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.

$$\begin{bmatrix} 24.2852642112099 & 24.3713475580178 & 23.8070233956105 \\ 24.2852642112099 & 6.3868425604432 & 2.56702339561049 \\ 3.15012997561382 & 24.3713475580178 & 2 \end{bmatrix}$$

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.

$$\begin{bmatrix} 6.38963156672388 & 21.1501336820103 & 24.2852480665093 \\ 6.38963156672388 & 3.15022406276185 & 6.3868420373549 \\ 6.38963156672388 & 24.2852480665093 & 24.3713446519717 \end{bmatrix}$$

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.