Assessing the Impact of Robust Stochastic Operators on Q-learning Efficiency in an OpenAI Gym Environment with a Large Observation Space

Henry Demarest

Irvington High School

Abstract:

Reinforcement learning is a form of machine learning that trains a digital model by associating a reward value with different actions based on how beneficial they are. Several algorithms have been developed to facilitate the management of Q-tables, data structures used to store information about the effectiveness of each action at each state. A new algorithm, known as Robust Stochastic Operators, has been shown to perform better than pre-existing algorithms, like the Bellman or Consistent Bellman, within relatively simple training environments. This experiment sought to determine if these benefits of the Robust Stochastic Operators hold true in more complex environments as well. To do this, a program that compares this algorithm and pre-existing algorithms was created for the relatively complex BipedalWalker-v3 environment in OpenAI Gym. After training the model with these different algorithms, the RSO algorithm had a higher average reward and a longer average survival time, which indicates that the benefits of this algorithm did in fact, hold true. This improvement could have significant implications in numerous new technologies that utilize machine learning, such as autonomous vehicles.

Acknowledgement of Major Assistance:

This research was conducted without major assistance from any mentors. All experimentation was done individually and mentors provided periodic guidance and feedback of which I am very grateful.

Table of Contents:

Lists of Figures, Tables, and Equations (in the order they appear):

Introduction:

In recent years, significant improvements in computing speed and memory capacity have led to increased opportunities for advancement in the field of machine learning. This field generally encompasses any attempt to give computers the ability to learn or recognize patterns and has become increasingly important in modern technological achievements. Seen in everything from autonomous cars to social media algorithms to facial recognition, machine learning continues to be used to accomplish tasks that are simply too complex for traditional "hard-coding" solutions. Other projects require machine learning to compensate for the slight discrepancies that exist between pieces of hardware, such as the small differences between sensors on self-stabilizing robots.[7] Because of its power in parsing immense quantities of data, machine learning technology has also established a significant presence in the corporate sector.[8]

Reinforcement learning is a form of machine learning that has played an important role in developing programs that control robots and other "agents" with clear goals. This is because reinforcement learning is based upon the idea of a "reward" which tells the computer how successful its last action was at growing closer to that goal.[3] By running many episodes of a simulation, the computer gradually learns to relate the ideal action with any observation state it may encounter. This information is usually stored in data structures, like deep neural networks or q-tables, or by using frameworks, such as TensorFlow or PyTorch.

To facilitate studies of Reinforcement Learning algorithms, computer scientists have developed toolkits through which algorithms can be tested in various types of situations. The toolkit that will be used for this project is OpenAI Gym which provides a wide array of different types of environments in which to run programs. These environments range in complexity from text-based games or simple physics examples to the difficult task of getting a 3-dimensional robot to walk [2]. The complexity of these environments is defined by how many dimensions its observation and action spaces occupy, which is similar to how many different variables define the current state. For instance, the relatively simple MountainCar environment communicates only the cart's velocity and position on the track, while the complex Zaxxon arcade game environment communicates 128 different pieces of information about the state. With toolkits like OpenAI Gym, reinforcement learning programs can quickly and objectively be tested digitally allowing for easier comparison between different algorithms.

Using OpenAI Gym, the efficiency of a new q-learning algorithm developed by researchers at IBM will be compared to more standard reinforcement learning algorithms. In their paper, these researchers describe how this novel algorithm performed better than conventional methods, such as the Bellman or Consistent Bellman algorithms, in relatively simple OpenAI Gym environments.[1] To expand upon this research, this project will study the new algorithm in more complex environments with larger observation spaces. If its efficiency is found to be better than the aforementioned conventional methods, this algorithm could be very important because any improvement in efficiency is extremely valuable and can save companies significant amounts of time and money. This type of development could also accelerate technological development in fields like autonomous robotics.

Studied Algorithms :

These types of reinforcement learning algorithms deal with stochastic situations in which the exact result of an action is somewhat random and demands a probabilistic approach. This approach is known as a Markov Decision Process where different actions are given different probabilities of occurring in different states, thus allowing an agent to traverse the stochastic situation. The algorithms studied in this experiment use a Q-table to manage the MDP and associate probabilities with the available actions at different states. Traditionally, Q-table problems have often used the Bellman Algorithm as shown below:[5]

Equation 1: $$Q'(s,\ a)\ =\ Q(s,\ a)\ +\ \alpha[R(s, a)\ +\ \gamma\, max_{b \in A}\, Q(s', b)\ -\ Q(s, a)]$$

This equation is used to calculate the new value for Q'(s, a) which is the new value for the q-table at state *s* for action *a* out of possible actions A. In the equation, Q(s, a) references the current value of the q-table at that state and action, α is the learning rate, R(s,a) is the reward obtained by taking action a at state s, γis the discount factor, and max Q(s', a,) is the maximum reward at the next state. By taking the maximum reward at the following state, this process enables the program to reward actions that will eventually lead to a higher reward, even if it seems like it could have a negative effect in the current state. By iterating this algorithm many times, the program gradually begins to identify action sequences that result in the best results.

This algorithm can experience issues when there is error or approximation in the data, however.[6] These negative effects occur because of the close proximity between the optimal and suboptimal actions. One solution to this problem has been the Consistent Bellman Algorithm which is presented in Equation 2 below:[1]

Equation 2: $$T_C Q(s, a)\ =\ R(s,\ a)\ +\ \gamma E_P[1_{s \neq s'}\, max_{b \in A}\, Q(s', b)\ +\ 1_{s = s'}\, Q(s, a)]$$

The important distinction between Equation 1 and Equation 2 can be seen in the dependence on the indicator function **1**. This function determines whether the next state *s'* will equal the current state *s* and changes the equation accordingly. Specifically, when the two consecutive states are equal, the Consistent Bellman Equation seen in Equation 2 will take the same action a, thus increasing the gap between optimal to suboptimal solutions.

Another attempt to improve the Bellman Equation was devised by Wu et al. and is comprised of a set of Robust Stochastic Operators.[1] This algorithm is described by the equation seen in Equation 3 below:[1]

Equation 3:

$$T_{\beta}Q(s,\ a) = R(s,a) + \gamma E_{P} max_{b \in A} Q(s',b) - \beta(max_{b \in A} Q(s,b) - Q(s,a))$$

The significant change that was implemented in Equation 3 is the addition of the third term which subtracts a value from the end result of the formula based on how much worse the current action is from the best known action at that state and on the value β. This value can be a random number and helps to separate the optimal states from the suboptimal states. In this experiment, these models are trained in a complex environment and then compared to determine which algorithm is most effective .

Statement of Purpose:

This experiment is intended to assess the benefits of Robust Stochastic Operators over conventional reinforcement learning algorithms in complex environments.

Methods and Materials:

A. Materials:

The only physical material necessary for this project would be a computer. Ideally, a computer with multiple CPU cores will make the execution of the experiment more efficient because multiple trials can be running at the same time. In terms of software, this experiment uses the scripting language Python, a code editor (which in this case is Atom), a terminal emulator, and the OpenAI Gym (an open-source framework which is available for free on the internet).

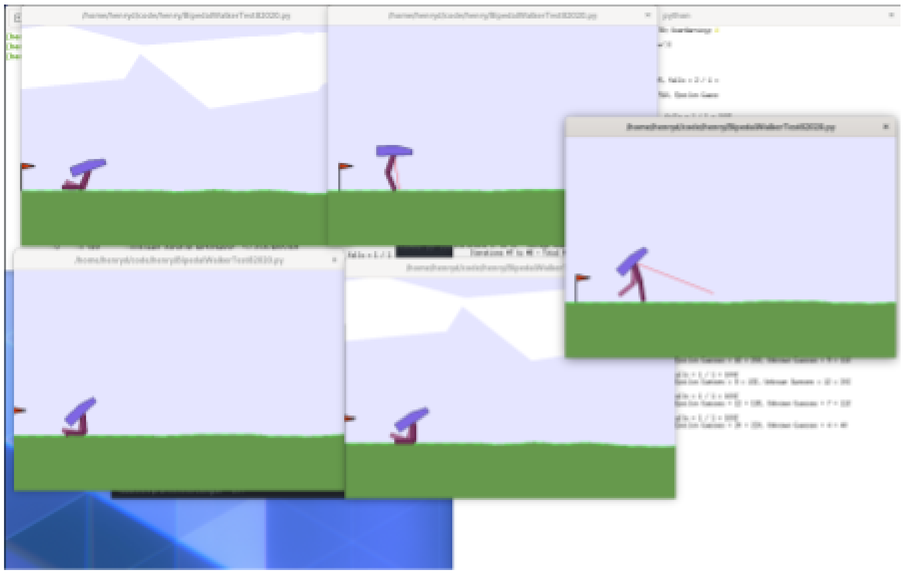


Figure 1: Screenshot of 5 CPU cores running experimental trials at the same time, dramatically reducing experiment time

B. Methods:

In order to fulfill the purpose of this experiment and assess the effectiveness of the various algorithms being studied, there are a number of steps that need to be taken. First, an environment must be identified that is sufficiently sophisticated to prove the algorithm's effectiveness on these complex problems, while also being simple enough to make q-learning a viable solution strategy. In this context, an environment can be judged on its "sophistication" by the number of dimensions in its state and action vectors and the range of values that each of those dimensions contain. These values are important because the relationship between memory usage and the number of state/action dimensions is exponential.

Once a suitable environment has been found, an existing piece of open source code must be found that uses a much simpler OpenAI Gym environment. Although not essential, this reduces the amount of work for the experimenter since the API between the program and OpenAI Gym is already established. It is also important that this code uses solely Q-learning and no neural networks or more sophisticated machine learning strategies since the algorithms being studied are meant to work with Q-tables.

This program then must be adapted for the OpenAI Gym environment selected at the beginning. This adaptation involves numerous changes and additions to the contents of the code, but the overall structure and interface with OpenAI Gym remains roughly the same. By the end, the program is then able to run with the new environment and should demonstrate some learning. The specific alterations and features of the code used in this experiment is described in greater detail in the “Code Overview” section.

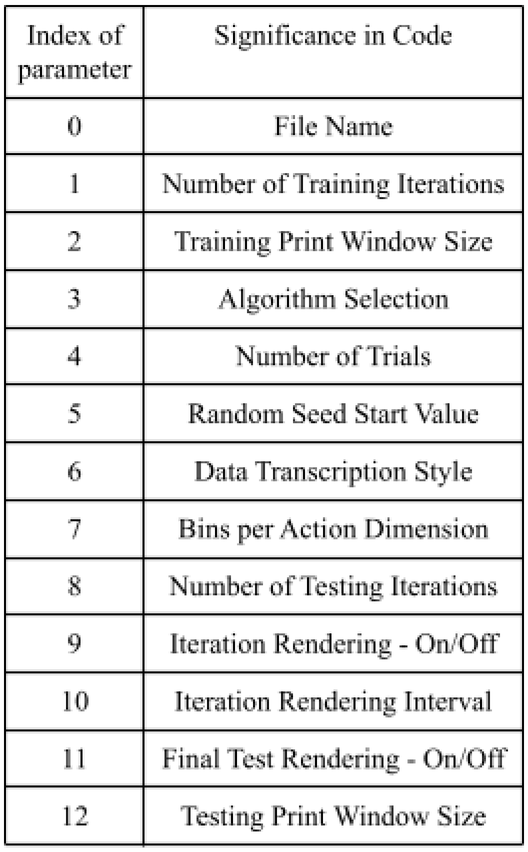
After making the significant changes necessary to change the environment with which the program interacts, the code then needs to be refined to best suit the learning needs of that environment. This primarily comes in the form of changing various constants that exist throughout the code that control how the reinforcement learning progresses (the exact constants used in the final experiment are seen in Table 2). Additionally, certain memory-intensive portions of the code can be adapted to reduce the usage of memory. These strategies for conserving memory are also described in the “Code Overview” section.

Finally, once the program has been finalized, 10 trials of the experiment should be conducted. Each trial would involve selecting a random “seed” value and running each of the algorithms through the program with that value. This “seed” value is used to ensure that the sequence of random numbers remains constant between experiments and that all of the algorithms are experiencing the same conditions within the environment. With all the data recorded, data analysis simply involves comparing the averages of the reward values obtained for the different algorithms. Comparing the iteration lengths of the final training iterations would also serve as a useful metric for comparing the different algorithms.

C. Code Overview

This section provides a cursory overview of how the code used in this project is able to both train and assess the various algorithms. This program was written in Python and designed to be highly adaptive through the use of multiple parameters that could specify the exact configuration of each test. These parameters can be seen in Table 1 and were used so that the program could be easily modified and would be more understandable for a potential user. Another core feature of the program is its focus on minimizing the amount of memory used by the program while handling the massive amount of data available in these complex environments.

The two main strategies used to mitigate this data usage were discretization and sparse matrices. First, discretization is a means of dividing ranges of data into bins, or ranges, that describe



Index of parameter	Significance in Code
0	File Name
1	Number of Training Iterations
2	Training Print Window Size
3	Algorithm Selection
4	Number of Trials
5	Random Seed Start Value
6	Data Transcription Style
7	Bins per Action Dimension
8	Number of Testing Iterations
9	Iteration Rendering - On/Off
10	Iteration Rendering Interval
11	Final Test Rendering - On/Off
12	Testing Print Window Size

Table 1: Code Parameters

the general area in which a data point exists.[4] This is vital for Q-learning in complex environment because many of the state and action dimensions are infinite, so essentially every state would be newly encountered and no learning would occur. On the other hand, if discretization is too aggressive and there are very few bins, the model would be unable to differentiate between potentially very different states. This balance between over and under-discretizing the state and action vectors was a vital consideration when creating the project and the values used for the final tests can be seen in Table 2.

The other means of improving data efficiency was the utilization of a sparse matrix to store the Q-table. This is a data structure in which values are only stored for states that have been visited and nothing is stored for unvisited states. This greatly improves memory efficiency because many states are difficult or impossible to reach despite being within the domain of the environment.

The basic structure of this code is centered around the idea of an iteration, in which the code runs through an entire simulation of the environment until either time runs out or the environment ends in some other way. Within each of these iterations, there are a number of steps which represent an instant of time in the environment. During each of these steps, the state is discretized, an action is selected, and the Q-table is updated accordingly.

The discretization of the state occurs in the aforementioned fashion, in which every component of the state vector is placed into a "bin" and the indices of those bins are used to create one scalar that represents the state. The program then selects the next action by either selecting a random action or referencing the Q-table at that state and selecting an action based on the model's previous experiences. Finally, it updates the Q-table by discretizing that selected state and passing the obtained reward, selected action, and current state into the method corresponding to the algorithm being tested. The code for these algorithms can be found in Appendix A.

At the end of every iteration of the training phase, the program records the total reward calculated from the sum of all individual step reward values. It also records how many steps the agent lasted without falling over. At the end of a certain number of iterations (Parameter 2), the program logs the average total reward and average step count in an output file. Following the training phase, the program then records the model in a separate data file and runs through more iterations with the program making what it believes is the optimal decision at each step. The average reward values for this final portion provides the researcher with a true metric of how effective the algorithm performed on the trial. These values are all logged in the output data file as well which is saved for future reference.

Results:

A. Environment Overview:

In order to test the different algorithms (Bellman, Consistent Bellman, and 3 variations of the Robust Stochastic Operators), all algorithms were applied to the "BipedalWalker-v3" environment, which is a part of the Box2d portion of OpenAI Gym. Every environment in the framework has a specific reward system, meaning that it calculates how well the model is doing in different ways. For example, the

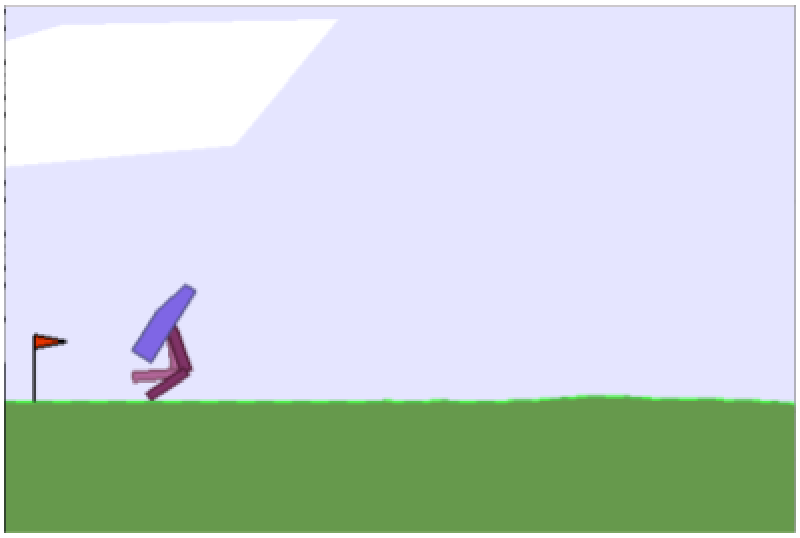


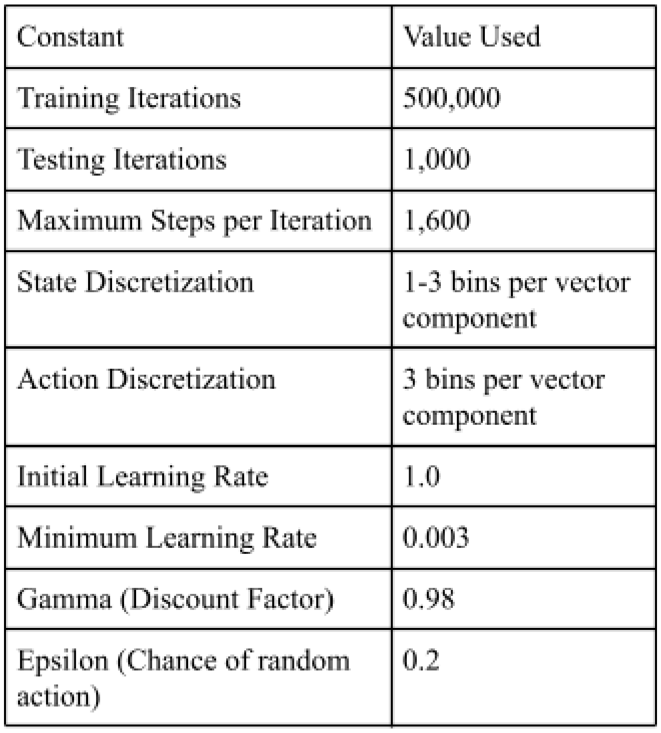
Figure 2: Image of agent in BipedalWalker-v3 environment

“MountainCar” environment gives you a reward based on how close you are to the top of the mountains (reaching the top of the mountain is the goal of the activity). In the “BipedalWalker-v3” environment, three factors play a role in the determination of the reward.

First, the distance that the model is able to travel in the game is a central element of the reward because this is the main objective of the environment. For this metric, the reward ranges from 0 to 300 based on the distance traveled. Next, the environment discourages falling with a 100 point reduction when the agent falls over. (“Falling over” in this environment is defined as any time when the “upper body” of the virtual walker makes contact with the floor). Finally, the environment encourages efficiency by taking away a small number of points for every movement.

This environment fits the requirements for this experiment because it has a 4 dimensional action space and a 24 dimensional state space making it considerably more complex than the simplest OpenAI Gym environments. Many of these dimensions can be somewhat overlooked, however, because they are meant to be used for harder versions of the environment with obstacles and dramatic terrain shifts. Because of this, it is an ideal choice for the experiment.

In order to adapt the program to work for this new environment, the discretization of the vectors and how those discretized vectors are input into the Q-table has to be changed. The constants that exist throughout the program have to be changed as well for this new environment and their final values can be seen in Table 2.



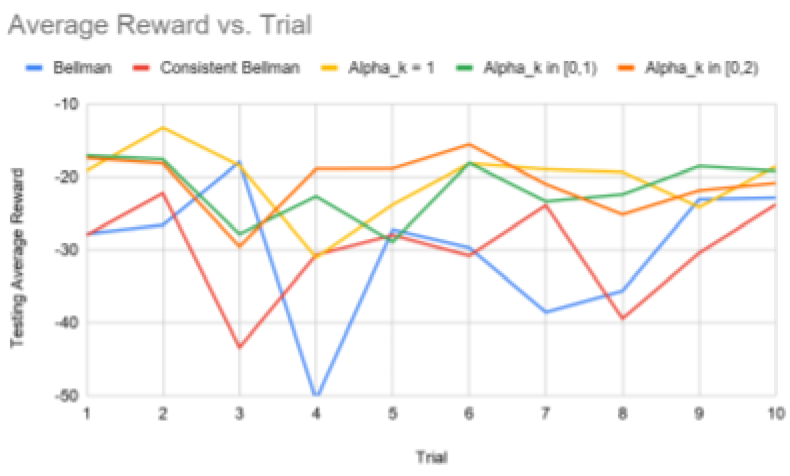
Constant	Value Used
Training Iterations	500,000
Testing Iterations	1,000
Maximum Steps per Iteration	1,600
State Discretization	1-3 bins per vector component
Action Discretization	3 bins per vector component
Initial Learning Rate	1.0
Minimum Learning Rate	0.003
Gamma (Discount Factor)	0.98
Epsilon (Chance of random action)	0.2

Table 2: Constants used in final testing

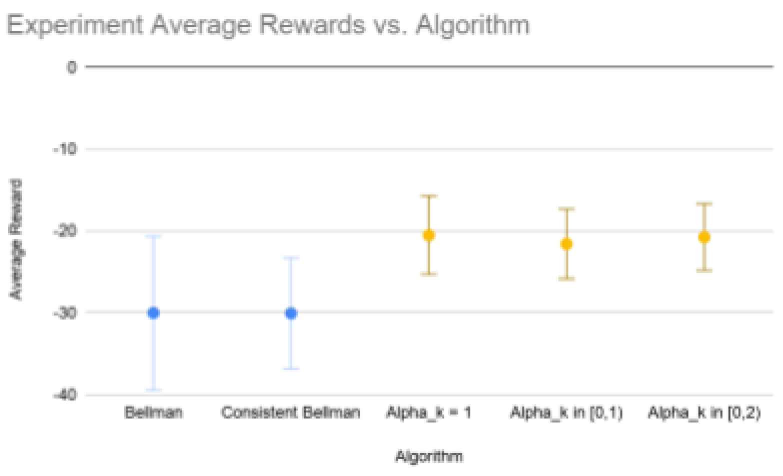
B. Experimental Results:

Ten trials were conducted for each of the five algorithms being studied in this experiment. As described previously, each of these trials involved training the model with a different random seed number for each trial (in this case, the seed number was calculated by $10532 + n$ for trial number $n > 0$) and then testing the model on 1000 new iterations. The average reward is the main metric by which the performance of the algorithm can be measured since the entire objective of reinforcement learning is to maximize the reward value.

The average reward values obtained by the various trials can be seen in Figure 3 below which depicts how the various algorithms compare in terms of the final average reward. The RSOs performed considerably better than the existing Bellman and Consistent Bellman counterparts. This can be seen in the fact that the three variations of RSOs tested obtained average reward values of -20.5, -21.6, and -20.7 which are considerably higher than the average rewards of the Bellman and Consistent Bellman which obtained values of -30.0 and -30.1 respectively. Since the goal of this environment is to attain a maximum reward, this demonstrates that the RSOs performed better than the other algorithms.



(a) Trial Rewards

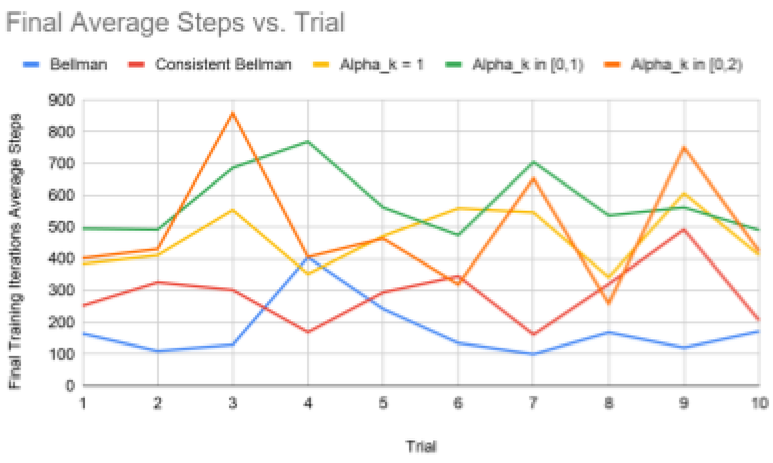


(b) Trial Averages

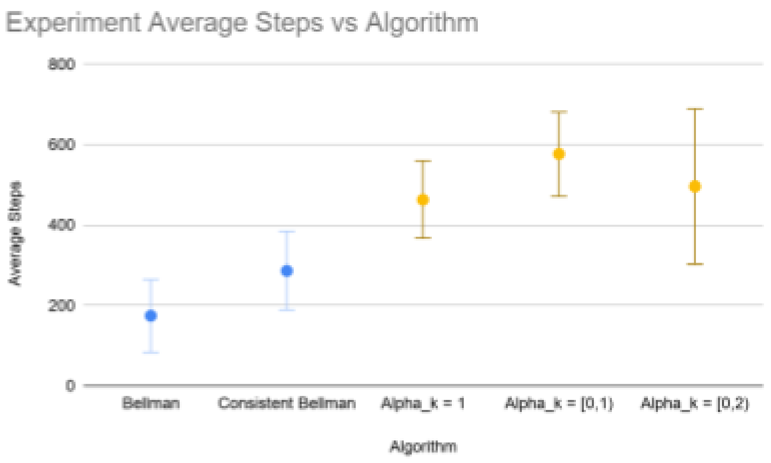
Figure 3: Both (a) and (b) display the average reward obtained by the model after 1000 testing iterations. (a) displays this final result for every trial with each algorithm represented in a different color. (b) shows the average rewards and standard devitation for each algorithm with the existing algorithms in blue and the novel RSOs in orange.

Furthermore, the sample standard deviations of the different algorithms reflect that the RSOs were more consistent and precise as well. While the reward data for the Bellman and Consistent Bellman have standard deviations of 9.41 and 6.78 respectively, the RSOs show standard deviations of 4.80, 4.24, and 4.09. These standard deviations can be seen in Figure 3b as well. This shows that the RSOs are more reliable and consistent than the other studied algorithms.

Similar trends are reflected when studying the average steps completed during the final 100 iterations of training. Having a greater number of steps is a more visual way of measuring the success of a model because it signifies how long the agent is able to remain upright without falling. This data can be seen in Figure 4 which depicts both the raw trial data (Figure 4a) and the trial averages and their standard deviations (Figure 4b). By this metric, the RSOs continue to demonstrate improved performance because they produced much higher average step counts than the other algorithms being studied. The RSOs had average step counts of 463.6, 577.0, and 496.6, which far outperformed the Bellman and Consistent Bellman algorithms which had



(a) Average Steps



(b) Trial Averages

Figure 4: (a) and (b) both display representations of the average number of steps taken by the model during the last 100 iterations of training. (a) shows the exact number of steps for each trial and algorithm while (b) shows the averages and standard deviation for the same data. In (b), blue shows the results of the pre-existing algorithms and orange shows the results of the novel RSO algorithm. A higher number of steps reflects a model with a better ability to keep itself upright without falling over.

average step counts of 174.3 and 286.7 respectively. This information confirms the findings of Figure 3 by showing that the improved reward value successfully led to a more effective model.

Despite this clear improvement of average steps, the RSOs did have slightly larger sample standard deviations within the data for average step count than the Bellman and Consistent Bellman algorithms, which could reflect that they may have been less consistent in this regard. This is not necessarily a problem, however, because it could indicate that the model occasionally exchanged some steps for a greater distance travelled or vice versa, attaining the higher average rewards seen in Figure 3 through either method.

Discussion and Conclusion:

The results described above demonstrate that RSOs performed better than the Bellman or Consistent Bellman algorithms when applied to this more complicated OpenAI Gym environment. Thus, it can be concluded that the benefits of RSOs that have been determined in relatively simple environments hold true for more complex environments as well.

This is impressive considering that none of the algorithms performed exceedingly well due to the fact that the volume of data involved was greater than is traditionally used in reinforcement learning problems where the solution must solely use Q-learning. Alterations, such as utilizing neural networks, would significantly improve the performance of the models, yet the algorithms could not be studied as independently as they are here. This explains why none of the algorithms were able to attain a positive average reward and the average number of steps was not close to the 1600 step maximum for each iteration. Despite this, the relative results between the algorithms still hold significance and reflect the RSOs heightened ability to distinguish between the optimal and suboptimal actions using the equation seen in Equation 3.

This confirmation that the benefits of the RSOs do not change in more complex environments could have implications in many technologies that involve reinforcement learning on a complex level. One of the most important and relevant applications would be in autonomous vehicles. Reinforcement learning is ideal for this type of application because the car has the capability to have a clear action, state, and reward system. For instance, a car could reward actions that improve safety and speed, while its state is determined by the variety of sensors located around the vehicle. The complexity of this system would require the implementation of other strategies as well, but if a Q-table is a part of the system, these RSOs could play an important role because of their enhanced performance.

References:

[1] Wu, Chai W, Squillante, Mark S, & Lu, Yingdong. “A Family of Robust Stochastic Operators for Reinforcement Learning.” *Arxiv.org*, 28 May 2019, arxiv.org/pdf/1805.08122.pdf.

[2] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Tang, J., & Zaremba, W. (2016, June 5). OpenAI Gym. Retrieved from https://arxiv.org/pdf/1606.01540.pdf.

[3] Kaelbling, L. P., Littman, M. L., & Moore, A. W. (1996). Reinforcement Learning: A Survey. Retrieved from https://www.cs.cmu.edu/~tom/10701_sp11/slides/Kaelbling.pdf.

[4] Liu, H., Hussain, F., Tan, C. L., & Dash, M. (n.d.). Discretization: An Enabling Technique. Retrieved from http://www.public.asu.edu/~huanliu/papers/dmkd02.pdf.

[5] Shyalika, C. (2019, November 16). A Beginners Guide to Q-Learning. Retrieved October 03, 2020, from https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c

[6] Bellemare, M. G., Ostrovski, G., Guez, A., Thomas, P. S., &amp; Munos, R. (n.d.). Increasing the Action Gap: New Operators for Reinforcement Learning. Arxiv.org, 15 Dec 2015, https://arxiv.org/pdf/1512.04860.pdf

[7] Rahman, M., Rashid, H., & Hossain, M. M. (2018, December 21). Implementation of Q learning and deep Q network for controlling a self balancing robot model. Retrieved from www.ncbi.nlm.nih.gov/pmc/articles/PMC6302870/pdf/40638_2018_Article_91.pdf.

[8] Bean, R. (2018, September 17). The State of Machine Learning in Business Today. Retrieved from https://www.forbes.com/sites/ciocentral/2018/09/17/the-state-of-machine-learning-in-business-today/#721b380c3b1d.

Appendix A:

```
def bellman(state1,action1,state2):
    q2 = getQTable(state2,q_table)
    return np.max(q2)


def consistent(state1,action1,state2):
    if state1 != state2:
        return bellman(state1, action1, state2)
    else:
        q2 = getQTable(state1, q_table)
        return q2[action1]


def learn_new(state1,action1,state2):
    q1 = getQTable(state1, q_table)
    q2 = getQTable(state2, q_table)
    maxq1 = np.max(q1)
    maxq2 = np.max(q2)
    alpha_k = 1.0 * random.random() / gamma
    return maxq2 - alpha_k * (maxq1 - q1[action1])
```

```
def learn_new_2(state1,action1,state2):
    q1 = getQTable(state1, q_table)
    q2 = getQTable(state2, q_table)
    maxq1 = np.max(q1)
    maxq2 = np.max(q2)
    alpha_k = 2.0 * random.random() / gamma
    return maxq2 - alpha_k * (maxq1 - q1[action1])


def learn_new_fixed(state1,action1,state2):
    q1 = getQTable(state1, q_table)
    q2 = getQTable(state2, q_table)
    maxq1 = np.max(q1)
    maxq2 = np.max(q2)
    alpha_k = 1.0 / gamma
    return maxq2 - alpha_k * (maxq1 - q1[action1])
```