VISION BASED ROBOT MANIPULATION TESTBED FOR REINFORCEMENT LEARNING

A PROJECT REPORT

submitted by

SREEJITH KRISHNAN R

TVE18ECRA17

to

the APJ Abdul Kalam Technological University in partial fulfillment of the requirements for the award of the Degree
of

Master of Technology

in

Electronics and Communication Engineering

with specialization in

Robotics and Automation

CET Centre for Interdisciplinary Research

College Of Engineering

Thiruvananthapuram

JULY 2020

DECLARATION

I undersigned hereby declare that the project report ***"VISION BASED ROBOT MANIPULATION TESTBED FOR REINFORCEMENT LEARNING"***, submitted for partial fulfillment of the requirements for the award of degree of Master of Technology of the APJ Abdul Kalam Technological University, Kerala is a bonafide work done by me under supervision of **Prof. Linu Shine**, Assistant Professor, Department of Electronics and Communication Engineering, College of Engineering, Thiruvananthapuram. This submission represents my ideas in my own words and where ideas or words of others have been included, I have adequately and accurately cited and referenced the original sources. I also declare that I have adhered to ethics of academic honesty and integrity and have not misrepresented or fabricated any data or idea or fact or source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the institute and/or the University and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been obtained. This report has not been previously formed the basis for the award of any degree, diploma or similar title of any other University.

Place:
Date: Sreejith Krishnan R

CET CENTRE FOR INTERDISCIPLINARY RESEARCH

COLLEGE OF ENGINEERING TRIVANDRUM

CERTIFICATE

This is to certify that the report entitled **"VISION BASED ROBOT MANIPULATION TESTBED FOR REINFORCEMENT LEARNING"** *submitted by* **SREEJITH KRISHNAN R,** *to the APJ Abdul Kalam Technological University in partial fulfillmentof the requirements for the award of the Degree of* **Master of Technology** *in* **Electronics and Communication Engineering** *with specialization in* **Robotics and Automation** *is a bonafide record of the project work carried out by him under our guidance and supervision. This report in any form has not been submitted to any other University or Institute for any purpose.*

Dr. Sindhu G
PG Coordinator
Dept. of Civil Engg.
College of Engineering
Trivandrum

Prof. Linu shine
Professor and Guide
Dept. of Electronics Engg.
College of Engineering
Trivandrum

Dr. Sindhu G
Head of the department
CET Center for Interdisciplinary Research
College of Engineering
Trivandrum

ACKNOWLEDGEMENT

I express my gratitude to **Prof. Linu Shine**, Associate Professor and Guide, Department of Electronics and Communication Engineering, College of Engineering, Thiruvananthapuram for all the necessary help and guidance extended to me towards the fulfillment of this work.

With my deep sense of gratitude, I would like to thank my respected teachers, **Dr. Ranjith S. kumar**, Associate Professor, Department of Mechanical Engineering and **Prof. P.S. Shenil**, Assistant Professor, Department of Electrical and Electronics Engineering College Of Engineering, Thiruvananthapuram, for their valuable suggestions and solutions.

I hereby acknowledge my sincere thanks to **Dr.Sindhu G**, Dean Research, College Of Engineering, Thiruvananthapuram, for her motivation and inspiration provided.

I also thank all the staff of College Of Engineering, Thiruvananthapuram for their wholehearted support and cooperation.

Finally I would like to acknowledge my deep sense of gratitude to all my well-wishers and friends who helped me directly or indirectly to complete this work.

SREEJITH KRISHNAN R

ABSTRACT

In order to assist in general tasks, autonomous robots should be able to interact with dynamic objects in unstructured environments. Robot manipulation of objects is the key component in all autonomous robot applications requiring interaction with environment. In vision based robot manipulation, robot has to measure the environment state using camera and take actions according the measured state and goal. The main challenges in here are interpreting the noisy high dimensional data from camera and deciding actions according to stochastic and non stationary environment state.

This project will develop a testbed for evaluating and developing various reinforcement learning algorithms for vision based robot manipulation tasks. Reinforcement learning algorithms usually have very low data efficiency and require lot of training data. Traditional testbeds available for robot manipulation tasks are not designed for parallel/distributed training making them slow for collecting training data. Developing a testbed which can run multiple simulations in parallel and is flexible enough for testing different reinforcement learning algorithms on different tasks like grasping, moving etc. will aid in developing RL algorithms faster and can be used as a standard framework for benchmarking different RL algorithms.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Background

In order to assist in general tasks, autonomous robots should be able to interact with dynamic objects in unstructured environments. Robot manipulation of objects is the key component in all autonomous robot applications requiring interaction with environment. In vision based robot manipulation, robot has to measure the environment state using camera and take actions according the measured state and goal. The main challenges in here are interpreting the noisy high dimensional data from camera and deciding actions according to stochastic and non stationary environment state.

Methods for robot manipulation can be broadly classified into traditional and data driven methods. In traditional methods, data from camera is interpreted using computer vision algorithms and actions are hand coded based on interpreted state. This approach works well for robots deployed for specific task like in automated production lines. But in stochastic and non stationary environments, it is not possible to hand code actions for all environment states. On the other hand data driven methods, particularly reinforcement learning have shown great potential in this use case. In reinforcement learning, an agent learns to take actions (policy) based on measured environment state by interacting with the environment through trial and error for maximising a feedback signal (reward or value function). The main challenge in this method is requirement of large amount of data for agent to learn. Collecting large amount of data from real robot will be slow and expensive and might require manual intervention. Instead, for faster and cheaper development, agent can be trained on simulated robot manipulation environment. However policies trained on simulated environments are not directly transferable to real robots due to various differences between simulation and real environments.

1.2 Research Gap

Current testbeds available for vision based robot manipulation for reinforcement learning have following limitations

- Slow training data collection speed
- Non standard/readily available frameworks used

1.3 Objectives

- Improve RL model training speed by running multiple simulations in parallel
- Use standard frameworks that support training distributed RL algorithms
- Flexibility for adding different types of robot manipulation tasks like grasping, moving etc.

1.4 Outline of Report

- Chapter 1 gives an introduction to project and outline of this report
- Chapter 2 gives an overview of previous works done
- Chapter 3 gives details of simulation and hardware setup used to train reinforcement learning agents
- Chapter 4 gives the specifications of simulation environment and hardware used for evaluating testbed
- Chapter 5 gives the performance of simulator and testbed evaluation results of PPO in table clearing environment

Chapter 2

Literature Review

Overall methods for robot manipulation can be classified into traditional and data driven methods. Traditional methods are rule based and can be applied for structured and deterministic environments. For stochastic and unstructured environments, data driven methods are used. In this literature review, we will focus on main data driven robot manipulation methods using reinforcement learning. In reinforcement learning, training data is generated by a simulator. Reinforcement learning methods have low sample efficiency and require lot of data for training. For fast training of reinforcement learning algorithms, effective architectures like GORILLA [6] is commonly used. In the following sections, various frameworks for efficient training of reinforcement learning algorithms and various commonly used reinforcement learning methods are discussed.

2.1 SURREAL: Open-Source Reinforcement Learning Framework and Robot Manipulation Benchmark

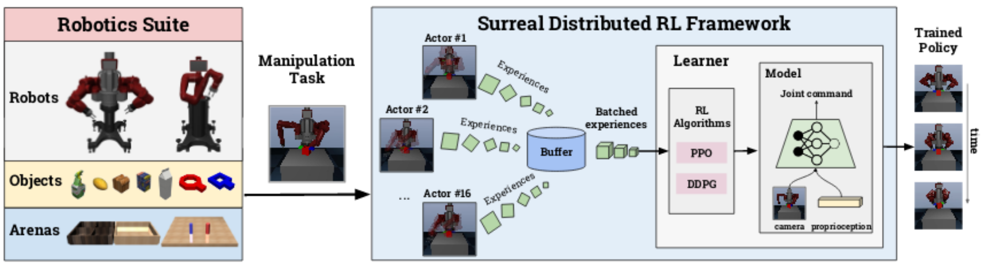


Figure 2.1: Surreal RL framework Source: [1]

SURREAL (Scalable Robotic Reinforcement Learning Algorithms) is an open-source framework for benchmarking reinforcement learning algorithms for different robot manipulation tasks. SURREAL framework consists of following major components

- Actors - for generating training data for RL algorithms via simulation
- Buffer - for storing generated data from actors
- Learning - Core RL algorithms which updates the parameters of the model by reading data from buffer
- Parameter server - Storing parameters of RL models

By providing a decomposed framework, SURREAL can enable scalable reinforcement learning and can speedup RL algorithm testing by increasing computational power. SURREAL framework can be deployed on major cloud computing providers like google cloud.

SURREAL also includes a robotics suite which provides environments for common robot manipulation tasks. This enables benchmarking RL algorithm on multiple robot manipulation tasks with little effort.

2.2 Reinforcement learning methods

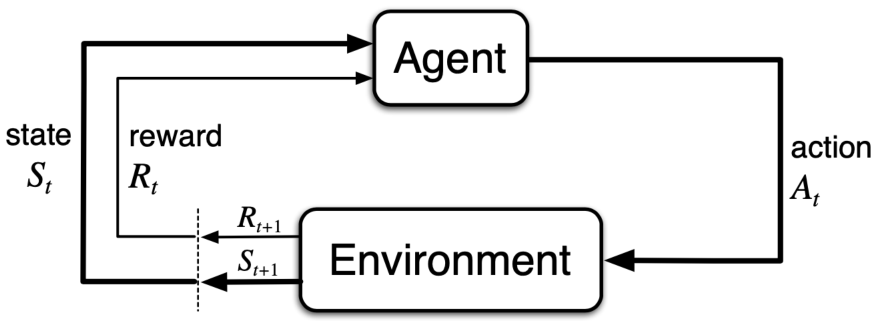


Figure 2.2: Reinforcement Learning; Source: [2]

In reinforcement learning, an agent learns to take actions (policy) based on measured environment state by interacting with the environment through trial and error for maximising a feedback signal (reward or value function). The main challenge in this method is requirement of large amount of data for agent to learn.

Collecting large amount of data from real robot will be slow and expensive and might require manual intervention. Instead, for faster and cheaper development, agent can be trained on simulated robot manipulation environment. However policies trained on simulated environments are not directly transferable to real robots due to various differences between simulation and real environments.

2.2.1 Comparing Task Simplifications to Learn Closed-Loop Object Picking

Work done by Michel Breyer et. al. [3] found that using autoencoder to reduce dimensionality of camera data and using curriculum learning reduced the training time of agents. Also by using shaped reward functions instead of sparse reward function, they obtained 98% success rate on simulated environment. They also found that using RANSAC for detecting and filtering surfaces from camera data while using policies trained from simulated environment on real robot gave 78% success rate. Figure 2.3 shows the network architecture used in this work.

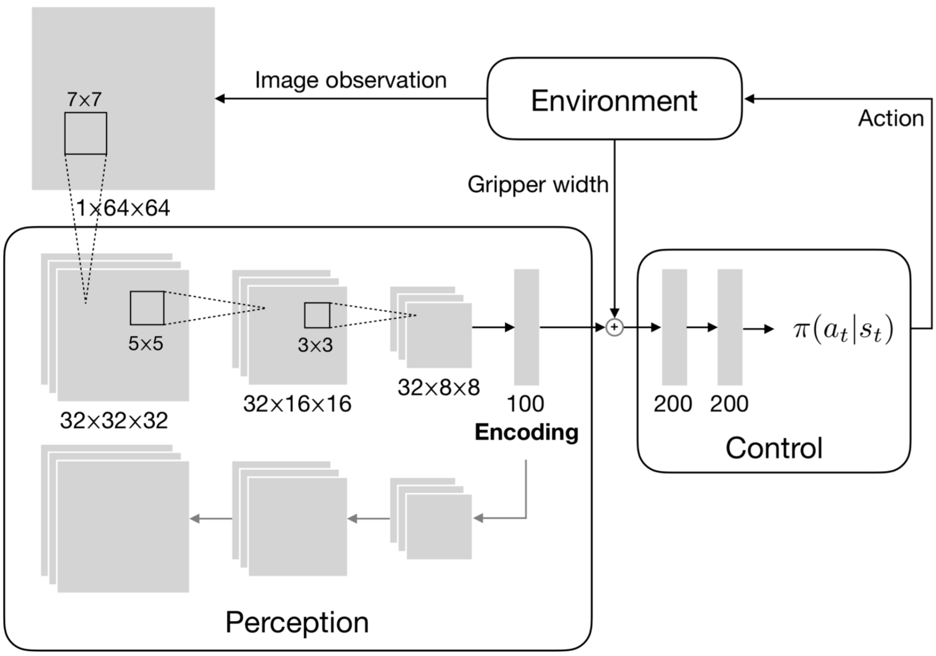


Figure 2.3: Comparing Task Simplifications to Learn Closed-Loop Object Picking: Network architecture; Source: [3]

2.3 Hierarchical reinforcement learning methods

Traditional reinforcement learning methods are data inefficient (require lot of data for training), difficult to scale (due to large action and/or state space) and brittle due to over specialisation (difficult to transfer their experience to new even similar environments) [4]. Hierarchical reinforcement learning are intended to address these issues by learning to operate on different levels of temporal abstraction.

2.3.1 Regularized Hierarchical Policies for Compositional Transfer in Robotics

This method proposes [5] hierarchical and modular policies for continuous control. The modular hierarchical policy used is defined as:

$$\pi_\theta(a|s,i) = \sum_{O=1}^{M} \pi_\theta^L(a|s,o)\pi_\theta^H(o|s,i) \tag{2.1}$$

Where π^H is the high level switching controller and π^L is the low level sub policy. The objective is to optimize

$$\underset{q}{max} J(q, \pi_{ref}) = E_{i \sim I}\left[E_{\pi, s\sim D}\left[\sum_{t=0}^{\infty}\gamma^t r_i(s_t, a_t)|s_{t+1} \sim p(.|_t, a_t)\right]\right] \tag{2.2}$$

subject to constraint

$$E_{s\sim D, i\sim I}\left[KL(q(.|s,i)||\pi_{ref}(.|s,i))\right] \leq \epsilon \tag{2.3}$$

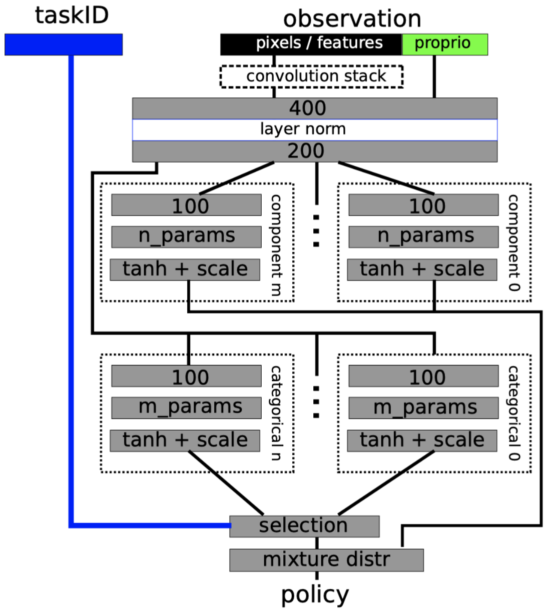


Figure 2.4: RHPO: Network architecture

Even though SURREAL provides an efficient framework which allows scaling reinforcement learning speed with computation power and includes a good collection of prebuilt standardized environments for common robot manipulation tasks, it does not provide prebuilt support for experiment logging and tracking and it does not provide integration with RayLib, a common framework used for reinforcement learning. RayLib also includes a good collection of standard reinforcement learning algorithms and has support for both Tensorflow and PyTorch deep learning frameworks.

Chapter 3

Proposed Method

Aim of this project is to develop a testbed that aids in the development of reinforcement learning algorithms for robot manipulation tasks. The test-bed should be able to scale the training speed with computational power by running multiple simulations in parallel. It should use standard frameworks that support training distributed RL algorithms. It should also have a flexible architecture that allows other researchers to add new types of robot manipulation task and test new RL algorithms.

Figure 3.1: Testbed methodology block diagram

Robot manipulation tasks can be evaluated on both simulation environment and real robot. The testbed will provide inbuilt support for simulated and real ABB IRB 120 robot. For training the RL algorithm, simulated environment will be used. This allows scaling training speed with computational power by running multiple instances of simulator in parallel. Policies learned from simulator can be evaluated on real ABB IRB 120 robot. A RGB-D camera will be mounted on the end-effector of robot. The only input to RL algorithm will be the RGB image with depth map from RGB-D camera. The RL algorithm will take this input and output the desired joint states of robot in order to achieve a particular task. Simulator / hardware will take the desired joint states as input and apply torques on joints to achieve desire joint state

3.1 Simulation setup

All reinforcement learning agents are first trained in a simulated environment. In this project the simulated environment is created using Bullet Physics Simulator.

3.1.1 Setup

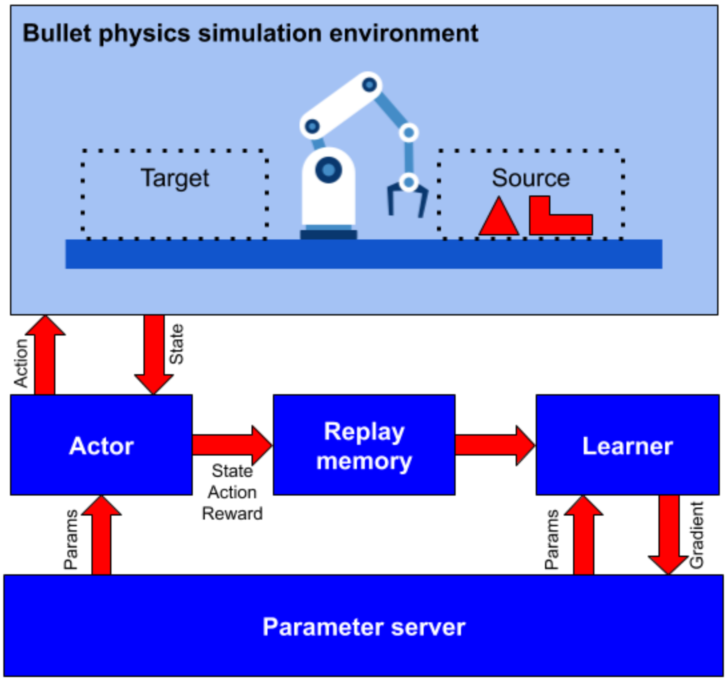


Figure 3.2: Standard simulation architecture

To reduce training time, agent training via simulation is parallelized and distributed similar to GORILA architecture [6]. In this project we use Ray RLLib [7] which provides abstractions for distributed reinforcement learning.

Environment

Simulation environment is developed using Bullet Physics simulator python module. Environment will be setup for a specific task like table clearing or stacking of objects. Environment will accept actions from specific task which must be in specific format according to action space of environment. When environment receives an action, it will apply the action to simulated environment using PyBullet APIs. After action is completed, the environment will record the state according to state space of environment and calculate the reward according to task setup of environment. Environment returns state and reward after action is applied.

PyBullet Simulator

PyBullet is a python module for Bullet Physics C SDK used for physics simulation in robotics, games, visual effects and machine learning, with a focus on sim-to-real transfer [8]. PyBullet can load visual, physical and other properties of a body from URDF, SDF and Mujoco formats. If required, 3D mesh of bodies can be loaded directly using PyBullet APIs. For simulating robots, PyBullet supports forward and inverse kinematics, collision detection, coordinate transformations, forward dynamics simulation, inverse dynamics computation, joint state position, velocity and force/torque control.

PyBullet supports rendering using CPU renderer and OpenGL renderer. Visualization can be optionally turned off in PyBullet. With visualization disabled, PyBullet uses CPU renderer for rendering images captured by camera API. This is especially useful in reinforcement learning where we want to collect data from simulated environment as fast as possible.

Actor

Actor is responsible for reading policy from the parameter server and execute actions is the simulation environment and save the action, returned state, returned reward in experience replay memory. When an actor is started, it will create a new simulation environment process.

- Start new simulation environment process
- Begin new episode
- Read policy ($\pi(a_t|s_t)$) and environment state (s_t). Evaluate and select action a_t for state s_t by evaluating policy $\pi(a_t|s_t)$. Execute the action a_t on simulation environment environment and read the returned reward r_t and new state of environment s_{t+1}. Save $[a_t, s_t, r_t, s_{t+1}]$ to replay buffer
- End episode when simulation environment terminates an episode

Replay buffer

Replay buffer stores the trajectory of episodes. Trajectory of an episode is a list of $[a_t, s_t, r_t, s_{t+1}]$. Since multiple actor process are supposed to add observations to replay buffer, replay buffer is usually a distributed database accessible by actor processes.

Learner

Learner reads the episode trajectories stored in replay buffer and optimize the policy stored in parameter server to maximize rewards. Learner process is specific to a reinforcement learning algorithm. Eg:- DDPG, PPO. Learner process uses deep learning frameworks like tensorflow or pytorch to model the policy, calculate the gradients for loss functions and do other computations.

Parameter server

Parameter server stores the policy. It is accessible by both actor and learner and also accepts gradient from learner and apply the gradient to update the policy represented by parameter server.

3.2 Hardware setup

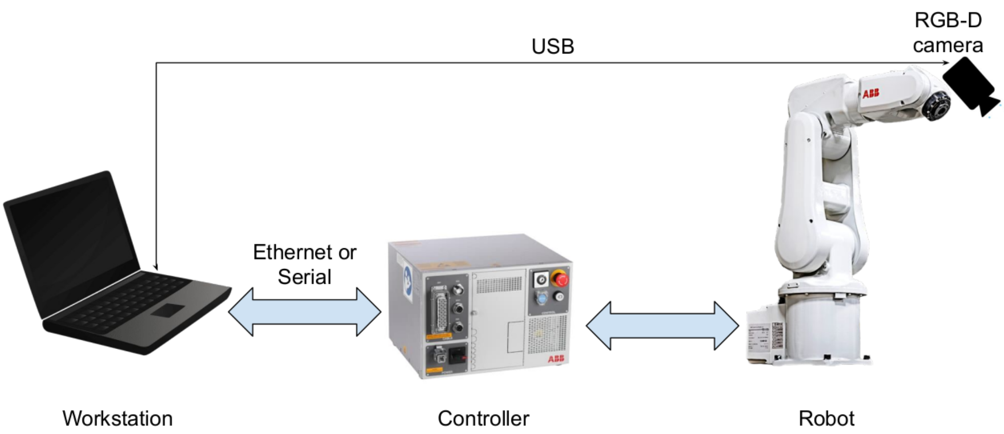


Figure 3.3: Hardware setup architecture

Policies learned from simulator will be evaluated on ABB IRB 120 robot. Depth camera mounted on endeffector will provide visual feedback. Camera will be directly connected to computer. Program running in the computer will directly connect with via serial or ethernet port in robot controller. RAPID program running on robot will read actions send from computer, execute it and send feedback to computer.

For ethernet connectivity, ABB robot studio license requires PC interface capability. This capability is not available in license available at robotics lab in CET. Also we were unable to establish bi-directional connection via virtual serial port between workstation and simulated ABB robot controller. Trying to establish bidirectional serial connection was also crashing the simulated ABB robot controller. So to keep it safe, we didn't tried to connect workstation and real ABB robot controller via serial port. This two limitations prevented us from connecting workstation with real robot controller and plan to integrate testbed with ABB IRB 120 hardware was dropped.

3.3 Reinforcement learning algorithm

Reinforcement learning algorithm takes RGB-D image as input and provides joint state as output. The testbed will have prebuilt support for following baseline RL algorithm.

3.3.1 Proximal Policy Optimization (PPO)

PPO is an on policy, stochastic and policy gradient based reinforcement learning method [9]. The objective function of PPO is

$$L^{CLIP}(\theta) = \hat{E}_t \left[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t) \right] \tag{3.1}$$

Where

- θ is the policy parameter
- $\hat{E}_t$ denotes the empirical expectation over timesteps
- r_t is the ratio of probability under the new and old policies, respectively
- $\hat{A}_t$ is the estimated advantage at time t
- ϵ is a hyperparameter, usually 0.1 or 0.2

Architecture

PPO requires two neural networks, value network to calculate advantage of taking an action and policy network for predicting mean and variance of an action to maximize reward at a given state. We use 3 layer CNN for extracting features from image of gripper camera. The input image shape is $84 \times 84 \times 4$. Fourth channel is depth value. The value network and policy network can share this 3 layer CNN for feature extraction. But since loss functions of value network and policy network are different, sharing the CNN layers might optimize the CNN layers for either value or policy network. This can be fixed by scaling the loss functions of value and policy network to same scale.

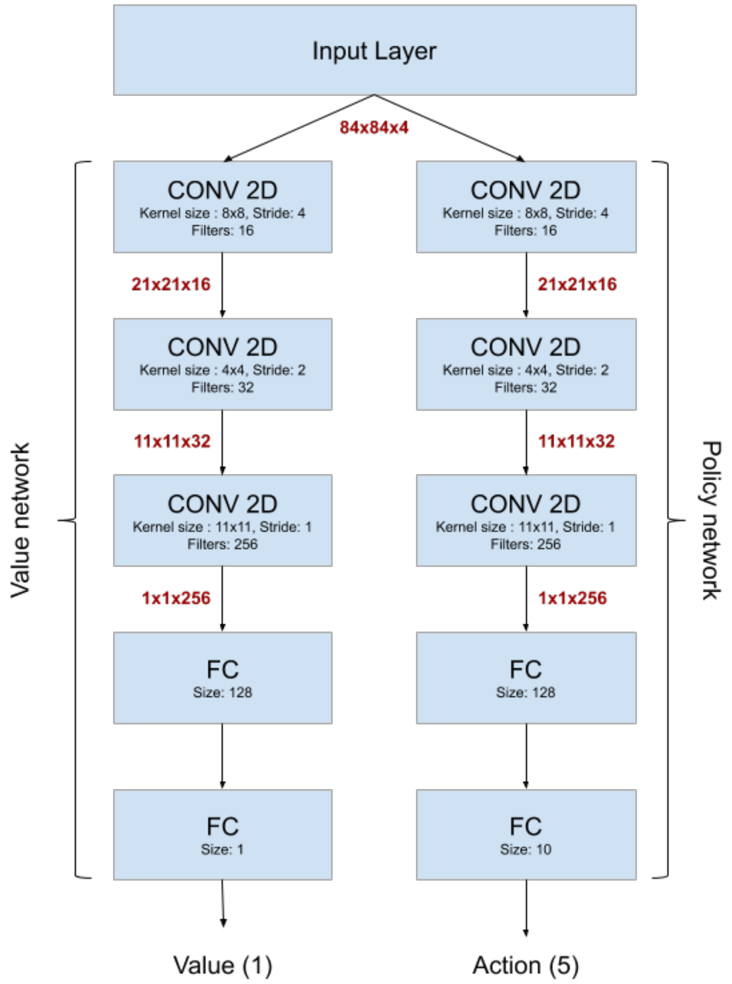


Figure 3.4: PPO network architecture

Chapter 4

Experimental setup

4.1 Simulation environment

4.1.1 Table Clearing Environment

In table clearing environment, main components are a 6 axis ABB IRB 120 robot with MetalWork W155032001 parallel jaw gripper and a table with yellow and green tray. Object at random position and orientation is inserted into the source tray (yellow). The task is to move the object from source tray (yellow) to destination tray (green).

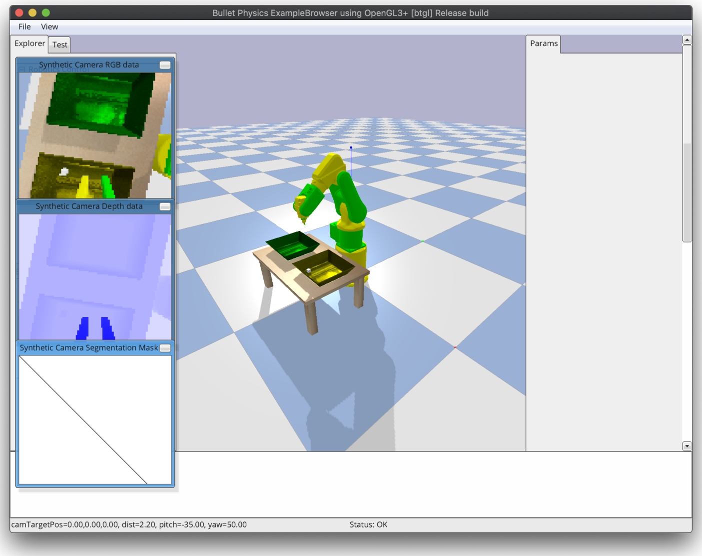


Figure 4.1: Table clearing pybullet environment

State space

A RGB-D camera is mounted on the end effector of the gripper. The output of this camera is shown in lFigure 4.2. The state space of the environment is a tensor of shape $84 \times 84 \times 4$. The first three channels are RGB pixel values and fourth channel is the depth value.

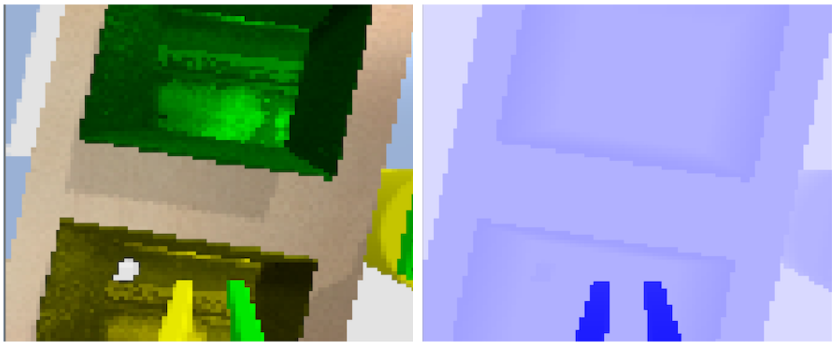


Figure 4.2: Gripper camera output; left: RGB, right: depth

Action space

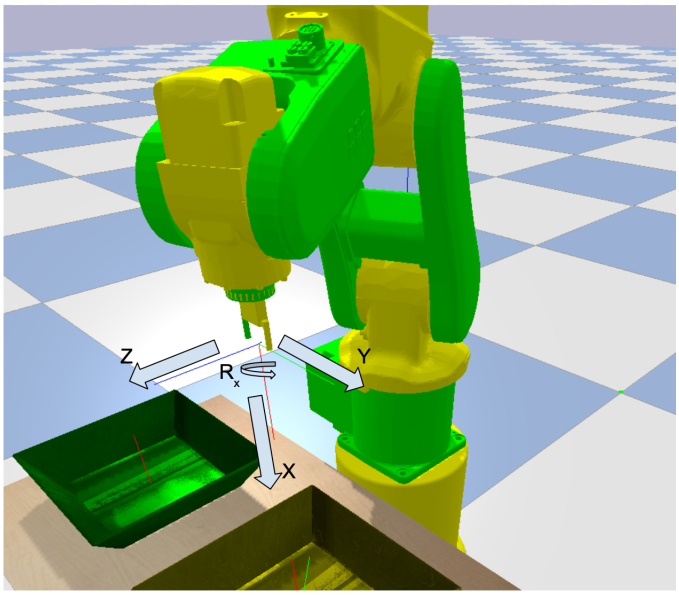


Figure 4.3: Table clearing environment action coordinate system

The end effector position can be controlled by making $[\delta x, \delta y, \delta z]$ movements with respect to $X, Y \& Z$ axis of a coordinate system attached to gripper as shown in

Figure 4.3. The end effector can be rotated about approach vector by δr_x. The gripper fingers can be closed by setting binary variable *open*. The maximum/minimum values of a single action is shown in Table 4.1

Action	Limit
$\delta x, \delta y, \delta z$	$[-1, 1]$ cm
δr_x	$[-10, 10] deg$
open	1, 0

Table 4.1: Table clearing environment action space

Reward

Let s_t and s_{t-1} be environment state at time t and $t-1$ respectively. The total reward returned by the environment for action a_t at time t is the sum of following rewards

- If object is not grasped and object has not moved far from initial position and gripper to object distance at time t is less than $t-1$, reward is +1 else -1
- If object is grasped and gripper to destination tray distance at time t is less than $t-1$, reward is +1 else -1
- For each action, reward of -1 is given to minimize time to complete task
- If body of robot including gripper touches any body other than target object, robot is assumed to be collided and a reward of -1000 is given
- If at time $t-1$, object is not grasped and at t, object is grasped, then a reward of +100 is given. Object is assumed to be grasped when target object is only in contact with gripper fingers and object is having a height of at least 5cm above source tray
- If at time $t-1$, object is grasped and at t, object is not grasped and target object is not at destination tray, object is assumed to be dropped and a reward of -200 is given
- If at time $t-1$, object is not at destination tray and at t, object is at destination tray, then object is assumed to be delivered and reward of +200 is given

Episode termination

A simulation episode is terminated at following conditions

- Robot or gripper body is in contact with any body other than target object (collision)
- Target object reached destination tray (delivered)
- Duration of episode is greater than 3 minutes
- Number of actions taken in episode is greater than 3000

4.2 Hardware specifications

4.2.1 Simulator benchmark

Simulator benchmark was done on a Lenovo Thinkpad X230 laptop with following specs

- Intel(R) Core(TM) i5-3320M CPU @ 2.60GHz processor
- 4GB RAM
- Integrated graphics

4.2.2 Testbed evaluation using PPO

Testbed evaluation using PPO was done on Azure cloud using Standard_NC12 VM with following specs

- Intel(R) Xeon(TM) E5-2690 v3 @ 2.60GHz processor x 12
- 112GB RAM
- 24GB NVIDIA K80 GPU

These VMs are optimized for compute-intensive algorithms like CUDA and OpenCL-based applications and simulations, AI, and Deep Learning.

Chapter 5

Experimental Results

In this chapter, the testbed is used for evaluation of various standard RL algorithms for vision based robot manipulation tasks.

5.1 Simulator performance

The simulation environment can be run in following configurations

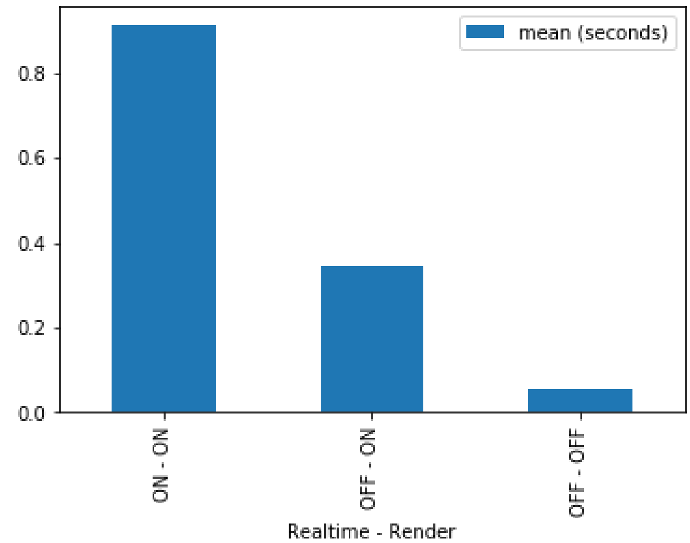
- Realtime mode - Simulation is run at realtime with rendering ON
- Fast rendering mode - Simulation is run rendering turned on but not at realtime
- Fast non rendering mode - Simulation is run with rendering and realtime turned OFF

The metric used for measuring performance of simulator is action time. Action time is the time taken by the simulator to execute a particular action

Table 5.1: Simulator performance (Mean action time in seconds) at different modes

Render — Realtime	**Mean**	**Std**	**Min**	**25%**	**50%**	**75%**	**Max**
ON — ON	0.912	2.783	0.138	0.358	0.374	0.39	26.33
ON — OFF	0.345	0.583	0.0793	0.198	0.206	0.214	4.342
OFF — OFF	0.053	0.023	0.041	0.046	0.047	0.048	0.201

Figure 5.1: Simulator performance at different modes (Mode vs Mean Action Time)



5.2 Table clearing environment

After training PPO agent on table clearing environment using 8 core 42GB RAM Nvidia T4 GPU system for 4M timesteps, results are shown below

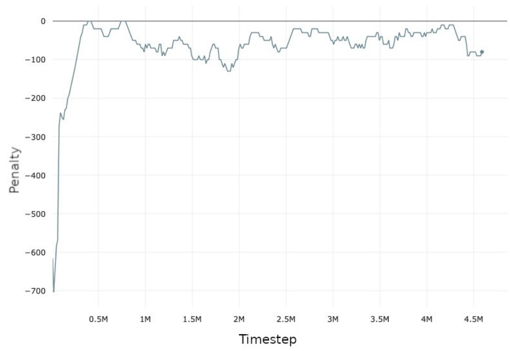


Figure 5.2: Collision penalty. Optimum value is 0

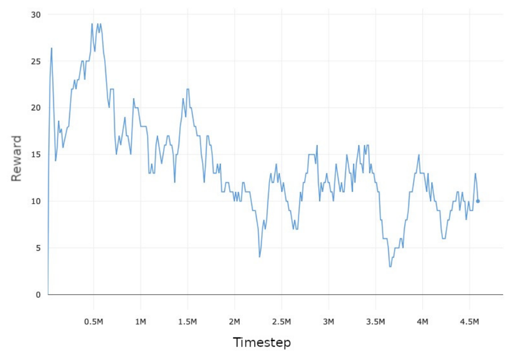


Figure 5.3: Grasp reward. Optimum value is 100

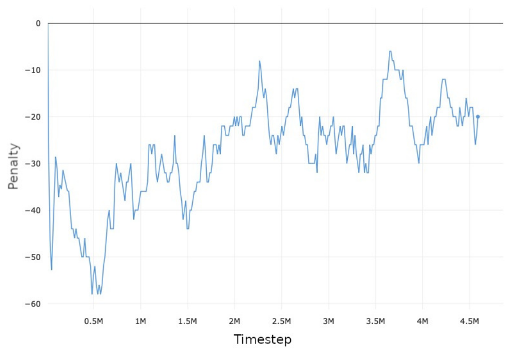


Figure 5.4: Drop penalty. Optimum value is 100

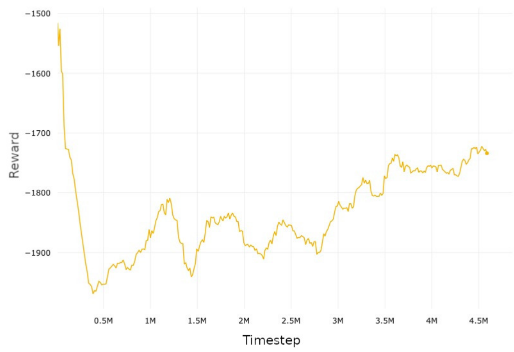


Figure 5.5: Mean episode reward. Optimum value ≥ 200

For attaining optimum reward, network needs to be trained for approximately 80M timesteps.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

A testbed that aids in the development and benchmarking of various RL algorithms for robot manipulation tasks is developed. By using RayLib, the testbed can scale training speed with computational power. Also the testbed is integrated with CometML platform for experiment logging and tracking. The testbed is evaluated using PPO RL algorithm and the simulator performance is benchmarked at various configurations. Evaluation using PPO RL algorithm showed its low sample efficiency and high reliance on reward shaping.

6.2 Future works

The current testbed is evaluated only using PPO algorithm. It would be beneficial to evaluate using other common RL algorithms like DDPG and RHPO. Also the testbed currently only provides table clearning robot manipulation task. Adding common tasks like stacking will be beneficial. Also evaluating the performance of policies trained using simulated environment on real robot will be beneficial

APPENDIX

Source code

URDF model of IRB 120 robot with MetalWorks gripper

Listing 6.1: geometry.xacro: Defines the geometric properties and constraints of different links in IRB 120 robot

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:property name="orig_foundation_link" value="0 0 0" />
  <xacro:property name="orig_foundation_joint" value="0 0 0.3" />

  <xacro:property name="orig_base_link" value="0 0 0" />
  <xacro:property name="orig_base_joint" value="0 0 0.166" />
  <xacro:property name="orig_base_joint_range" value="${[-pi*165/180, pi
      ↪ *165/180]}" />

  <xacro:property name="orig_link1" value="0 0 -0.166" />
  <xacro:property name="orig_link1_joint" value="0 0 0.124" />
  <xacro:property name="orig_link1_joint_range" value="${[-pi*110/180, pi
      ↪ *110/180]}" />

  <xacro:property name="orig_link2" value="0 -0.058 0.270" />
  <xacro:property name="orig_link2_joint" value="0 0 0.270" />
  <xacro:property name="orig_link2_joint_range" value="${[-pi*110/180, pi
      ↪ *70/180]}" />

  <xacro:property name="orig_link3" value="0.07196 0 -0.06615" />
  <xacro:property name="orig_link3_joint" value="0.14961 0 0.0706" />
  <xacro:property name="orig_link3_joint_range" value="${[-pi*160/180, pi
      ↪ *160/180]}" />

  <xacro:property name="orig_link4" value="0.1524 0 0" />
  <xacro:property name="orig_link4_joint" value="0.1524 0 0" />
  <xacro:property name="orig_link4_joint_range" value="${[-pi*120/180, pi
      ↪ *120/180]}" />
```

```
  <xacro:property name="orig_link5" value="-0.30886 0 -0.630" />
  <xacro:property name="orig_link5_joint" value="0.059 0 0" />
  <xacro:property name="orig_link5_joint_range" value="${[-pi, pi]}" />

  <xacro:property name="orig_link6" value="0 0 0" />
  <xacro:property name="orig_link6_rpy" value="0 ${pi/2} 0" />
  <xacro:property name="orig_link6_joint" value="0 0 0" />

  <xacro:property name="orig_gripper_base" value="0 0 0" />
  <xacro:property name="orig_gripper_base_rpy" value="${pi/2} 0 ${pi/2}" />
  <xacro:property name="orig_gripper_base_joint" value="0 0 0" />
  <xacro:property name="orig_gripper_base_joint_range" value="${[0, 0.012]}" />

  <xacro:property name="orig_gripper_finger1" value="0 0 0" />
  <xacro:property name="orig_gripper_finger1_rpy" value="${pi/2} 0 ${pi/2}" />

  <xacro:property name="orig_gripper_finger2" value="0 0 0" />
  <xacro:property name="orig_gripper_finger2_rpy" value="${-pi/2} 0 ${-pi/2}"
      ↪ />
</robot>
```

Listing 6.2: physics.xacro: Defines the mass and moment of inertia of different links in IRB 120 robot

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
    <xacro:property name="mass_base_link" value="6.215" />
    <xacro:property name="com_base_link" value="-0.04204 8.01E-05 0.07964" />
    <xacro:property name="com_base_link_rpy" value="0 0 0" />
    <xacro:property name="inertia_base_link" value='${dict(
        ixx="0.0247272", ixy="-8.0784E-05", ixz="0.00130902",
        iyy="0.0491285", iyz="-8.0419E-06", izz="0.0472376"
    )}' />

    <xacro:property name="mass_link1" value="3.067" />
    <xacro:property name="com_link1" value="9.77E-05 -0.00012 0.23841" />
    <xacro:property name="com_link1_rpy" value="0 0 0" />
    <xacro:property name="inertia_link1" value='${dict(
        ixx="0.0142175", ixy="-1.28579E-05", ixz="-2.31364E-05",
        iyy="0.0144041", iyz="1.93404E-05", izz="0.0104533"
    )}' />

    <xacro:property name="mass_link2" value="3.909" />
    <xacro:property name="com_link2" value="0.00078 -0.00212 0.10124" />
    <xacro:property name="com_link2_rpy" value="0 0 0" />
    <xacro:property name="inertia_link2" value='${dict(
        ixx="0.0603111", ixy="9.83431E-06", ixz="5.72407E-05",
        iyy="0.041569", iyz="-0.00050497", izz="0.0259548"
```

```
    )}' />

    <xacro:property name="mass_link3" value="2.944" />
    <xacro:property name="com_link3" value="0.02281 0.00106 0.05791" />
    <xacro:property name="com_link3_rpy" value="0 0 0" />
    <xacro:property name="inertia_link3" value='${dict(
        ixx="0.00835606", ixy="-8.01545E-05", ixz="0.00142884",
        iyy="0.016713", iyz="-0.000182227", izz="0.0126984"
    )}' />

    <xacro:property name="mass_link4" value="1.328" />
    <xacro:property name="com_link4" value="0.2247 0.00015 0.00041" />
    <xacro:property name="com_link4_rpy" value="0 0 0" />
    <xacro:property name="inertia_link4" value='${dict(
        ixx="0.00284661", ixy="-2.12765E-05", ixz="-1.6435E-05",
        iyy="0.00401346", iyz="1.31336E-05", izz="0.0052535"
    )}' />

    <xacro:property name="mass_link5" value="0.546" />
    <xacro:property name="com_link5" value="-0.00109 3.68E-05 6.22E-05" />
    <xacro:property name="com_link5_rpy" value="0 0 0" />
    <xacro:property name="inertia_link5" value='${dict(
         ixx="0.000404891", ixy="1.61943E-06", ixz="8.46805E-07",
         iyy="0.000892825", iyz="-1.51792E-08", izz="0.000815468"
    )}' />

    <xacro:property name="mass_link6" value="0.137" />
    <xacro:property name="com_link6" value="-0.00706 -0.00017 -1.32E-06" />
    <xacro:property name="com_link6_rpy" value="0 0 0" />
    <xacro:property name="inertia_link6" value='${dict(
         ixx="0.001", ixy="0", ixz="0",
         iyy="0.001", iyz="0", izz="0.001"
    )}' />
</robot>
```

Listing 6.3: macros.xacro: Helper xacro function to generate common urdf code of different links in IRB 120 robot

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:property name="orig_rpy" value="${pi/2} 0 0" />
  <xacro:property name="orig_xyz" value="0 0 0" />
  <xacro:property name="joint_axis" value="0 1 0" />
  <xacro:property name="joint_range" value="${[-pi, pi]}" />
  <xacro:property name="joint_xyz" value="0 0 0" />
  <xacro:property name="joint_rpy" value="0 0 0" />
  <xacro:property name="joint_type" value="revolute" />

  <xacro:macro name="irb_120_link" params="name mesh:=0 orig_rpy:=^ orig_xyz:=^
```

```
    ↪  parent:=0 joint_axis:=^ joint_range:=^ joint_xyz:=^ joint_rpy:=^
    ↪ joint_type:=^ mass:=0 com:=0 com_rpy:=0 inertia:=0 mesh_type=dae
    ↪ mesh_scale=1">
    <xacro:if value="${mesh == 0}">
      <xacro:property name="mesh" value="${name}" />
    </xacro:if>

    <link name="${name}">
      <visual>
        <geometry>
          <mesh filename="../cad/${mesh}.${mesh_type}" scale="${mesh_scale}" />
        </geometry>

        <origin rpy="${orig_rpy}" xyz="${orig_xyz}" />
      </visual>

      <collision>
        <geometry>
          <mesh filename="../cad/${mesh}.${mesh_type}" scale="${mesh_scale}" />
        </geometry>

        <origin rpy="${orig_rpy}" xyz="${orig_xyz}" />
      </collision>

      <xacro:unless value="${inertia==0}">
        <inertial>
          <mass value="${mass}" />
          <origin xyz="${com}" rpy="${com_rpy}" />
          <inertia ixx="${inertia['ixx']}" ixy="${inertia['ixy']}"
                   ixz="${inertia['ixz']}" iyy="${inertia['iyy']}"
                   iyz="${inertia['iyz']}" izz="${inertia['izz']}"
          />
        </inertial>
      </xacro:unless>

    </link>

    <xacro:unless value="${parent == 0}">
      <joint name="joint_${parent}_${name}" type="${joint_type}">
        <axis xyz="${joint_axis}"/>
        <limit effort="1000.0" lower="${joint_range[0]}" upper="${joint_range
            ↪ [1]}" velocity="0.5"/>
        <origin rpy="${joint_rpy}" xyz="${joint_xyz}"/>
        <parent link="${parent}"/>
        <child link="${name}"/>
      </joint>
    </xacro:unless>
  </xacro:macro>
</robot>
```

Listing 6.4: gripper.xacro: URDF of MetalWork W155032001 gripper

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro">
  <xacro:include filename="geometry.xacro" />
  <xacro:include filename="macros.xacro" />

  <xacro:irb_120_link
    name="gripper_base"
    orig_xyz="${orig_gripper_base}"
    orig_rpy="${orig_gripper_base_rpy}"
    parent="link6"
    joint_type="fixed"
    joint_xyz="${orig_link6_joint}"
  />

  <link name="gripper_finger1">
    <contact>
      <lateral_friction value="1"/>
      <spinning_friction value="1"/>
    </contact>
    <collision>
      <geometry>
        <box size="0.0025 0.005 0.025" />
      </geometry>
      <origin rpy="1.5707963267948966 0 1.5707963267948966" xyz="0.0445 -0.0025
          ↪ 0"/>
    </collision>
    <visual>
      <geometry>
          <mesh filename="../cad/gripper_finger.dae" scale="1"/>
      </geometry>
      <origin rpy="1.5707963267948966 0 1.5707963267948966" xyz="0 0 0"/>
    </visual>
  </link>

  <joint name="joint_gripper_base_gripper_finger1" type="prismatic">
    <axis xyz="0 -1 0"/>
    <limit effort="1000.0" lower="0" upper="0.012" velocity="0.5"/>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <parent link="gripper_base"/>
    <child link="gripper_finger1"/>
    <dynamics damping="0" friction="0" spring_reference="0" spring_stiffness="0"
        ↪  />
  </joint>

  <link name="gripper_finger2">
```

```
    <contact>
      <lateral_friction value="1"/>
      <spinning_friction value="1"/>
    </contact>
    <collision>
      <geometry>
        <box size="0.0025 0.005 0.025" />
      </geometry>
      <origin rpy="-1.5707963267948966 0 -1.5707963267948966" xyz="0.0445 0.0025
          ↪  0"/>
    </collision>
    <visual>
      <geometry>
          <mesh filename="../cad/gripper_finger.dae" scale="1"/>
      </geometry>
      <origin rpy="-1.5707963267948966 0 -1.5707963267948966" xyz="0 0 0"/>
    </visual>
  </link>

  <joint name="joint_gripper_base_gripper_finger2" type="prismatic">
    <axis xyz="0 1 0"/>
    <limit effort="1000.0" lower="0" upper="0.012" velocity="0.5"/>
    <origin rpy="0 0 0" xyz="0 0 0"/>
    <parent link="gripper_base"/>
    <child link="gripper_finger2"/>
    <dynamics damping="0" friction="0" spring_reference="0" spring_stiffness="0"
        ↪  />
  </joint>
</robot>
```

Listing 6.5: main.xacro: Generate URDF of IRB 120 robot with MetalWork W155032001 gripper

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://ros.org/wiki/xacro" name="irb_120">
  <xacro:include filename="geometry.xacro" />
  <xacro:include filename="physics.xacro" />
  <xacro:include filename="macros.xacro" />

  <xacro:irb_120_link
    name="foundation"
    orig_xyz="${orig_foundation_link}"
    orig_rpy="0 0 0"
    mesh="foundation"
    mesh_type="stl"
    mesh_scale="0.001"
  />

  <xacro:irb_120_link
```

```
  name="base"
  parent="foundation"
  orig_xyz="${orig_base_link}"
  joint_type="fixed"
  joint_xyz="${orig_foundation_joint}"
  mass="${mass_base_link}"
  com="${com_base_link}"
  com_rpy="${com_base_link_rpy"
  inertia="${inertia_base_link}"
/>

<xacro:irb_120_link
  name="link1"
  parent="base"
  orig_xyz="${orig_link1}"
  joint_axis="0 0 1"
  joint_xyz="${orig_base_joint}"
  joint_range="${orig_base_joint_range}"
  mass="${mass_link1}"
  com="${com_link1}"
  com_rpy="${com_link1_rpy}"
  inertia="${inertia_link1}"
/>

<xacro:irb_120_link
  name="link2"
  parent="link1"
  orig_xyz="${orig_link2}"
  joint_axis="0 1 0"
  joint_xyz="${orig_link1_joint}"
  joint_range="${orig_link1_joint_range}"
  mass="${mass_link2}"
  com="${com_link2}"
  com_rpy="${com_link2_rpy}"
  inertia="${inertia_link2}"
/>

<xacro:irb_120_link
  name="link3"
  parent="link2"
  orig_xyz="${orig_link3}"
  joint_axis="0 1 0"
  joint_xyz="${orig_link2_joint}"
  joint_range="${orig_link2_joint_range}"
  mass="${mass_link3}"
  com="${com_link3}"
  com_rpy="${com_link3_rpy}"
  inertia="${inertia_link3}"
/>
```

```xml
  <xacro:irb_120_link
    name="link4"
    parent="link3"
    orig_xyz="${orig_link4}"
    joint_axis="1 0 0"
    joint_xyz="${orig_link3_joint}"
    joint_range="${orig_link3_joint_range}"
    mass="${mass_link4}"
    com="${com_link4}"
    com_rpy="${com_link4_rpy}"
    inertia="${inertia_link4}"
  />

  <xacro:irb_120_link
    name="link5"
    parent="link4"
    orig_xyz="${orig_link5}"
    joint_axis="0 1 0"
    joint_xyz="${orig_link4_joint}"
    joint_range="${orig_link4_joint_range}"
    mass="${mass_link5}"
    com="${com_link5}"
    com_rpy="${com_link5_rpy}"
    inertia="${inertia_link5}"
  />

  <xacro:irb_120_link
    name="link6"
    parent="link5"
    orig_xyz="${orig_link6}"
    orig_rpy="${orig_link6_rpy}"
    joint_axis="1 0 0"
    joint_xyz="${orig_link5_joint}"
    joint_range="${orig_link5_joint_range}"
    mass="${mass_link6}"
    com="${com_link6}"
    com_rpy="${com_link6_rpy}"
    inertia="${inertia_link6}"
  />

  <xacro:include filename="gripper.xacro" />
</robot>
```

PyBullet Simulation environments

Listing 6.6: env/base.py: Base class for simulation environments

```python
import time

import cv2
import pybullet as pb

from ..sensors.camera import Camera


class Environment(object):

    def __init__(self, pb_client=pb, step_size=1./240., realtime=True):
        self._pb_client = pb_client

        self._step_size = step_size
        self._last_step_time = time.time()
        self._realtime = realtime

        self._setup()
        self._timer = 0.0

        self._camera = Camera(
            pb_client,
            view_calculator=lambda: ((2, 2, 2), (-2, -2, -2), (0, 0, 10)),
            resolution=(600, 600)
        )

    def spin(self, count=None):
        while count is None or count > 0:
            self.step()
            count = count - 1 if count is not None else None

    def step(self):
        self._pb_client.stepSimulation()
        self._timer += self._step_size
        current = time.time()
        elapsed = current - self._last_step_time
        to_sleep = max(0., self._step_size - elapsed)

        if self._realtime and to_sleep > 0:
            time.sleep(to_sleep)

        self._last_step_time = current + to_sleep

    def _setup(self):
        self._step = 0

    def reset(self):
        self._pb_client.resetSimulation()
```

```
        self._setup()

    def camera(self):
        return self._camera

    @property
    def pb_client(self):
        return self._pb_client

    @property
    def time(self):
        return self._timer
```

Listing 6.7: env/irb120.py: Simple test environment with IRB 120 robot for debugging URDF model

```
import os
import time

import numpy as np
import pybullet as pb
import pybullet_data

from ..data import abb_irb120
from ..utils import assert_exist


REVOLUTE_JOINT_INDICES = (1, 2, 3, 4, 5, 6)
GRIPPER_INDEX = 7
GRIPPER_FINGER_INDICES = (8, 9)
MOVABLE_JOINT_INDICES = REVOLUTE_JOINT_INDICES + GRIPPER_FINGER_INDICES


class IRB120(object):

    def __init__(self, pb_client=pb, gravity=(0, 0, -9.81), realtime=True,
        ↪ joint_state_tolerance=1e-3,
                gripper_max_force=250.):
        self._urdf_robot = abb_irb120()
        assert_exist(self._urdf_robot)

        self._urdf_plane = os.path.join(pybullet_data.getDataPath(), "plane.urdf
            ↪ ")
        assert_exist(self._urdf_plane)

        assert pb_client is not None
        self._pb_client = pb_client

        self._gravity = gravity
```

```
        self._realtime = realtime
        self._joint_state_tolerance = joint_state_tolerance
        self._gripper_max_force = gripper_max_force

        self._setup()

    def _setup(self):
        is_connected, _ = self._pb_client.getConnectionInfo()
        if not is_connected:
            raise Exception('Bullet physics client not connected')

        gx, gy, gz = self._gravity
        self._pb_client.setGravity(gx, gy, gz)

        self._plane_id = self._pb_client.loadURDF(self._urdf_plane)
        self._robot_id = self._pb_client.loadURDF(self._urdf_robot, useFixedBase
            ↪ =True)

        self._pb_client.enableJointForceTorqueSensor(self._robot_id,
            ↪ GRIPPER_FINGER_INDICES[0])
        self._pb_client.enableJointForceTorqueSensor(self._robot_id,
            ↪ GRIPPER_FINGER_INDICES[1])

    @property
    def joint_states(self):
        return np.array([i[0] for i in self._pb_client.getJointStates(self.
            ↪ _robot_id, MOVABLE_JOINT_INDICES)])

    @property
    def revolute_joint_states(self):
        return np.array([i[0] for i in self._pb_client.getJointStates(self.
            ↪ _robot_id, REVOLUTE_JOINT_INDICES)])

    @property
    def gripper_finger_joint_states(self):
        return np.array([i[0] for i in self._pb_client.getJointStates(self.
            ↪ _robot_id, GRIPPER_FINGER_INDICES)])

    @property
    def gripper_pose(self):
        gripper_state = self._pb_client.getLinkState(self._robot_id,
            ↪ GRIPPER_INDEX)
        px, py, pz = gripper_state[0]
        ax, ay, az = pb.getEulerFromQuaternion(gripper_state[1])

        return np.array([px, py, pz, ax, ay, az])

    @property
    def joint_state(self):
```

```
        return np.array([i[0] for i in self._pb_client.getJointStates(self.
            ↪ _robot_id, REVOLUTE_JOINT_INDICES)])

    @property
    def gripper_state(self):
        return np.array([i[0] for i in self._pb_client.getJointStates(self.
            ↪ _robot_id, GRIPPER_FINGER_INDICES)])

    def reset(self):
        self._pb_client.resetSimulation()
        self._setup()

    def move_absolute(self, pose):
        px, py, pz, ax, ay, az = pose

        joint_states = pb.calculateInverseKinematics(
            self._robot_id,
            GRIPPER_INDEX,
            (px, py, pz),
            pb.getQuaternionFromEuler((ax, ay, az))
        )[:-2]

        self._move_joint(joint_states)

    def move_relative(self, pose_diff):
        dpx, dpy, dpz, dax, day, daz = pose_diff
        px, py, pz, ax, ay, az = self.gripper_pose

        return self.move_absolute((
            px + dpx, py + dpy, pz + dpz,
            ax + dax, ay + day, az + daz,
        ))

    def _move_joint(self, joint_states):
        assert len(REVOLUTE_JOINT_INDICES) == len(joint_states)

        self._pb_client.setJointMotorControlArray(
            self._robot_id,
            REVOLUTE_JOINT_INDICES,
            pb.POSITION_CONTROL,
            joint_states
        )

        self._wait_for_joint_state(joint_states)

    def _wait_for_joint_state(self, target_state):
        while np.any(np.abs(self.joint_state - target_state) > self.
            ↪ _joint_state_tolerance):
            self._tick()
```

```python
def hold_gripper(self):
    self._move_gripper([0, 0])
    self._wait_for_gripper_hold()

def release_gripper(self):
    self._move_gripper([0.012, 0.012])
    self._wait_for_gripper_open()

def _move_gripper(self, position):
    assert len(position) == len(GRIPPER_FINGER_INDICES)

    self._pb_client.setJointMotorControlArray(
        self._robot_id,
        GRIPPER_FINGER_INDICES,
        pb.POSITION_CONTROL,
        position,
        forces=(self._gripper_max_force, ) * 2
    )

def _wait_for_gripper_open(self):
    ul1 = self._pb_client.getJointInfo(self._robot_id,
        ↪ GRIPPER_FINGER_INDICES[0])[9]
    ul2 = self._pb_client.getJointInfo(self._robot_id,
        ↪ GRIPPER_FINGER_INDICES[1])[9]
    while np.any(np.abs(self.gripper_state - [ul1, ul2]) > self.
        ↪ _joint_state_tolerance):
        self._tick()

def _wait_for_gripper_hold(self):
    while True:
        self._tick()
        fx, fy, fz, _, _, _ = self._pb_client.getJointState(self._robot_id,
            ↪ GRIPPER_FINGER_INDICES[1])[2]
        diff = np.abs(np.sqrt(np.sum(np.array([fx, fy, fz]))**2) - 9.81)

        if diff > 0.05:
            break

def spin(self):
    while True:
        self._tick()

def _tick(self):
    if self._realtime:
        time.sleep(1./240.)

    self._pb_client.stepSimulation()
```

Listing 6.8: env/table_clearing.py: Table clearing reinforcement learning environment for IRB 120 robot with MetalWorks gripper environments

```
import math
import uuid

import pybullet as pb
import numpy as np

from . import base
from ..entity.ground import Ground
from ..entity.irb120 import IRB120, GRIPPER_FINGER_INDICES, GRIPPER_INDEX,
    ↪ GRIPPER_ORIGIN_OFFSET
from ..entity.table import Table
from ..entity.tray import Tray
from .. import interrupts
from ..interrupts import CollisionInterrupt, TimeoutInterrupt


class Environment(base.Environment):

    def __init__(self, *args, debug=False, **kwargs):
        self._debug = debug
        super(Environment, self).__init__(*args, **kwargs)

    def _setup(self):
        super(Environment, self)._setup()

        self._pb_client.setGravity(0, 0, -9.81)

        self._ground = Ground(self._pb_client)
        self._robot = IRB120(self._pb_client, debug=self._debug)
        self._table = Table(self._pb_client, position=(0, -0.4, 0), scale=0.5)

        self._src_tray = Tray(self._pb_client, position=(0.2, -0.4, self._table.
            ↪ z_end),
                              scale=0.5, color=(1, 1, 0, 1), debug=self._debug)
        self._dest_tray = Tray(self._pb_client, position=(-0.2, -0.4, self.
            ↪ _table.z_end),
                               scale=0.5, color=(0, 1, 0, 1), debug=self._debug)

        if self._debug:
            self._pb_client.addUserDebugLine(
                (GRIPPER_ORIGIN_OFFSET, 0, 0),
                (GRIPPER_ORIGIN_OFFSET + 0.1, 0, 0),
                (1, 0, 0),
                parentObjectUniqueId=self.robot.id,
                parentLinkIndex=GRIPPER_INDEX
            )
```

```
            self._pb_client.addUserDebugLine(
                (GRIPPER_ORIGIN_OFFSET, 0, 0),
                (GRIPPER_ORIGIN_OFFSET, 0.1, 0),
                (0, 1, 0),
                parentObjectUniqueId=self.robot.id,
                parentLinkIndex=GRIPPER_INDEX
            )

            self._pb_client.addUserDebugLine(
                (GRIPPER_ORIGIN_OFFSET, 0, 0),
                (GRIPPER_ORIGIN_OFFSET, 0, 0.1),
                (0, 0, 1),
                parentObjectUniqueId=self.robot.id,
                parentLinkIndex=GRIPPER_INDEX
            )

    def step(self):
        super(Environment, self).step()
        self.robot.update_state()

        if self._debug:

            self._pb_client.addUserDebugLine(
                self.robot.gripper_pose[0],
                self.robot.gripper_pose[0] + 0.002,
                lineColorRGB=(0, 0, 0),
                lineWidth=3,
                lifeTime=10.
            )

    def new_episode(self, *args, **kwargs):
        return Episode(self, *args, **kwargs)

    @property
    def robot(self):
        return self._robot

    @property
    def src_tray(self):
        return self._src_tray

    @property
    def dest_tray(self):
        return self._dest_tray


class Action(object):
```

```python
    def __init__(self, dx=0., dy=0., dz=0., dyaw=0., dpitch=0., droll=0.,
        ↪ open_gripper=True,
                 x=None, y=None, z=None, yaw=None, pitch=None, roll=None):
        self._dx = dx
        self._dy = dy
        self._dz = dz

        self._dyaw = dyaw
        self._dpitch = dpitch
        self._droll = droll

        self._open_gripper = open_gripper

        self._x = x
        self._y = y
        self._z = z

        self._yaw = yaw
        self._pitch = pitch
        self._roll = roll

    def apply(self, env: Environment):
        current_position, current_orientation = env.robot.gripper_pose

        position, orientation = env.pb_client.multiplyTransforms(
            current_position,
            current_orientation,
            (self._dx, self._dy, self._dz),
            env.pb_client.getQuaternionFromEuler((self._dyaw, self._dpitch, self
                ↪ ._droll))
        )

        if self._x is not None:
            position[0] = self._x

        if self._y is not None:
            position[1] = self._y

        if self._z is not None:
            position[3] = self._z

        orientation = np.array(env.pb_client.getEulerFromQuaternion(orientation)
            ↪ )

        if self._yaw is not None:
            orientation[0] = self._yaw

        if self._pitch is not None:
            orientation[1] = self._pitch
```

```
        if self._roll is not None:
            orientation[2] = self._roll

        move_interrupt = env.robot.set_gripper_pose(position, env.pb_client.
            ↪ getQuaternionFromEuler(orientation))
        finger_interrupt = env.robot.set_gripper_finger(self._open_gripper)

        return interrupts.all(move_interrupt, finger_interrupt)


class Episode(object):

    _env: Environment

    def __init__(self, env, target_position=None, target_orientation=None):
        self._id = uuid.uuid4()
        self._env = env
        self._start_time = self._env.time
        self._num_actions = 0

        if target_position is None or target_orientation is None:
            target_position, target_orientation = self._generate_target_pose()

        self._target = self._env.src_tray.add_cube(target_position,
            ↪ target_orientation)
        self._collision_interrupt = CollisionInterrupt(self._env.robot.id, [self
            ↪ ._target.id])

    def _generate_target_pose(self):
        tray = self._env.src_tray

        x_range = tray.x_span / 2 - 0.025
        y_range = tray.y_span / 2 - 0.025
        z_min = 0.05
        z_max = 0.15

        position = np.random.uniform([-x_range, -y_range, z_min], [x_range,
            ↪ y_range, z_max])
        orientation = np.random.uniform([0, 0, 0], [2*math.pi] * 3)

        return position, pb.getQuaternionFromEuler(orientation)

    def act(self, action: Action, timeout=5.):
        self._num_actions += 1
        interrupt = action.apply(self._env)
        interrupt = interrupts.any(self._collision_interrupt, TimeoutInterrupt(
            ↪ self.env, timeout), interrupt)
        interrupt.spin(self._env)
```

```python
    def state(self):
        return EpisodeState(self)

    @property
    def id(self):
        return self._id

    @property
    def env(self):
        return self._env

    @property
    def target(self):
        return self._target

    def cleanup(self):
        return self._target.remove()

    @property
    def start_time(self):
        return self._start_time

    @property
    def num_actions(self):
        return self._num_actions


class EpisodeState(object):

    def __init__(self, episode: Episode):
        self._episode = episode

        self._gripper_pos, _ = self._episode.env.robot.gripper_pose
        self._target_pos = self._episode.target.position

        self._d_target_gripper = self._calc_d_target_gripper()
        self._d_gripper_src_tray = self._calc_d_gripper_src_tray()
        self._d_gripper_dest_tray = self._calc_d_gripper_dest_tray()

        self._grasped = self._calc_grasped()
        self._collided = self._calc_collided()

        self._gripper_cam = self._episode.env.robot.capture_gripper_camera()
        self._time = episode.env.time

        self._reached_src_tray = self._calc_reached_src_tray()
        self._reached_dest_tray = self._calc_reached_dest_tray()
        self._done = self._calc_done()
```

```python
def _calc_d_target_gripper(self):
    return np.linalg.norm(self._gripper_pos - self._target_pos)

def _calc_d_gripper_src_tray(self):
    return np.linalg.norm(np.array(self._gripper_pos) - self._episode.env.
        ↪ src_tray.position)

def _calc_d_gripper_dest_tray(self):
    return np.linalg.norm(self._gripper_pos - self._episode.env.dest_tray.
        ↪ position)

def _calc_grasped(self):
    contact_f1 = len(self._episode.env.pb_client.getContactPoints(
        bodyA=self._episode.env.robot.id,
        bodyB=self._episode.target.id,
        linkIndexA=GRIPPER_FINGER_INDICES[0]
    )) > 0

    contact_f2 = len(self._episode.env.pb_client.getContactPoints(
        bodyA=self._episode.env.robot.id,
        bodyB=self._episode.target.id,
        linkIndexA=GRIPPER_FINGER_INDICES[1]
    )) > 0

    target_min_z = min(self._episode.target.z_start, self._episode.target.
        ↪ z_end)
    raised = target_min_z - self._episode.env.src_tray.z_start > 0.01

    other_contacts = len([i for i in self._episode.env.pb_client.
        ↪ getContactPoints(
        bodyA=self._episode.target.id) if i[2] != self._episode.env.robot.id
            ↪ ]) > 0

    return contact_f1 and contact_f2 and raised and not other_contacts

def _calc_collided(self):
    points = self._episode.env.robot.contact_points()
    exceptions = [self._episode.target.id]
    collisions = [p[2] for p in points if p[2] not in exceptions]
    return len(collisions) > 0

def _calc_reached_src_tray(self):
    src_tray = self._episode.env.src_tray
    return self._d_gripper_src_tray < (min(src_tray.x_span, src_tray.y_span)
        ↪  / 2.0) - 0.01

def _calc_reached_dest_tray(self):
    dest_tray = self._episode.env.dest_tray
```

```
        return self._d_gripper_dest_tray < (min(dest_tray.x_span, dest_tray.
            ↪ y_span) / 2.0) - 0.01

    def _calc_done(self):
        return self._collided or self._reached_dest_tray

    @property
    def gripper_pos(self):
        return self._gripper_pos

    @property
    def d_target_gripper(self):
        return self._d_target_gripper

    @property
    def d_gripper_src_tray(self):
        return self._d_gripper_src_tray

    @property
    def d_gripper_dest_tray(self):
        return self._d_gripper_dest_tray

    @property
    def grasped(self):
        return self._grasped

    @property
    def collided(self):
        return self._collided

    @property
    def reached_src_tray(self):
        return self._reached_src_tray

    @property
    def reached_dest_tray(self):
        return self._reached_dest_tray

    @property
    def done(self):
        return self._done

    @property
    def gripper_camera(self):
        return self._gripper_cam

    @property
    def time(self):
        return self._time
```

```
    def __str__(self):
        return '%s: %s' % (self.__class__, {
            'd_tg': self.d_target_gripper, 'd_gd': self.d_gripper_dest_tray, '
                ↪ grasped': self.grasped, 'collided': self.collided
        })
```

Listing 6.9: entity/base.py: Base class for objects in simulation environment

```
import pybullet as pb
import numpy as np


class Entity(object):

    def __init__(self, urdf, pb_client=pb, position=(0, 0, 0), orientation=(0,
        ↪ 0, 0, 1),
                 fixed_base=False, scale=1, debug=False):
        self._debug = debug
        self._pb_client = pb_client
        self._id = pb_client.loadURDF(
            urdf,
            basePosition=position,
            baseOrientation=orientation,
            useFixedBase=fixed_base,
            globalScaling=scale
        )
        self._position = position
        self._orientation = orientation

        if debug:
            self._pb_client.addUserDebugLine(
                (0, 0, 0),
                (0, 0, 0.1),
                (1, 0, 0),
                parentObjectUniqueId=self.id
            )

            self._pb_client.addUserDebugLine(
                (0, 0, 0),
                (0, 0.1, 0),
                (0, 1, 0),
                parentObjectUniqueId=self.id
            )

            self._pb_client.addUserDebugLine(
                (0, 0, 0),
                (0.1, 0, 0),
                (0, 0, 1),
```

```
            parentObjectUniqueId=self.id
        )

    @property
    def id(self):
        return self._id

    @property
    def pose(self):
        pos, orientation = self._pb_client.getBasePositionAndOrientation(self.
            ↪ _id)
        return np.array(pos), np.array(orientation)

    @property
    def position(self):
        return self.pose[0]

    @property
    def orientation(self):
        return self.pose[1]

    @property
    def orientation_euler(self):
        return self._pb_client.getEulerFromQuaternion(self.orientation)

    @property
    def bounding_box(self):
        start, end = self._pb_client.getAABB(self._id)
        return np.array(start), np.array(end)

    @property
    def x_start(self):
        return self.bounding_box[0][0]

    @property
    def x_end(self):
        return self.bounding_box[0][1]

    @property
    def y_start(self):
        return self.bounding_box[0][1]

    @property
    def y_end(self):
        return self.bounding_box[1][1]

    @property
    def z_start(self):
        return self.bounding_box[0][2]
```

```
    @property
    def z_end(self):
        return self.bounding_box[1][2]

    @property
    def x_span(self):
        return self._axis_span(0)

    @property
    def y_span(self):
        return self._axis_span(1)

    @property
    def z_span(self):
        return self._axis_span(2)

    def _axis_span(self, axis):
        start, end = self.bounding_box
        return end[axis] - start[axis]

    def transform(self, position, orientation):
        return self._pb_client.multiplyTransforms(self.position, self.
            ↪ orientation, position, orientation)

    def contact_points(self):
        return self._pb_client.getContactPoints(self._id)

    def remove(self):
        self._pb_client.removeBody(self._id)
```

Listing 6.10: entity/table.py: Loads table object into simulation environment

```
import os

import pybullet as pb
import pybullet_data

from .base import Entity


class Table(Entity):

    def __init__(self, pb_client=pb, position=(0, 0, 0), orientation=(0, 0, 0,
        ↪ 1), fix_base=True, scale=1., **kwargs):
        table = os.path.join(pybullet_data.getDataPath(), 'table', 'table.urdf')
        super(Table, self).__init__(table, pb_client, position, orientation,
            ↪ fix_base, scale, **kwargs)
```

Listing 6.11: entity/tray.py: Loads tray object into simulation environment

```
import os
import numpy as np

import pybullet as pb
import pybullet_data

from .base import Entity
from .cube import Cube


class Tray(Entity):

    def __init__(self, pb_client=pb, position=(0, 0, 0), orientation=(0, 0, 0,
        ↪ 1), fix_base=False,
                scale=1., color=None, **kwargs):
        table = os.path.join(pybullet_data.getDataPath(), 'tray', 'traybox.urdf'
            ↪ )
        super(Tray, self).__init__(table, pb_client, position, orientation,
            ↪ fix_base, scale, **kwargs)

        if color is not None:
            self._pb_client.changeVisualShape(self._id, -1, rgbaColor=color)

    def add_cube(self, position, orientation):
        position, orientation = self.transform(position, orientation)
        return Cube(pb_client=self._pb_client, position=position, orientation=
            ↪ orientation, debug=self._debug)

    def add_random_cube(self):
        bb = self.bounding_box
        pos = np.random.uniform(bb[0] + [0.02, 0.02, self.z_end], bb[1] - [0.02,
            ↪  0.02, self.z_end+0.1])
        ori = np.random.uniform((0, ) * 3, (np.pi * 2, ) * 3)
        cube = Cube(pb_client=self._pb_client, position=pos,
                    orientation=self._pb_client.getQuaternionFromEuler(ori),
                        ↪ debug=self._debug)
        return cube
```

Listing 6.12: entity/cube.py: Loads cube object into simulation environment

```
import os

import pybullet as pb
import pybullet_data

from .base import Entity
```

```
class Cube(Entity):

    def __init__(self, pb_client=pb, position=(0, 0, 0), orientation=(0, 0, 0,
    ↪ 1), fix_base=False, scale=0.4, **kwargs):
        table = os.path.join(pybullet_data.getDataPath(), 'cube_small.urdf')
        super(Cube, self).__init__(table, pb_client, position, orientation,
        ↪ fix_base, scale, **kwargs)
        self._pb_client.changeDynamics(
            bodyUniqueId=self._id,
            linkIndex=-1,
            lateralFriction=0.5,
        )
```

Listing 6.13: entity/irb120.py: Loads IRB 120 robot with MetalWorks gripper object into simulation environment

```
import math
import logging
from functools import lru_cache

import pybullet as pb
import numpy as np

from .. import interrupts
from ..data import abb_irb120
from .base import Entity
from ..sensors.camera import Camera
from ..interrupts import NumericStateInterrupt, BooleanStateInterrupt
from ..filter import MovingAverage

REVOLUTE_JOINT_INDICES = np.array((1, 2, 3, 4, 5, 6))
GRIPPER_INDEX = 7
GRIPPER_FINGER_INDICES = np.array((8, 9))
MOVABLE_JOINT_INDICES = np.hstack([REVOLUTE_JOINT_INDICES,
↪ GRIPPER_FINGER_INDICES])
FINGER_JOINT_RANGE = np.array([
    [0., 0.012],
    [0, 0.012],
])

GRIPPER_ORIGIN_OFFSET = 0.057


def to_deg(ori):
    return np.array(pb.getEulerFromQuaternion(ori)) * 180. / math.pi


class IRB120(Entity):
```

```
    def __init__(self, pb_client=pb, position=(0, 0, 0), orientation=(0, 0, 0,
        ↪ 1),
                 fixed=True, scale=1., max_finger_force=500., debug=False,
                    ↪ gravity=9.81):
        self._debug = debug

        urdf = abb_irb120()
        super(IRB120, self).__init__(urdf, pb_client, position, orientation,
            ↪ fixed, scale)

        self._max_finger_force = max_finger_force
        self._gravity = gravity

        self._gripper_cam = Camera(
            self._pb_client,
            resolution=(84, 84),
            fov=90.,
            near_plane=0.001,
            far_plane=2.,
            view_calculator=self._gripper_cam_view_calculator,
            debug=debug
        )

        self._pb_client.enableJointForceTorqueSensor(self.id,
            ↪ GRIPPER_FINGER_INDICES[0])
        self._pb_client.enableJointForceTorqueSensor(self.id,
            ↪ GRIPPER_FINGER_INDICES[1])

        self._grasp_interrupt = BooleanStateInterrupt(lambda: self.grasp_force >
            ↪  5)
        self._grasp_force_filter = MovingAverage(count=120, shape=(1, ))

    def update_state(self):
        force = self._grasp_force_state
        self._grasp_force_filter.update(force)

    @property
    def revolute_joint_state(self):
        return np.array([i[0] for i in self._pb_client.getJointStates(self._id,
            ↪ REVOLUTE_JOINT_INDICES)])

    @property
    def finger_joint_state(self):
        return np.array([i[0] for i in self._pb_client.getJointStates(self._id,
            ↪ GRIPPER_FINGER_INDICES)])

    @property
    def gripper_pose_euler(self):
        gripper_state = self.gripper_pose
```

```
        px, py, pz = gripper_state[0]
        ax, ay, az = pb.getEulerFromQuaternion(gripper_state[1])

        return np.array([px, py, pz]), np.array([ax, ay, az])

    @property
    def gripper_pose(self):
        gripper_state = self._pb_client.getLinkState(self._id, GRIPPER_INDEX)

        position, _ = self._pb_client.multiplyTransforms(
            gripper_state[0],
            gripper_state[1],
            (GRIPPER_ORIGIN_OFFSET, 0, 0),
            (0, 0, 0, 1)
        )

        return np.array(position), np.array(gripper_state[1])

    @lru_cache()
    def _joint_range(self):
        num_joints = self._pb_client.getNumJoints(self.id)
        lower_limits = [self._pb_client.getJointInfo(self.id, i)[8] for i in
            ↪ range(num_joints)]
        upper_limits = [self._pb_client.getJointInfo(self.id, i)[9] for i in
            ↪ range(num_joints)]

        return np.array(lower_limits), np.array(upper_limits)

    @property
    @lru_cache()
    def revolute_joint_range(self):
        ll, ul = self._joint_range()
        return ll[REVOLUTE_JOINT_INDICES], ul[REVOLUTE_JOINT_INDICES]

    @property
    @lru_cache()
    def finger_joint_range(self):
        ll, ul = self._joint_range()
        return ll[GRIPPER_FINGER_INDICES], ul[GRIPPER_FINGER_INDICES]

    def set_gripper_pose(self, position, orientation):
        position, _ = self._pb_client.multiplyTransforms(
            position,
            orientation,
            (-GRIPPER_ORIGIN_OFFSET, 0, 0),
            (0, 0, 0, 1)
        )

        common_params = dict(
```

```
        bodyUniqueId=self.id,
        endEffectorLinkIndex=GRIPPER_INDEX,
        targetPosition=position,
        targetOrientation=orientation,
        maxNumIterations=1000,
        residualThreshold=0.00001,
        jointDamping=(1e-50,) * len(MOVABLE_JOINT_INDICES),
    )

    joint_states = pb.calculateInverseKinematics(
        **common_params,
        solver=self._pb_client.IK_DLS
    )[:-2]

    ll, ul = self.revolute_joint_range
    if np.any(np.isnan(joint_states) | (ll > joint_states) | (ul <
    ↪ joint_states)):
        joint_states = pb.calculateInverseKinematics(
            **common_params,
            lowerLimits=list(self._joint_range()[0][MOVABLE_JOINT_INDICES]),
            upperLimits=list(self._joint_range()[1][MOVABLE_JOINT_INDICES]),
            jointRanges=(2 * math.pi, ) * len(MOVABLE_JOINT_INDICES),
            restPoses=(0, ) * len(MOVABLE_JOINT_INDICES),
            solver=self._pb_client.IK_SDLS,
        )[:-2]

    return self.set_revolute_joint_state(joint_states)

def move_gripper_pose(self, dposition, dorientation):
    current_position, current_orientation = self.gripper_pose

    position, orientation = self._pb_client.multiplyTransforms(
        current_position,
        current_orientation,
        dposition,
        dorientation
    )

    return self.set_gripper_pose(position, orientation)

def set_revolute_joint_state(self, joint_states):
    assert len(REVOLUTE_JOINT_INDICES) == len(joint_states)

    ll, ul = self.revolute_joint_range
    limit_joint_states = np.maximum(np.minimum(joint_states, ul), ll)

    if np.any(joint_states != limit_joint_states):
        logging.debug('Out of bound joint states')
        return self._make_revolute_joint_interrupt(self.revolute_joint_state
```

```
        ↪ )

    for i, v in enumerate(REVOLUTE_JOINT_INDICES):
        self._pb_client.setJointMotorControl2(
            bodyUniqueId=self._id,
            jointIndex=v,
            controlMode=pb.POSITION_CONTROL,
            targetPosition=joint_states[i],
            positionGain=0.3,
            velocityGain=1.,
            maxVelocity=4.,
        )

    return self._make_revolute_joint_interrupt(limit_joint_states)

def set_gripper_finger(self, open):
    if open:
        return self.open_gripper()
    else:
        return self.close_gripper()

def open_gripper(self):
    return self.set_finger_joint_state(FINGER_JOINT_RANGE[:, 1].ravel())

def close_gripper(self):
    return interrupts.any(self.set_finger_joint_state(FINGER_JOINT_RANGE[:,
        ↪ 0].ravel()), self._grasp_interrupt)

def set_finger_joint_state(self, joint_states):
    assert len(joint_states) == len(GRIPPER_FINGER_INDICES)

    self._pb_client.setJointMotorControlArray(
        self._id,
        GRIPPER_FINGER_INDICES,
        pb.POSITION_CONTROL,
        joint_states,
        forces=(self._max_finger_force,) * 2,
        positionGains=(0.3,) * len(joint_states),
        velocityGains=(1.,) * len(joint_states)
    )

    return self._make_finger_joint_interrupt(joint_states)

def _make_revolute_joint_interrupt(self, target_state):
    assert len(target_state) == len(REVOLUTE_JOINT_INDICES)
    interrupt = NumericStateInterrupt(target_state, lambda: self.
        ↪ revolute_joint_state, tolerance=1e-4)
    return interrupt
```

```
def _make_finger_joint_interrupt(self, target_state):
    assert len(target_state) == len(GRIPPER_FINGER_INDICES)
    interrupt = NumericStateInterrupt(target_state, lambda: self.
        ↪ finger_joint_state, tolerance=1e-3)
    return interrupt

@property
def _grasp_force_state(self):
    fx, fy, fz, _, _, _ = self._pb_client.getJointState(self.id,
        ↪ GRIPPER_FINGER_INDICES[1])[2]
    diff = np.abs(np.sqrt(np.sum(np.array([fx, fy, fz])) ** 2) - self.
        ↪ _gravity)

    return diff

@property
def grasp_force(self):
    return self._grasp_force_filter.get()

def reset_joint_states(self):
    for i in MOVABLE_JOINT_INDICES:
        self._pb_client.resetJointState(self.id, i, 0)

def capture_gripper_camera(self):
    return self._gripper_cam.state

def _gripper_cam_view_calculator(self):
    position, orientation = self.gripper_pose

    offset = np.array([-0.03, 0, 0.02])

    eye, _ = self._pb_client.multiplyTransforms(
        position,
        orientation,
        offset,
        (0, 0, 0, 1)
    )

    to, _ = self._pb_client.multiplyTransforms(
        position,
        orientation,
        offset + (100, 0, 0),
        (0, 0, 0, 1)
    )

    up, _ = self._pb_client.multiplyTransforms(
        position,
        orientation,
        offset + (0, 0, 100),
```

```python
            (0, 0, 0, 1)
        )

        return eye, to, up
```

Listing 6.14: entity/ground.py: Loads ground object into simulation environment

```python
import os

import pybullet as pb
import pybullet_data

from .base import Entity


class Ground(Entity):

    def __init__(self, pb_client=pb):
        plane = os.path.join(pybullet_data.getDataPath(), 'plane.urdf')
        super(Ground, self).__init__(plane, pb_client)
```

Listing 6.15: sensors/base.py: Base class for sensors in simulation

```python
import pybullet as pb

from ..utils import unimplemented


class Sensor(object):

    def __init__(self, pb_client=pb):
        self._pb_client = pb_client

    @property
    def state(self):
        unimplemented()
```

Listing 6.16: sensors/camera.py: Loads RGB-D depth camera sensor into simulation

```python
import pybullet as pb

from .base import Sensor
from ..utils import clip_line_end


class Camera(Sensor):
```

```python
    def __init__(self, pb_client=pb, resolution=(320, 240), fov=60, near_plane
        ↪ =0.01, far_plane=100.,
                 view_calculator=lambda: ((0, 0, 1), (0, 0, 0), (1, 0, 1)), debug
                     ↪ =False):
        super(Camera, self).__init__(pb_client)
        self._res_x, self._res_y = resolution

        self._projection_matrix = pb_client.computeProjectionMatrixFOV(
            fov,
            self._res_x / self._res_y,
            near_plane,
            far_plane,
        )
        self._view_calculator = view_calculator
        self._debug = debug

    @property
    def state(self):
        eye, to, up = self._view_calculator()

        view_matrix = self._pb_client.computeViewMatrix(
            eye,
            to,
            up,
        )

        _, _, rgb, depth_map, _ = self._pb_client.getCameraImage(
            width=self._res_x,
            height=self._res_y,
            renderer=pb.ER_BULLET_HARDWARE_OPENGL,
            flags=pb.ER_NO_SEGMENTATION_MASK,
            viewMatrix=view_matrix,
            projectionMatrix=self._projection_matrix,
        )

        if self._debug:
            self._pb_client.addUserDebugLine(
                eye,
                clip_line_end(eye, to),
                (1, 0, 0),
                lifeTime=1.
            )

            self._pb_client.addUserDebugLine(
                eye,
                clip_line_end(eye, up),
                (0, 0, 1),
                lifeTime=1.
            )
```

```
        return rgb, depth_map
```

Listing 6.17: interrupts.py: Interrupts used to detect different simulator states like collision state

```
import pybullet as pb
import numpy as np

from simulator.utils import unimplemented
from simulator.env import base


class Interrupt(object):

    def should_interrupt(self):
        unimplemented()

    def spin(self, env, max_time=None):
        counter = 0
        while not self.should_interrupt() and (max_time is None or counter <
            ↪ max_time * 240):
            env.step()
            counter += 1


class TimeoutInterrupt(Interrupt):

    def __init__(self, env: base.Environment, timeout: float):
        super(TimeoutInterrupt, self).__init__()
        self._start = env.time
        self._timeout = timeout
        self._env = env

    def should_interrupt(self):
        return (self._env.time - self._start) > self._timeout


class NumericStateInterrupt(Interrupt):

    def __init__(self, target_state, state_reader, tolerance=1e-4):
        super(NumericStateInterrupt, self).__init__()
        self._state_reader = state_reader
        self._target_state = target_state
        self._tolerance = tolerance

    def should_interrupt(self):
        current_state = self._state_reader()
        return np.all(np.abs(current_state - self._target_state) <= self.
```

```
        ↪ _tolerance)


class BooleanStateInterrupt(Interrupt):

    def __init__(self, state_reader):
        super(BooleanStateInterrupt, self).__init__()
        self._state_reader = state_reader

    def should_interrupt(self):
        interrupt = self._state_reader()
        return interrupt


class CollisionInterrupt(Interrupt):

    def __init__(self, target, exclusions=tuple(), pb_client=pb):
        super(CollisionInterrupt, self).__init__()
        self._pb_client = pb_client
        self._target = target
        self._exclusions = exclusions

    def should_interrupt(self):
        points = self._pb_client.getContactPoints(self._target)
        collisions = [p[2] for p in points if p[2] not in self._exclusions]
        return len(collisions) > 0


class ComposeInterrupts(Interrupt):

    def __init__(self, interrupts, decision_maker):
        super(ComposeInterrupts, self).__init__()
        self._interrupts = interrupts
        self._interrupted = []
        self._decision_maker = decision_maker

    @property
    def interrupts(self):
        return self._interrupts

    def should_interrupt(self):
        self._interrupted = [i for i in self._interrupts if i.should_interrupt()
            ↪ ]
        return self._decision_maker(self, self._interrupted)


def all(*interrupts):
    return ComposeInterrupts(interrupts, lambda i, v: len(i.interrupts) == len(v
        ↪ ))
```

```python
def any(*interrupts):
    return ComposeInterrupts(interrupts, lambda _, v: len(v) > 0)


def compose(*interrupts, decision_maker):
    return ComposeInterrupts(interrupts, decision_maker)
```

Listing 6.18: filter.py: Moving average filter for smoothening readings from gripper force sensor

```python
import numpy as np


class MovingAverage(object):

    def __init__(self, count=240, shape=(1, )):
        self._count = count
        self._index = -1
        self._data = np.zeros((count, ) + shape)
        self._calculate()

    def _calculate(self):
        self._current = np.mean(self._data, axis=0)

    def update(self, value):
        self._index = self._index + 1

        if self._index >= self._count:
            self._index = 0

        self._data[self._index] = value
        self._calculate()

        return self.get()

    def get(self):
        return self._current
```

Listing 6.19: utils.py: Common utility functions

```python
import os

import numpy as np


def assert_exist(file):
    assert os.path.exists(file)
```

```
def unimplemented():
    raise Exception('not implemented')


def clip_line_end(start, end, length=0.01):
    vec = np.array(end) - np.array(start)
    dir = vec / np.sqrt(np.sum(vec**2))
    clipped = dir * length

    return clipped + start
```

Evaluation using PPO reinforcement learning algorithm

Listing 6.20: comet.py: Logs experiment results and model weights to comet.ml platform

```
import threading
from xmlrpc.server import SimpleXMLRPCServer
from xmlrpc.client import ServerProxy

from comet_ml import Experiment, ExistingExperiment


def new_experiment(experiment_key=None, disabled=False):
  params = dict(
    api_key="api-key",
    project_name="project-name",
    workspace="workspace-name",
    disabled=disabled,
  )

return Experiment(**params) if experiment_key is None else
  ExistingExperiment(
    **params,
    previous_experiment=experiment_key
  )


def new_rpc_experiment_logger(experiment: Experiment, host='localhost', port
    ↪ =8085):
  methods = [i for i in dir(experiment) if callable(getattr(experiment, i))]
  log_methods = [i for i in methods if i.startswith('log_')]

  server = SimpleXMLRPCServer((host, port), allow_none=True)
```

```
  def make_handler(method):
    def handler(args, kwargs):
      getattr(experiment, method)(*args, **kwargs)

    return handler

  for method in log_methods:
    server.register_function(make_handler(method), method)

  def client_generator():
    return ExperimentProxy(host, port)

  def worker():
    server.serve_forever()

  threading.Thread(target=worker).start()

  return server, client_generator


class ExperimentProxy(object):

  def __init__(self, host, port):
    self._host = host
    self._port = port
    self._setup()

  def _setup(self):
    self._proxy = ServerProxy('http://{}:{}'.format(self._host, self._port),
        ↪ allow_none=True)

  def __setstate__(self, state):
    self._host = state['host']
    self._port = state['port']
    self._setup()

  def __getstate__(self):
    return {
    'host': self._host,
    'port': self._port,
    }

  def __getattr__(self, item):
    if item in ('__setstate__', '__getstate__', '_setup'):
      return getattr(self, item)

    attr = getattr(self._proxy, item)
```

```
    def proxy_caller(*args, **kwargs):
      return attr(args, kwargs)

    if item.startswith('log_'):
      return proxy_caller

    return attr
```

Listing 6.21: train.py: PPO model trainer

```
import os
import logging

from comet import new_experiment, new_rpc_experiment_logger

import numpy as np
import ray
import click

from ray.rllib.agents.ppo import PPOTrainer, DEFAULT_CONFIG as
    ↪ PPO_DEFAULT_CONFIG
from ray.rllib.agents.ddpg.apex import ApexDDPGTrainer,
    ↪ APEX_DDPG_DEFAULT_CONFIG
from ray.rllib.agents.ddpg import DDPGTrainer, DEFAULT_CONFIG as
    ↪ DDPG_DEFAULT_CONFIG
from ray.tune.logger import pretty_print
from ray.rllib.models import MODEL_DEFAULTS

import envs
import models


def train(environment='table-clearing-v0', iterations='1000', num_gpus='1',
    ↪ checkpoint=None, model=None,
         num_workers='1', render='0', comet='0', save_frequency='10', algorithm
             ↪ ='PPO', config_trainer={},
         comet_key=None, log_level="DEBUG", object_store_memory=None,
             ↪ worker_memory=None):
    parse_memory = lambda v: None if v is None else int(v) * 1024 * 1024

    ray.init(
        object_store_memory=parse_memory(object_store_memory),
        memory=parse_memory(worker_memory)
    )

    iterations = int(iterations)
    save_frequency = int(save_frequency)
    num_gpus = int(num_gpus)
    num_workers = int(num_workers)
```

```python
render = render == '1'
comet = comet == '1'

comet = new_experiment(disabled=comet is False, experiment_key=comet_key)
comet_rpc_server, comet_client_gen = new_rpc_experiment_logger(comet, '
    ↪ localhost', 8089)

model_config = MODEL_DEFAULTS.copy()

if model is not None:
    model_config = {
        "custom_model": model,
        "custom_options": {}
    }

config = {
    "num_gpus": num_gpus,
    "num_workers": num_workers,
    "env_config": {
        "render": render
    },
    "callbacks": {
        "on_episode_step": _make_episode_step_handler(comet_client_gen()),
        "on_episode_end": _handle_episode_end,
    },
    "model": model_config,
    "log_level": log_level,
}

trainer = _get_trainer(algorithm, environment, config, config_trainer)

if checkpoint is not None:
    trainer.restore(checkpoint)

comet.set_model_graph(trainer.get_policy().model.base_model.to_json())
logging.info(trainer.get_policy().model.base_model.summary())

for i in range(iterations):
    result = trainer.train()
    print(pretty_print(result))

    check_point = None
    if i % save_frequency == 0 or i == iterations-1:
        check_point = trainer.save()
        print('Checkpoint saved at {}'.format(check_point))

    if comet is None:
        continue
```

```
        comet.log_current_epoch(i)

        metrics = [
            'episode_reward_max', 'episode_reward_mean', 'episode_reward_min',
            'episode_len_mean', 'episodes_total', 'timesteps_total'
        ]

        for metric in metrics:
            comet.log_metric(metric, result[metric])

        if check_point is not None:
            comet.log_asset_folder(os.path.dirname(check_point), step=i)


def _get_trainer(name, env, defconfig, config_trainer):
    if name == 'PPO':
        return _trainer_ppo(env, defconfig, **config_trainer)
    elif name == 'DDPG':
        return _trainer_ddpg(env, defconfig, **config_trainer)
    elif name == 'APEX_DDPG':
        return _trainer_apex_ddpg(env, defconfig, **config_trainer)
    else:
        raise Exception('unknown algorithm {}'.format(name))


def _trainer_ddpg(env, defconfig):
    config = DDPG_DEFAULT_CONFIG.copy()
    _copy_dict(defconfig, config)

    config["use_state_preprocessor"] = True
    config["pure_exploration_steps"] = 20000
    config["learning_starts"] = 10000

    trainer = DDPGTrainer(config=config, env=env)

    return trainer


def _trainer_apex_ddpg(env, defconfig):
    config = APEX_DDPG_DEFAULT_CONFIG.copy()
    _copy_dict(defconfig, config)

    config["use_state_preprocessor"] = True
    config["exploration_should_anneal"] = True

    trainer = ApexDDPGTrainer(config=config, env=env)

    return trainer
```

```python
def _trainer_ppo(env, defconfig):
    config = PPO_DEFAULT_CONFIG.copy()
    _copy_dict(defconfig, config)

    config["lambda"] = 0.95
    config["kl_coeff"] = 0.5
    config["vf_clip_param"] = 100.0
    config["entropy_coeff"] = 0.01

    config["train_batch_size"] = 1024
    config["sample_batch_size"] = 200
    config["sgd_minibatch_size"] = 512
    config["num_sgd_iter"] = 30
    config["batch_mode"] = "complete_episodes"

    trainer = PPOTrainer(config=config, env=env)

    return trainer


def _handle_episode_end(info):
    episode = info['episode']
    _flatten_info(episode.last_info_for(), episode.custom_metrics)
    logging.info('Episode completed info: {}'.format(episode.custom_metrics))


def _flatten_info(info, out, prefix=None):
    for k, v in info.items():
        name = '{}_{}'.format(prefix, k) if prefix is not None else k
        if isinstance(v, dict):
            _flatten_info(v, out, name)
        else:
            out[name] = v


def _make_episode_step_handler(c):
    def handler(info):
        episode = info["episode"]
        step = episode.length

        if step % 1000 != 0 or c is None:
            return

        obs = episode.last_raw_obs_for()
        rgb = np.array(obs)[:, :, :3]
        c.log_image(rgb.tolist(), name=str(episode.episode_id), overwrite=True)

    return handler
```

```
def _copy_dict(src, dest):
    for k, v in src.items():
        dest[k] = v


@click.command('train')
@click.argument('name', type=click.STRING)
@click.option('--config', '-c', type=(str, str), multiple=True)
@click.option('--config_trainer', '-ct', type=(str, str), multiple=True)
def main(name, config, config_trainer):
    train(name, **dict(config), config_trainer=dict(config_trainer))
```

Listing 6.22: evaluate.py: PPO model evaluator

```
import ray
import click

from ray.rllib.agents.ppo import PPOTrainer, DEFAULT_CONFIG
from ray.rllib.rollout import rollout

import envs


def evaluate(env, model):
    ray.init()
    config = DEFAULT_CONFIG.copy()
    config["env_config"] = {"render": True}
    config["num_workers"] = 1

    trainer = PPOTrainer(config=config, env=env)
    trainer.restore(model)

    rollout(trainer, env, 1000)


@click.command('evaluate')
@click.argument('name', type=str)
@click.argument('model', type=click.Path(exists=True))
def main(name, model):
    evaluate(name, model)
```

Bibliography

[1] L. Fan, Y. Zhu, J. Zhu, Z. Liu, O. Zeng, A. Gupta, J. Creus-Costa, S. Savarese, and L. Fei-Fei, "Surreal: Open-source reinforcement learning framework and robot manipulation benchmark," in *Conference on Robot Learning*, 2018.

[2] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* The MIT Press, second ed., 2018.

[3] M. Breyer, F. Furrer, T. Novkovic, R. Siegwart, and J. Nieto, "Comparing task simplifications to learn closed-loop object picking using deep reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 4, pp. 1549–1556, April 2019.

[4] T. Gradient, "The promise of hierarchical reinforcement learning." https://thegradient.pub/the-promise-of-hierarchical-reinforcement-learning/, 2019.

[5] M. Wulfmeier, A. Abdolmaleki, R. Hafner, J. T. Springenberg, M. Neunert, T. Hertweck, T. Lampe, N. Siegel, N. Heess, and M. Riedmiller, "Regularized hierarchical policies for compositional transfer in robotics," 2019.

[6] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver, "Massively parallel methods for deep reinforcement learning," *CoRR*, vol. abs/1507.04296, 2015.

[7] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, K. Goldberg, and I. Stoica, "Ray rllib: A composable and scalable reinforcement learning library," *CoRR*, vol. abs/1712.09381, 2017.

[8] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning." http://pybullet.org, 2016–2019.

[9] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.