Intelligent Aim Training: Reinforcement Learning Applied

Sriram Boddeda (5033) Mark Cunningham (4033)

Abstract

This paper is concerned with finding the optimal algorithms and environment parameters to spawn targets for a user playing an aim training game, such that an intelligent reinforcement learning agent minimizes the player's score. The idea behind this approach is to expose the player to their weaknesses more often than a random spawning system, resulting in higher human accuracy over many episodes. The outcomes of the paper are planned to be used in BeatShot, a game developed in Unreal Engine releasing on Steam this summer.

1. Domain

We have two separate environments for this project. The first is a custom Gymnasium environment that uses a static two-dimensional array representing the accuracy for all locations in a grid for an example player. The observation space includes the current target spawn position and the previous target spawn position, which are coordinates in a 2-dimensional grid. Likewise, the action space is any point in the 2-dimensional grid. The example player has near-perfect accuracy in the middle of the grid, slowly falling off toward the edges. The lower right section of the grid contains the lowest values, including one point of 0% accuracy. The chance the agent receives a reward is inversely proportional to the player's accuracy at that location. For example, if the player has 30% accuracy at that location, the agent has a 70% chance to receive a reward of 1 and otherwise 0.

The second environment uses BeatShot in tandem with Python running in Visual Studio Code. The implementation isn't ideal since VSCode must be open and executing while playing the game, but nonetheless, the two are able to communicate. When the Python program is executed, it listens for changes in a file that only the game can write to. Conversely, the game reads spawn locations from a file that only the Python program can write to. When a target has either timed out or been destroyed by the user in the game, the location of the target and the results of the player hitting or missing are added to a queue. This queue is popped every time the game spawns a new target. Since the game can have multiple targets up at any point in time, the queue is used to avoid a situation where a target timed out at the same time a target has been destroyed by the player, which could cause some timing issues for the Python program.

2. Hypotheses

1. Varying the learning rate while keeping γ and ϵ constant across episodes will allow our algorithms to converge.

2. The Q-Learning algorithm will perform better when compared to a random spawning system inside the Gym environment.

3. The Sarsa algorithm will perform better when compared to a random spawning system inside the Gym environment.

4. Both the Q-Learning and Sarsa algorithms will converge faster with lower values of ϵ.

3. Learning Methods

Our experiments use temporal difference algorithms because we believe it to be the best fit for a dynamic environment like ours. The algorithms are able to respond to changes in the player's accuracy after every target unlike Monte-Carlo, which would only provide updates after each episode.

3.1. Sarsa

Mark implemented the Sarsa algorithm by creating a Q-Table in Python, which is a matrix with shape (Width, Height, Width, Height). The matrix can best be described with an example: Q-Table[1,2] is a matrix with size $Width * Height$, where each element is the Q-Value for previously spawning a target at location (1,2) and spawning the next target at that new position. We update our Q-table by receiving the observation state and reward from the environment after every time step. For any given state, our Q-function is represented by Equation 1. For all equations, P_1 is the previous position and P_2 is the current position.

$$Q(P_1, P_2) = Q(P_1, P_2) + \alpha[R + \gamma Q(P_1', P_2') - Q(P_1, P_2)] \quad (1)$$

3.2. Q-Learning

Sriram chose to implement Q-Learning. Since it's very similar to Sarsa, we just had to change the TD-target part of the Sarsa Q-Value equation. Equation 2 shows the Q-function we used to update our Q-Table. We were able to easily query the Q-table to find the max Q-value of any position by finding the argmax of the first two indices of the Q-table, which also gave us our next action (location) to take.

$$Q(P_1, P_2) = Q(P_1, P_2) + \alpha[R + \gamma max_a Q(P_1', a') - Q(P_1, P_2)] \quad (2)$$

4. Experiments and Results

4.1. Gym Environment

In order to compare the algorithms, we found that linearly increasing the learning rate, α over the course of the episodes while keeping ϵ and γ constant allowed our algorithms to converge. Having a high learning rate towards the beginning allowed both algorithms to place a high value on exploration, while slowly decreasing it meant that they could reap the rewards from having explored. The rewards per episode across 10,000 episodes for completely random spawning, Sarsa, and Q-Learning are shown in Figure 1. Both Sarsa and Q-Learning were able to effectively learn how to spawn targets with a strategy to lower the player's score, which increased the rewards they received as the episodes progressed.

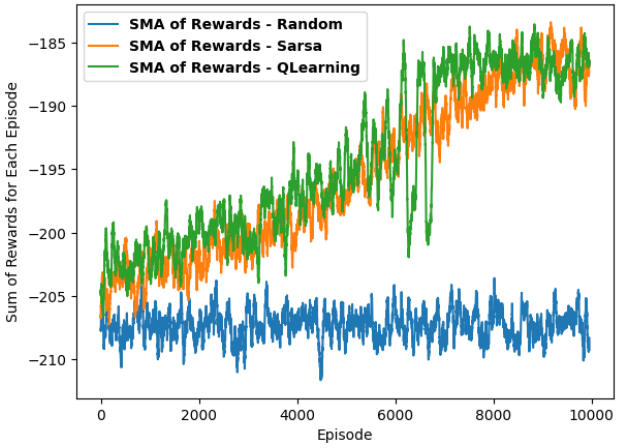

Figure 1. Total Reward Per Episode for all 3 Gym experiments at the end of 10,000 episodes of length 300, $\epsilon = 0.9$, $\gamma = 0.9$.

The average accuracy value of the static 2-dimensional array is 69%, so it makes sense that the average reward that the random spawning system received was 69% of an episode length. It looks like Sarsa and Q-Learning didn't actually perform much better than the completely random spawning system, but this is because we didn't change our policy throughout our experiments thus far. When we save our Q-Tables from our experiments and change ϵ to zero, both algorithms only choose location (8,3), which would make for a boring aim trainer.

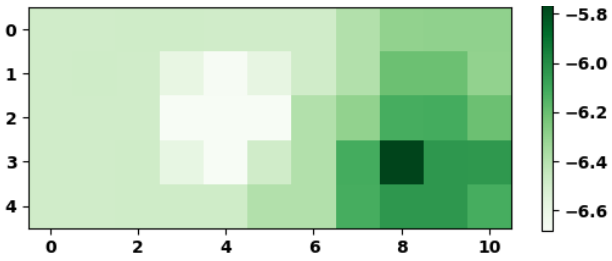

Figure 2. The average Q-values per episode for Sarsa

The heat-map shown in Figure 2 provides a visualization for Sarsa's Q-Table. Since both algorithms converge to the same reward value, Sarsa and Q-Learning have identical heat-maps, with the only difference being the absolute magnitude, while their relative magnitude from position to position is the same. Unsurprisingly, the rewards were highest where the player accuracy was lowest. For example, location (8,3) had the single highest Q-Value in both tables because it was the only accuracy value that was zero.

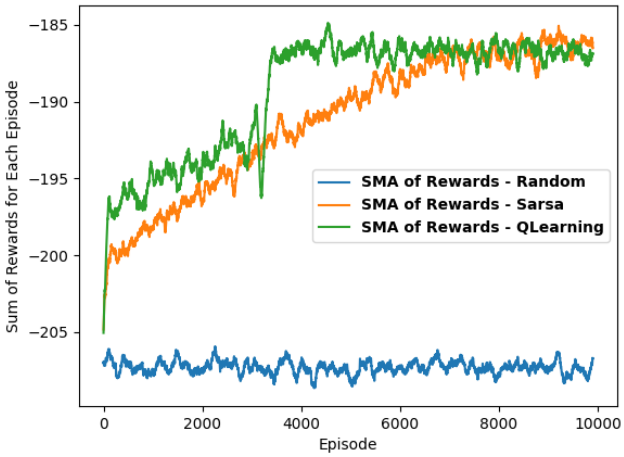

Figure 3. Total Reward Per Episode using non-binary rewards. 10,000 episodes of length 300, while linearly decreasing α, $\epsilon = 0.9$, $\gamma = 0.9$.

When performing our experiments in the Gym environment, we were initially only providing binary rewards of 0 and 1 since the act of destroying a target is most often a binary action. We realized that both algorithms would converge towards the static accuracy values provided to the environment. Figure 3 shows the results for providing the actual float accuracy values as rewards instead of using binary rewards. The algorithms not only learned quicker but also had less variance due to the random nature of binary rewards.

Q-Learning appears to outperform Sarsa in all of our ex-

periments. This is likely a result of Q-Learning being an off-policy learning algorithm that uses the maximum Q-Value instead of the Q-Value at the next state like Sarsa. Both algorithms converge at around -187. when $\epsilon = 0.9$.

Earlier we hypothesized that lower the value of ϵ would decrease the time needed for both algorithms to converge, but we were incorrect, as Figure 4 shows. Decreasing ϵ from 0.9 to 0.5 changed the level at which the algorithms converge, from -187 up to -100. The rate at which they increased their reward was accelerated, but their slopes were almost identical when compared to Figure 3.

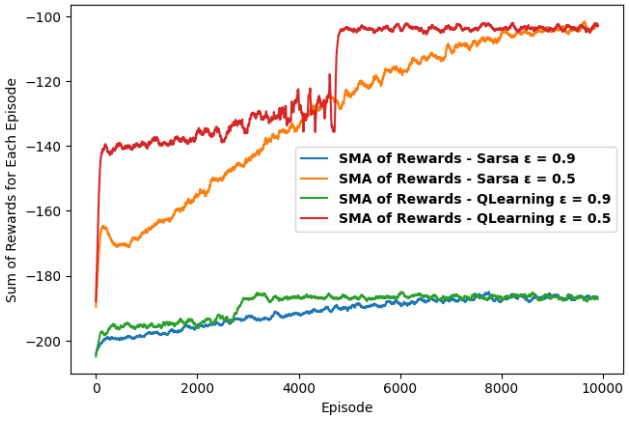

Figure 4. Total Reward Per Episode using non-binary rewards for Sarsa and Q-Learning. 10,000 episodes of length 300, while linearly decreasing α, $\gamma = 0.9$.

4.2. Unreal Engine Environment

To preface this section, accumulating enough data to achieve statistical significance is not within the realm of possibility for a two-person group project, but rather we aim to show that our Python experiments were able to be translated into a tangible real-world implementation. Mark recorded a video of himself playing BeatShot with the Python program running alongside it available here on YouTube, using Sarsa as the learning algorithm. The top right corner is the console of VSCode, which prints out what Python read from the game and what Python is writing to the game. A few games were played before the video so that the algorithms had some starting data to work with. Figure 5 is the resulting Q-Table that was saved to file after the completion of the video. The map has much less contrast when compared to Figure 2 because the algorithms did not have much data to work with. The game doesn't always immediately spawn what Python tells it to, since Python will often want to spawn a target where one is already active. Instead, the game adds the location to a queue and will spawn it the next time the location becomes available.

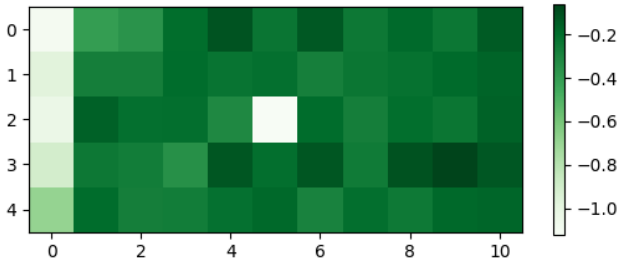

Figure 5. The average Q-values after 1200 time-steps across four episodes in the Unreal Engine Environment.

5. Related Work / Literature Review

5.1. Convergence

Since we didn't prove any convergence at the time of our Project Checkpoint, we looked for papers which could help us understand how to achieve it. Since Sarsa is an on-policy algorithm, the learning rate needs to be changed over time since the algorithm must have the freedom to exploit the observed actions towards the limit (Singh et al., 2000).

5.2. Motor Learning In Video Games

The effects of video games on long-term motor learning have been an interest for researchers since their inception. In this study, data obtained from online games is examined to uncover the truth about how people acquire and retain complicated movements over a long time period. The researchers investigated how players enhance and maintain their skills, as well as how these abilities change over time. According to the study, people who played the game for an extended period gained more motor skills than those who played for a shorter duration (Listman JB, 2015). Additionally, the most skilled players continued to improve their performance over time, which suggests that it is possible to enhance one's abilities in games over a prolonged duration. This research has significance since it demonstrates that games can be utilized to instruct and enhance movements and that prolonged practice can lead to long-lasting improvements in motor performance. These results could prove to be useful for our program, which learns by playing against a human player over time.

5.3. Q-Learning Applications

When estimating the true action values using Q-Learning in stochastic environments, (Jang et al., 2019) discusses the various issues that may arise and their solutions. A bias in the estimation error can cause a gradual increase in the approximation error, leading to significant overestimations. To address this, researchers have developed the Weighted Estimator method to reduce variance and process multiple variables randomly. Another problem is the potential impair-

ment of the reward function, which can occur due to bugs or improper modifications by the agent. The "Corrupt Reward MDP" approach solves this issue by extending MDP with a "Corrupt Reward Runction" and defining formalities and measurement methods to address corruption in various agents. Reward shaping is another novel approach that is designed to address the slow progress issue of Q-learning that results from delayed feedback or reward. Reward shaping is a technique that accelerates learning by integrating expertise from the domain into Q-learning. This technique has been implemented in both single-agent and multi-agent systems.

5.4. Learning to be a Bot

Reinforcement learning algorithms have been used to train artificial agents to play against human players in first-person shooter (FPS) video games (Chaplot & Lample, 2017). The primary objective of the bot is to effectively navigate the gaming environment and make optimal decisions regarding movement and shooting. The effectiveness of the bots was evaluated by the authors using various algorithms such as Q-Learning, SARSA, and Deep Q-Networks, through the use of a well-known video game. The bots were tested against human players and compared to hard-coded bots developed by experienced game developers. The results indicated that the bots were able to learn and enhance their gameplay over time, with some of them successfully defeating human players. In addition, the paper discusses some of the challenges associated with obtaining an efficient reward function due to the large datasets required for training the agents. Our 11x5 grid took thousands of episodes to converge, so we can relate to these challenges.

5.5. Aim Training in Review

Competitive FPS games require players to have accurate aim to defeat their opponents. Hence, the rise of FPS aim training games, where the primary goal of the game is to practice aiming in a closed environment. The most popular game in this genre is Aim Lab. Researchers analyzed the effect of training in Aim Lab on performance in the FPS shooter known as Valorant and found that it led to an increase in players' damage per round (Roldan & Prasetyo, 2021). Precision and speed-themed tasks were attributed to the increase. Could the results be even greater with the help of machine learning?

6. Summary and Future Work

The results of our combined effort on this project will be used to implement an intelligent aim trainer in BeatShot. Since Unreal Engine uses C++, the algorithms will unfortunately need to be rewritten. Mark does not look forward to implementing four-dimensional arrays without Numpy arrays. Alternatively, perhaps there is a way to create a standalone Python program that can be packaged with the game and somehow executed in C++, however, we could not find any promising evidence of this. Since BeatShot already stores player accuracy data, the Python program could be used to generate notable spawn locations for the game even if Python isn't executing in real-time, similar to how we used a static accuracy grid for our Gym environment experiments. The data obtained from our experiments as well as the experience with environments and experimentation will prove useful for implementing practical solutions involving reinforcement learning.

References

Chaplot, D. S. & Lample, G. (2017). Arnold: An autonomous agent to play fps games. *Proceedings of the AAAI Conference on Artificial Intelligence*, *31(1)*.

Jang, B., Kim, M., Harerimana, G. & Kim, J. W. (2019). Q-learning algorithms: A comprehensive classification and applications. *IEEE Access*, *7*, 8.

Listman JB, Tsay JS, K. H. M. W. H. D. (2015). Long-term motor learning in the "wild" with high volume video game data.

Roldan, C. J. & Prasetyo, Y. (2021). Evaluating the effects of aim lab training on filipino valorant players' shooting accuracy (pp. 465–470).

Singh, S., Jaakkola, T., Littman, M. & Szepesvári, C. (2000). Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, *38*, 287–308.