Mancala

The game mancala is a two-player game played with a Mancala board and 48 stones/marbles/pieces. The board is made up of two rows of six pockets, with 4 stones placed in each pocket to start the game, and each player also having a pocket to their right (their "store or "Mancala"). The objective of the game is for players to collect the most pieces by the end of the game on either their side of the board or their mancala. The gameplay is as follows (Endless Games):

1. The game begins with one player picking up all of the pieces in any one of the pockets on his/her side
2. Moving counterclockwise, the player deposits one of the stones in each pocket until the stones run out.
3. If you run into your own Mancala (store), deposit one piece in it. If you run into your opponent's Mancala, skip it and continue moving to the next pocket.
4. If the last piece you drop is in your own Mancala, you take another turn.
5. If the last piece you drop is in an empty pocket on your side, you capture that piece and any pieces in the pocket directly opposite.
6. Always place all captured pieces in your Mancala (store).
7. The game ends when all six pockets on one side of the Mancala board are empty, or a player places 25 or more pieces in their Mancala
8. The player who still has pieces on his/her side of the board when the game ends captures all of those pieces.
9. Count all the pieces in each Mancala. The winner is the player with the most pieces.

Setup

Game Environment

The game environment was implemented as a class, with member functions similar to the ones found in the environments for OpenAI Gym. This allowed for familiarity and compatibility with agents when training the reinforcement learner. The complete code for the mancala environment can be found in MancalaEnv.py

Board

The environment for the game is rendered as a numpy array, where:

1. The 1st to 6th digits indicate the number of pieces in the first player's board
2. The 7th element is the number of pieces in the first player's mancala
3. The 8th to 13th element represent the number of pieces in the second player's mancala
4. The 14th element in the number of pieces in the second player's mancala

A sample representation of this is as follows:

```
Board = [4, 4, 0, 5, 5, 5, 1, 4, 4, 4, 4, 4, 4, 0]
```

Action

The actions in the game were passed as integers representing the location of the hole from which stones would be moved. Given the implementation of a board as a numpy array, these actions for the players

were defined as 0:5 for player 1, and 7:12 for player 2. Taking an action on the board was implemented in the MancalaEnv() class as step(). The step function took in an action as well as a player id (1 or 2), and using a reward function (described in the next section), returned the board state after the opposing agent (also described in later section) had made a move, the reward as described by the reward function, a *done* flag describing if the game was finished or not, and a *win* flag to indicate if the agent had won the game. To help with exploration and random sampling of actions, a sample() function was created as well to return a randomly sampled action based on the player id.

Rewards

Three different reward functions were implemented in the environment, which were variables in many experiments of the project, and were used to encourage different behaviour in the agent The reward structures are discussed as follows:

Sparse Reward

This reward structure provided minimal information, where a reward of +10 was provided if the agent on the game, and returned -10 if the agent lost the game to the opponent. The code for the reward structure can be found in the sparse_reward() method.

Staying Ahead

The reward structure for staying ahead was developed to encourage the agent to maintain a winning score. +10 and -10 were awarded for winning/losing the game respectively, but a reward of 0.1 was also provided if the action selected by the agent resulted in a score indicating the agent was winning and 0 otherwise. The code for this reward structure can be found in the keep_ahead_reward() method.

Falling Behind

In this reward structure, the agent was punished for taking actions that resulted in a score indicating the agent was behind. Similar to the previous structures, a reward of ±10 was also provided upon completion of the matches, however a reward of -0.1 was given if the action resulted in the agent falling behind. The overall aim is similar to the previous reward structure, however it is a different way to encourage/disincentivize a certain behaviour, and could have different impacts on the strategy of the game. The code for this structure could be found in the dont_fall_behind_reward_method()

Agents Used

Deep Q-Networks:

The reason behind using deep Q methods was rooted in the large number of states contained in the environment. Given 12 states and 48 pieces on the board, a simplified upper bound for the state space would yield 12^{48} possible states. Relying on tabular methods, the state space is far too large for the value of each state-action to be stored and indexed efficiently. Instead, by using a neural network as an approximation of the state-action, a much smaller representation can be created, and relied on after training to provide results that would be expected if a tabular method was to be used to capture state-action values over the entire state space. To implement the Deep Q network, I used an implementation of Deep Q Networks used to solve the CartPole environment, and added functions in order to make the agent work with the Mancala environment I had written (Link provided below references). Code for the neural network used can be found in the DQN() class. For training, the neural network architecture relied on:

- **Layers:** The input of the neural network was of size 14, to match the representation of the state space. This fed into a 64 unit hidden layer, which in turn fed into a 64^2=4096 hidden layer. This hidden layer then fed into a 64 unit hidden layer, which then produced a 6 unit output
- **Activation Function:** The activation action used between the layers was a Leaky Rectified Linear Unit (Leaky ReLU). One disadvantage of the simple ReLU is that it can give a value of 0 for negative values and keep the optimization stuck with a gradient of 0. This activation function The optimizer used in overcomes this by adding a very small slope for the negative values of input. Combined with the fact that the reward structure for the sparse reward will provide many values of 0, this function is preferred over the simple ReLU function.

DQN vs DQN + Replay

Two agents were used in the training, a simple DQN (in the class DQN())which used a neural network to generate an action given a game state, and a DQN that included a replay memory (in DQN_replay()). The DQN including replay was used because it was believed that decorrelating experiences through random sampling could produce improved performance.

Opponents

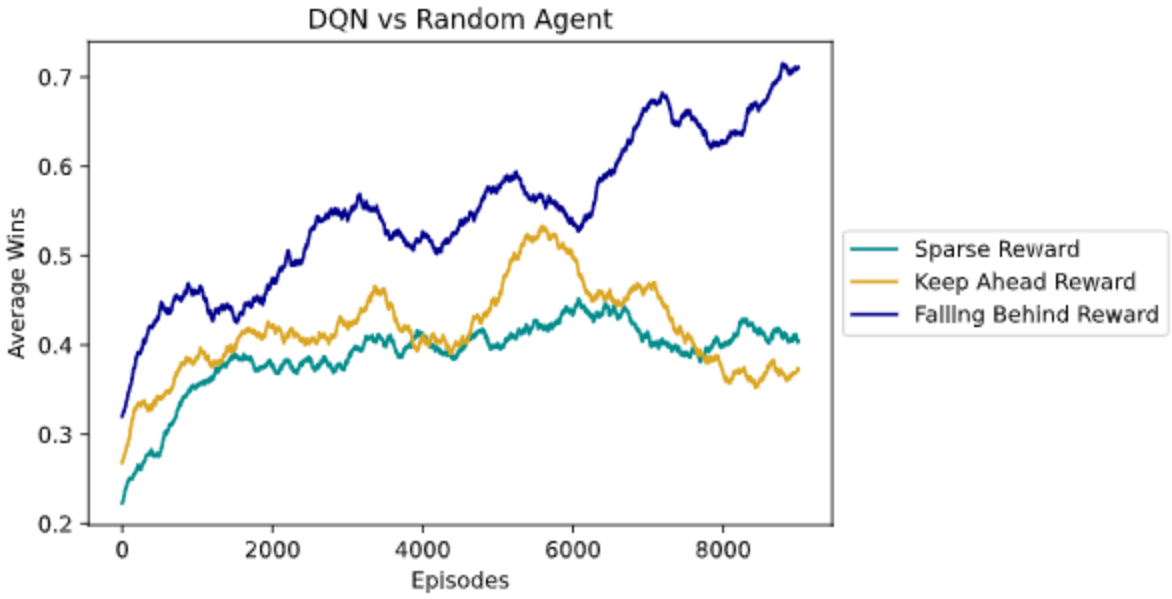
Three opponents were used in the agent did the following:

1. Random agent: This agent selected actions randomly from available actions.
2. Exact agent: This agent made a move that ensured the last piece ended in the mancala, and selected a random action if that was not available
3. Max agent: This agent tests every possible move and chooses the one that maximizes the score for the agent.

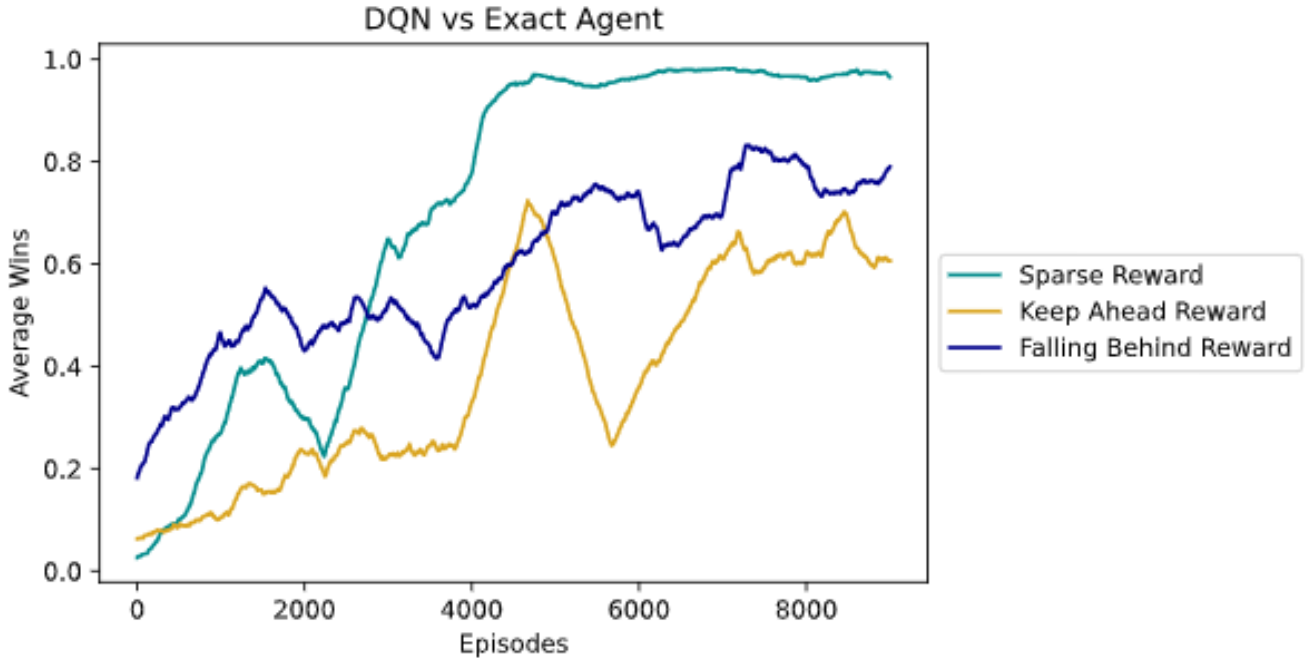
Procedure

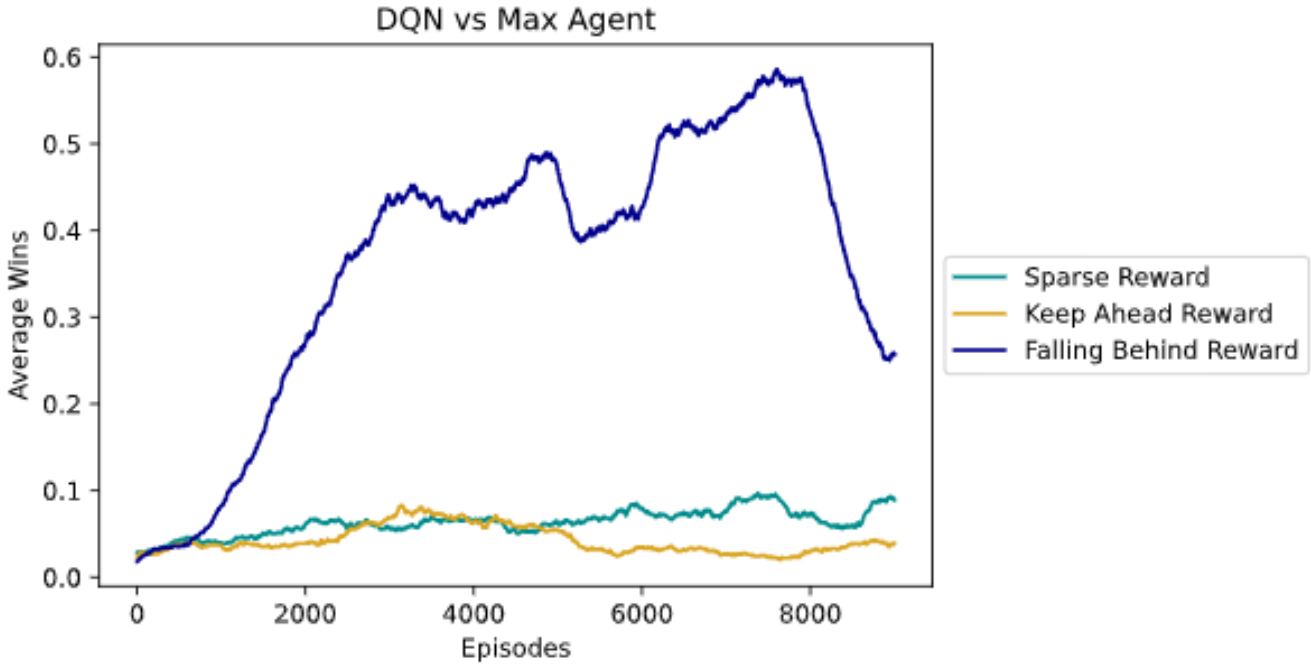
Training

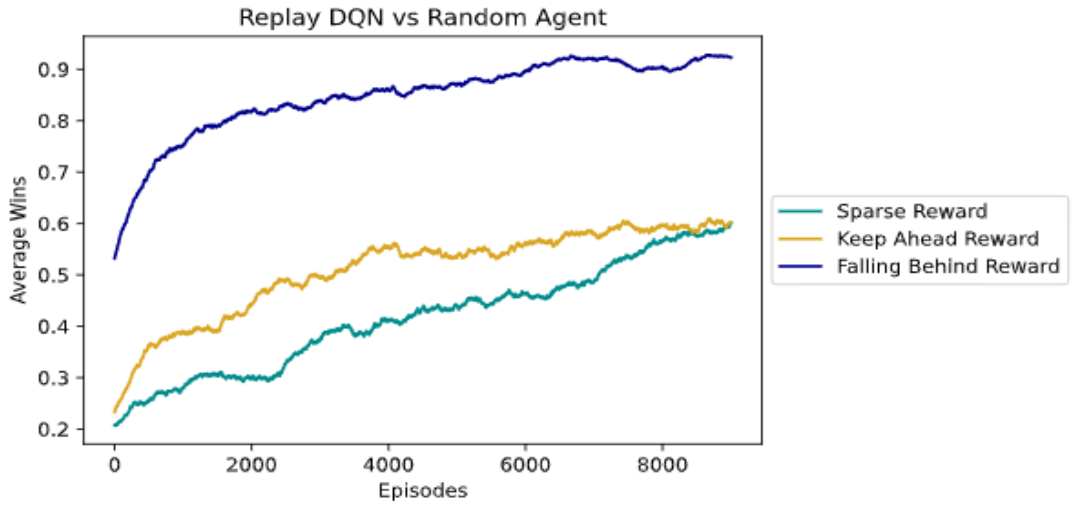
In training, the agents were run against each of the opponents using the different reward structures. Given that there were multiple ways to win the game, I decided to run 10,000 iterations of the agents

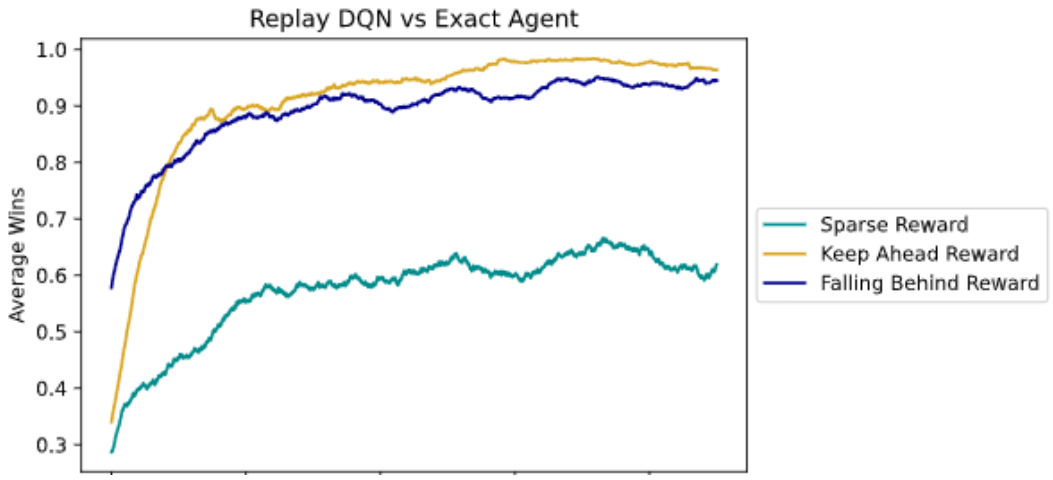


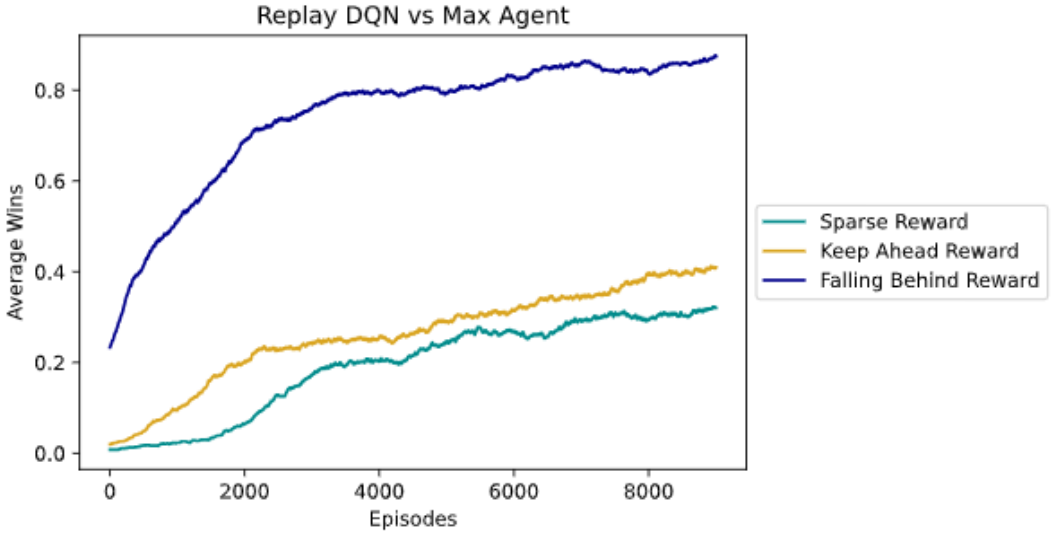
for the training phases, and then once a model was decided on for the final phase, 20,000 episodes would be run to fully train the agent. The results are as follows:






Replay DQN vs Random Agent
Average Wins
Episodes
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
0
2000
4000
6000
8000
Sparse Reward
Keep Ahead Reward
Falling Behind Reward


Replay DQN vs Exact Agent
Average Wins
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1.0
Sparse Reward
Keep Ahead Reward
Falling Behind Reward


Replay DQN vs Max Agent
Average Wins
Episodes
0.0
0.2
0.4
0.6
0.8
0
2000
4000
6000
8000
Sparse Reward
Keep Ahead Reward
Falling Behind Reward

From the images, it can be seen that the reward structure where falling behind is punished is the most effective. Against the max agent which maximizes the score after testing possible moves, the falling behind reward structure performs much better than the other two reward structures. In the case of other opponents, the falling behind reward performs fairly well, coming up short against the sparse and keep ahead reward structures when facing the exact agent with a simple DQN and DQN with replay, respectively. In the final DQN, the falling behind reward structure will be used to train the agent.

When comparing the use of DQN vs DQN with Replay, it is also apparent from the results that the DQN with Replay performs much better than the DQN alone. This confirms the use of randomized experience replay in function approximators; by sampling randomly, the experiences are de-correlated from the past actions of the agent, which enables the agent to handle new experiences better. In the final agent, the DQN with experience replay will be used.

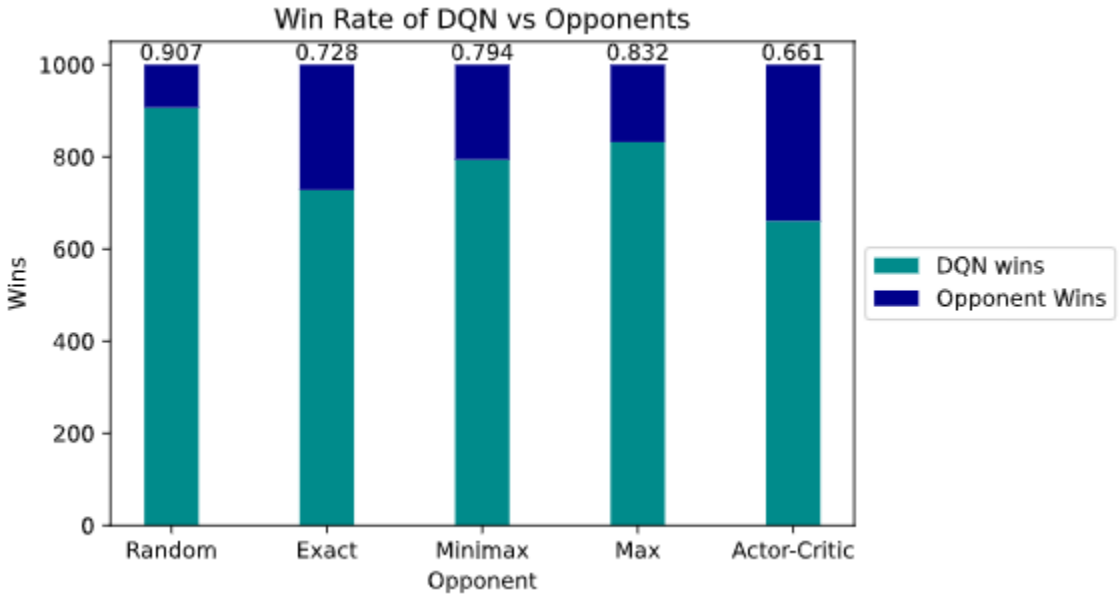
Final Agent

To train the agent, a DQN + replay architecture with the fall behind reward structure implemented. It was trained against an exact agent, a random agent and then a max agent in that order. The model was then saved, so that it can be used in a notebook for testing (testing.ipynb) .

Testing

To test the agent, the agent was pitted for 1000 matches against the following agents:

1. Random Agent
2. Exact Agent
3. Minimax Agent- Run the minimax algorithm to help the opponent maximize their action based on future possibilities.
4. Max Agent
5. Actor Critic Model- This agent was provided as the method trained using the other agents in the original Github repository, so I decided to see how my agent would perform in comparison (results discussed in following section, they come with an asterisk).



Results & Improvements.

The DQN agent performed fairly well, winning over 50% for all matches. One important note for these results was that in training, I made a large oversight by always setting the DQN agent to be player 1. This had important consequences because in testing, I thought I could flip the board, and have the agent make decisions based on the new board, but I observed that the DQN performed much more poorly even against the simpler agents. Therefore the results observed here are for if the DQN + replay agent is allowed to move first.

Human Vs DQN

The code for this is implemented in human_play.py, and allows for a human to play against the agent on the terminal. From playing the DQN myself, I discovered that the strategy of the DQN was to take advantage of the capture rule. In playing the DQN, the agent always opened by moving the stones on the last hole, which immediately allows for a capture if the pieces opposite the hole are not moved. This could be a result of the training, where random/exact agents in opening might have missed this step, and allowed the DQN to build a 7-1 advantage in just two moves, which greatly increases the chances of the agent getting the +10 reward. Throughout the game this "counterattack mindset" was prevalent, where the DQN would almost always try to use the capture rule, and always protect itself from getting its pieces captured. An unintended strategy implemented by the DQN was also to intentionally provide invalid moves. As part of the repository I used/the environment in training, inputting a move to a hole with 0 pieces was invalid, and did not change the game state. When playing the DQN, in order to protect the configuration of pieces it had, it would intentionally play an invalid move, so that it could keep the pieces as they are. In some games, it forced my hand and led to the DQN winning, but in other cases I was able to win by making sure I kept an option of capturing the DQN pieces (to force the DQN to play invalid moves), while slowly filling up my Mancala.

Conclusion

Lessons Learned

Overall, my project ended much different than I had started out. I wanted to work on one of the OpenAI robotics challenges, but after running into multiple problems (described below), I decided to pursue a new project, but was able to apply a lot of what I had learned in terms of RL concepts and programming, to successfully complete this project.

Obstacles:

Multiple times in the robotics project, I ran into a problem where the version of python used for one part of the project (for example the learning algorithm) was not compatible with the version used for other parts. Running the files created a lot of deprecation warnings/errors, and I ended up spending a lot of time debugging code trying to get a successful version running. For example a lot of the algorithms I tried to run were based on tensorflow 1.14, which was written for Python 3.7.0. Upon downloading Python 3.7.0 and reverting tensorflow to a previous version, some commands for the mujoco-py library required newer versions of Python. Though I was unable to overcome this obstacle, which drove the decision to work on the Mancala project, I still learnt a lot about managing versions of packages as well as different versions of python (pip is now my favourite command), and searching up documentation on methods based on different versions within a package.

Another major obstacle I ran into was my approach to solving the robotics environments. To start the project, I decided to implement a DQN to solve the environment, but realized that one of the shortcomings of the DQN is suffering from catastrophic forgetting, where new experiences can make the agent perform worse over training. Though this is mitigated with (random) experience replay, the DQN overall performs horribly for tasks with non-stationary goals, as new experiences will continuously be needed to be observed. Furthermore, the action space for the fetch environment was continuous, and properly discretizing the action space was a challenge; I tried to use the value of the output to one-hot encode the joints at a fixed velocity, then tried to use the output to select from a few arrays setting combinations of joints to different velocities, all to no avail. Overall, it took me about a week to get the DQN environment set up, and from watching videos of other methods, I realized I would have to almost basically scrap the DQN approach for other methods. This obstacle however ended up being a benefit later on in the project because it helped me firmly understand how to apply the DQN to the mancala environment.

Struggles

When working on the project, I would say I originally struggled with understanding how to use the mujoco-py library in an effective method. As mentioned above, I spent about a week trying to write the DQN code and training loop using TensorFlow, then PyTorch. In this week, I struggled with taking the possible outputs of the mujoco-py environment (the states, possible actions) and using them in the training loop. To manage this, I ended up writing many functions that could parse the outputs and return values to be used with the neural network libraries. Though this method worked, it was not very efficient. However, I think struggling through this helped develop my understanding of the PyTorch library, as well as searching up documentation for methods in PyTorch. Furthermore, in trying to avoid writing many different functions for the Mancala task, I decided to take the mancala game board and wrap it in a class containing methods used by the OpenAI gym environments, so that I can maintain similar code to what I had done for the mujoco-py method. This ended up being more efficient and easier to implement and understand.

Frustrations

When working on the Mancala project, a major frustration I ran into was not planning experiments carefully. When running the training, I would use a large number of iterations, and running one experiment alone could take up to an hour, and I had 6 experiments to go through. More often than not, I would let an experiment run, only to find out there was something I missed/ made a mistake on, which meant that the time spent was basically wasted. To overcome this, I started spending time making sure everything was et up properly; running a small number of iterations, and making sure the outputs contained expected results/no missing values. In addition, I started scheduling my work around the training, by working on other parts of the project while the training was being conducted, or leaving the training to run before going to bed. Overall, I think what made this a frustration for me was that it was very easy to miss something, and sometimes it could be hours of the day wasted because at the end of the day, I was the one who was not careful enough.

Learnings

I learned a lot from this project. In addition to gaining a strong understanding of Deep- Q networks, the project made me a stronger programmer. I became more comfortable with obtaining code from online sources and making modifications as needed to serve my purposes, rather than writing something to

make what I had work with what I had downloaded. For example, writing an environment for the Mancala game board helped me simplify the training function by using methods I was familiar with when using OpenAI Gyms.

From this experience as well, I think one part of the project I could have worked on more is relying on GitHub for version management. To manage my code better, I downloaded GitHub Desktop, and linked the folder to the workspace folder in Visual Studio Code. This enabled me to work directly on my files in Visual Studio Code, but as a result, I ended up doing less commits of my files over time, meaning I was less able to track my progress over time. In the future, I need to make sure I make an effort to constantly commit files, or break down the project into specific tasks, and commit when those tasks are completed in order to better record progress made using GitHub.

References

Endless Games. "Classic Mancala Instructions." 2015. *Endless Games.* 18 04 2021. <https://endlessgames.com/wp-content/uploads/Mancala_Instructions.pdf>.

Versloot, Chris. "Leaky ReLU: improving traditional ReLU." 15 October 2019. *Machine Curve.* Website. 19 April 2021.

CartPole DQN implementation: https://github.com/ritakurban/Practical-Data-Science/blob/master/DQL_CartPole.ipynb