Playing simple games with reinforcement learning

A final project for Data Analysis IV

true true

For our final project for Data Analysis IV, we decided to focus on reinforcement learning. We believe this was a good choice, as reinforcement learning is relatively intuitive and has a variety of applications. We decided that some simple traditional games would be a good area of application, specifically rock paper scissors and Connect Four. We explain further our motivation and methodology in the following sections, each focusing on one of the two games.

Can we use reinforcement learning to train an AI to win best of three matches in rock paper scissors (RPS)?

I chose this specific game as it was very simple to conceive this best of three match as a reduced 'maze' type problem. Taking what I have previously learned regarding Q-learning based reinforcement learning, I knew that I would be able to parameterize this in that framework. Additionally, compared to some other simple games, Rock Paper Scissors (RPS), has very clearly definined winning, losing, and drawing conditions, and has a minimum amount of hidden information or other data that needs to be tracked, other than a given type of hand thrown in a match and what the outcome of each match.

Conceptually, my goal was to create a system that would throw a given sign more often than others. Essentially, a robot that, while it does throw every sign, tends to throw one sign more often than others. Additionally, I would like to represent 3 separate states in each system, representing the 3 potential rounds in a best of three match for RPS. Ideally, we can determine the success of our given agent by examining the Q-matrix, and seeing if the column representing the ideal sign to throw is the largest for each given 'round' in our best of three set.

The main concept that we are adding, as something of value in our examination of reinforcement learning, is the addition of an element of stochasticity. This randomness in a sense represents how much variance there naturally is in a game of RPS, even with the understanding that an individual agent may or may not prefer to throw one sign more often than others.

A priori, we choose to define a successful outcome as a Q-matrice produced after training that has 'winning' choices against an agent that tends to throw a particular sign, in at least two of our three rows, representing two of our three rounds in a best of three. We can interpret this as our agent winning more often than not against our biased sign throwing robot.

Method

Below, you can see our initial parameterization of our R-matrix for a round in which our 'robot' throws chooses to throw rock. Examining it specifically, the way it is designed is such that each row of our matrix represents the choice of what to throw or what value a throw has in a particular round. The columns represent the value of what a given sign would have against specifically rock in this condition. So, expanding further, the columns represent, in order, the value of the agent throwing rock, paper, or scissors, against our robot throwing rock. We chose to represent a tie as an option with some amount of value, given that it isn't quite a loss (and generally is considered a breakeven or re-do in actual play), thus we assigned this outcome a positive value, but a relatively small one, of 10. For a robot throwing rock, the tie scenario is our agent throwing rock. We chose to represent a winning outcome as an option with the largest amount of value, given that this is our desired outcome, thus we chose a value of 100, significantly larger than the mild positive value of reaching a tie. For a robot throwing rock, the winning scenario is our agent throwing paper. We chose to represent a losing outcome as an option with no value, given that this is the absolute must avoid outcome, thus we chose a value of 0, much smaller than our winning or tying outcomes. For a robot throwing rock, the losing scenario is throwing scissors.

$$\begin{bmatrix} 10 & 100 & 0 \\ 10 & 100 & 0 \\ 10 & 100 & 0 \end{bmatrix}$$

Below, we can see the exact same parameterization as described above, but for a robot that throws paper in a round. Respectively, the tying, winning, and losing choices for our agent are paper, scissors, and rock. This is just a simple rotation of values in the columns of our matrix as compared to our rock throwing scenario.

$$\begin{bmatrix} 0 & 10 & 100 \\ 0 & 10 & 100 \\ 0 & 10 & 100 \end{bmatrix}$$

Lastly, we can see another parameterization, this time for a scissors throwing robot. Again, we see that our tying, winning, and losing choices have shifted again respectively to scissors, rock, and paper

$$\begin{bmatrix} 100 & 0 & 10 \\ 100 & 0 & 10 \\ 100 & 0 & 10 \end{bmatrix}$$

The next issue was determining how to have stochasticity in our robot playing RPS. Interestingly enough, there was a natural point in our Q-learning algorithm to introduce stochasticity. We decided to implement this by modifying our reward matrix. In standard Q-learning, the reward matrix is static, as the 'maze' for example, that you are trying to solve, stays the same. We believed that a worthwhile implementation of randomness was instead to sample a new reward matrix for each iteration of our Q-learning algorithm. This clearly adds randomness representative or similar to the randomness that would be present in a game of RPS. Then, after having the sampling equation defined, it is relatively simple to increase the likelihood of sampling a matrix representing our robot throwing a given sign (in this case, rock) more often than the others. This is a clean representation of how a robot can still have some randomness in it's actions, but would tend to throw one shape more often than others. We implemented this by sampling from a list of matrices, wherein there was a 50% chance of sampling the rock matrix, 25% of sampling the paper, and 25% of sampling the scissors.

With regards to our output, for this problem, there are a great deal of our parameters that we were able to choose and modify. Namely - The reward values in our given matrix (could be tuned up or down), the ratio of sampling for a given sign to be thrown (50%, 25%, 25% is relatively crude, we could even do 90/5/5, or 80/10/10), the gamma value in our Q-learning algorithm, the learning rate in our Q-learning algorithm, and lastly the number of iterations of our algorithm.

To summarize for our initial attempt, we chose reward values of 100 for wining, 10 for tying, 0 for losing, and a sampling ratio of 50%, 25%, 25% for our preferred sign and the other signs respectively. Additionally, our gamma value was set to 0.9, our learning rate was set to 0.2, and our iterations were set to 100.

Output

Below we can see the output of our first attempt in our Q-matrix. Note that this is against an agent trained to throw rock, so with regards to scoring, we wish to see the greatest values in our second column (throwing paper against rock), for at least two of our rows.

$$\begin{bmatrix} 21.0452679023135 & 23.7881482224164 & 3.04526790231352 \\ 21.0452679023135 & 6.37134822241643 & 24.2852679023135 \\ 21.0452679023135 & 21.1502432382345 & 5.80704378288221 \end{bmatrix}$$

Examining our Q-matrix, our criteria has indeed been fulfilled. In the first row we see that our paper column has the largest value, 24.3 is larger than both 24.28, and 23.8. In our second row, rock has the largest value, which while isn't ideal, is not outright horrible (as a draw is better than a loss), of 24.28 against 6.38 and 2.56. In our final row however, paper again has the largest value of 24.37.

We were relatively fortunate to obtain ideal results with our initial choice of parameters. However, we were interested in potentially obtaining greater separation in our results (ideally, with wins in all three rounds/rows instead of just two).

We then chose to play around with some of our parameters, seeking to see if there was any improvement we could have. A priori, it seems like upweighting the sampling rate to increase bias and increasing the iterations to reduce variance would be sucessful. Our next iteration of our Q-matrix with our sampling rate increased to 2/3, 1/6, 1/6 for our favored sign vs the others, and iterations increased to 1000.

$$\begin{bmatrix} 6.38963156672388 & 24.3868420373549 & 24.2852480665093 \\ 6.38963156672388 & 3.15022406276185 & 6.3868420373549 \\ 6.38963156672388 & 24.2852480665093 & 24.3713446519717 \end{bmatrix}$$

Interestingly enough, changing these parameters did not lead to a better Q-Matrix. We can see this in the results above. While paper has a decently large value in two of the columns, this agent actually responds most commonly to scissors, which is actually a poor outcome. This illustrates a concern that perhaps the direction we chose to modify these parameters is incorrect, or at the very least, not fully understood.

We then chose to play around again, with other parameters. Specifically, we redid our previous series of analysis, but with the specific noted difference in scores, from 100, 10, 0, to 200, 1, and -1, for winning, tying, and losing respectively.

$$\begin{bmatrix} 8.98048780487805 & 48.780487804878 & 48.780487804878 \\ 8.98048780487805 & 48.780487804878 & 1.81648780487805 \\ 8.98048780487805 & 48.780487804878 & 8.980487804878 \end{bmatrix}$$

After modifying our scores in our reward matrix, we finally see the ideal amount of separation we were desiring in our results. It is very clear that these changes in the parameters leads to the best results we have seen so far. There is large difference in the absolute values of the correct scores in 2 of our rows, and the winning strategy (to regularly throw paper if the robot throws rock) is found immediately.

Conclusion

For this specific problem, I did not find any outside code, other than modifications of code I had already constructed for the purposes of teaching myself Q-learning earlier in the semester. The main challenge I faced was designing the structure of the problem itself, and how to make iterations on the parameters to optimize the output. Furthermore, another challenge I faced was having some modifications of the parameters not affecting the output in exactly the way I had predicted. The only improvements I could consider making would be extensions of this to solve other games similar to RPS, but with greater numbers of signs (Rock, Paper, Scissors, Lizard, Spock is one example of a more complex game). Alternatively, it would be interesting to sample actual human behavior and use that as our basis for the parameter choices, and to see if a more realistic origin for our data leads to more realistic or improved results.

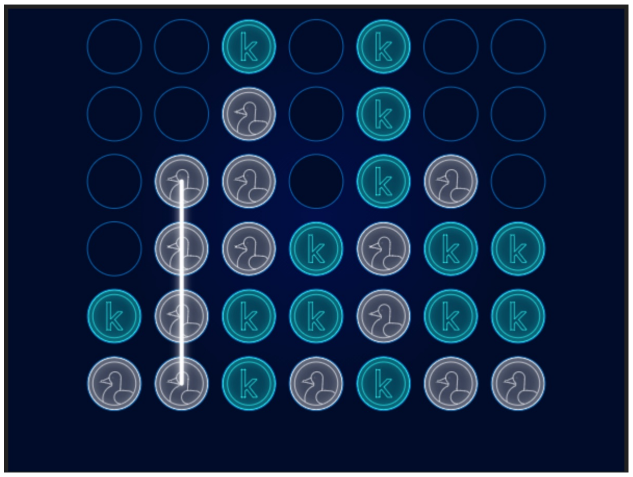
Using deep Q-learning to play Connect Four

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 ϵ used for the ϵ decay to allow the agents to explore the environment more initially. The initial ϵ 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.

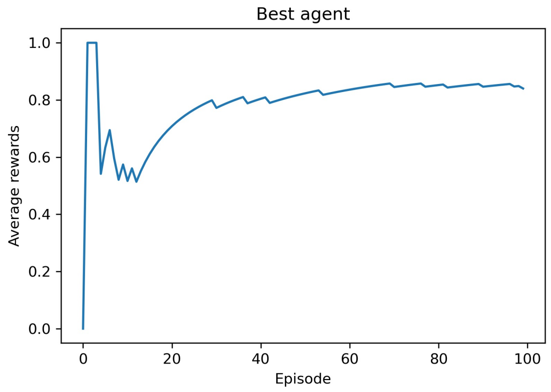
Finally, Monte Carlo dropout was added after the first layer and the second layer. A dropout rate of 0.2 was used since this is fairly standard. Monte Carlo dropout is widely known as a way to automatically improve model performance by randomly switching off neurons, which prevents overfitting. We also 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 ϵ equal to 0.1 or 0.15. The original algorithm used Adam as the optimizer, did not use dropout, and set the initial ϵ 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.

Results

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

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 may be a better optimizer. The average winning percentage for agents trained using ϵ equal to 0.15 was 0.83, while the average winning percentage for agents trained using ϵ 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 ϵ 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 to test our agents against the negamax agents 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.

Overall conclusion

This project was an extremely satisfying opportunity to grow as students and statisticians. It is always important to try and see if we can grow and adapt to new problems, and getting to choose our own area of focus for this last project led to ample learning. One of the biggest problems we dealt with during this project was efficiently collaborating across two different programming languages on two separate but related problems. We used GitHub for collaboration, and as compared to our previous project in Data III (which our group members also worked on together), we had a significantly easier experience, reflective of our increased mastery of the platform. Additionally, integrating images of Python output and our R figures was a challenge but a worthwhile one with regards to producing a final report and product that was relatively easy to use.