FINAL REPORT

Deep Reinforcement Learning for Simulated Robotic Manipulation

Group Members:
Jerry Zhang | jz2966
Wenhao Li | wl2655
Zhenyang Du | zd2219

VIDEO RESULTS: https://www.youtube.com/watch?v=K-foX756KTc&t=48s
CS 6731: Humanoid Robotics | Professor Peter Allen

ABSTRACT

OpenAI's Hindsight Experience Replay offers an incredibly insightful technique for reinforcement learning with sparse rewards. In our paper, we recreated and evaluated a variety of their robotic manipulation tasks and experiments: 1. Reaching a target, 2. Pushing an object to a target. 3. Picking and object up and placing it at a target, in which we respectively achieved testing accuracies of 96%, 97.5%, and 95%. Additionally, we attempted to add vision to our robot and obstacles to our environment to further investigate the limits and intricacies of HER.

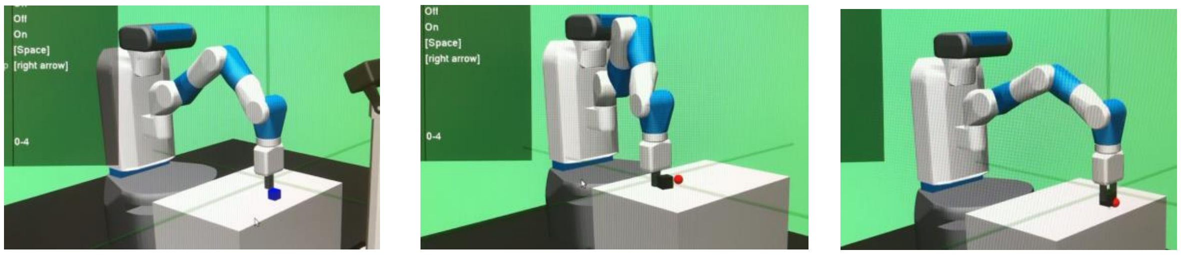
INTRODUCTION

In the last few years, the field of deep reinforcement learning has become one of the most exciting fields in artificial intelligence due to its ability to consolidate the ability of neural networks to understand and represent the world with the ability to act on that representation. Robotics is amongst the many fields that show great promise with deep reinforcement learning applications as researchers uncover ways to utilize robotic sensor data to create models that allow robots to learn about their environments and act accordingly. Additionally, the current era of artificial intelligence has been coupled with the incredible growth of computational power, which has vastly contributed to the influx in new discoveries found in recent AI papers. Computational power has created a landscape in which roboticists can now easily develop and rapidly tune their models in simulation. Robotic reinforcement learning has already shown the capability of GPU-accelerated learning for model-based [1] and model-free [2] deep reinforcement leaning environments.

Using MuJoCo [3] (a multi-joint dynamics physics engine), OpenAI gym [4] (a toolkit/python-wrapper for reinforcement learning algorithm development), and OpenAI baselines (a reinforcement learning implementation repo) [5], our group set out to recreate experiments and results from OpenAI's Hindsight Experience Replay (HER) paper [6]. Using a sparse reward, we recreated three of their simulated robotic manipulation tasks: 1. reaching a target, 2. pushing an object to a target, and 3. picking an object up and placing it at a target. Additionally, we experimented with parameter tuning, utilizing images and training convolutional neural networks to provide us information for our models' observation space. We also experimented with initializing obstacles in our environment, which forced us to create denser rewards. If we had more time, we would have focused these latter two tasks.

BACKGROUND

The purpose of our project is to utilize deep reinforcement learning techniques to teach our robot to perform robotic manipulation tasks show in Figure 1. To do so, we decided to recreate methodologies found in the HER paper [6].



A. B. C.

Figure 1: *Robotic manipulation tasks, A. reach, B. push, C. pick and place*

OpenAI's paper utilizes Deep Deterministic Policy Gradients (DDPG), combined with Universal Value Function Approximators (UVFA), and their novel HER technique to train their RL model for the manipulation tasks.

Deep Deterministic Policy Gradient

The off-policy algorithm used with HER was DDPG [10], which is a model-free RL algorithm designed for continuous action spaces, which in our robotic environment, is (x,y,z) coordinates. With DDPG, two MLP's are maintained: Actor + Critic. The actor, or target policy, maps policy π from state to action, and thus optimizes a neural network to output actions based on current state. This neural network is trained via mini-batch descent on Loss $L_a = -E_s[Q(s, \pi(s))]$, where s sampled from replay buffer. The critic network is an action-value function approximator which estimates the action-value (Q-function), and it is also parameterized by a neural network. With DDPG, we can concurrently learn a Q-function and policy.

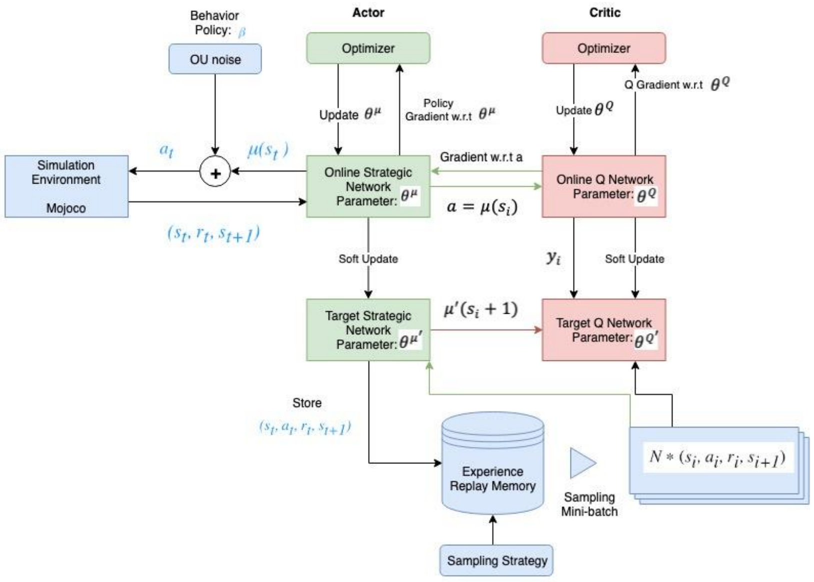


Figure 2: DDPG Framework

Figure 2 provides a descriptive framework that describes DDPG. The replay buffer is an important element of DDPG, as it allows the algorithm to train on a wide variety of past experiences. The buffer store transition tuples: $(s_t,\ a_t,\ r_t,\ s_{t+1})$, subsequently allowing the DDPG agent to learn offline by gathering experiences collected from environment agents + sampling experiences from a large replay memory buffer across a set of unrelated experiences. This enables a very effective and faster training process.

Hindsight Experience Replay

Sparse rewards are considered to be an incredibly difficult challenge within the field of reinforcement learning. OpenAI's development of Hindsight Experience Replay aims to provide a technique for sample efficient learning for sparse and binary rewards [6]. It is important to note that this technique is meant to be coupled with off-policy RL algorithms.

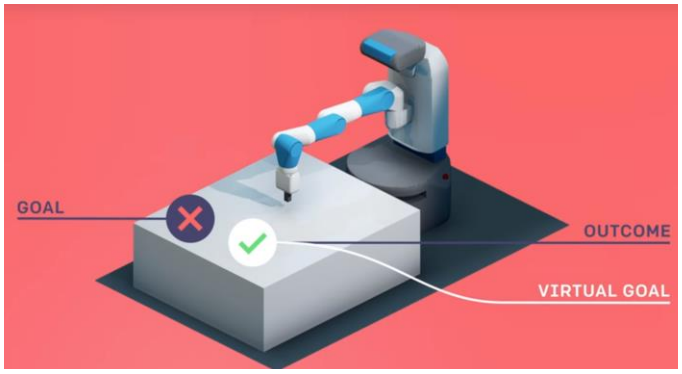


Figure 3: HER Example

The brilliant insight behind HER is that it reevaluates the final goal that was achieved through a stored transition tuple that includes information about the state, action, reward. In Figure 3, the image shows a robot aiming to push an object to the Goal marker, however, it ends up pushing the object towards the Outcome marker. Instead of initializing a new episode and ignoring this failure case, HER decides to store a transition tuple that stores the virtual goal instead of the actual achieved goal. The general idea is that with HER, an algorithm will be able to learn from all episodes stored in the buffer, even if the episode was initially categorized as unsuccessfully.

With this formulation, HER is able to train agents to learn to achieve multiple different goals. It is also important to note that in the paper [6], HER performs better at training an agent for multiple different goals than with a single goal. With more successful episodes, the replay buffer will store more relevant information, which can be called upon during gradient descent for minimization of the loss function.

At every episode store the replay buffer is stored with Original Goal $(s_t,\ a_t,\ r_t,\ s_{t+1},\ g)$, as well as the Virtual Goal $(s_t,\ a_t,\ r_t,\ s_{t+1},\ m(s_T))$, where s_T is the final state in each episode

Additionally, the HER algorithm takes advantage of Universal Value Function Approximators (UVFA), which add the goal as an input to both the actor-critic neural networks. Thus, we can define the Q-function as $Q^{\pi}(s_t, a_t,\ g) = \ E[R_t|s_t, a_t,\ g_t]$, and the policy as $\pi: S\ x\ G \rightarrow A$, where the reward $r_t = r_g(s_t,\ a_t)$.

Algorithm 1 provides a more formal description of the HER algorithm [6].

Algorithm 1 Hindsight Experience Replay (HER)

```
Given:
  • an off-policy RL algorithm $\mathbb{A}$,                          ▷ e.g. DQN, DDPG, NAF, SDQN
  • a strategy $\mathbb{S}$ for sampling goals for replay,           ▷ e.g. $\mathbb{S}(s_0, \ldots, s_T) = m(s_T)$
  • a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \rightarrow \mathbb{R}$.    ▷ e.g. $r(s, a, g) = -[f_g(s) = 0]$
Initialize $\mathbb{A}$                                               ▷ e.g. initialize neural networks
Initialize replay buffer $R$
for episode = $1, M$ do
    Sample a goal $g$ and an initial state $s_0$.
    for $t = 0, T-1$ do
        Sample an action $a_t$ using the behavioral policy from $\mathbb{A}$:
                $a_t \leftarrow \pi_b(s_t||g)$                         ▷ || denotes concatenation
        Execute the action $a_t$ and observe a new state $s_{t+1}$
    end for
    for $t = 0, T-1$ do
        $r_t := r(s_t, a_t, g)$
        Store the transition $(s_t||g,\ a_t,\ r_t,\ s_{t+1}||g)$ in $R$        ▷ standard experience replay
        Sample a set of additional goals for replay $G := \mathbb{S}($current episode$)$
        for $g' \in G$ do
            $r' := r(s_t, a_t, g')$
            Store the transition $(s_t||g',\ a_t,\ r',\ s_{t+1}||g')$ in $R$              ▷ HER
        end for
    end for
    for $t = 1, N$ do
        Sample a minibatch $B$ from the replay buffer $R$
        Perform one step of optimization using $\mathbb{A}$ and minibatch $B$
    end for
end for
```

METHODS

As shown in Figure 1, three different robotic manipulation tasks were specified:

1. **Reach**: A box is randomly initialized on top of the table with a random uniform distribution of 150 mm in the x and y directions. The task is for the gripper to come into close contact with box .
2. **Pushing:** A box and goal are both randomly initialized on a table, and the task is to move to target goal on table. During this task, the robot fingers locked to prevent grasping.
3. **Pick and Place**: Box is randomly initialized on table with the same distribution as before, and target position is initialized with z-axis of uniform distribution between 0 and 350 mm from the top of the table. The fingers are not locked.

Observation space [25 values]: grip pos {3}, gripper state {3}, grip translational velocity {3}, gripper rotational velocity {2}, object position {3}, object relative position {3}, object rotation {3 }, object translational velocity { 3}, object rotational velocity{2}.

Action Space [4 values]: gripper position{3-dim: x, y, z of end effector} and distance b/w the gripper fingers.

Goals: randomly initialized and represented by a 3-dimensional position array where the goal object or target is located.

Reward: At every timestep, a goal of -1 or 0 was returned; 0 if object position is within 5 cm of the target position, -1 otherwise. Also, there was a discount factor $\gamma = .98$ to prioritize immediate rewards.

Training

The three tasks were trained via the actor-critic paradigm and optimized via gradient descent on the loss function with the experience replay buffer, as shown in Fig 2.

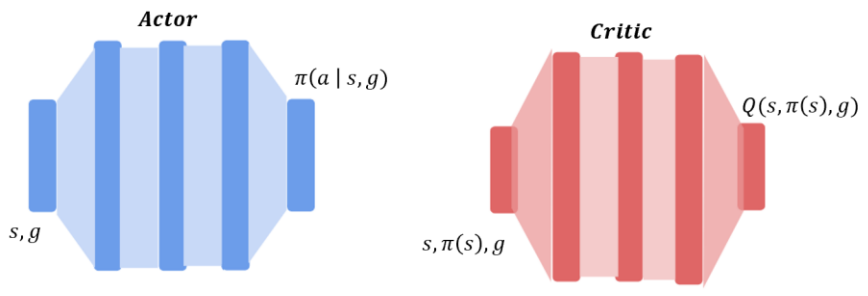


Figure 4: Actor-critic network utilized to train models for manipulation tasks. Three hidden layers were used for both networks, each of size 256.

CURRENT WORK + NEXT STEPS

Following the implementation and recreation of the models found in OpenAI's HER paper [6], we decided to attempt to extend the paper's work by implementing vision and obstacles.

Vision

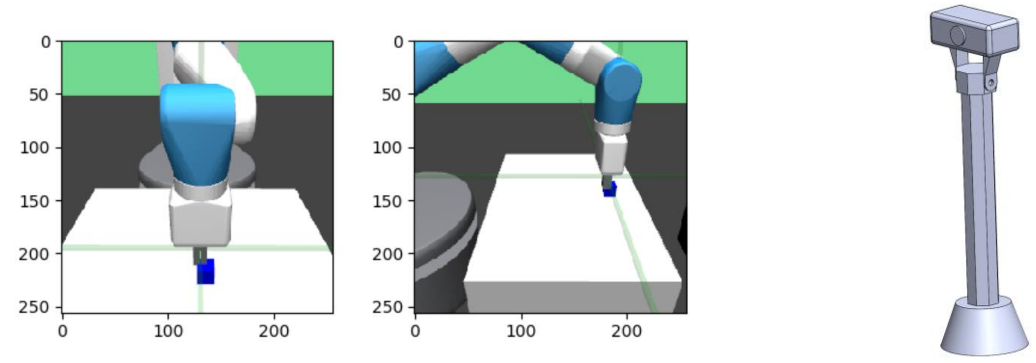


Figure 5: Left: Front view and side view camera images. Right: CAD model of the camera imported into MuJoCo

For vision-based RL, we adjusted the observation space to be the following:

Observation space [14 values]: grip pos {3}, gripper state {3}, grip translational velocity {3}, gripper rotational velocity {2}, object position {3}

By only retaining object position information and removing all other object information, we ensured that only camera information was utilized for object state values. We also experimented with training CNN's to output object pos + velocity information as well. (Note this information can be readily called from the MuJoCo simulation environment).

To implement vision, we generate fake simulated training data with our previously successful policies for reach, pick and place, and push. Unfortunately, due to the time constraint of the semester, we were only able to successfully finish train the CNN for a Fetch Reach model with cameras. Issues regarding vision implementation with Push and Pick + Place can be found in Appendix A.2

Obstacles

A potential avenue of exploration and analysis for robotic manipulation tasks with HER involves the inclusion of obstacles. Figure 6 shows a screenshot of an obstacle imported for the Reach task and environment.

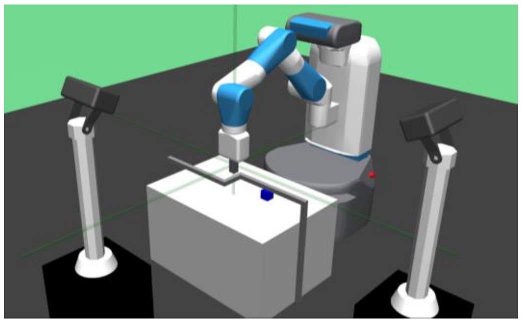
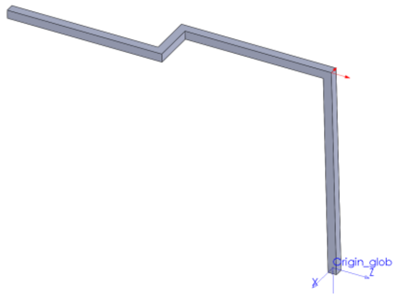



Figure 6: Left: image of Fetch Reach with Obstacle. Right: CAD image of obstacle

With and without vision, we were unfortunately unable to train a model to successfully reach our desired goal, which is most likely due to how the HER replay buffer stores transition tuples. Appendix A.3 expands more on the roadblocks we faced with this route.

RESULTS | https://www.youtube.com/watch?v=K-foX756KTc&t=48s

Fetch Reach

Training and testing data for the fetch reach model can be seen below:

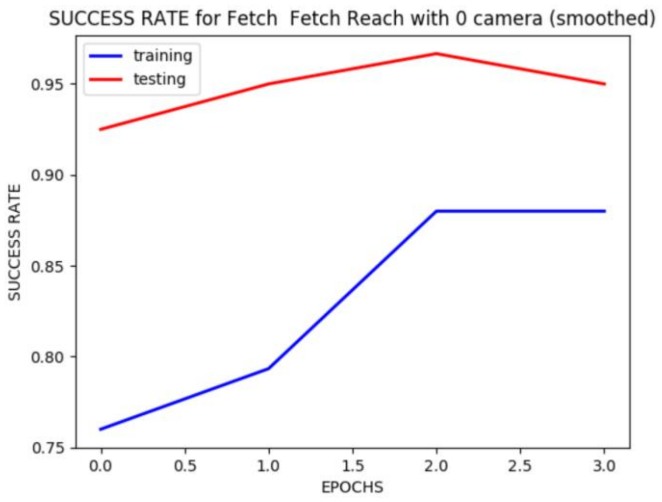
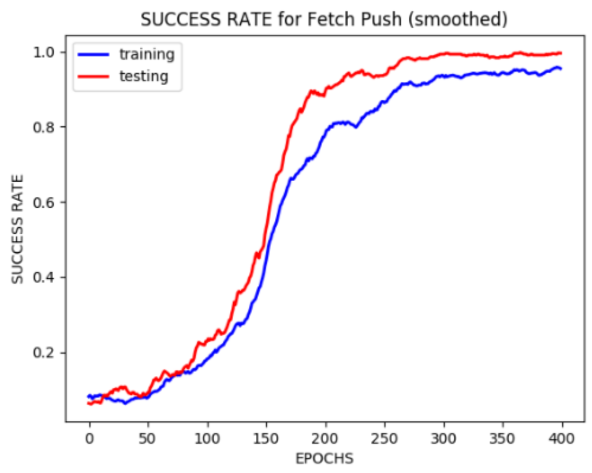


Figure 7: Epochs vs Training/Testing Success Rate for Fetch Reach

Fetch Push

Training and testing data for the fetch push model can be seen below:



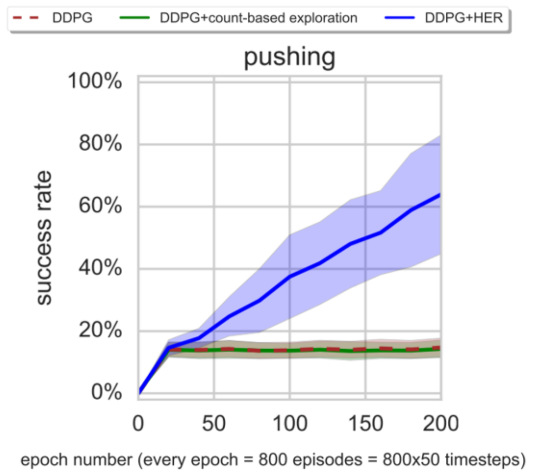
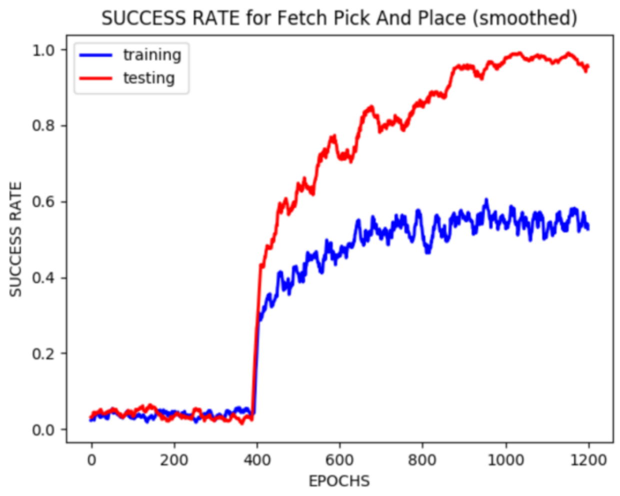


Figure 8: Epochs vs Training/Testing Success Rate for Fetch Reach.
Left: Recreated Plot Right: Plot from Open AI [6]

Fetch Pick and Place



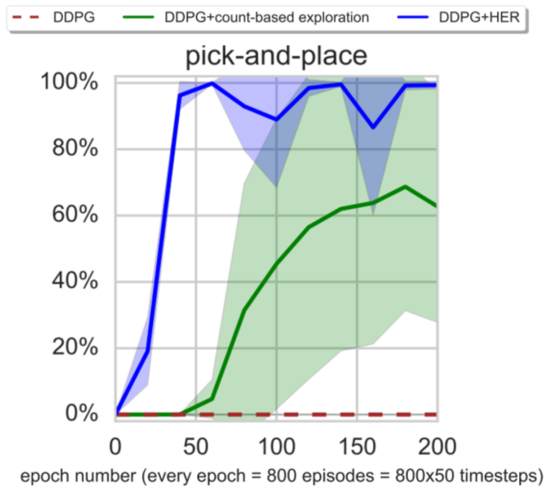


Figure 9: Epochs vs Training/Testing Success Rate for Fetch Reach.
Left: Recreated Plot Right: Plot from Open AI [6]

Fetch Reach with 2 Cameras

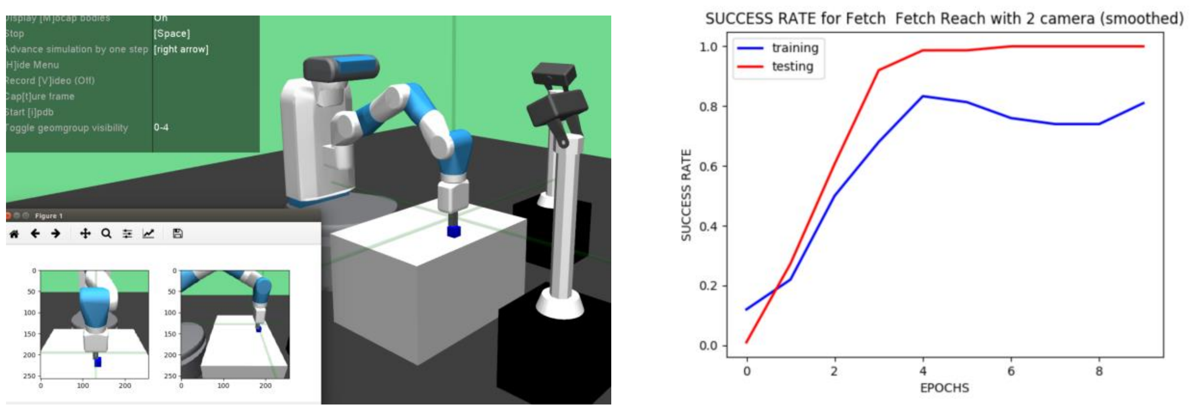


Figure 10: Left: Simulation of Fetch Reach with 2 cameras. Right: Epochs vs Training/Testing Success Rate for Fetch Reach with two cameras.

DISCUSSION AND CONCLUSION

Fetch Reach, as seen in Figure 7, converges to a near perfect testing success rate very quickly. This is most likely due to the fact that at every episode, in accordance with HER, a virtual goal is being input into the transition tuple for the replay buffer, even if it does not reach its actual goal. Because this occurs at every episode, the loss function gets optimized very quickly. This specific task was not compared against in the HER paper due to its rapid convergence.

From Figures 8, it can be seen that we were roughly able to recreate the Fetch Push results from the HER paper. At 200 epochs, we were able to achieve a very similar testing rate of the mean of ~60%. Our results for Fetch Pick and Place, as seen in Figure 9, do differ. Only after ~400 epochs, does our model find solutions to increase our success rate. This result is due to the fact that the OpenAI researchers began their training with a deliberate single successful state in which the box is grasped, and then start half their task from this state. With this handicap, they were able to make exploration of the task much easier, and immediately store many more successful transition tuples in their replay buffer for the loss function to call during gradient descent.

In order to make their manipulation models train very efficiently, they introduced an incredibly robust observation space, which includes many values that would be very hard to accurately acquire if this model was transferred to a real-world Fetch. Values such as object relative position to the goal, object translational, and object rotational velocity would be very difficult to acquire in real life. In MuJoCo, however, these values are very easily callable.

In order to make a more accurate simulation environment, we decided to remove many of these object values from the observation space and utilize vision to find object position. By generating simulated data, utilizing a VGG16 pretrained CNN, and further training our network, we were able to successfully perform our Reach task with our constrained observation space as shown in Figure 10. However, we were unable to do so for Push and Pick + Place. Appendix A.2 expands more upon this.

If we had a more time, there were a few routes we wanted to explore in order to better improve our Deep RL model with our implementation of vision and/or with obstacles. Instead of using HER, which focuses on very sparse rewards, we would potentially want to experiment with utilizing PPO [9] for policy gradient descent as well as shifting our focus from model-free RL to model-based RL methodologies. Additionally, we would love to have deployed one of our models on a real robot, which most likely require some level of domain randomization on image data. It would have also been very interesting to experiment with dynamics randomization as seen in [8] in order to improve the real-world transfer process.

Authors' Notes: In our project proposal, we specified that our main goal in taking upon this project was to expose ourselves to deep reinforcement learning in robotics, and utilize vision to do so if possible. Of course, at the time, we all had a very naïve understanding of RL, let alone RL for robotics. This project was a phenomenal learning experience, and we regret that we did not have more time to expand upon our results.

APPENDIX A: PROJECT CHALLENGES

A.1 *Simulation Environment Issues*

This was an enormous bottleneck for our project. At a high level, we were unable to find a method, with the current resources were using (ROS, Ubuntu 16.04), to train Reinforcement Learning algorithms with Gym and Gazebo environments. We utilized poorly written guides from ROS wiki [8], spent nearly a month attempting to solve these issues, went to office hours numerous times, and manually configured packages to work with our virtual environment and with ROS. The final issue we faced was that we were unable to call our custom ROS services and messages in our Python 3.5 environment, even though there were callable in Python 2.7. We had to use a 3.5 venv as well in order to utilize OpenAI baselines and OpenAI gym. After a month of struggling, we did finally find a potential workaround using gym-gazebo2 that required a complete overhaul of our current systems with its ROS2 + Ubuntu 18.04 LTS dependencies. In our progress reports, we did state that were going to experiment with a couple of simulation environments at once, which did help get the ball rolling once we switched to MuJoCo. Unfortunately, we lost a lot of time in the semester by the time we made the switch.

A.2 *Adding Vision for Push and Pick + Place Tasks*

Fetch Push and Fetch Pick + Place tasks were much more difficult for us to train than Fetch Reach. We created and trained separate CNN's to output object position and object velocity [6-dim array]. We trained our model with 5000 images of simulated training data from our original successful Pick + Place and Push tasks. We realized that in order to truly capture velocity information into our model we may need to input an RCNN or multiple images. Thus, we decided to incorporate two cameras, which unfortunately did not make our model any better. We experimented with different pretrained convolutional networks such as VGG16, Dense101, and ResNet50 to incorporate, which also did not aid in improving our model. A major flaw in our attempt to implement vision the size of the images we created for our dataset. Due to the lack of time, we only utilized 32x32 RGB images to train our network, which may have been too low of a resolution. A single pixel represents relatively large area. Generating 10000 simulated 32x32 images for our dataset took us ~ 4 hours. Originally, we wanted to generate 10000 256 x 256 RGB images, but we realized that would take us ~40 hours to complete. Additionally, a dataset that large (10000x256x256x3) would have been too massive for our laptops to read and load. If we had more powerful machines, we definitely would have produced my robust data, which would have hopefully have encoded relevant velocity data for Push and Pick and Place tasks. We also wanted to implement a method to concurrently train our CNN and actor-critic networks.

A.3 *Adding Obstacles to the simulation environment while using HER*

With our current model and the inclusion of a bracket-like obstacle, we trained a policy with HER and with our reach environment as previously stated. Unfortunately, even after much parameter tuning and training, we were unable to create a model that yielded a testing accuracy of more than 10% (after 48 hours with a CUDA machine with 7 CPUs). We realized that the major flaw in our logic was how our new denser reward function conflicted with how HER intrinsically operates. We included a denser reward function that would return -1 at every timestep when any robot link was in contact with the obstacle as well as when the robot/object was more than 5 cm away from the goal. Unfortunately, what we failed to catch was that with the HER algorithm, every episode was being stored into the replay buffer, regardless of whether the robot contacted the obstacle or not. If we had more time, we would have to rewrite our reach environment and edit the way replay buffers are stored to make sure transition tuples for episodes where there was obstacle contact would not be categorized and stored as cases with virtual goals in the replay buffer.

APPENDIX B: Division of Labor

Wenhao Li: Created and trained all CNN models; analyzed and created all data models; algorithm development; trained and tested manipulation tasks; research

Jerry Zhang: Created the custom simulation environments; trained and tested manipulation tasks; reward engineering; create URDF files and .xmls from CAD for MuJoCo; research

Zhengyang Du: Attempted to figure out ROS/gym implementation; Created a dual-boot CUDA machine for training; research; create CAD files

REFERENCES

[1]: https://ieeexplore.ieee.org/document/6386109
[2]: https://arxiv.org/pdf/1810.05762.pdf
[3]: http://www.mujoco.org/
[4]: https://gym.openai.com/
[5]: https://github.com/openai/baselines/tree/master/baselines/her
[6]: https://arxiv.org/pdf/1707.01495.pdf
[7]: https://arxiv.org/pdf/1509.02971.pdf
[8]: http://wiki.ros.org/openai_ros
[9]: https://arxiv.org/abs/1707.06347
[10]: https://arxiv.org/pdf/1509.02971.pdf