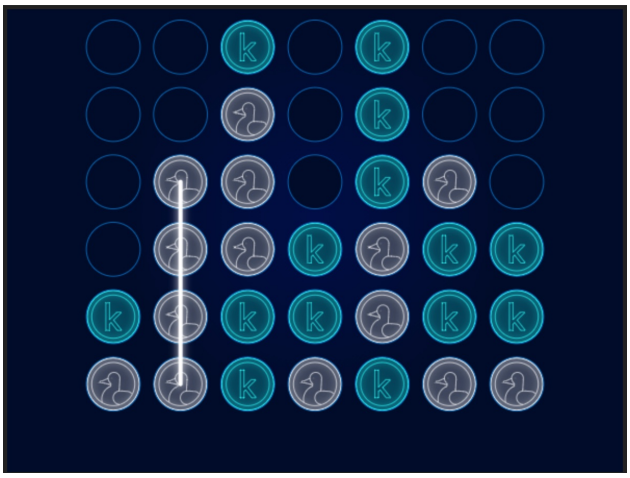
**Using deep Q-learning to play Connect Four**

**Motivation** Connect Four is another simple two-player game that involves dropping colored discs into a seven-column, six-row vertically suspended grid. The objective is to be the first player to form a horizontal, vertical, or diagonal line of four discs. The game has 4,531,985,219,092 possible states, so it is unfeasible to use standard Q-learning as opposed to with rock paper scissors. Instead, deep Q-learning was used, with the Q-table in Q-learning being replaced with a deep neural network.

For convenience, Kaggle's `connectx` environment was used. The environment is part of a Kaggle competition called Connect X in which participants build an algorithm that can play Connect Four and then play against each other to see whose algorithm is the best. With `connectx`, one can train an algorithm to play Connect Four without actually building the game. The `connectx` environment can only be interacted with using Python, and the algorithm used here was adapted from www.voigtstefan.me/post/connectx/ due to the complexity of deep Q-learning algorithms. In addition to the fact that `connectx` can only be interacted with using Python, we used Python as opposed to R here due to our unfamiliarity with the language and the challenge that would bring.

Below is an example of what the `connectx` environment looks like. There are two agents that can be used for training: a "random" agent and a "negamax" agent. The random agent makes moves at random, while the negamax agent makes moves that are better than random but still beginner-level. Most participants use the random agent to train their model and the negamax agent to test its performance. The illustration shows the result of two random agents playing each other.



**The deep Q-learning algorithm** Most of the source code obtained from the blog site was kept the same, with some modifications. Similar to the source code and other examples found online, the neural network was trained using `connectx`'s random agent. Eight different agents were trained, and we determined which agent performed the best against the random agent during training. There seemed to be some syntax errors in the source code that made it difficult to test the trained agent against `connectx`'s negamax agent (see the challenges section). However, the training results do illustrate the effectiveness of Monte Carlo dropout, which is the main contribution of this portion of the project.

The three-layer neural network was kept mostly the same as the source code except that RMSprop was considered as an optimizer in addition to Adam. This is because RMSprop seemed to be another common optimizer that is used to train deep Q-learning models for playing Connect Four and it is possible that it leads to better performance than Adam. We also changed the $\epsilon$ used for the $\epsilon$ decay to allow the agents to explore the environment more. The initial $\epsilon$ in the code is 0.1, but we set it to 0.15 for some agents. We thought allowing the agents to explore their environment more could result in better performance. Finally, Monte Carlo dropout was added after the first layer and the second layer. A dropout rate of 0.2 was used since this is fairly standard. Monte Carlo dropout is widely known as a way to automatically improve model performance by randomly switching off neurons, which prevents overfitting. We also tried changing the

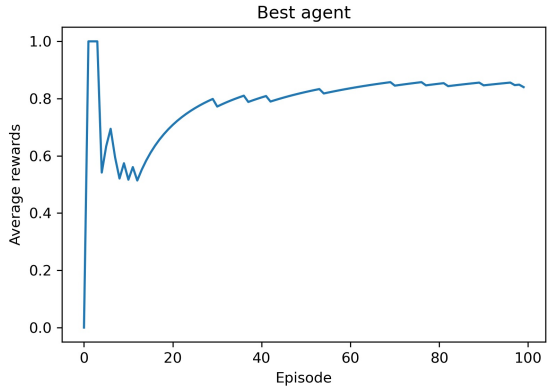
learning rate, discount rate, and penalization for making an impossible move, but these often resulted in the agent getting stuck or not learning anything.

Altogether, eight different agents were trained, with or without Monte Carlo dropout, with RMSprop or Adam as the optimizer, and with the initial $\epsilon$ equal to 0.1 or 0.15. The original algorithm used Adam as the optimizer, did not use dropout, and set the initial $\epsilon$ equal to 0.1. Due to lack of computational power, only 100 episodes were performed for each agent and a batch size of 20 was used. Initial weights for each layer in the neural network were obtained from the original source, and these weights were updated by the algorithm after each iteration. The proportion of times that each agent won against `connectx`'s random agent was recorded, and the agent that had the highest winning percentage was considered the best.

| Dropout | Optimizer | Initial exploration rate | Pct. of games won |
|---|---|---|---|
| Yes | RMSprop | 0.15 | 0.86 |
| Yes | Adam | 0.10 | 0.86 |
| Yes | RMSprop | 0.15 | 0.86 |
| No | RMSprop | 0.15 | 0.83 |
| No | Adam | 0.10 | 0.79 |
| Yes | Adam | 0.10 | 0.78 |
| No | RMSprop | 0.15 | 0.77 |
| No | Adam | 0.10 | 0.75 |

**Results**

The winning percentage for each agent is shown in the table above. Three agents tied for the best, all of which were trained using Monte Carlo dropout. Agents that were trained using dropout had an average winning percentage of 0.84, while agents that were trained without dropout had an average winning percentage of 0.785, so dropout seems to improve agent performance. Agents that were trained using RMSprop as the optimizer had an average winning percentage of 0.83, while agents that were trained using Adam had an average winning percentage of 0.795. While a smaller improvement than implementing dropout, it appears that RMSprop was a better optimizer. The average winning percentage for agents trained using $\epsilon$ equal to 0.15 was 0.83, while the average winning percentage for agents trained using $\epsilon$ equal to 0.1 was 0.795, so allowing the agents to explore their environment more did improve their performance. Overall, all three adjustments seemed to result in a better agent.




The plot above shows the average rewards obtained by one of the best agents for each episode. This is the agent that was trained using dropout, RMSprop as the optimizer, and initial $\epsilon$ equal to 0.15. An agent was rewarded 0.5 for making a move when the game is not over, 1 for winning the game, and 0 for losing the game. If the agent made an impossible move (i.e., trying to use a column that was already completely filled), it was penalized -10. It is unclear why the average reward was very high for the first few episodes, but the agent clearly learned something. The plots of the average rewards for all of the agents looked similar.

**Challenges** The main challenge we faced with this portion of the project was getting the source code to work properly. Due to updates in the `kaggle_environments` package, the source code initially failed to run. This was after cloning the GitHub repository and running it as the author intended. After searching online, we found that the `board` attribute of `state` had to be referred to as `state['board']` rather than

`state.board`. While this is a relatively trivial error, it took several hours to debug the code and determine that was the issue. In addition, we were never able to get the code that tests our agents against the negamax agent to work. This is likely due to more syntactical issues in the code.

In addition, it was difficult to find complete source code online that worked as is. We had the intention of using a deep neural network to train the agent, and there was a lack of examples that used a deep neural network to play Connect Four. Most of the examples on blog sites were not complete and left out portions of the code, so they failed to run. It took several hours to find complete source code for what we specifically wanted to accomplish.

Our lack of understanding of the Python programming language also made this portion of the project difficult. For example, it was difficult to completely understand the source code, and our lack of understanding made debugging the code difficult as well. However, this project also served as an opportunity to improve our Python skills, and it was extremely useful from that perspective.

**Possible improvements** The most obvious improvement that could be made is testing the agents against negamax so we can get an idea of how well they perform against a nonrandom opponent. Unfortunately, this was difficult due to errors in the source code. We could also train the agents for many more iterations. 40,000 episodes seems to be the standard amount for training, but this could take days on our machine. For the sake of this project, we only performed 100 iterations for each agent. It would also be interesting to consider other types of algorithms that are used for playing Connect Four. Two very common ones are the minimax algorithm and Monte Carlo tree search, but neither of these involve reinforcement learning.

Most significantly, it appears that Monte Carlo dropout greatly improves deep learning models, including in the reinforcement learning setting. This portion of the project was fun but also challenging due to our limited understanding of Python. It is interesting that a robot can be trained to win at Connect Four by playing a random agent.