Using Q Learning and Deep Q Learning for Maze Solving Agents

Intro

This project explored the design, implementation, and optimization of agents trained using Q Learning and Deep Q Learning to solve different types of mazes; mazes with keys for the agent to pick up and doors to unlock, as well as mazes with no doors or keys. The inspiration of this project was a general interest in video game AI.
By sharing my experience creating this project, I hope to demonstrate some of the considerations and possible surprises that can come with Q Learning. I also explain my decisions and thought processes as I gained new information.
The main focus will be on my attempts at optimizing the speed and accuracy of training for each agent, inspired by the research paper, Optimal Values Selection of Q-learning Parameters in Stochastic Mazes, by Xiaolin Zhou. Their problem was very similar to mine, having to do with training agents via Q Learning to navigate mazes, and their focus on tweaking parameters to get better results inspired me to create loss graphs and spend a lot of time trying to make the agents more accurate, and to make training more efficient.
I will discuss my design choices, and why I chose Q Learning and Deep Q Learning, as well as the challenges I faced in training the agents and the difficulty of incorporating keys and doors. I will also compare training methods and parameters, share some data I collected, and explain why I chose the training methods and the specific agents to be included in the final project.

Q Learning

Q Learning is a reinforcement learning algorithm which teaches an agent behavior by having it try different actions and associating the result/reward with the state it was in previously and the action it took. So if the agent is in the same situation again, it knows the result of that action from last time. Eventually, it has enough experiences that, ideally, it knows the best action to take in every state. By chaining these memorized actions and states together, the agent can bring itself to some desired outcome.

As a quick sequential break down of the algorithm, you start with a blank Q table where the rows are states and the columns are actions.
Next, we need to choose an action to start filling the Q table. For this we can use epsilon-greedy, which balances exploration (choosing random actions) and exploitation (choosing the best action in the current state that we know of). We start with the probability of choosing exploration very high, and then as time goes on, we start making more and more choices based

on past knowledge.
Having made a choice, we update the Q table using an equation called the Bellman equation:

```
Q_table[state][action] = Q_table[state][action] + (a[state][action] * (reward + gamma * max(Q_table[new_state]) - Q_table[state][action]))
```

"a", the learning rate, is a table that determines how much the Q value is updated. Gamma is a value that determines the importance of future rewards over immediate rewards.
Once the Q table is update, it repeats from there as many times as is wanted.

This is a great choice for something like simple maze solving, because it's so lightweight and efficient for the problem at hand. Using more advanced AI learning methods would probably be overkill.
I had two Q Learning tasks at this point: one, create an efficient Q Learning function for training agents for solving mazes with no doors and keys; two, create one for doors and keys.

Maze without doors (13x13)

Figuring out how to train an agent to get to the goal tile in a maze with no keys or doors was relatively simple. There aren't a lot of different rewards to fine tune, or complex states.
For the state representation, I chose a simple approach: treat the entire maze as the state. This way, each possible position in the maze is considered a distinct state.

After that I needed to set rewards. Firstly, the agent should want to get to the goal tile, so I gave it a random high reward, 1,000. Next, I didn't want the agent hitting walls, so I gave that a -100 penalty so it learned to never do it. After that, I thought about giving a penalty for retracing steps, so the agent is encouraged to find the shortest path, but I found that the agent already did that with the given rewards/penalties.

After I got it working, it was time to optimize parameters. I wanted the user to be able to create a maze, or randomly generate one, and to train an agent to solve it at the click of a button. The less time the user waits, the better, but I didn't want to sacrifice accuracy.

I tried a few different options, three of which were:

1. 100 episodes, and .979 epsilon decay rate.
2. 500 episodes, and .9955 decay rate.
3. 1,000 episodes, and .9977 decay rate.

(To decide on epsilon decay rates, I used the equation: 1.0 × decay^episodes. I aimed for an epsilon value of around 0.1 at the end of training, so I adjusted the decay so that the equation would give a value near 0.1 after at the end of training).

The first option seemed promising, but it would screech to a halt during training on more complex mazes, indicating to me that the agent hadn't acquired enough knowledge about the later parts of the maze yet as the epsilon was lowered. These parameters are unusable.
The second one works well. I wrote a function to test it's speed and accuracy, and it gets through 100 randomly generated mazes with 100% accuracy in around 56 seconds. For my project, this seemed great, because the user gets to train an agent almost instantly.
The third one also works well, but it is slower, as expected. Running the same test function on it, it completed with a 100% accuracy in 1 minutes and 31 seconds.
For good measure, I tried to manually create mazes designed to break #2 in order to declare #3 the winner, but it didn't want to break. #2 was just better. That is the one I used for 13x13 with no doors in my project.

Maze with doors (13x13)

I knew this would be trickier, and it took me quite a bit longer to get it right. The maze itself no longer sufficed as a state, but it was a good start. The reason why it isn't enough is because, where we humans might see two different states, the agent doesn't. For example, if the agent needs to go down a dead end hallway to pick up a key and then retrace its steps, the agent wont see going through the hallway *to* the key as a different state than going back out of the hallway after attaining the key, so it will walk into a trap and never come out of that hallway. So we need to give the agent differentiation between the two.

I did this by giving it a "key_state," telling it which key(s) it has in it's inventory. Now, the state will be different going into the hallway vs out of it. It's possible that's the only other state I needed, but I added a couple more, which may have been overkill.
For good measure I added boolean state "on_key" to make sure it knew when it was on a key. I also added boolean state "door_adj" to let the agent know when it was adjacent to a door and ready to open it.

Rewards for picking up keys and opening doors were a major struggle. No matter what I tried, the agent was inconsistent and would walk into loops. Some combinations, like a high reward for picking up keys, and a lower reward for opening doors seemed to create fairly okay agents, but they didn't satisfy me. Finally, I tried 0 reward for picking up keys and 0 reward for opening doors and... it was perfect. Literally, a 100% accuracy. Goes to show, rewards aren't always a good thing.

Like with Q Learning without doors and keys, I tried a few different parameters to try to optimize. I tried 500 episodes and 0.9955 decay, since that was best last time, but it was terrible at handling slightly more complex mazes, so I amped it up to 1,000 episodes and 0.9977 decay. This was significantly better, but would still get stuck when I pushed the complexity further. I finally decided on 10,000 episodes and .9997 decay after I realized it's really not terribly slow, and it will be far more reliable if the user tries to test difficult mazes.

Deep Q Learning

Having completed the last two challenges, I wanted to tackle a new one: generalized knowledge. I wanted agents that weren't trained on just one maze, but many, and could solve new mazes put in front of them. For this, it's ill advised to use a Q table, since a table doesn't scale well. Rather than a table, we can use a neural network which approximates the Q function, which is far more efficient for generalizing knowledge.
As stated, Deep Q Learning uses neural nets to learn. Here's a quick rundown my process:

First, a neural network is initialized, as well as a target network which keeps the main network more stable. Then an optimizer, which updates the weights of the neural net, and replay buffer, which stores pas experiences, are initialized.
We then begin running the first episode with a random maze. Like in standard Q Learning, we use epsilon-greedy to choose either an explore or exploit action. We take the action and then store the experience using the replay buffer.
After a few stored experiences, we take a batch of them, calculate what the network currently thinks the Q value of the state action pair is, and then compare it to what it should be according to the target network. Then we calculate loss, backpropagate, and and occasionally update the target network.
Rinse and repeat.

I knew this Deep Learning agent would take much more time to train than the other Q Learning agents, and I didn't want waiting time to be a big issue on this project, so I scaled the maze down for this to 7x7.
The network architectures I used for the agent trained on mazes with no keys or doors was this:

```
class QNetwork(nn.Module):
    '''QNetwork class, used for dq no doors agent'''
    def __init__(self):
        super(QNetwork, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(49, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 4),
        )
    def forward(self, x):
        return self.net(x)
```

I thought one hidden layer would be sufficient for a fairly simple problem.

For the agent trained on mazes with keys and doors, I chose this architecture:

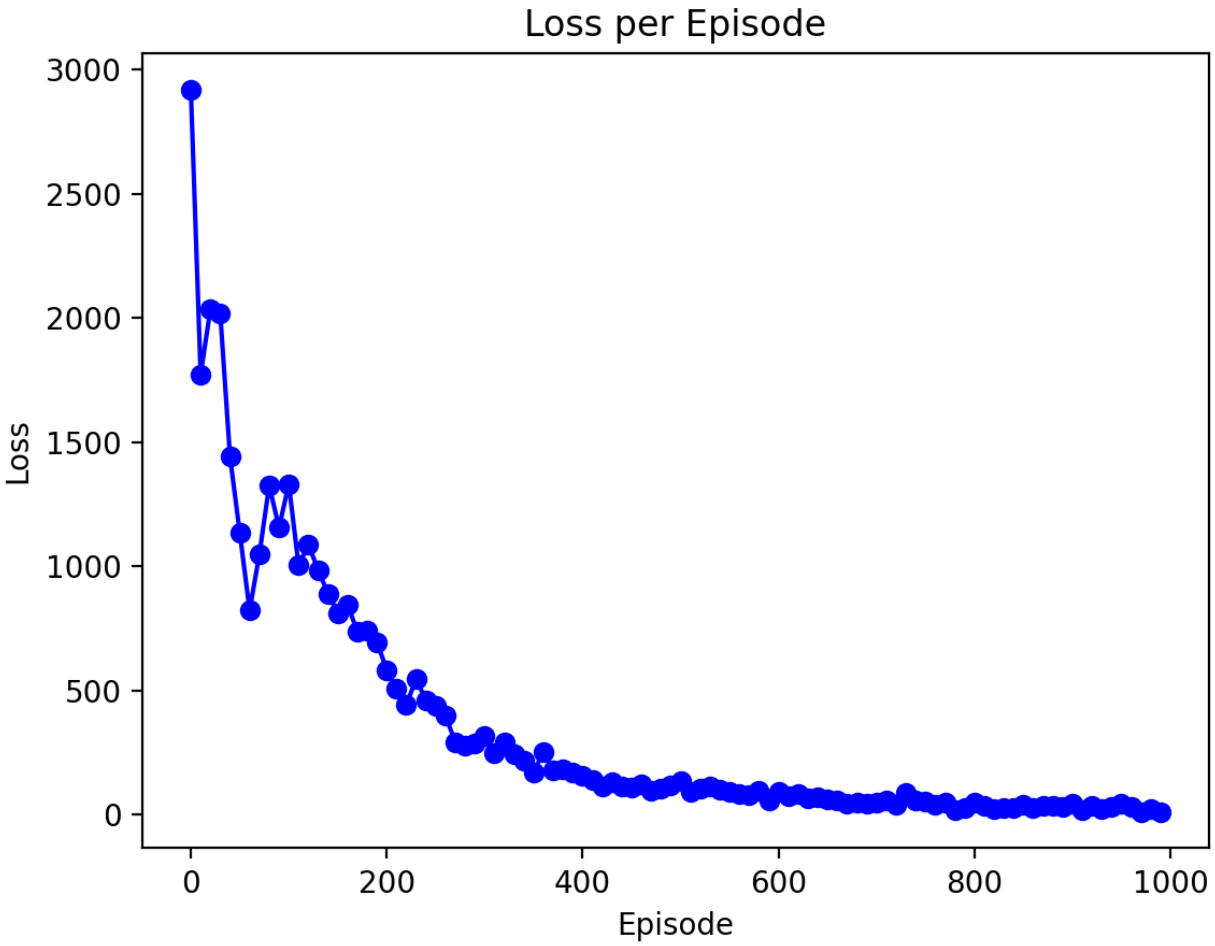
```
class QNetworkDoors(nn.Module):
    '''Qnetwork class, for dq doors agent'''
    def __init__(self):
        super(QNetworkDoors, self).__init__()
        self.net = nn.Sequential(
            nn.Linear(52, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, 6),
        )
    def forward(self, x):
        return self.net(x)
```

I added two more hidden layers due to the added complexity of learning how keys and doors function. I thought I'd rather be on the safe side and over-equip the agent than to under-equip.

I think the states on_key and door_adj were much more important for these Deep Q Learning agents than the Q Learning ones. on_key, because when the agent is on top of a key, they are blind to the key, and so need some indication that a key is ready to be picked up. For the Q learning agents, they'd eventually learn which tile holds a key, and they will pick it up there every time. For the Deep Learning agents, they don't have that same consistency to depend on. My reasoning is similar in the case of door_adj, though it's likely less necessary. It's more difficult for the Deep Learning agent to learn the conditions under which a door is within reach and ready to be opened just by using the maze state, since an agent and a door can be in all kinds of different places on the grid. I thought I'd help the agent out by giving it a clear boolean indicator of when a door is within reach, so that it could listen for that signal. My reasoning was that this might be easier for an agent early on in training, and so require less overall training.

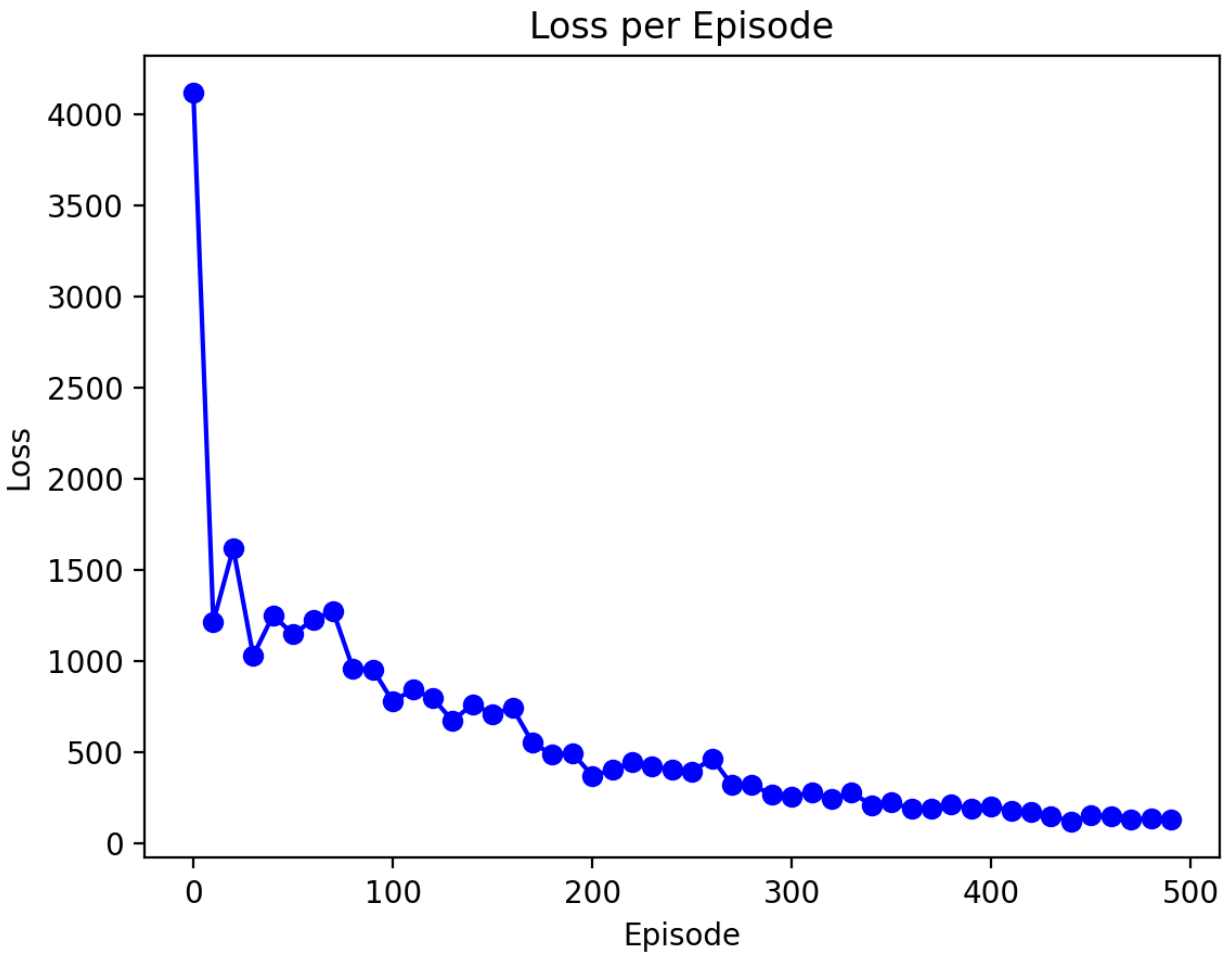
Maze without doors (7x7)

This was quite an easy training process. Again, it was best to have no rewards for picking up keys and opening doors. I assumed that I'd need more training episodes since the agent had to learn general knowledge, so I started with 1,000 episodes and 0.9977 decay, and the agent had perfect accuracy on 1,000 randomly generated mazes. The loss graph can be seen below.



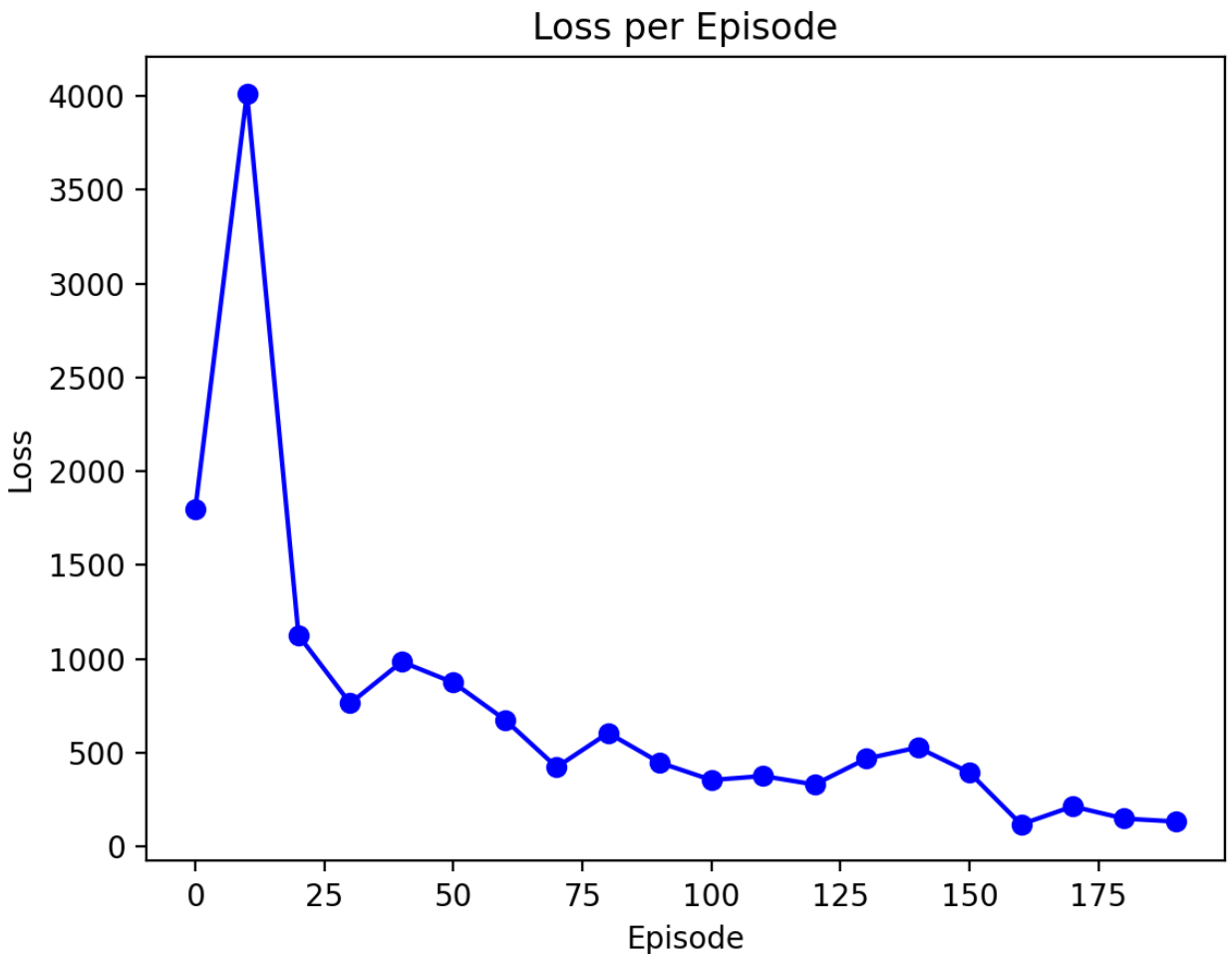
(I read that some fluctuations in loss is to be expected with Deep Q Learning, but that it's a good sign if it is generally trending downward. Some of those early bumps had me worried).

Of course, I wondered if I could lower the number of episodes to make training faster (not that it mattered to the user anymore). So I tried 500 episodes with 0.9955 decay, and again this agent had perfect accuracy. Loss graph below.



Not as neat of a curve, but it produced an excellent agent.

However, when I tried 200 episodes with 0.989 decay, accuracy dropped to 87%. Loss graph below.



I went with the first agent. Even though the second one was seemingly just as good, I think having trained for double the episodes makes it that much more of a safe pick.

Maze with doors (7x7)

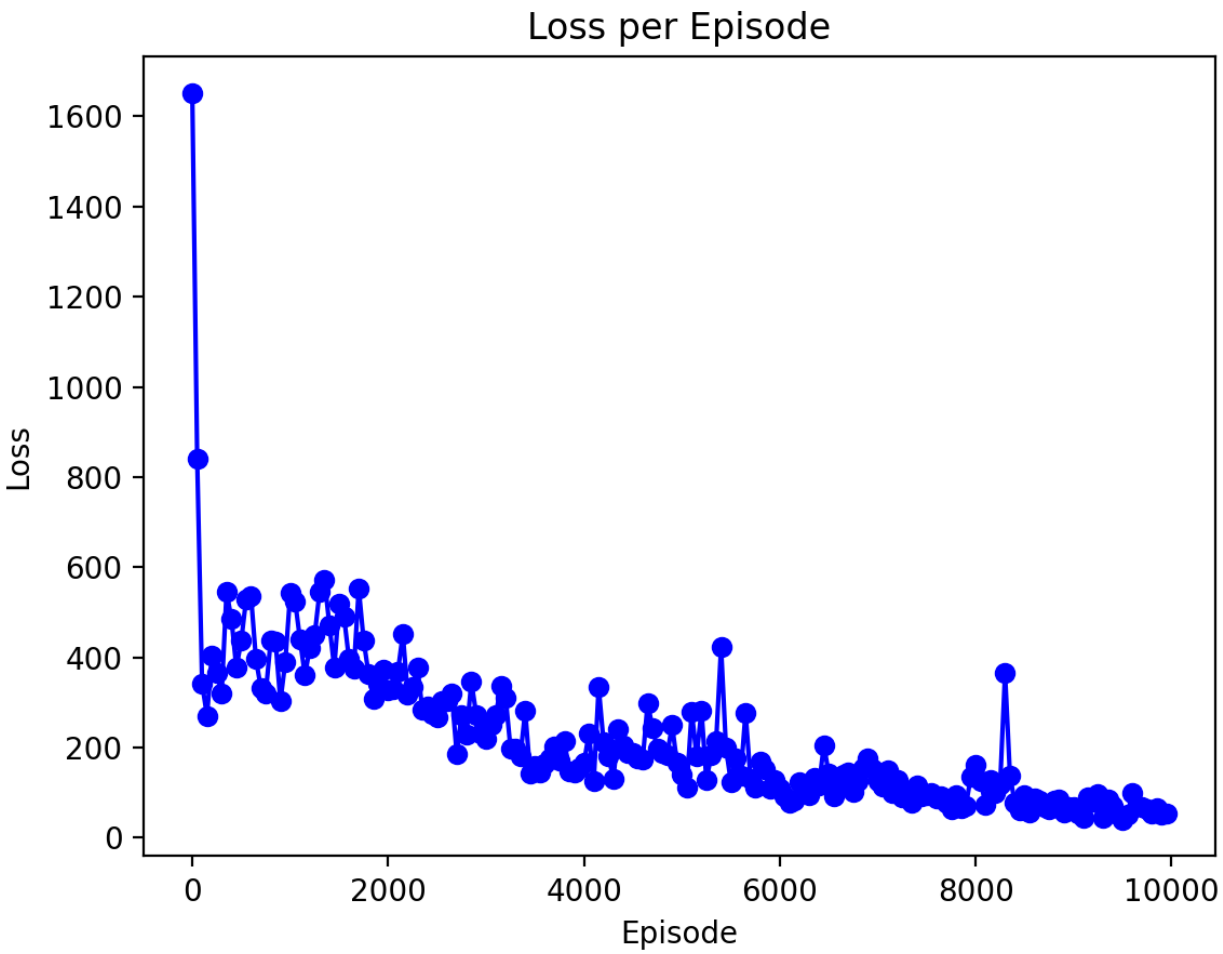
This was the "final boss" of my project, and took me the most time. I trained many agents hoping to reach 100% accuracy, but started off quite far off. I trained many agents, and none were that good, and I started attributing some of that to the fact that my maze generator wasn't consistent about placing doors and keys. Rather than always dropping a yellow key, a red key, a yellow door, and a red door every time, it'd sometimes not drop a door or a key here and there, resulting in only 2 or 3 items in the maze; never in a way that made the maze unsolvable, but perhaps it was variant enough to make generalized knowledge difficult to attain. I changed this, and agents still were not good.

For some reason, no matter what rewards I set, the agent would end up in a loop, commonly trying to open doors infinitely. Weirdly, agents were getting through training, meaning they weren't hitting infinite loops then, but then when I tested the agents, they would hit loops. Then it occurred to me: the key_state wasn't being reset properly each time a new randomly generated maze was created after each episode. This means that if an agent had picked up a

key in the beginning of training, it had door access *for the rest of training*. This trained agents to think they could open doors whenever they want, and so they would attempt to do so in an infinite loop when I tested them.

With this fixed, accuracy went up quite a bit, but I still had to mess around with rewards. I tried to transfer the wisdom I learned from Q Learning and use 0 rewards for picking up keys and opening doors and that wisdom didn't transfer *at all*. These agents required more instant indication of success. So I kept messing around with rewards, and it seemed like a higher reward for picking up keys than doors worked well.

The best agent I managed to train was one with 10,000 episodes and .999 decay. It has a 95% accuracy. The loss graph is below.



As you can see, this one fluctuated a lot, but it ended up with a good agent.

Conclusion

This was a good experience to see how Q Learning and Deep Q Learning compare, and to see some of the difficulties in training agents. The strength of Q Learning is that it's so simple, and training a new agent for every new maze actually has its perks. It's quite fast if you optimize

even a little bit, and it's not difficult to reach an accuracy of 100% or close to it. But of course, the simplicity of Q learning also comes with downsides. It's not the most flexible and scales very poorly. Deep Q Learning is a bit more difficult to work with from the start, but even in this project alone, I could see the vast amount of possibilities that this method of learning unlocks. I believe I chose the correct methods for the tasks at hand, and I'm happy with the result.

Citation

Zhou, Xiaolin. "Optimal values selection of Q-learning Parameters in Stochastic Mazes." In Journal of Physics: Conference Series, vol. 2386, no. 1, p. 012037. IOP Publishing, 2022.