Carnegie Mellon University
The Robotics Institute

16-662A: Robot Autonomy
Project Report

Visual Pushing and Grasping

Team 11:

Team Members:
Sai Shruthi Balaji
Aditya Gaddipati
Shruti Gangopadhyay
Kevin Gmelin

Date: May 05, 2022

1. Motivation

Robotic grasping has been an active area of research in robotics for a long time, and it is still nowhere near human-like performance. Achieving generalizable and successful grasps, especially in challenging cluttered environments, is not an easy feat. Consider an environment where cluttered objects are tightly packed together. In this case, it is not possible for a standard parallel-jaw gripper to grasp an individual object since there is no gap for the gripper fingers to be inserted into. Hence, there arises a need for the robot to perform a non-prehensile action such as a push to create some space for better grasps. Recent advances in this area rely increasingly on perception, and leverage reinforcement learning techniques to let the robot learn from exploration by obtaining rewards for successful grasps. [1] is one such approach that learns the synergies between pushes and grasps using visual deep Q-learning. [2] is a very similar approach that uses a composite suction-based manipulator to grasp objects. [3] is another approach that improves the reward strategy on top of [1] to converge faster towards more successful grasps.

Our project is a re-adaptation of [1] to enable the Franka Panda robot to learn generalizable robotic grasping by leveraging pushing actions. We also attempted to make improvements to the approach by implementing approaches such as double Q-learning [4], importance sampling during prioritized replay, and a piecewise reward strategy from [3] and observed various results that are detailed in the below sections.

2. Overview of the approach

The project aims to leverage the synergy between pushing and grasping actions for successful grasping in a cluttered environment. This has been achieved by training the policy through self-supervised trial and error, to learn pushes that enable future grasps and grasps that can leverage past pushes. Figure 1 represents the approach taken.

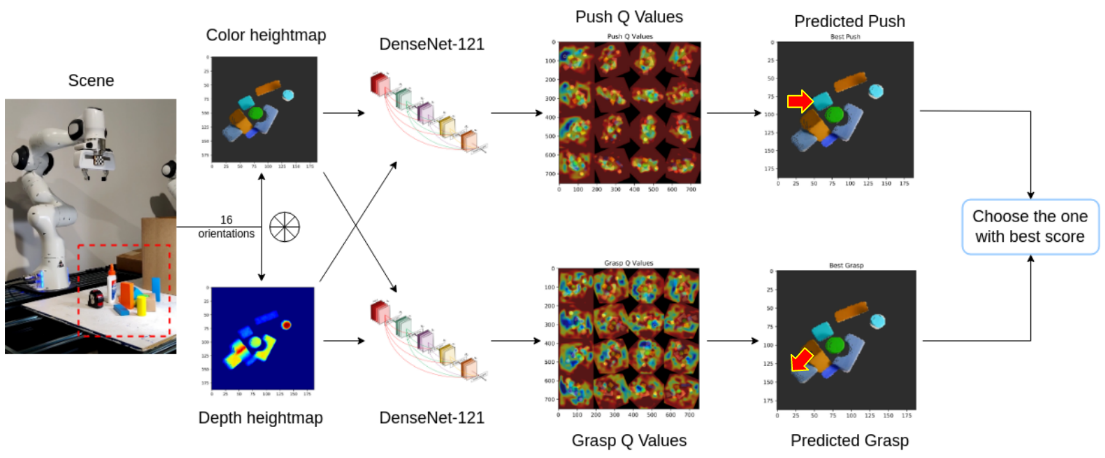


Figure 1: Visual representation of the approach

The setup for our project consists of a Franka arm, Azure kinect camera and a cluttered environment. RGB-D data from the kinect camera is projected onto color and depth heightmaps. These heightmaps are then fed as inputs to two fully convolutional networks (DenseNet-121) which define the deep Q function. The FCNs perform pixel-wise sampling of end-effector orientations and locations for the pushing and grasping primitives. The heightmaps are rotated in 16 different orientations to account for various pushing and grasping orientations and fed into the FCNs. One FCN outputs pixel-wise Q values for executing a push while the other FCN outputs the pixel-wise Q values for executing a grasp. Finally, the action that generates the highest Q value, i.e, the highest reward, is chosen and executed.

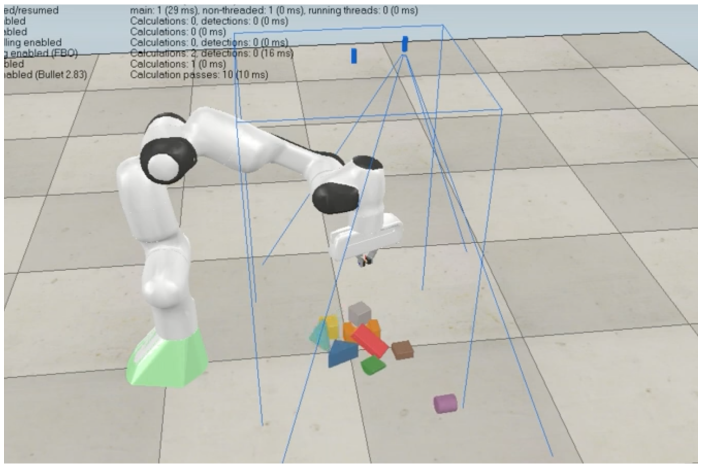
3. Methods Implemented

The main methods implemented include creation of a setup for training the robot in CoppeliaSim, implementation of the motion primitives and sensor preprocessing on the real Franka Arm, and modification of the RL algorithm and reward function used by [1].

3.1 Simulation

In order to speed up the process of training the RL algorithm, the team decided to set up a CoppeliaSim environment for training the Franka. The team based our simulation on the simulation provided by the authors from [1] which uses a UR5 arm. The team used the default Franka arm and Franka gripper models provided by CoppeliaSim. All joints were changed from torque/force mode to IK (inverse kinematics) mode. An inverse kinematics tip and target dummy points were defined at the end of the gripper. The tip was set to be a child of the Franka Arm's last link. The target was set to be a child only of the first link of the Franka Arm itself. The team added an IK link between the tip and target. This allowed the ability to control the Franka's end-effector pose by moving the target. All links were made non-dynamic. A small gap was added to the minimum gripper closing width. This prevented a problem where the gripper's jaws would occasionally get stuck together after closing the gripper. The simulation camera was modified to have the same intrinsic matrix, extrinsic matrix, and resolution as the real system. The Franka was also modified to be invisible to the camera, ensuring the robot didn't get in the way of generating orthographic heightmaps.

Within the actual code base, the limits of the workspace were modified to reflect the real system. The original code base had hardcoded the size of some arrays used when computing the loss function. This caused a problem when the workspace limits were adjusted, so the team modified the trainer code to scale for an arbitrarily specified workspace. Finally, the team adjusted the thresholds for determining if the gripper is open or closed. The end result was a robot that could learn to push and grasp blocks via its interactions in CoppeliaSim. This simulation is shown below in figure 2.



```
ed/resumed              main: 1 (29 ms), non-threaded: 1 (0 ms), running threads: 0 (0 ms)
abled                   Calculations: 0, detections: 0 (0 ms)
abled                   Calculations: 0 (0 ms)
lling enabled           Calculations: 0, detections: 0 (0 ms)
g enabled (FBO)         Calculations: 2, detections: 0 (16 ms)
bled                    Calculations: 1 (0 ms)
nabled (Bullet 2.83)    Calculation passes: 10 (10 ms)
```

Figure 2: Simulation of a Franka arm in CoppeliaSim

3.2 Real System

The simulation and the real system were set up in parallel by the team.

We first had to create motion primitives for pushing, grasping, and for resetting the work scene. The pushing primitive goes to a specified x,y position on the table at a given yaw angle. The gripper is kept pointing perpendicular to the table. The z position is set to be 1 cm above the height of whatever block/table is located at that x,y position. In this primitive, the robot first closes its gripper and goes to the starting point. It then moves its gripper 10 centimeters to the right relative to the gripper's x-axis. Finally, the robot raises its gripper and returns home.

The grasp primitive is similar to the push primitive in that it will go to a specified x,y position with a given yaw. It first travels to a pre-grasp height. Then, it lowers the gripper to be about 1 cm lower than the heightmap value at that x,y position and closes the gripper. The system detects if an object has been grasped by checking if the gripper width is above a fixed threshold. If the gripper is fully closed (no object was grasped), the robot returns home. Otherwise, the robot moves the newly-grasped object to a predefined location above a cardboard box. Once above the cardboard box, the robot checks again if the gripper is still not fully-closed. If it is fully-closed, the system knows that it dropped the object. Otherwise, it will open the gripper to drop the item into the box. Finally, it resets back to home.

The final primitive that was implemented was a reset scene primitive. This primitive is used to re-add blocks to the scene after all blocks have been grasped and placed in the cardboard box. In this primitive, we grasp the edge of the cardboard box and drag it to the middle of the workspace.

We then have the robot raise the cardboard box. The box has an open bottom that results in the blocks falling out into the workspace. We then place the box back to its original position. This allows the robot to be self-supervised during training by continually resetting the scene when it detects there are no more blocks left.

All of the above motion primitives were implemented using the FrankaPy API.

In addition to these motion primitives, we had to implement some sensor preprocessing steps to get the full RL algorithm running on the real system. The input to the Q network is a depth and color orthographic heightmap. After utilizing the code from [1] for generating such orthographic heightmaps (using the updated camera calibration), the team found that the top left corner of the workspace had a height more than 2 centimeters greater than bottom right corner due to inaccuracies in the calibration. We corrected this by generating a heightmap of the table with no blocks on it, saving this heightmap, and then subtracting this table heightmap from new heightmaps. The team also found that, when using weights from simulation, the robot would occasionally make pushes in open space above shadows on the table. To improve performance and minimize this sim2real gap, the team added code to identify all parts of the heightmap with height close to 0 and change their color to be the same color as the simulation table. We found that this helped when running the simulation weights on the real system.

With the above changes, the team was able to successfully run the simulation-learned weights on the real system. Furthermore, with the reset scene primitive, the team was able to further train the robot on the real system.

3.3 RL Algorithm Adjustments

The team attempted two different adjustments to the original RL algorithm implementation: using double Q-learning and adjusting the reward function.

The first adjustment the team tried was to use double Q-learning. As mentioned by [1], "Additional extensions to deep networks for Q-function estimation such as double Q-learning, and dueling networks have the potential to improve performance, but are not the focus of this work". The purpose of double Q-learning is to counter the fact that typical Q-learning tends to overestimate action values. It does this by changing how targets for the Q network are calculated. Regular Q-learning uses the following target:

$$Y_t^Q = R_{t+1} + \gamma \, max_a Q(s_{t+1}, a; \theta_t)$$

In double Q-learning, the target is:

$$Y_t^{Double\ Q} = R_{t+1} + \gamma \, Q(s_{t+1}, argmax_a Q(s_{t+1}, a; \theta_t); \theta_t^-)$$

Here, $Q(s, a; \theta_t^-)$ is a fixed target network used for evaluating the value of being in the next state. In double Q-learning, we get the value of state s_{t+1} by picking an action using the original Q network, and we evaluate this action using the fixed target network. As shown by [4], this can reduce overestimation to improve performance. In our implementation, the fixed target network is set equal to the online network every 50 iterations. A comparison in training results from running regular Q-learning and double Q-learning using the Franka arm in CoppeliaSim is shown below in figure 3. As can be seen, double Q-learning had slightly better performance after 2000 iterations, as expected. However, to be confident that this is due to double Q-learning and not due to random chance, the team would have to rerun this experiment multiple times.

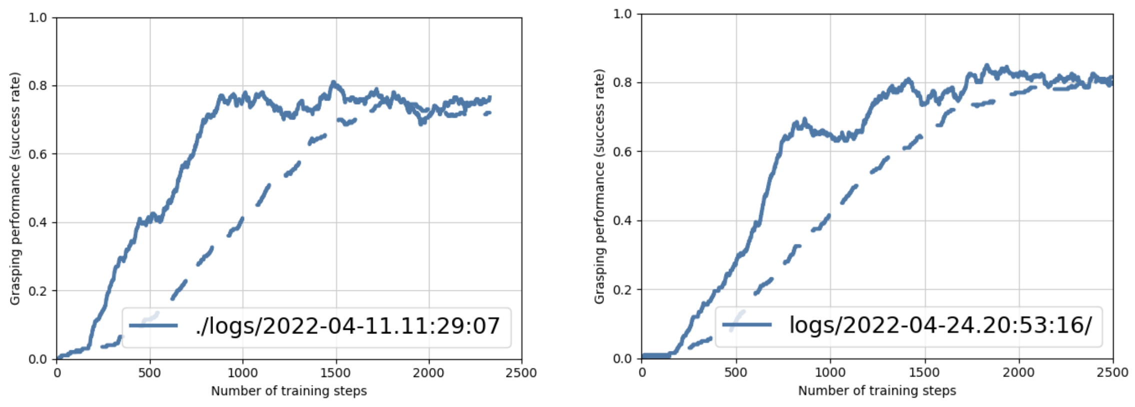


Figure 3: Grasp success using regular Q-learning (left). Grasp success using double Q-learning (right). The solid line indicates grasp success rate. The dotted line indicates percentage of pushes that are followed by a successful grasp

The second adjustment we tried was modifying the reward function. We implemented a piecewise reward strategy using the approach from [3]. The downside of the reward functions used in [1] is that it does not penalize failed grasps, and gives equal rewards to pushes irrespective of the magnitude of change in the scene. Therefore, our piecewise reward strategy is as shown in Table 1.

Table 1: Piecewise rewards

Primitive	Performance	Reward
Grasp	● Success ● Failure	+1 -1
Push	● Success ○ 1-2% change in scene ○ 2-5% change in scene ○ 5+% change in scene ● Failure	 +0.3 +0.5 +0.7 -0.1

This reward strategy showed comparatively better results while training the robot in the real world, giving more preference to significant pushes which were in turn used for successful grasps. The percentage of failed grasps also visibly reduced.

4. Demo/Evaluation

For evaluation, we created a challenging cluttered scene and computed the percentage of successful pushes and grasps. A push is considered successful if it causes changes in the scene, and a grasp is considered successful if the robot correctly holds the object from the time of picking up until the time of dropping the object into a bin. The robot was trained for 2500 iterations on simulation, and for 200 iterations on the real robot. A piecewise reward strategy was used when the robot was trained in the real world.

Percentage of successful pushes: **70%**
Percentage of successful grasps: **77.78%**

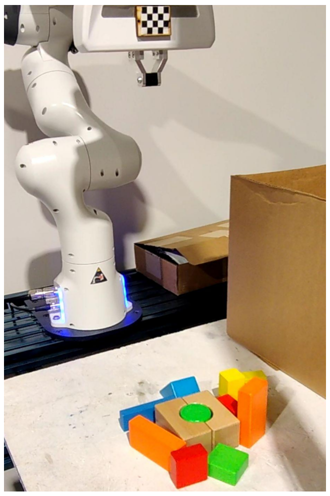


Figure 4: Evaluation scene

The attached video demonstrates all the above explained scenarios including training in simulation, sim2real transfer, real world training and evaluation.

5. Key Challenges

Some of the key challenges faced in the project are as follows:

1. The extrinsic calibration of the overhead camera was imperfect. This resulted in inaccurate heightmaps which lead to bad pushing actions.

2. Since the table on which Franka was placed was skewed, the orthographic projection from the camera resulted in an inaccurate heightmap. To tackle this problem, manually calculated Z offsets were used to correct for the skew.
3. The training of the RL policy was compute-intensive and time consuming.
4. Since the policy was trained in simulation, sim2real gap was evident. For example, the table color and shadows seemed to be affecting the quality of pushes.
5. The RL policy was learning shortcuts to generate more reward such as repeated pushes only to cause a change in the environment.

6. Future Work

In this project, the RL policy was trained in simulation and deployed on the real robot. Future work can include training the policy for more iterations on the real robot in order to bridge the sim2real gap. New reward functions and penalties can be explored for strengthening the pushing and grasping synergy and to avoid learning shortcuts. Some experimentation with the Q-value function is possible such as extending the Densenet architecture to make it wider and deeper and trying other CNN architectures. At inference time, some optimization can be done to avoid repetitive attempts of previously failed pushes and grasps.

7. References

[1] Andy Zeng et al. Learning Synergies between Pushing and Grasping with Self-supervised Deep Reinforcement Learning. IROS 2018. arXiv:1803.09956

[2] Yuhong Deng et al. Deep Reinforcement Learning for Robotic Pushing and Picking in a Cluttered Environment. IROS 2019. 10.1109/IROS40897.2019.8967899

[3] Peng Gang et al. A Pushing-Grasping Collaborative Method Based on Deep Q-Network Algorithm in Dual Perspectives. arXiv:2101.00829

[4] Van Hasselt, Hado, Arthur Guez, and David Silver. "Deep reinforcement learning with double q-learning." *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. No. 1. 2016.