Real-World Projectile Catching with Reinforcement Learning: Empirical Analysis using Discretized Simulations

Bryon Kucharski, Adam Ziel, Michael Hickey, Collin Travers
Electrical and Computer Engineering Department
Wentworth Institute of Technology
Boston, MA, USA
{kucharskib, ziela, hickeym2, traversc}@wit.edu

***Abstract*—Robotic projectile catching has previously been done with path planning, in which the trajectory is predicted using basic kinematic equations. In addition, reinforcement learning (RL) has proven successful in mastering video games, such as 8-bit Atari 2600 games like PacMan or Breakout. We evaluate the performance of two RL algorithms, Q-Learning and a Deep Q-Network (DQN), applied to a simulation of a novel application of catching a projectile. A continuous physical environment is translated into a discretized, limited state and action space, and then solved using RL.**

***Index Terms*—reinforcement learning, discretization, robotics, simulation**

I. Introduction

From a high-level, reinforcement learning (RL) is a branch of artificial intelligence in which an agent aims to collect sufficient experience in a state-space, and then leverage this experience to determine the optimal action to take for any given state. Therefore, the training iterations required to solve a given environment scale exponentially with the dimensionality of the state space. State-of-the-art RL research has been largely performed in the domain of video games, most famously with DeepMinds generalized, pixel-based Deep Q-Network (DQN), used for solving a wide array of classic Atari 2600 games [1], [2]. Video games appear to be the ideal testing ground for RL algorithms due to their discrete state-spaces, clearly defined reward schemes, lack of noise in determining state, and lack of liability otherwise present in training real-world agents. Progress in physical RL agents, on the other hand, has lacked due to failing to meet these criteria based on their inherent continuous nature. Discretizing continuous spaces has proven to be successful in solving continuous environments with RL [3], [4]. Environment discretization aims to shrink an enormous state space into one in which it is more computationally feasible to gain sufficient experience of the state and action spaces. The core aim of this discretization is to reduce the state and action space, or set of all possible states, and therefore reduce the required training iterations to learn a given task.

Research has been conducted towards robotic projectile catching using a variety of approaches. Kober, Glisson, and Mistry [5] demonstrated a juggling humanoid robot which used explicit kinematic equations to predict the landing locations of the balls. Supervised learning algorithms [6] such as Support Vector Machines and Gaussian process regression have been used to learn the trajectory of a ball. The authors have not found any research on projectile catching using RL. Our research utilizes the discretization methods along with the RL algorithms applied to a new domain of catching a projectile in simulation. The future work section describes the initial steps taken to transition this simulation into the real world.

II. Methodology

Unity, a popular 3D game engine, was used to create a simulated a projectile catching environment. Fig. 1 shows a spherical projectile being launched off a ramp, which is then caught by the cuboid RL agent. The decision was made to roll a ball down a ramp, rather than just throw a ball freely, for several reasons. First, as a means of limiting the state-space, since freely throwing a ball can yield trajectories with many different initial and final positions in 3D space, as well as evoke drastically different types of parabolic paths. These considerations were also shared by the aim to minimize the window of sensing and have sensing mechanisms remain statically placed in the environment with limited fields of view. Lastly, since projectile catching is an extremely time-critical task, a ramp was used to increase the balls total travel time, allotting a longer sensing window, and therefore maximizing the number of full propagations through our signal path, and consequently predictions, per episode. Though the RL agent can move instantaneously in simulation, a physical robot would require nontrivial time to complete actions.

When viewing the environment from a top-down camera view, the four features extracted to compose a state were the ball's x-position and speed down the ramp, and the robot's x and y positions. These features were then discretized uniformly such that a grid was imposed over the environment, and then the state of each feature was the grid space that it occupied. An episode began when a new random spawning location and initial speed were generated, and ends after the ball lands in the

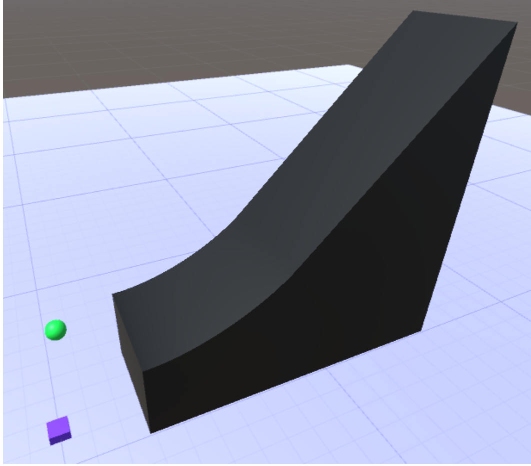

Fig. 1. Simulated environment in Unity.

robots catching area. A catch occurs if the robot's coordinates are the same as the ball 's coordinates at the end of an episode.

A. Simulation

The simulation was used to test different state sizes of both reinforcement learning algorithms. The initial state is created by selecting a random number between 1 and the number of x states for the ball x and robot x features, and a random number between 1 and the number of y states for the robot y and ball velocity features. The agent then performs a simulated number of episodes. Each episode the agent performs number of y states multiplied by the number of x states iterations, which is the max possible grid space away. An episode begins when a new random location is generated, and the episode ends after the total amount of iterations.

III. Reinforcement Learning Algorithms

As previously mentioned, successful attempts to catch a projectile have often used trajectory planning, where the landing position is calculated and sent to the robot. In our research, reinforcement learning has been used to teach the robot the landing position of the ball, rather than calculating it. Two versions of reinforcement learning algorithms were analyzed. Each algorithm had the same state, action, and reward setup. Every iteration, the agent in a current state interacts with the environment by taking an action chosen by the RL algorithm. Upon interaction, the environment returns a new state prime and reward for that action. After enough iterations, agent will converge to the ideal action to take in every state.

A. State-Action Representation

Utilizing the discretization methods, the state of the reinforcement learning algorithms includes a list of four features including the robot’s x grid location, the robot's y grid location, the ball's x grid location, and the ball's speed. The agent has the option to choose from five different actions which consist of move up, down, left, right, or stay

B. Reward Function

The reward is given to the robot after every action is taken and is based on the proximity of the robot’s coordinates to the ball’s coordinates. The distance between the robot and the ball is calculated using (1) at every iteration

$$R(x_r, y_r, x_b, s_b) = \sqrt{(x_r - x_b)^2 + (y_r - s_b)^2} \quad (1)$$

where x_r is the x position of the robot, y_r is the y position of the robot, x_b is the x position of the ball, and s_b is the speed of the ball. If the agent takes an action that decreases this distance, it receives a reward of +1. If an episode is at an end and the robot is in the same grid location as the ball, it is a catch and the agent receives a +10 reward. For all else, the agent receives a -1 reward. The robot speed and the robot y position are discretized on the same scale, meaning a speed of 0 corresponds to grid position 0 for the robot, a speed of 1 corresponds to grid position 1, etc. Since the ball speed is known at the beginning of an episode in simulation, the reward is given to the agent at each iteration as opposed to once per episode. This allows for the agent to receive constant positive or negative reward which will converge faster than a reward once per episode.

C. Q-Learning

The first algorithm is a simple temporal-difference (TD) learning algorithm called Q-Learning [7]. The algorithm calculates a scalar value, called the Q-Value, which is a measure of value of taking action a in state s. The agent uses (2) to calculate the Q-Values for each possible state-action pair during training. s_t is the current state at time t, a_t is the current action taken at time t, s_{t+1} is the new state of the agent, and r_t is the reward for taking the action.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha * (r_t + \gamma max\left(Q(s_{t+1})\right) - Q(s_t, a_t)) \quad (2)$$

The learning rate α was set to 0.5. γ is a hyperparameter which controls how much the agent takes into consideration of the future Q-Value at time s_t and is explained more in section E.

All Q-Values are stored in a Q-Table, with rows representing states and columns representing the Q-Value for every action in that state. When the agent needs to select an action, the Q-Table is referenced to determine which action results in the highest Q-Value in the current state.

D. Deep Q-Network

The second algorithm is a combination of deep learning and reinforcement learning, titled Deep Q-Network (DQN) [1], [2]. DQN uses a neural network as an approximator of the Q-Values instead of referencing a table in Q-Learning. Each iteration during training, the agent collects experiences, or combinations of states, actions, and rewards, and saves them as replay memory. Once enough experiences are collected, the network is trained by selecting random experiences from replay memory. The agent predicts the Q-Value for the current

state, then predicts a target Q-Value for the next state. The weights of the network are updated using (3)

$$\Delta w = \alpha * [(r_t + \gamma * Q(s_{t+1}, a_t, w)) - Q(s_t, a_t, w)] \nabla_w Q(s_t, a_t, w) \quad (3)$$

where w is the weights of the neural network, s_t is the current state at time t, a_t is the current action taken at time t, s_{t+1} is the new state of the agent, and r_t is the reward for taking the action. γ is once again discussed in section E.

The network is optimized by taking the mean squared error between the predicted Q-Values and the target Q-Values.

The network architecture used for the DQN algorithm is derived from [8] with three hidden layers and takes a vector of $< x_r, y_r, x_b, s_b >$ as the input. The output is the Q-Value for every action. For each hidden layer, the number of nodes is set to 20, 18, and 10, respectively. The learning rate is set to 0.001 [8].

E. Common Hyperparameters

The agents choose an action with a decaying ϵ-greedy strategy, ϵ meaning percent of the time the agent picks a random action, and 1-ϵ percent of the time the agent picks the action with the highest Q-Value for the current state. This strategy is to encourage the agent to explore the environment at the beginning of training where not a lot of information is known, then slowly lower the exploration rate as it learns. The value of starts at 1 and decays exponentially by an ϵ-decay rate of 0.995 after every training step until it reaches an ϵ-min of 0.01. The parameters of initial ϵ, ϵ-min, and ϵ-decay were chosen to match the DeepMind Atari research [3]. The discount rate γ is the rate at which the agent considers the predicted future reward when calculating the Q-Value. Based on the way the reward function is setup, the discount rate is set to 0.0 because the agent will always receive a positive reward if it moves closer to the landing location. Thus, the goal for the agent is to always perform actions so that it immediately moves closer to the landing location.

IV. RESULTS

The simulation results were collected by running 40,000 simulated ball rolls at various levels of state-spaces. 5x5 represents 5 x grid spaces and 5 y grid spaces, and so on. Every 100 ball rolls, the average accuracy was calculated as the total number of catches over 100. The agent should have a relatively low accuracy to start, and gradually increase in catch percentage as it trains. Ideally, the agent would converge to a 100 catch percentage. The vertical lines represent the point of convergence, or the point at which the average number of catches plateaus. The number above the vertical line represents roughly how many ball rolls it took to converge. The dashed vertical line is the convergence point for Q-Learning while the dotted vertical line is the convergence point for DQN. Fig. 2 shows the comparison of both algorithms in a small state space, where the initial four features to the simulation can range from 1-5 (top) and 1-10 (bottom). Both

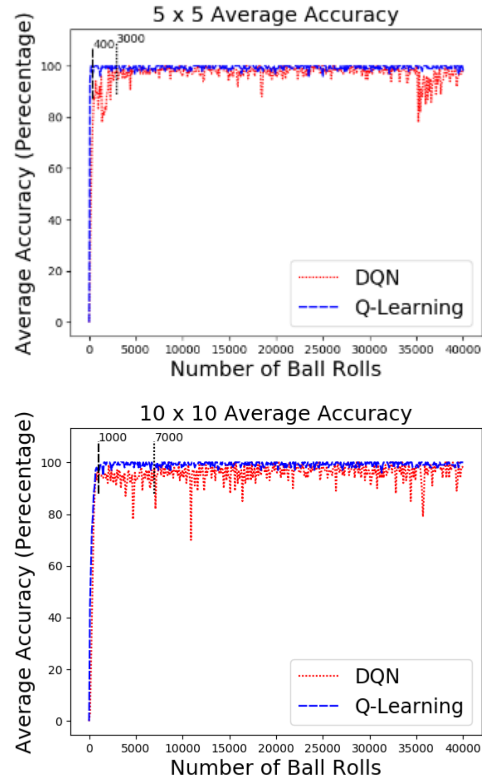

Fig. 2. Average Catches in a 5x5 (above) and 10x10 (below) state-action space. Q-Learning is dashed line and DQN is dotted line. Q-Learning outperforms DQN in these smaller state spaces.

algorithms perform well in both state spaces, but the Q-Learning agent converges more quickly. In the 5x5 state space, Q-Learning convergences after 400 ball rolls, while DQN converges after 3,000 ball rolls. In the 10x10 state space, Q-Learning converges after 1,000 ball rolls and DQN converges after 7,000 ball rolls. Q-Learning converges more quickly in the smaller state spaces because the size of the Q-Table is relatively small which requires fewer ball rolls to determine the optimal actions.

In larger state spaces shown in Fig. 3, the size of the Q-Table expands, requiring more iterations to determine the optimal actions. In each situation both algorithms still converge, but DQN converges more quickly than Q-Learning. In the 15x15 and 20x20 state space, DQN converges as 5,000 iterations, and in the 25x25 state space converges at 7,000 rolls. Q-Learning converges at 6,000, 12,000, and 25,000 ball rolls for 15x15, 20x20, and 25x25 state sizes. Fig. 4 shows an even larger state space, where each of the four input features can range between 1 and 35. The DQN algorithm converges at 15,000 ball rolls while the Q-Learning algorithm does not converge in the 40,000 ball rolls. At an even larger state space, Q-Learning needs even more ball rolls to determine optimal actions.

These results prove that Q-Learning is not as scalable as the DQN algorithm. As the state size increases, the size of the Q-Table increases as well, which requires more simulations to converge. DQN does not need to be trained in as many

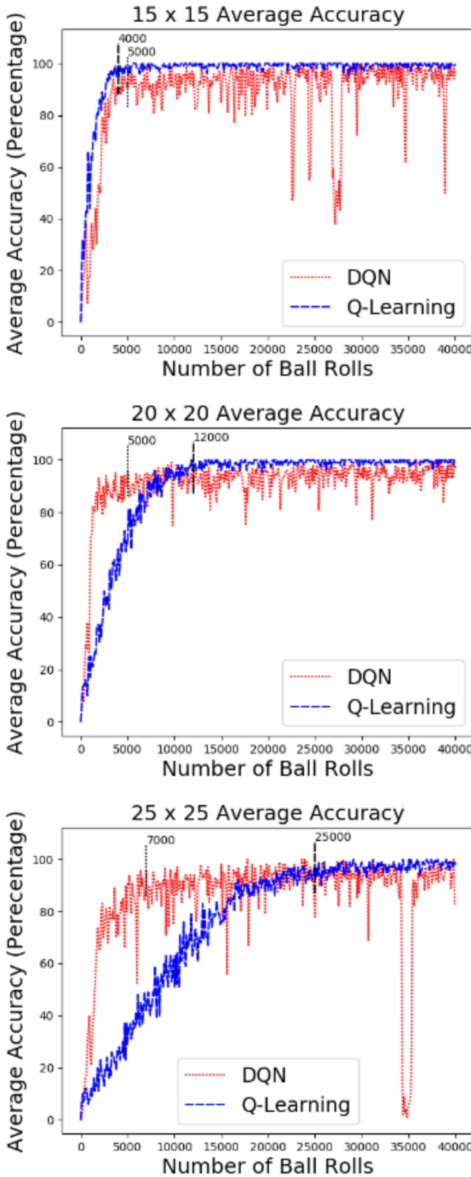

Fig. 3. Average Catches in a 15x15 (top), 20x20 (middle), and 25x25 (bottom) state-action space. Q-Learning is dashed line and DQN is dotted line. DQN begins to outperform Q-Learning.

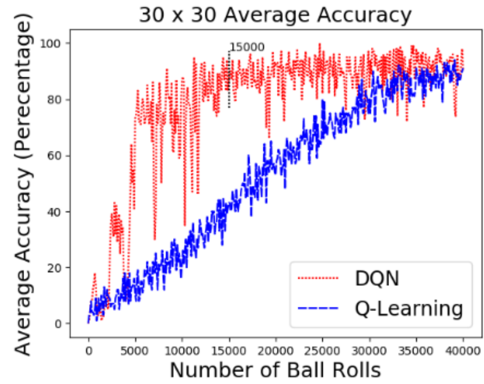

Fig. 4. Average Catches in a 30x30 state-action space. Q-Learning is dashed line and DQN is dotted line. DQN greatly outperforms Q-Learning, while Q-Learning does not converge in the simulated number of ball rolls.

Fig. 5. Real world implementation of the environment. The ramp is made out of plywood and the robot is a 2D gantry robot controlled by an Arduino.

episodes as Q-Learning since a neural network is approximating the Q-Values that guide the actions. In a situation where we are discretizing the environment, the goal is to gradually increase the state and action spaces as close to continuous as possible. The DQN algorithm proves to be a better choice for this situation.

V. Conclusion

Overall, this project demonstrated a successful method to predict the landing location of a projectile using reinforcement learning. We showed that environment discretization can enable simple physical tasks to become solvable with RL. In very limited state spaces, Q-Learning outperforms DQN. However, with an increased number of state spaces, DQN outperforms Q-Learning and converges more quickly.

VI. Future Work

The simulation provides reassurance that an agent could learn the optimal actions to take in order to catch a projectile in a discretized environment. Seen in Fig. 5, initial work has been done to transition the simulated agent into the real world. A camera is used to detect the location of the ball rolling down the ramp, and a photo-gate is used to measure the speed of the ball. The robot's grid position is calculated as a function of it's motor values. The real world environment has not yet been tested thoroughly enough to provide statistically significant results.

Future work may done to improve the baseline RL algorithms as well. The current algorithms predict one action based on one given state. In the future, it may be beneficial to input a sequence of states to the algorithms. Ideally, the vector representation of the state could be replaced with a sequence of images. A list of improvements DQN algorithm [9], [10] have recently been published and may benefit the performance of the algorithm.

VII. Acknowledgements

We would like to thank our advisor, Professor Aaron Carpenter, as well as Professors Memo Ergezer and Nate

Derbinsky for guidance throughout the project.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[3] W. D. Smart and L. Kaelbling, "Practical reinforcement learning in continuous space," in *Proceeding of the Seventeenth International Conference on Machine Leaning (ICML 2000)*, 2000, pp. 903–910.

[4] W. T. Uther and M. M. Veloso, "Tree based discretization for continuous state space reinforcement learning," in *American Association for Artificial Intelligence*, 1998.

[5] J. Kober, M. Glisson, and M. Mistry, "Playing catch and juggling with a humanoid robot," in *Humanoid Robots (Humanoids), 2012 12th IEEE-RAS International Conference on*. IEEE, 2012, pp. 875–881.

[6] R. Lampariello, D. Nguyen-Tuong, C. Castellini, G. Hirzinger, and J. Peters, "Trajectory planning for optimal robot catching in real-time," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 3719–3726.

[7] R. S. Sutton, A. G. Barto *et al.*, *Reinforcement learning: An introduction*. MIT press, 1998.

[8] K. Kim. (2017) Deep q-learning with keras and gym. [Online]. Available: https://keon.io/deep-q-learning/

[9] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[10] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International Conference on Machine Learning*, 2016, pp. 1995–2003.