Bottle Flipping with Reinforcement Learning*

Bryson Gullett and Taylor Rogers
Department of Electrical Engineering and Computer Science
The University of Tennessee, Knoxville
Knoxville, Tennessee 37916 USA
bgullet1@vols.utk.edu and troger28@vols.utk.edu

***Abstract*—This project involves applying reinforcement learning (RL) to the task of flipping a water bottle. A custom environment was developed through the Unity game engine, and the task was iteratively approximated as a Markov Decision Process (MDP). Three RL approaches were used to train a robotic arm simulation to solve the bottle flipping task: Proximal Policy Optimization (PPO), Deep Q-Networks (DQN), and Advantage Actor Critic (A2C). This paper compares the different RL approaches' effectiveness in performing the bottle flipping task. Based on both quantitative and qualitative results, PPO was found to be the most successful.**

***Index Terms*—reinforcement learning, bottle flip, robotic arm simulation**

Fig. 1. KUKA's heuristically programmed bottle flipping robot

I. Introduction

In popular culture, flipping a water bottle has become a notable trend due to its significant social media presence beginning in 2016. A successful "bottle flip" is defined by tossing a partially-filled water bottle such that its cap is pointing downward at least once during its rotation and the final resting state of the water bottle is with the cap pointing upward relative to the surface it lands on. A bottle flip may be considered even more successful by maximizing the number of end-over-end rotations the bottle does or by landing it in interesting locations.

This project applies reinforcement learning (RL) to solve the task of controlling a simulated robotic arm to flip a water bottle as many times as possible and land it upright. Furthermore, the project compares various reinforcement learning methods in order to find out which method can train the best bottle flipping agent in a timely manner. The three reinforcement learning techniques we use are Proximal Policy Optimization (PPO), Deep Q-Networks (DQN), and Advantage Actor Critic (A2C).

Though this project is not the first to try and solve the bottle flipping task with reinforcement learning, it is the first known attempt to solve the problem using a simulation environment. We brought this project to life by creating a custom bottle flipping simulation environment in the Unity game engine with the help of Unity's ML-Agents toolkit.

II. Previous Work

Reinforcement learning has previously been used to train robots to accomplish a variety of tasks. In order to train robots to perform in real-world situations, these robots are often first simulated in software. Robotic software simulation combined with machine learning allows researchers to train models that can either be applied to physical robots or be used to find the quickest techniques of training robots in real life.

Although flipping a water bottle is not a common robotics control task in RL research, it has been approached before. Yang, Young-Ha, et al. explores how different reward functions may be designed to improve a robot's learning of this task. Their work helps inspire ours and highlights a foundational challenge or aspect of this project - framing the task as a Markov Decision Process (MDP) [1].

Reinforcement learning has also been used to train a robotic arm to flip a pancake [2]. Kormushev, Calinon, & Caldwell used Policy learning by Weighting Exploration with the Returns (PoWER), a hybrid RL implementation by Kober et al [5]. This research focused on training motor primitives associated with a physical robot to flip and catch custom-made artificial pancakes. The method, PoWER, is developed specifically for use with physical robotics. Although our project involves a simulated robotic arm, and thus we should not use PoWER itself, this work motivated ours. In order to use this project to expand upon the content covered in

*This work was completed for Reinforcement Learning at The University of Tennessee, Knoxville.

Reinforcement Learning at the University of Tennessee, we were eager to compare methods discussed in class.

Outside of machine learning research, robots have been heuristically programmed to solve bottle flipping successfully in the past as well. Specifically, the KUKA robotics and automation company created a bottle flipping robot (shown in Figure 1) with the goal of using the robot as a platform to develop future streamlined machine learning robotics tools [4]. However, due to the current heuristic nature of the robot's software, programming this bottle flipper involved detailed knowledge regarding modeling the dynamics of both the robot and its environment. Reinforcement learning does not require such model knowledge.

All of these works serve as inspiration and references towards our goal of applying reinforcement learning to flip a water bottle in simulation.

III. Domain Problem

A. Creating the Environment

This task involves flipping a water bottle in a simulated environment. We began the project by creating the bottle flipping environment with the Unity game engine. The small bottle flipping Unity game includes a pivoting arm, a pivoting hand connected to the arm, and a water bottle attached to the hand as the only functional game objects. The resulting bottle flipping game environment is depicted in Figure 2. Unity's physics engine allows the bottle and appendages to move and rotate similar to a real solid object and person's straightened arm, respectively. We used only two joints with the arm having an unlimited range of motion for simplicity. For the same reason, we approximated the water bottle using a rectangular prism of the same base dimensions as a 16.9 fluid ounce water bottle. Using Unity's ML-Agents toolkit [6], which includes a Unity package and Python package, we were then able to wrap the Unity bottle flipping game such that it can be interaction with like an OpenAI gym environment in Python.

B. Iterative MDP Approximation

In order to apply reinforcement learning to this problem and our Unity environment, we had to approximate the problem as a Markov Decision Process. Developing an effective approximation of an MDP was a significant portion of this project. We began by deciding what observations we wanted the arm agent to have that would approximate the states for the problem as an MDP. The initial observations were the current target velocity for each joint motor and the current timestep. The idea behind these observations was to give the machine learning agent what it needed in order to learn a policy that implements the target motor velocity versus time curve that successfully lands a bottle flip with the maximum number of flips. However, even before any training, we decided that we wanted to instead give the agent a better picture of what the current positions and motions of the appendages moving the bottle looked like. We thought that using arm and hand angles and angular velocities would better capture the current state of the system in the framing of the task as an MDP.

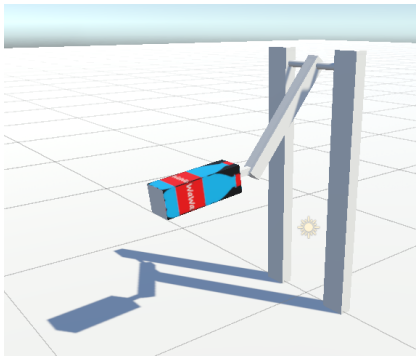

Fig. 2. Water bottle flipping training environment in the Unity game engine

In order to avoid further increasing complexity, we decided against using current target motor velocities as observations. Notice that our observation space only contains continuous values; this is because bottle flipping is a very precise task that would not benefit much from discretizing the observation space for potentially faster training.

We decided to make the action space discrete rather than continuous because this would limit the size of the action space and theoretically reduce training time without losing much precision. Therefore, we decided from the beginning to have three different action branches: change in arm target motor velocity, change in hand target motor velocity, and releasing the bottle. The former two action branches each have three possible values: 0 (do nothing), 1 (increase by one deg/sec), or 2 (decrease by one deg/sec). The releasing the bottle action branch has two possible values: 0 (do not release the bottle) and 1 (release the bottle).

At the beginning of the project, we decided to define the reward structure such that a successful bottle flip earns a large positive reward, a failed bottle flip earns a negative reward, and flipping the bottle more times in a single release grants a more positive reward. The goal of this structure was to cause the agent to learn primarily to flip the bottle and land it upright and, secondarily, to flip the bottle as many times as possible.

While observing initial PPO training, it was clear that the initial reward structure was not effective in framing the task. Instead of the arm learning to flip the bottle as we intended, it first learned to "throw" the bottle as hard as possible and flip it many times without landing it upright. This highlighted that the reward structure did not effectively encourage what we intended, so it was adjusted. Iteratively adjusting the rewards resulted in the following final bottle flipping MDP approximation attributes:

- State/Observation Space:
  - Arm angle and angular velocity
  - Hand angle and angular velocity
- Action Space:
  - Increase (1), decrease (2), or keep constant (0) the current target arm angular velocity by one deg/sec
  - Increase (1), decrease (2), or keep constant (0) the current target hand angular velocity by one deg/sec

  - Keep holding onto (0) or release (1) the bottle
- Reward Structure:
  - Rewards applied at bottle release:

$$R_{release} = \frac{|v_{arm}| + |v_{hand}|}{50} - 1$$

Where:

* v_{arm} is the angular velocity in deg/sec of the arm.
* v_{hand} is the angular velocity in deg/sec of the hand.

  - Rewards applied at the end of the episode:

$$R_{end} = \frac{d}{10} + s$$

Where:

* d is the number of degrees the bottle flips in the XY plane after release.
* s equals +100000 if the bottle is released, reaches a resting position, and is upright after rotating at least 180 degrees.
* s equals -8 if the bottle is released, reaches a resting position, and either does not rotate at least 180 degrees or is not upright.
* s equals -10 if the bottle is not released or does not reach a resting position after 500 timesteps.

IV. Reinforcement Learning Methods

A. Proximal Policy Optimization

To begin this application of reinforcement learning, a baseline algorithm was established. Proximal Policy Optimization (PPO) was used to train the robotic arm simulation first. PPO is a policy gradient method, meaning the policy is being improved or learned directly. Like A2C, PPO is normally implemented using an actor deep neural network and a critic deep neural network. However, unlike A2C, PPO uses clipping methods in its policy objective function. This clipping constrains the size of the policy update and prevents major unstable performance drops from occurring. PPO may be used for problems with both discrete and continuous action spaces. PPO was the initial approach used in this project because it was appropriate for the task, convenient with the libraries used, and extremely popular for similar control problems. Unity's ML-Agents toolkit includes a basic PPO implementation, and the hyperparameters were adjusted as necessary to apply that implementation to this task.

B. Deep Q-Networks

Other approaches and their results were then compared to those with PPO through Python programs. A Deep Q-Networks implementation was developed with Keras-RL, Keras' RL library [7]. Deep Q-Networks is an extension of Q-learning. In contrast with tabular Q-learning, DQN uses a deep neural network to approximate the RL state action value function. The policy is then derived from the approximated value function.

C. Advantage Actor Critic

An Advantage Actor Critic program was also developed to compare its application to the water bottle flipping task. A2C from Stable Baselines3 is usable with multi-discrete action spaces [3]. Considering many other RL libraries' algorithms are not compatible with multi-discrete action spaces like that for this problem, A2C is particularly useful in this project. A2C, which is another policy gradient RL method, includes two branches of deep neural networks: an actor, which estimates actions, and a critic, which estimates the value function to evaluate the actions. Like PPO, A2C also uses the advantage function to reduce its estimation variance. Therefore, the only real difference between PPO and A2C is that PPO utilizes clipping to prevent too significant policy update steps. In fact, researchers have proven that A2C can simply be considered a special case of PPO [8].

V. Code Design

Along with code for the Unity environment (written in C#, specifically BottleFlip/BottleFlipEnv/Assets/ArmAgent.cs), a Python program was developed to run Deep Q-Networks (located at BottleFlip/DQN_with_KerasRL.py). This code utilizes the Keras-RL and Unity's ML-Agents libraries. The code is sequential and does not contain unnecessary functions or classes as it is relatively short. Instead of writing DQN from scratch, Keras-RL reinforcement learning libraries were used to keep the code simple and to practice utilizing different RL and deep learning libraries [7]. Also, Unity's mlagents_envs Python package was used to wrap the custom Unity bottle flipping game as an OpenAI gym-style reinforcement learning environment [6].

Another Python program was developed to run Advantage Actor Critic (located at BottleFlip/A2C_with_Stablebaselines3.py) using Stable Baselines 3 instead of Keras-RL as our choice of reinforcement learning library [3]. As in the DQN code, the custom environment was wrapped as an OpenAI-style gym using Unity's mlagents_envs Python package to ensure it worked as intended with the RL library. No additional functions or classes were written.

Note that PPO training is already integrated into Unity's ML-Agents toolkit [6]; therefore, there was no need to write any code for the PPO implementation. Instead, we tuned PPO hyperparameters using a configuration file (located at BottleFlip/bottleflip_config_ppo.yaml).

VI. Results Analysis

For each bottle flip training run, we plotted the average episode length in timesteps as a function of the total timesteps trained. This data was displayed in TensorBoard to compare the training results between PPO and A2C. Along with these quantitative results, we more qualitatively analyzed the training progress. While the agent was learning with each algorithm implemented, we monitored the bottle flipping game in the Unity Editor to view the agent's activity. Results for each method used in this project are discussed below.

A. PPO Results

We trained an agent using PPO for approximately 32 million timesteps or about 75 hours. The PPO average return per episode versus total timesteps trained graph is depicted in Figure 3. Initially, the average return was negligible. This is justified by the agent exploring without yet flipping the bottle much or successfully landing a bottle flip upright and receiving the large reward. After a few hundred thousand timesteps though, the agent completed its first successful bottle flip. Upon discovering the massive reward from completing a successful flip, the agent gradually started to successfully complete more and more bottle flips. This progress is clearly shown through a rapid increase in average return per episode from around timestep 500,000 to 2.8 million. Although the average return per episode appeared to be approaching a plateau after 15 million timesteps, it spiked as the agent somewhat suddenly perfected its technique. The final average return per episode of about 99,000 means that the agent learned to successfully complete a bottle flip on approximately 99% of episodes, better than the average human's performance. While adjusting the reward structure and gaining familiarity with the environment, both team members manually controlled the agent. In 100-200 attempts of flipping the bottle, we successfully landed a bottle flip roughly 5-15 times.

The main reason that the final trained agent cannot flip the bottle successfully 100% of its attempts is because PPO learns a stochastic policy function, meaning that the probability of taking a non-optimal action (e.g. releasing the bottle early) likely turned out to be greater than zero (though still very small). Nevertheless, the final agent's 99% success rate is still enough to say that the agent successfully solved the bottle flipping problem (or at least the one we made in the Unity game engine).

From a qualitative standpoint, the agent trained via PPO first started to learn to throw the bottle as hard as possible using a windmill pitching motion. However, while exploring with its stochastic policy, it eventually found out that if it drops the water bottle early, lets it slide along its arm, and lets it collide and flip off of its hand, it can consistently perform a water bottle flip as defined by the environment. This is an interesting way of solving the bottle flip problem that we had not previously imagined. This is the technique that the agent perfected as training continued to achieve its final 99% success rate.

Although the agent trained with PPO did learn to successfully flip a water bottle, it did not learn the true optimal policy. Compared to the agent's learned policy, the bottle can be flipped more times while still landing it upright; consequently, the agent's approach of just flipping the bottle once is sub-optimal.

B. DQN Results

While developing a DQN approach to the task of flipping a water bottle, we observed minimal learning. After about 4 million training timesteps, the mean reward remained in the negatives. In order to dedicate resources towards the other algorithms' training, we decided to terminate the DQN training. We were unable to retrieve the evaluation metrics from this training, but we observed the agent's activity qualitatively in the Unity Editor. One observation worth noting is that when the agent was training with DQN, the arm did not move. The hand was controlled and the bottle was released, but the arm action was clearly stagnant. The agent had minimal success attempting to flip the bottle with only the hand.

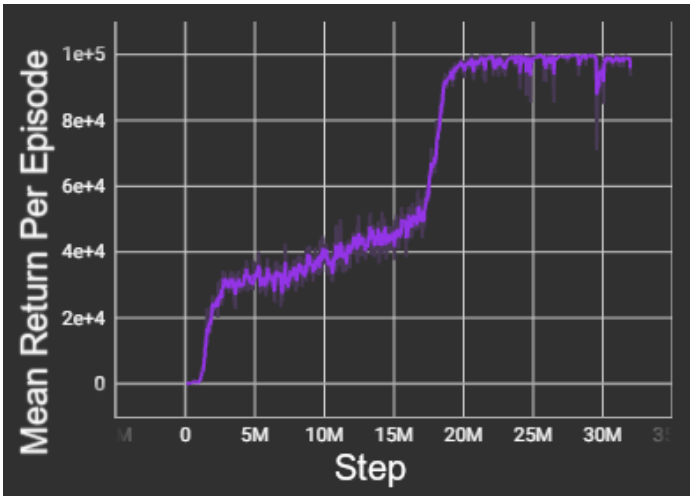

Fig. 3. Best PPO training run mean return per episode vs timestep

One explanation for the poor performance of DQN on this problem is that the action space defined is multi-discrete (i.e. there are multiple action branches with non-binary action values). Although approaches may be adjusted to account for this type of action space, many RL libraries and methods only support single, discrete action spaces. With the MDP described in Section III, the agent has 3 avenues through which to act: the arm angular velocity, the hand angular velocity, and the grip on the bottle. With a standard DQN model, the action space was flattened, preventing the agent from utilizing all actions. This explains why the arm of the agent did not move during training - that particular action branch may have been ignored. The reduction of control here significantly limits the agent's ability to learn effectively. It makes sense that DQN is not recommended for multi-discrete action space environments due to these potential incompatibilities.

Although training towards this task with DQN did not yield expected results, this highlights the problem's complexity. Instead of more simple tasks in which a single action can be taken, having multiple actions to take simultaneously increases complexity. With the multi-discrete action space for this task being significantly reduced, it makes sense that the DQN-trained agent learned minimally.

C. A2C Results

The third and final RL method we applied was A2C for 14 million training timesteps or about 24 hours. The A2C average return per episode versus total timesteps trained graph

is depicted in Figure 4. Although the A2C-trained agent saw success at times, going through periods with over 90% successful bottle flips between 4-5 million training timesteps, it is not stable. The agent moved away from the more successful technique it learned and plateaued to an average return per episode of just about 29. This average return indicated that the agent completes a bottle flip on zero out of one hundred attempts; however, the average return of 29 does mean that the bottle was flipped without landing upright. 5-6 million timesteps into training, the average return per episode started becoming consistently low. Since the return began to never change and the bottle in the Unity Editor was moving in the exact same failed way, we decided to end the training run.

This instability may be explained by hyperparameters not being tuned appropriately. As we discussed in class, settings such as the learning rate can drastically affect training. If the learning rate is too high, an algorithm may temporarily find an optimum but then jump out of that area. This is one explanation for what occurred in the A2C training. However, we did decrease the learning rate in a separate training run and found similar performance trends. Nevertheless, if the hyperparameters were more carefully tuned for this task, the training may be more successful.

Beyond quantitative training metrics, we also observed the agent's performance here. Interestingly, the agent learning using A2C adopted a bottle flipping technique mirroring humans. After roughly 24 hours of training, the arm swings underhanded, and the hand flicks upwards while releasing the bottle. Considering the agent initially explored various techniques such as throwing the bottle overhanded, and the PPO-trained agent adopted a unique technique, this A2C-trained agent's activity is intriguing.

Another interesting observation during the A2C training is that the agent seems to fail landing the bottle upright in the same manner. Time and time again, the bottle flipped just over a full flip. The bottle landed upright then fell towards the arm agent, indicating that the agent flipped the bottle slightly too much. A possible explanation for the agent abandoning the successful flipping technique it found earlier in the learning and settling on this unsuccessful flipping is that the reward structure encourages the agent to flip the bottle more total degrees.

D. Comparison of Methods

By comparing the results from applying PPO, DQN, and A2C to this bottle flipping task, it is clear that the PPO-trained agent adopted the most successful technique. Its average return per episode after 14 million timesteps was roughly 43,000. This contrasts starkly with the A2C-trained agent, which had an average return per episode around 0 after the same number of timesteps. DQN yielded the most poor results as the agent's actions were significantly reduced. In terms of the observed technique, the A2C-trained agent moved in a way most similar to humans approaching this task.

The results of this application project are reasonable. DQN, the poorest performing approach used, is unsuitable for problems with multi-discrete action spaces. With this in mind, the DQN-agent should not have learned well. In contrast, PPO and A2C are compatible with this type of task, and training with those methods was much more successful. PPO training resulted in higher average return per episode than A2C in the same number of timesteps. This is as expected because we were not able to spend many hours fine tuning the A2C hyperparameters; therefore, PPO reasonably trained more successfully than A2C with its clipping technique that prevents performance collapses (as described in Section IV-A).

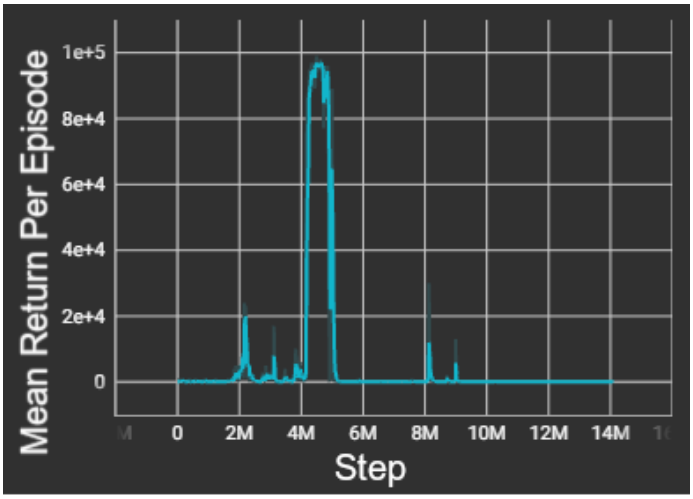

Fig. 4. Best A2C training run mean return per episode vs timestep

VII. Conclusions

Through successfully (and unsuccessfully) applying reinforcement learning to flip a water bottle, we gained tangible experience using a custom environment, framing a problem as an MDP, and solving it with the help of reinforcement learning libraries. These experiences are useful for applying reinforcement learning to real-world problems in the future.

One thing we learned through this project was to be more intentional about choosing which RL approaches to use. As seen through our results with DQN, certain methods are less appropriate than others for particular MDP structures. In future research we will remember this and be more careful to consider our MDP attributes when exploring RL methods. Additionally, this challenge we faced sheds light on an area for improvement in RL research. Despite multi-discrete being an accepted action space type in gym environments, many RL libraries' algorithms' are not compatible with such spaces. One way RL libraries may be advanced is adding functionality to support OpenAI Gym's MultiDiscrete action spaces.

There are numerous ways to advance this project. First, with additional resources, it may be interesting to more intentionally tune hyperparameters associated with A2C. With appropriate settings, A2C may be more successful on this task than it previously was, and we may train an agent with A2C longer. It is clear from the training data using PPO that the agent learned significantly between 24 and 48 hours into training. Due to

time and computational limits, we were able to run A2C for only 24 hours. Considering the training with PPO made notable improvements after this window, it may be interesting to run A2C for longer to explore if it yielded a similar jump later into training.

In order to further improve upon the application of reinforcement learning to flipping a water bottle, the state and action spaces used may be further explored. How may defining the action space differently, maybe as a continuous space, affect training?

Overall, this project was a successful extension of our Reinforcement Learning coursework. Both team members gained hands-on experience framing the complex task as a Markov Decision Process and implementing reinforcement learning approaches. It was interesting to use and compare methods we discussed in lectures on the bottle flipping task. We enjoyed analyzing the results and discussing possible explanations based on our knowledge. Out of the three methods applied (PPO, DQN, and A2C), PPO yielded the most successful training.

APPENDIX

In order to complete this application project in the time provided, tasks were distributed between team members. Although efforts were completed collaboratively, below is a breakdown of which parts each team member took the lead in.

- Bryson: custom environment, PPO configurations, hyperparameter tuning
- Taylor: DQN code, A2C code, initial report outline

Upon completion of the code and training, the analysis and report were accomplished together.

REFERENCES

[1] Yang, Young-Ha et al. “Designing an Efficient Reward Function for Robot Reinforcement Learning of The Water Bottle Flipping Task,” Journal of Korea Robotics Society, vol. 14, no. 2, The Korea Robotics Society, 31 May 2019, pp. 81–86. Crossref, doi:10.7746/jkros.2019.14.2.081.

[2] Kormushev, Petar & Calinon, Sylvain & Caldwell, Darwin. (2010). Robot Motor Skill Coordination with EM-based Reinforcement Learning. Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS). 3232 - 3237. 10.1109/IROS.2010.5649089.

[3] “A2C” A2C - Stable Baselines3 1.7.0a5 documentation. [Online]. Available: https://stable-baselines3.readthedocs.io/en/master/modules/a2c.html. [Accessed: 12-Dec-2022].

[4] KUKA - Robots & Automation. ”KUKA Robot Bottle Flip Challenge - Machine Learning in Practice,” YouTube, Apr 19, 2017. [Video file]. Available: https://www.youtube.com/watch?v=HkCTRcN8TB0. [Accessed: December 13, 2022].

[5] J. Kober, “Reinforcement learning for motor primitives,” Master’s thesis, University of Stuttgart, Germany, August 2008.

[6] Juliani, A. et al. ”Unity: A general platform for intelligent agents,” arXiv:1809.02627 [cs], 2020

[7] Plappert, Matthias. keras-rl. GitHub repository. Available: https://github.com/keras-rl/keras-rl.

[8] Huang, S. et al. ”A2C is a special case of PPO,” arXiv:2205.09123 [cs], May 2022