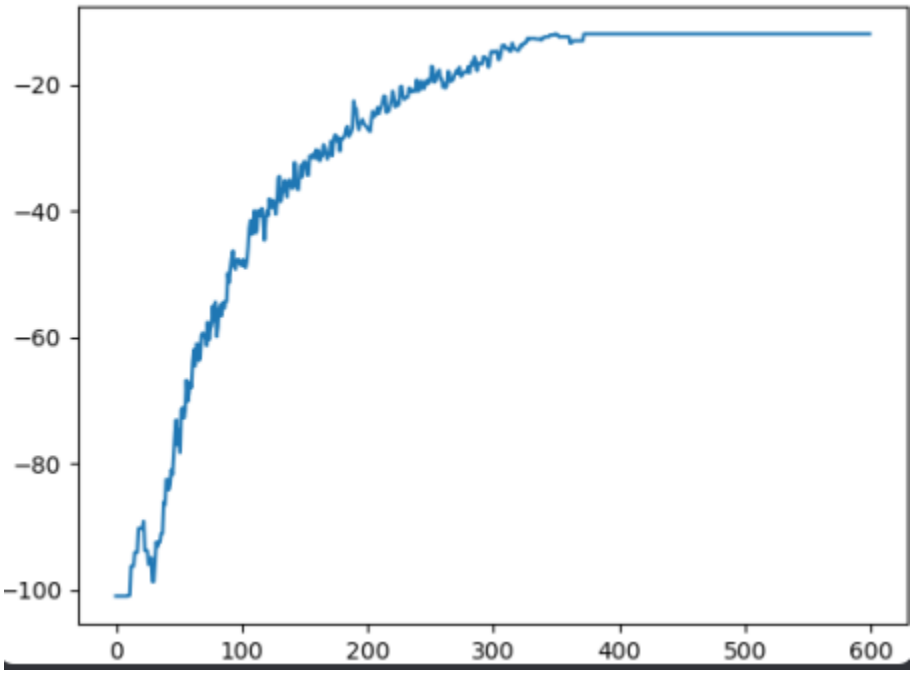
After running the model for 600 episodes with a learning rate set to 0.1, a discount factor set to 0.9, and an epsilon set to 0.001, we get the following graph. With the y-axis being the average reward and the x-axis being the episode.



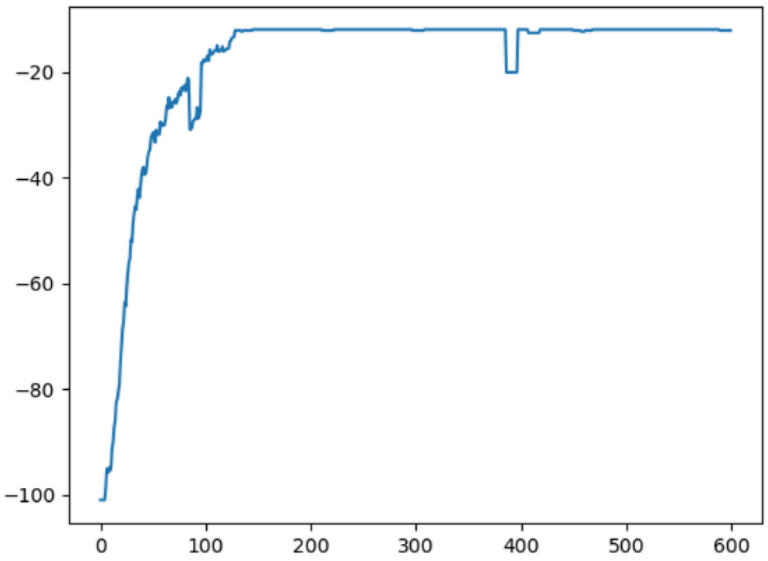
The graph is plotted every 10 episodes. From the resulting graph, we see a large change in rewards from episodes 10 to 180. The change starts to flat out at over 200 episodes. With the current graph produced, changing the learning rate, discount rate, and epsilon is tested. Firstly with epsilon: the learning rate and discount rate remained the same, but epsilon is set to 0.01 instead of 0.001.



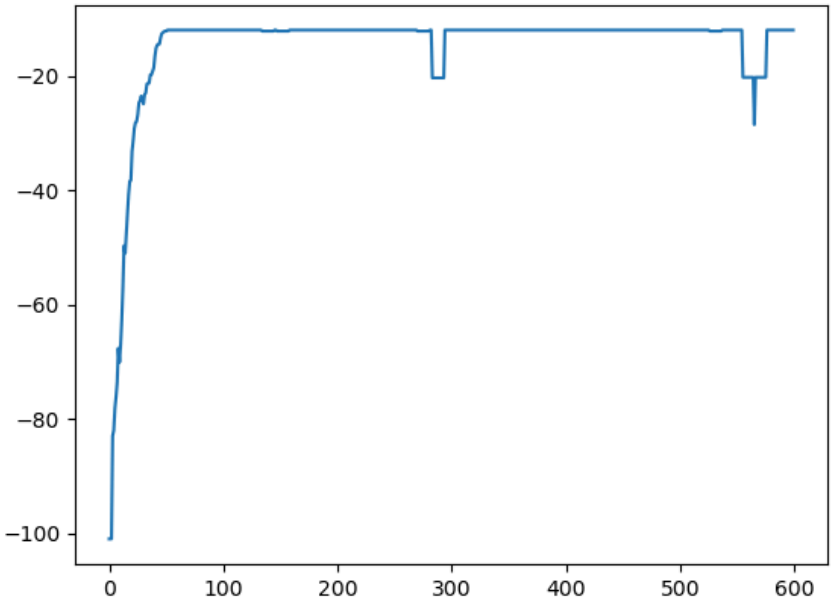
In the above graph, we set the epsilon to 0.01. We see that the model converges to the solution at around 300 episodes. Compared to the original, the model takes longer to discover the optimal solution, as well as being much more noisier. There are many episodes in which, the model does more exploration than exploitation. From this graph, having a smaller epsilon relatively small prevents too much exploration. Excess exploration leads to increased training time for the model. Next epsilon and learning rate stayed the same, but the discount rate was changed to 0.5. The discount rate is the amount in which the model wants to use future predicts.



The graph shows that by reducing the discount rate, the model is 100 episodes longer to converge to the optimal solution compared to the original settings. Although this model takes longer to converge, the model learns and reduces rewards at the same rate compared to the original model at 0-200 episodes. Going from 200 episodes until the optimal solution is achieved takes longer for a smaller discount value. Lastly, the learning rate was set to 0.25 while maintaining the discount value and epsilon to the original values. The following graph was produced.



From the graph, we see a much faster convergence to the optimal solution, but when the model takes an exploration route over the exploitation, the model will deviate very far from the optimal solution as we see in episode 400. Increasing the epsilon rate will increase the amount of drops in average rewards. Only one drop is present as a relatively low learning rate has been chosen. The same can be said when the learning rate is set to 0.75. See the graph below.



The graph with a 0.75 learning rate learns even faster, but when the model decides on exploration, even bigger drops are experienced. Another important note is that at the episodes close to 600, we see significant drops in average reward compared to the 0.25 and original 0.1 values. Although the path was discovered, the Q-table values are not optimized at each position in the Q-table, as the model will want to apply exploitation more than exploration. Having a learning rate that isn't too large and too small will yield the best Q-table values in all states. The original values train the model faster, more accurately, and more complete in terms of Q-table values compared to the changes applied.

Conclusion

After running the model by changing the learning rate, discount rate, and epsilon, having too high or too low of any of the three values results in a less efficient, less accurate, and less complete model. Epsilon-Greedy helped ensure that the model doesn't get stuck at a local solution. The ability for the model to branch out ensures that the agent is able to explore other states in hopes that a more optimal, faster, and efficient solution can be found compared to the local solution. The cliff walking problem looks like a straightforward problem to solve, but can be big challenge for the agent if the character's start and end states are changed to random areas on the map.