PREDICTING OBJECT DYNAMICS FROM IMAGE SEQUENCES: A PRECURSOR TO AUTONOMOUS OBJECT MANIPULATION

A PROJECT REPORT

Submitted by

S. UTHIRA LAKSHMI (119005122)

In partial fulfillment of the requirements for the award of the degree of

Bachelor of Technology
in
Electrical & Electronics Engineering

School of Electrical & Electronics Engineering
SASTRA DEEMED TO BE UNIVERSITY
(A University established under section 3 of the UGC Act, 1956)
Tirumalaisamudram
Thanjavur-613401

JUNE-2019

BONAFIDE CERTIFICATE

This is to certify that the project work entitled **"PREDICTING OBJECT DYNAMICS FROM IMAGE SEQUENCES: A PRECURSOR TO AUTONOMOUS OBJECT MANIPULATION"** is a bonafide record of the work carried out by

Uthiralakshmi S (119005122)

student of final year B.Tech., Electrical and electronics, in partial fulfillment of the requirements for the award of the degree of B. Tech in Electrical and electronics engineering of the **SASTRA DEEMED TO BE UNIVERSITY, Tirumalaisamudram, Thanjavur – 613 401,** during the year 2018-2019. This project work was done as a part of Internship in a Research project looking at Intelligent Robot Manipulation approaches at Lincoln Centre for Autonomous Systems, University of Lincoln, Lincoln, United Kingdom.

Harit Pandya
Research Fellow,
Lincoln Centre for Autonomous Systems,
University of Lincoln, Lincoln, United Kingdom (UK).

Project Viva-voce held on _____

Examiner - I **Examiner - II**

ACKNOWLEDGEMENT

The satisfaction that accompanies the successful completion of this project would be incomplete without the mention of the people who made it possible, without whose constant guidance and encouragement would have made efforts go in vain. I consider myself privileged to express gratitude and respect towards all those who guided me through the completion of this project.

It would have been impossible to have completed this project without the kind support from SASTRA Deemed to be University.

I wish to thank my parents for financing my studies in this college as well as for constantly encouraging me to learn engineering and providing this opportunity which cannot be thanked enough.

My sincere thanks to **Dr. S. Vaidhyasubramaniam**, Vice-Chancellor, SASTRA Deemed to be University, for his support to perform my work.

My utmost thanks to **Dr. K. Thenmozhi,** Dean, SEEE for her consistent and immense belief in me and support for my work.

My heartfelt thanks to **Dr. K. Vijayarekha,** Associate Dean, EEE for her motivation and support in this endeavor. It was of great value as I progressed in this project.

I convey thanks to my project guide **Dr. Harit Pandya**, University of Lincoln, for providing encouragement constant support and guidance which was of great help to complete this project successfully.

I would also like to thank **Dr. Aravinda Srinivasan, Dr. Gerrhard Neumann, Dr. Marc Hanheide** without whose help I would not have been able to complete this project successfully.

I am immensely grateful to **Dr. M. Sridharan**, **Dr. N. Subashini**, and **Dr. J. Jayachandran**, the project coordinators, for their help and guidance.

Last but not the least, I would like to thank the Almighty for giving me the strength, knowledge, ability and opportunity to undertake this project and complete it to the best of my abilities. Without His blessings, it would have been impossible for me to have made any progress.

ABSTRACT

KEYWORDS: Robot manipulation, Machine learning, Neural networks, Degrees of freedom, Sequence prediction, Long short term memory network, Encoder, Decoder

Robot manipulation is an upcoming area in the field of robotics. A basic task of any robotic application is picking and placing objects and interacting with the environment which forms the fundamental tasks in manipulation. Intelligent robot manipulation involves using algorithms and machine learning approaches to manipulate and make the robot arm decide which method can be used to perform various tasks. Machine learning algorithms and techniques involve training neural networks on a dataset to learn features in an image, predict sequences, classify images and other tasks that requires intelligent learning.

In this project an attempt is made to understand the physics of the simulation environment for making a robot build and break a tower of blocks repeatedly. The robot is manipulated to perform push and grasp actions on blocks of different color.

Further, the project involves obtaining a dataset of RGB images containing various positions and orientation of the blocks when it is being broken from simulation environment and using it as the dataset to predict the subsequent sequences by using an architecture combining long short term memory neural network with convolutional encoder down-sampling layers and convolutional nonlinear decoder-up sampling layers.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

DOF – Degrees of freedom

ROS – Robot operating system

RViz- Robot visualizer

RGB- Red green blue

NN – Neural network

CNN – Convolutional neural network

RNN- Recurrent neural network

LSTM – Long short term memory

DH parameter- Denavit–Hartenberg parameter

SDF- Standard format

URDF- Universal robot description format

TF- Transformations

3D – 3 Dimensional

TCP/IP- Transmission control protocol/Internet protocol

OSRF- Open source robotics foundation

NOTATIONS

x, y, z – denotes distance along 3 Cartesian coordinate axes

DH parameters a_{i-1}, α_{i-1}, d_i and θ_i denote the link length, link twist, link offset and joint angle respectively and Z_i denoted the Z axis

W- Weights or strength of neurons

x – Input signal

b – Bias

f(x)- Function

σ(x) -Sigmoid function

tanh(x) -Tanh function

max(x)- Maximum function

Li -Sum of loss values of all incorrect classes

yi --Correct class score

yj -Current class score

fyi -Score function of correct class

fj -Score function for current class

Δ - Margin value

w_j- jth row of W reshaped as a column.

CHAPTER 1

INTRODUCTION

In the recent times, there has been a substantial development in the field of Robotics and Artificial Intelligence. The ability to play around and interact with the surrounding is the basic quality of all beings. Robots are almost able to interact with environment like humans and animals can do. Robot manipulation refers to ways robots interact with the environment.

While robots interact with the objects around by grasping or pick and place, humans use a variety of methods to interact and manipulate objects in the environment. This restricts the number of tasks and complexity of tasks that can be performed by the robots. As much it is important to identify the type and number of tasks that can be achieved by robots, it is also important to find the best way to reach at the solution. Humans can understand and predict the best way to attain the solution to reach and manipulate objects very easily, in fact in just a matter of seconds they can easily find out the best answer. In the contrast, a robot cannot identify its environment very easily, it requires lot of training and understanding to be able to arrive at a solution. Although robots can be very accurate and efficient at doing any given task, they still require lot of learning at the initial development stages.

Intelligent Robot Manipulation refers to methods and practices where a robot achieves best solution while manipulating and interacting with objects and environment by using suitable algorithms to learn and train. Intelligent way of manipulating the robots involves first developing an efficient method to arrive the desired goal then implementing that method to make the robot interact in real time.

Robot manipulation is a field with lot of applications. Robots can help humans in doing day to day activities. Just as hands are an important and integral part for humans, Robotic arm acts as a primary means of interacting with the environment. Specially designed Robots can assist disabled to manipulate or carry objects and do daily chores. Nuclear robots can carry harmful and toxic waste using their arms. Further, specialized applications can be found when a robot is able to understand the physics of the environment and predict future sequences. The above can be achieve by using machine learning. With above implementation, robots can help in assisting more complex tasks like preventing fall of any object or trajectory of any object for locating. The basic idea can be further implemented in more complex modelling and training. The combination of Robot manipulation with time sequence prediction using recurrent neural networks is foundation to establishing a Humanoid robot. Although the speed of understanding and computational capability of human brain cannot be compared with that of a robot, this combination can be compared to the connection between human brain with its memory of environment and object and hands picking up objects. Human like accuracy of detection and prediction can be achieved by using various modified type of recurrent neural networks.

In this project, a 7 DOF Panda arm of Franka Emika Robot is simulated in Gazebo Simulation environment in integration with ROS and RViz. The Panda robot consists of an arm and a gripper attached to it. The panda robot is manipulated to construct a tower of blocks of different color and break it repeatedly. In this process, the pose and orientations of the blocks are recorded and used as a dataset to train and test a recurrent neural network to predict the possible future state of the blocks. The project is aimed to be implemented in real time on Franka Emika Panda Arm with 7 Degrees of Freedom for manipulating, testing the sequence prediction.

CHAPTER 2

LITERATURE REVIEW

The list of Prerequisites needed for doing this project are discussed in this section. The important know-hows include understanding Franka Emika Panda Robot and its characteristics, exploring its Forward and Inverse Kinematics and Dynamics, understanding about various simulation software packages used, understanding what is a Neural network, Convolutional neural network, Recurrent neural network, Long short term memory network, understanding the architecture of various networks that are used and knowing the ways to interpret and implement it, understanding about programming languages used, hardware specifications of computer for running the simulation.

2.1.1 Franka Emika Robot

Fig 2.1 Franka Emika Panda Robot
(Ref. www.franka.de)

Franka Emika Panda Robot is a product of German engineering, inspired by human agility and sense of touch. It is a very sensitive and an extraordinary tool for manipulating tasks. It has 7 Degrees of freedom with torque sensors build on all the 7 axes which makes easily maneuverable. It is a bench-mounted robot with integrated joint modules. It has an ability to learn and train and can be used for performing a multitude of tasks.

The key features:

- 7 DOF
- Sensors in all 7 axes
- Ability to learn and train

Why FRANKA EMIKA?

Franka Emika Panda arm has 7 DOF with sensibility in all 7 axes. It is more economical and has high sensitivity as compared to other robot arms. It can be easily used to manipulate objects and interact with the environment. It can be made helpful in working in Nuclear environments for moving or disposing harmful objects and in factories for moving, pushing, lifting objects and it can also be useful in agricultural fields to pick tomatoes or mushrooms. It can be controlled through Master-slave configurations, has the ability to teach and learn, easy to integrate with ROS. Franka panda robot can be simulated in Gazebo simulation by integrating it with RViz for motion planning, tracking and visualizing using markers and also can be simulated in Vrep simulation environment. Franka panda robot is setup on a stationary working cell assembly with a Kinect camera mounted. In this setup, multiple Franka Panda robots can be mounted in parallel and controlled by master robot.

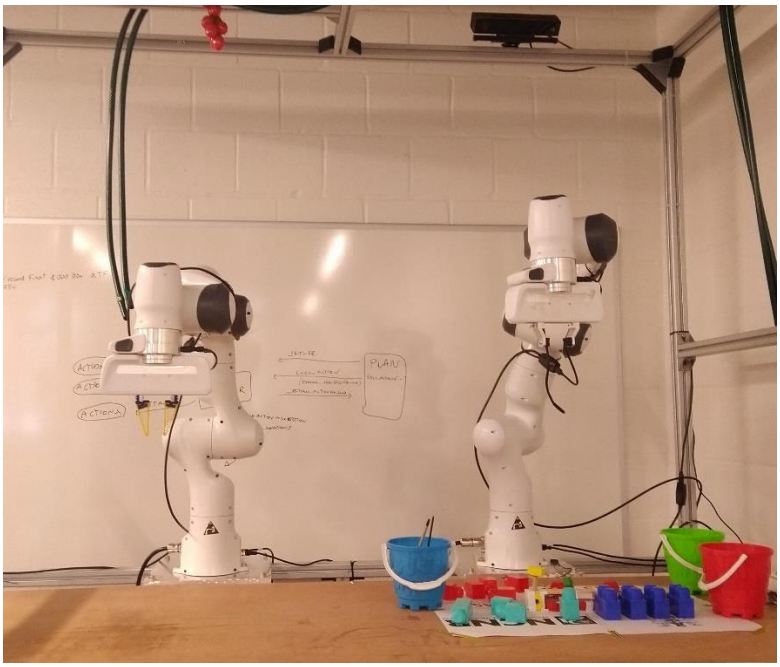


Fig. 2.2. Franka Emika panda robot setup

2.1.2 Robot Kinematics and Dynamics

Kinematics is study of motion of objects without considering the cause of motion. Robot Kinematics is study of robot manipulator motion without including forces, moments which are the cause of motion. It deals with the study of interconnection between the links and joints to analyze the position, velocity and acceleration of each. The motion parameters are position, velocity, acceleration, pose (position and orientation). There are two types of defining spaces in Kinematics: Joint space and Task space. The transformations between the two spaces can be decomposed into rotation and translation. Joint space encompasses the angular coordinates and task space encompasses the Cartesian coordinates like x, y, z.

The two types of Kinematics are forward kinematics and inverse kinematics, which specifies the relationship between the end-effector coordinates in task spaces and joint coordinates in local joint spaces. In forward kinematics, we specify the end-effector coordinates in terms of local joint coordinates and inverse kinematics, joint space coordinates are specified in terms of the end-effector

coordinates. A good robot kinematic model is essential to obtain Forward and Inverse kinematic solution. Denavit-Hartenberg method that uses four parameters is the most common method for describing the robot kinematics. These parameters a_{i-1}, α_{i-1}, d_i and θ_i are the link length, link twist, link offset and joint angle, respectively. A coordinate frame is attached to each joint to determine DH parameters. Z_i axis of the coordinate frame is pointing along the rotary or sliding direction of the joints.

Computing a forward kinematics solution is far easier than computing an inverse kinematics solution. The inverse kinematics solution might contain anomalies or singularities or multiple solutions.

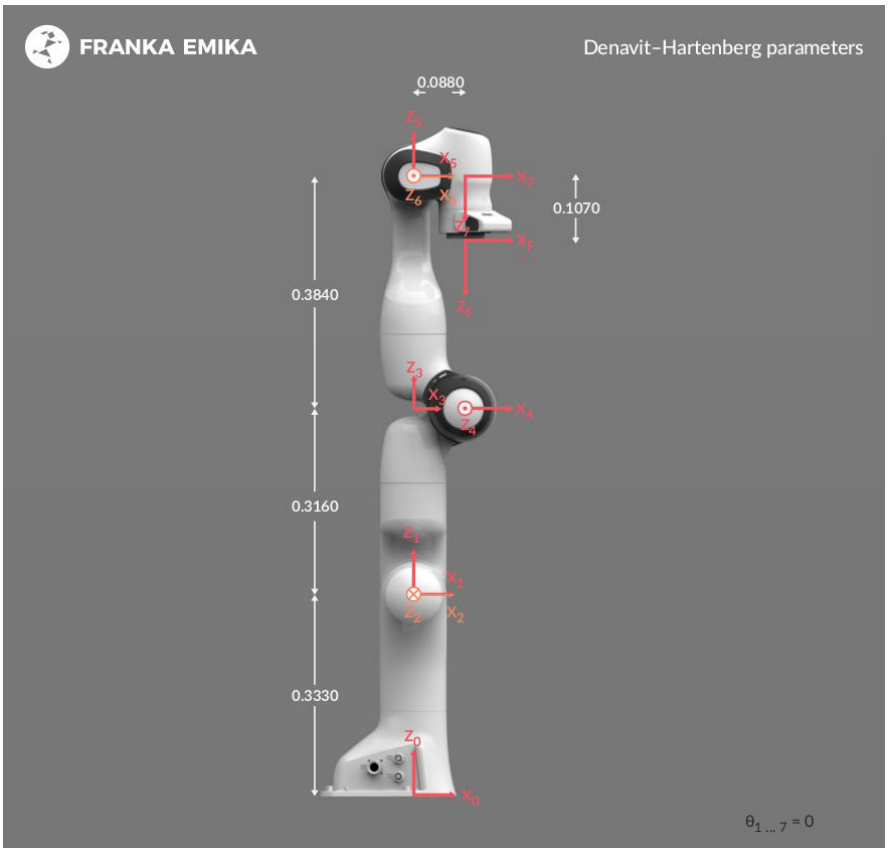


Fig. 2.3 Panda's Kinematic Chain
(Ref. www.franka.de)

Joint	a(m)a(m)	d(m)d(m)	α(rad)α(rad)	θ(rad)θ(rad)
Joint 1	0	0.333	0	θ1θ1
Joint 2	0	0	−π2−π2	θ2θ2
Joint 3	0	0.316	π2π2	θ3θ3
Joint 4	0.0825	0	π2π2	θ4θ4
Joint 5	-0.0825	0.384	−π2−π2	θ5θ5
Joint 6	0	0	π2π2	θ6θ6
Joint 7	0.088	0	π2π2	θ7θ7
Flange	0	0.107	0	0

Table 2.1 D-H Parameters for Panda Robot

(Ref. www.franka.de)

2.2 Software/Package required for Simulation

- ROS (Robot operating system)
- Gazebo Simulation environment
- V-Rep Simulation software

2.2.1 ROS (Robot operating system)

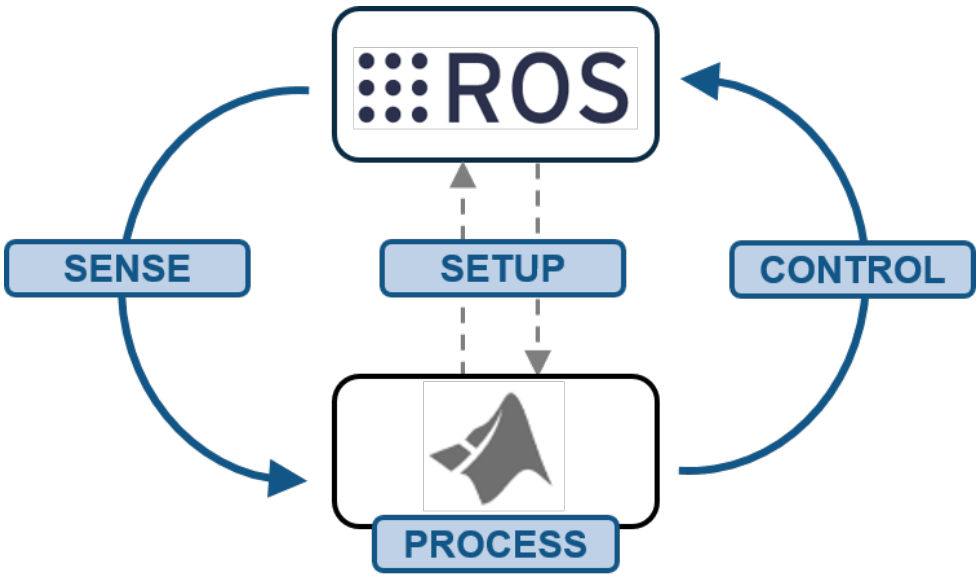


Fig. 2.4 ROS overview

(Ref. http://wiki.ros.org/)

ROS is an open source meta-operating system software platform that helps in creating robotics application for software developers. It provides hardware abstraction, device drivers, low-level device management, libraries, visualizers, message-passing, package management, and more. The ROS runtime "graph" is a peer-to-peer network of processes (potentially distributed across machines) that are loosely coupled using the ROS communication infrastructure. ROS implements several different styles of communication, including synchronous RPC-style communication over services, asynchronous streaming of data over topics, and storage of data on a Parameter Server.

ROS is similar to Ubuntu with respect to publication of distributions. Each distribution is in relation with one of the Ubuntu version. ROS distribution is released every year in May. The distributions released in even years support LTS release of Ubuntu. ROS has an official website with documentation and tutorials for Beginner and intermediate level of user. It also offers tutorials for its supported inbuilt software like TF, RViz, MoveIt!

Framework of ROS:

ROS has three levels of concepts: File system level, Computation Graph level and the Concept level. It also has two types of names – Package Resource Names and Graph Resource Names. ROS resources that are present in a disk are covered by ROS Filesytem level concepts. It specifies details about Packages, Metapackages, Package manifests, Repositories, message types, Service types.

In Computation Graph level, the computation graph is the peer-to-peer network of ROS processes that are processing data together. The basic Computation Graph concepts of ROS are nodes, Master, Parameter Server, messages, services, topics, and bags, all of which provide data to the Graph in different ways.

In Community level, ROS resources and used for community use for exchange of knowledge and software. Distributions, Repositories, The ROS wiki, Bug ticket system, Mailings list, ROS answers and blogs. On a higher level, ROS core platform attempts to be architecture-agonistic. It provides several different modes of communicating data (topics, services, Parameter Server). In order to know how these modes are named and how they are implemented, it is required to have higher level ROS concepts for building larger systems on top of ROS. There are stacks like common, common_msgs and geometry which provide higher level concepts: Coordinate Frames/Transforms, Actions/Tasks, Message Ontology, Plugins, Filters, Robot model, Cheat sheet. One more important thing to build larger systems on top of ROS are the ROS client libraries. ROS client libraries are collections of code that are very user friendly, it takes some of the concepts from ROS and makes them accessible via code.

In order to understand Implementation of ROS, it is very essential to understand the connection between Master, Nodes, Parameter server, how connections are made to topics, messages and services. Although, for an end-user it is not mandatory to understand the concepts of

implementation, it is very much an essential detail for those who want to integrate their systems with ROS. In the nutshell, the most important and basic concepts one need to understand in ROS are ROS Master, ROS packages and dependencies, ROS nodes, ROS topics, ROS services and messages, ROS client server framework, ROS Publisher Subscriber framework, tools available for editing, debugging, recording, plotting, graphs.

Starting ROS Master:

The command used for starting ROS Master is as follows:

```
$ roscore
```

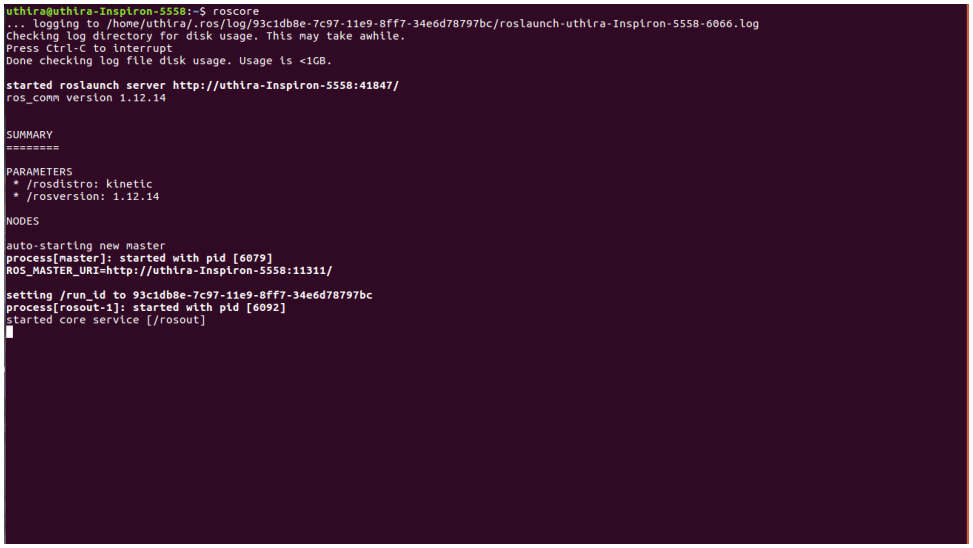


Fig. 2.5 Roscore

Creating a ROS workspace (catkin workspace):

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make
```

catkin_make command automatically creates CMakeLists.txt. Catkin workspace consists of 4 directories namely **build devel install src** folders.

In order to source the setup, the below command is used:

```
$ source devel/setup.bash
```

Creating a sample package with dependencies:

The command below creates a sample package with dependencies.

```
$ catkin_create_pkg beginner_tutorials std_msgs rospy roscpp
```

Displaying ROS nodes list:

The command below can be used for displaying node list:

```
$ rosnode list
```

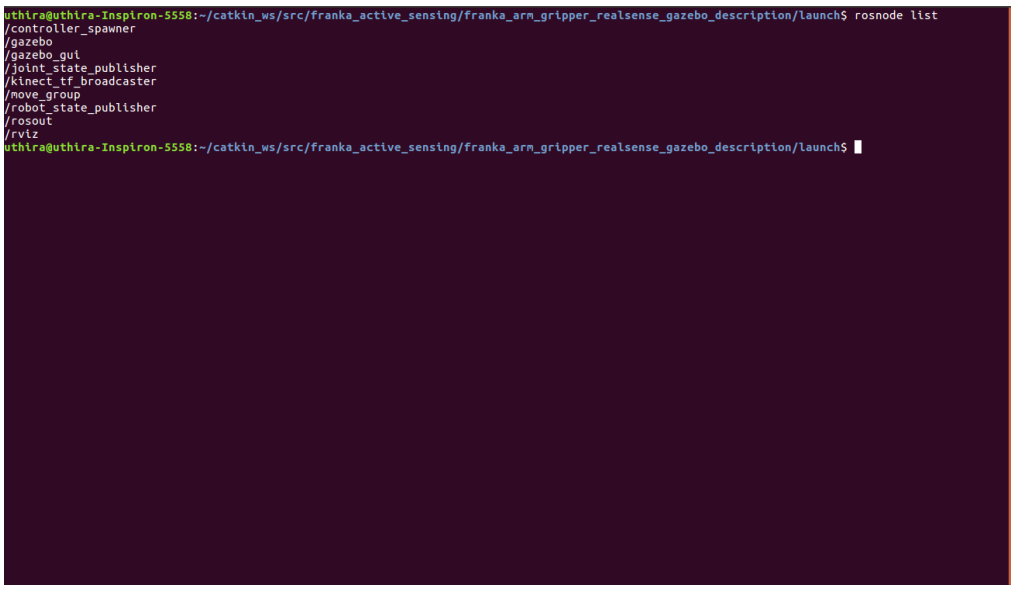


Fig. 2.6 Rosnode list

Displaying ROS topic list:

The command below can be used for displaying topic list:

```
$ rostopic list
```

```
uthira@uthira-Inspiron-5558:~/catkin_ws/src/franka_active_sensing/franka_arm_gripper_realsense_gazebo_description/launch$ rostopic list
/arm_controller/command
/arm_controller/efforts
/arm_controller/follow_joint_trajectory/cancel
/arm_controller/follow_joint_trajectory/feedback
/arm_controller/follow_joint_trajectory/goal
/arm_controller/follow_joint_trajectory/result
/arm_controller/follow_joint_trajectory/status
/arm_controller/state
/attached_collision_object
/clicked_point
/clock
/collision_object
/execute_trajectory/cancel
/execute_trajectory/feedback
/execute_trajectory/goal
/execute_trajectory/result
/execute_trajectory/status
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/gazebo_gui/parameter_descriptions
/gazebo_gui/parameter_updates
/initialpose
/joint_states
/kinect_camera/depth/camera_info
/kinect_camera/depth/depth_image_raw
/kinect_camera/depth/image_raw
/kinect_camera/depth/image_raw/compressed
/kinect_camera/depth/image_raw/compressed/parameter_descriptions
/kinect_camera/depth/image_raw/compressed/parameter_updates
/kinect_camera/depth/image_raw/compressedDepth
/kinect_camera/depth/image_raw/compressedDepth/parameter_descriptions
/kinect_camera/depth/image_raw/compressedDepth/parameter_updates
/kinect_camera/depth/image_raw/theora
/kinect_camera/depth/image_raw/theora/parameter_descriptions
/kinect_camera/depth/image_raw/theora/parameter_updates
/kinect_camera/depth/points
/kinect_camera/depth/points2
/kinect_camera_ir/depth/camera_info
```

Fig. 2.7 Rostopic list

Displaying ROS parameter list:

The command below can be used for displaying ROS parameter list:

```
$ rosparam list
```

```
thira@uthira-Inspiron-5558:~/catkin_ws/src/franka_active_sensing/franka_arm_gripper_realsense_gazebo_description/launch$ rosparam list
arm_controller/gains/panda_arm_joint1/d
arm_controller/gains/panda_arm_joint1/p
arm_controller/gains/panda_arm_joint2/d
arm_controller/gains/panda_arm_joint2/p
arm_controller/gains/panda_arm_joint3/d
arm_controller/gains/panda_arm_joint3/p
arm_controller/gains/panda_arm_joint4/d
arm_controller/gains/panda_arm_joint4/p
arm_controller/gains/panda_arm_joint5/d
arm_controller/gains/panda_arm_joint5/p
arm_controller/gains/panda_arm_joint6/d
arm_controller/gains/panda_arm_joint6/p
arm_controller/gains/panda_arm_joint7/d
arm_controller/gains/panda_arm_joint7/p
arm_controller/joints
arm_controller/type
arm_id
gazebo/auto_disable_bodies
gazebo/cfm
gazebo/contact_max_correcting_vel
gazebo/contact_surface_layer
gazebo/erp
gazebo/gravity_x
gazebo/gravity_y
gazebo/gravity_z
gazebo/max_contacts
gazebo/max_update_rate
gazebo/sor_pgs_iters
gazebo/sor_pgs_precon_iters
gazebo/sor_pgs_rms_error_tol
gazebo/sor_pgs_w
gazebo/time_step
gazebo_gui/auto_disable_bodies
gazebo_gui/cfm
gazebo_gui/contact_max_correcting_vel
gazebo_gui/contact_surface_layer
gazebo_gui/erp
gazebo_gui/gravity_x
gazebo_gui/gravity_y
gazebo_gui/gravity_z
gazebo_gui/max_contacts
gazebo_gui/max_update_rate
```

Fig. 2.8 Rosparam list

The main list of characteristic steps of installing ROS/understanding ROS that are required for a beginner to understand the ROS framework and use it for a certain application are listed below:

Step 1: Installing the appropriate version of Ubuntu

Step 2: Installing the latest and stable version ROS (recent versions ROS Indigo or ROS Kinetic)

Step 3: Installing and configuring ROS environment

Step 4: Setting up ROS workspace (catkin or rosbuild)

Step 5: Setting up file system and understanding the concept of rospackages

Step 6: Understanding ROS server, nodes, topics, messages, services, parameter services

Step 7: Understanding the tools in ROS: rqt graph, rqt plot

Step 8: Understanding the concept of publisher and subscriber and implementing in python/C++

Step 9: Understanding the concept of server/client

Step 10: understanding editing options, recording rosbags and other editing tools available

Step 11: Understanding the tools inbuilt with ROS: TF, RViz, MoveIt!

Step 12: Understanding complex concepts of ROS like building your own ROS package and running ROS in multiple system etc.

Step 13: Starting to use ROS in user specific application or integrating ROS with a specific simulation software and building the catkin/rosbuild.

2.2.2 Gazebo Simulation

Gazebosim is a simulation software that is capable of simulating population of robots in both indoor and outdoor environments. It has multiple physics engine, high-quality graphics and convenient programmatic and graphical interfaces.

Features:

- Dynamics Simulation
- Advanced 3D Graphics
- Sensors and Noise
- Plugins
- Robot Models
- TCP/IP Transport
- Cloud Simulation
- Command Line Tools

The various components of Gazebo are world files, model files, environment variables, gazebo server, gazebo client, Plugins. The world description file contains all the elements in a simulation, including robots, lights, sensors, and static objects. This file is formatted using SDF (Simulation Description Format), and typically has a .world extension. These files are used by Gazebo server to populate the world scene.

The model files are subparts of world file, used for re-use and simplifying the world files. The model file also comes in SDF format. The models files can range from simple shapes to complex robots which consists of joints, links, visual, collision, plugins. The environment variables are used for communication between gazebo server and client and for locating files. The gazebo server is the workhouse of Gazebo, it simulates the world file using physics and sensor engine. The gazebo

client is used to visualize the various elements of the world file. Plugins provide a simple and convenient mechanism to interface with Gazebo. Gazebo uses a distributed architecture with separate libraries for physics simulation, rendering, user interface, communication, and sensor generation. Additionally, gazebo provides two executable programs for running simulations: gazebo server and gazebo client.

Gazebo has an official website with well documented content and user friendly tutorials. The tutorials are wide ranging from beginner: understanding the overview to very high advanced version. Gazebo also offers further documentation through OSRF Bit bucket where many world files, model files can be accessed for free.

Starting:

The command used for starting Gazebo is as follows:

```
$ Gazebo
```

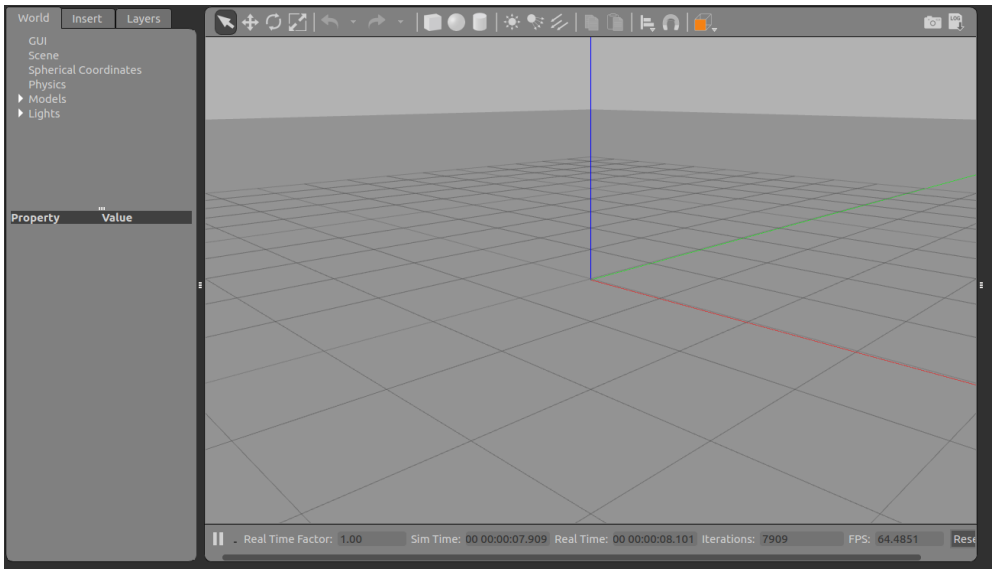


Fig. 2.9 Gazebo Simulation Environment

The list of main steps involved in simulating a robot model in Gazebo are listed below:

Step 1: Install ROS and follow the tutorials

Step 2: Install the appropriate version of Gazebo

Step 3: Install gazeboros packages and setup the ROS workspace

Step 4: Create your own or import a robot model

Step 5: Create a Gazebo world file and include the robot model and other components

Step 6: Connect the robot with ROS and include the ROS control packages

Step 7: Tele-operate the robot, add camera and other functionalities to the robot

Step 8: Visualize the robot with RViz and use MoveIt! for motion planning

Creating World file:

There are two ways to create/populate world in Gazebo: The IDE of gazebo environment is very interactive, offers lot of menus and list of worlds, models, built in plugins to modify scene properties, physics GUI, editing and navigating the objects/models.

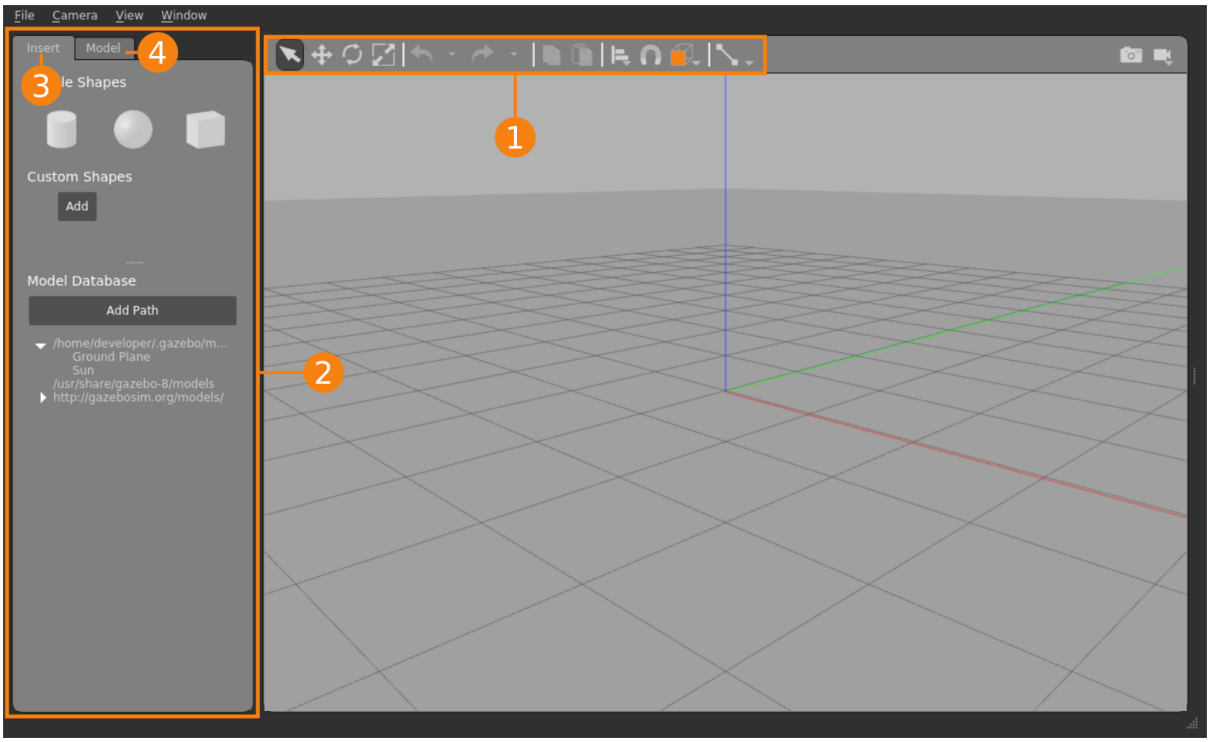


Fig. 2.10 Gazebo model editor

(Ref. www.gazebosim.org)

1. **Toolbar - Contains tools for editing the model**

2. **Palette - Also known as Left Panel. Has two tabs for editing the model.**

3. **Insert tab - Tools for adding links and nested models**

4. **Model tab - Allows editing model properties and contents**

Another way to create a world is by creating a world file following SDF specification.

Sample example of world file is given below.

```
<?xml version="1.0"?>
<sdf version="1.4">
  <world name="default">
    <include>
      <uri>model://ground_plane</uri>
    </include>
    <include>
```

```
      <uri>model://sun</uri>
        </include>
    <model name="my_mesh">
      <pose>0 0 0  0 0 0</pose>
      <static>true</static>
      <link name="body">
        <visual name="visual">
          <geometry>
            <mesh><uri>file://my_mesh.dae</uri></mesh>
          </geometry>
        </visual>
      </link>
    </model>
  </world>
</sdf>
```

Fig. 2.11 World file

(Ref. www.gazebosim.org)

Different models are added to the world file, position and orientation are specified for the models. The color/ visibility properties of the world can be set in the world file. Some of the physics properties can be set by using World file.

Creating a model:

One can add already existing models from online or build a model using URDF specification. Joints, links, physics, force parameters for different joints/links are specified in model SDF file.

```
<?xml version='1.0'?>
<sdf version="1.4">
<model name="my_model">
  <pose>0 0 0.5 0 0 0</pose>
  <static>true</static>
```

```
    <link name="link">
      <inertial>
        <mass>1.0</mass>
        <inertia>
          <ixx>0.083</ixx>       <!-- for a box: ixx = 0.083 * mass * (y*y + z*z) -->
          <ixy>0.0</ixy>         <!-- for a box: ixy = 0 -->
          <ixz>0.0</ixz>         <!-- for a box: ixz = 0 -->
          <iyy>0.083</iyy>       <!-- for a box: iyy = 0.083 * mass * (x*x + z*z) -->
          <iyz>0.0</iyz>         <!-- for a box: iyz = 0 -->
          <izz>0.083</izz>       <!-- for a box: izz = 0.083 * mass * (x*x + y*y) -->
        </inertia>
      </inertial>
      <collision name="collision">
        <geometry>
          <box>
            <size>1 1 1</size>
          </box>
        </geometry>
      </collision>
      <visual name="visual">
        <geometry>
          <box>
            <size>1 1 1</size>
          </box>
        </geometry>
      </visual>
    </link>
  </model>
</sdf>
```

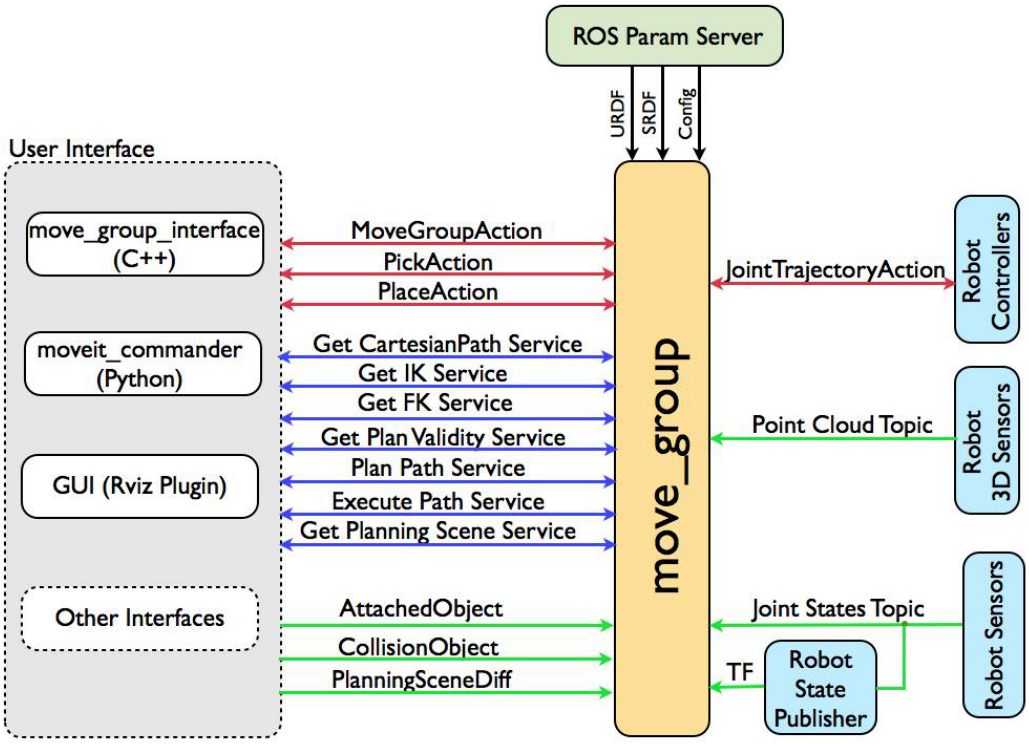
Fig. 2.12 Sample SDF file

(Ref. www.gazebosim.org)

Setting up controllers:

The actuator of every robot has to be controlled. MoveIt! set-up assistant is used for setting up controllers for specific robot part simulated in Gazebo environment. MoveIt! is a package of ROS. MoveIt is an open source software used for mobile manipulation, motion planning, control and navigation, kinematics, 3D perception. It incorporates lot of applications, futuristic design of robots and widely used in domestic, industrial and Research and development areas. It is very easy to use and can be transported to any platform. MoveIt is also integrated with ROS and used mainly is a motion planning tool for ROS interface applications.

Fig. 2.13 System Architecture for move group node



(Ref. www. moveit.ros.org)

The figure above shows the high-level system architecture for the primary node provided by MoveIt called move group. This node serves as an integrator: pulling all the individual components together to provide a set of ROS actions and services for users to use.

There are three ways of interfacing move group node. One is C++ move group interface, Python moveit-commander interface and using GUI RViz plug-in.

RViz is a 3-D visualization tool for ROS. It is helpful in visualizing the sensor data and state information. Using RViz, one can view the current configuration of the robot or its virtual simulation. RViz can get and display the sensor content in ROS topics related to camera.

RViz is a very useful tool used for setting up markers using ROS topics and visualize and track the motion and plan the configuration. RViz can be integrated with Gazebo and MoveIt for planning, visualizing motion trajectories.

Starting:

The command used for starting Rviz is as follows:

```
$ rosrun rviz rviz
```

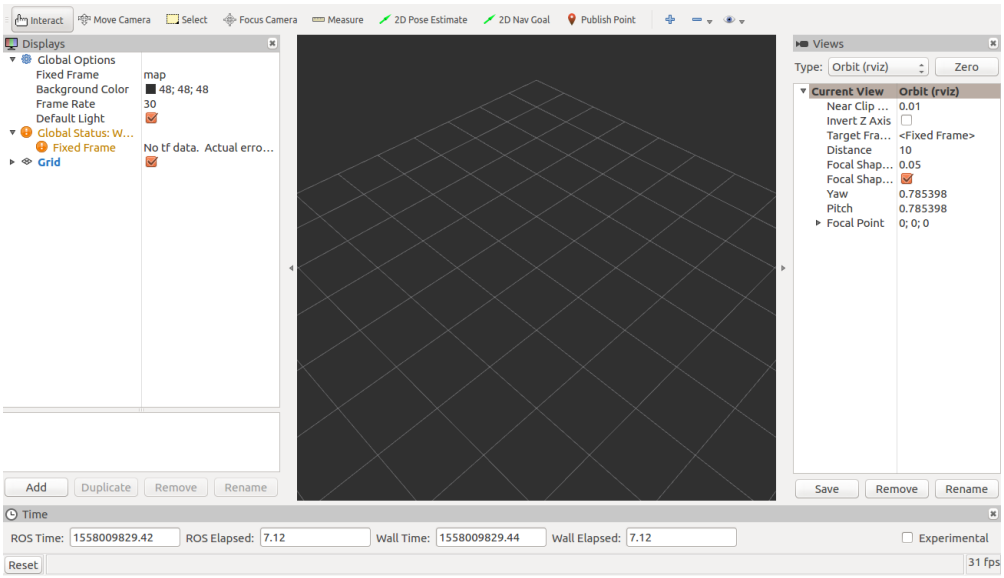


Fig. 2.14 RViz Simulation Environment

Motion planning:

After the controllers are set up, the next step will be to the desired target position. In order to achieve this the motion has to be planned. There are two widely used motion planning packages – ROS stack and OMPL (Open Motion Planning Library). An important tool for motion planning is solving the inverse kinematic chain. Fast IK solver is used for Gazebo ROS integration.

2.2.3 V-REP

V-REP is an integrated simulation development software which can be used to simulate multiple robotic models with easy programming with 6 different program languages, ROS interfaces, External API.

Features:

- 6 Programming approaches (Embedded script, Add on, Plugins, Remote API client, ROS node, BlueZero node)
- Powerful API framework for the 6 programming approaches
- 4 Dynamics/Physics engine (Bullet, ODE, Vortex, Newton)
- Inverse and forward kinematics control
- Customized dynamic control of particles
- Collision detection and distance calculation
- Cross platform and portable
- Proximity and vision sensor simulation
- Building block concept of building any model/sensor
- Path and motion planning

- Custom user interfaces
- Integrated edit modes, Data recording and visualizing, Model browser, Import/export, Full scene hierarchy viewer
- RRS interface and motion library

Starting:

The command for starting up Vrep is as follows and is typed in the root directory of Vrep:

```
$ ./vrep.sh
```

Simulation Environment:

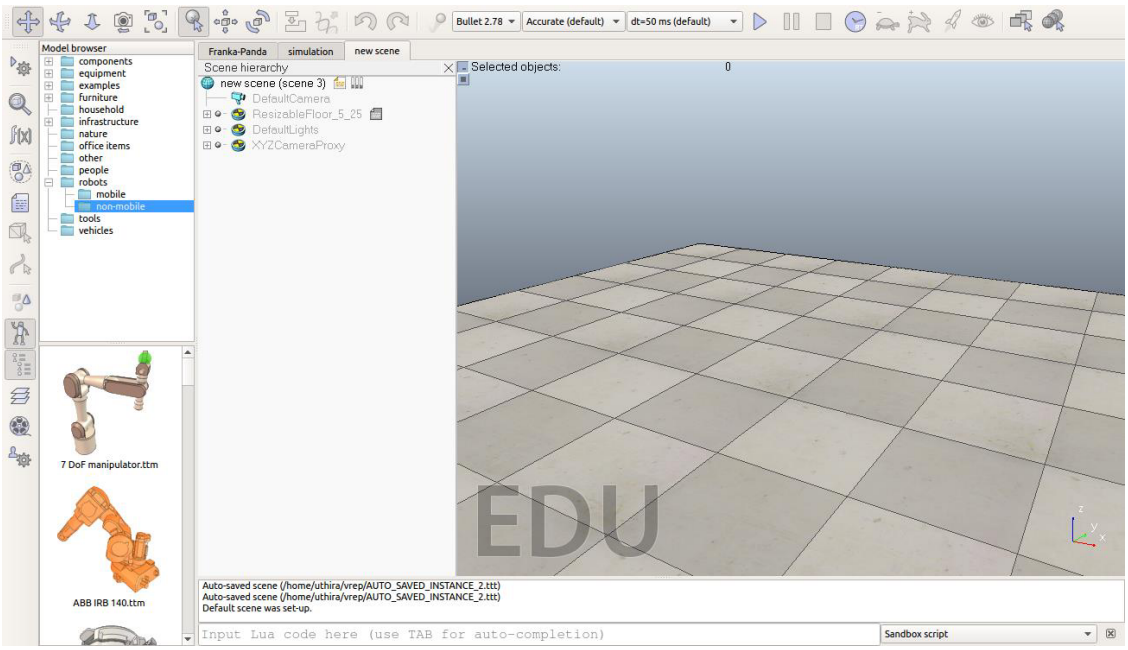


Fig 2.15 Vrep Simulation Environment

The steps involved simulating a robot model in V-Rep:

Step 1: Creating model file of the robot by setting up joints, link properties and saving it in *.ttm* format

Step 2: Configuring dynamics properties and the controller setup

Step 3: Attaching main script ad child script using Lua commander scripting

Step 4: Configuring motion planning

Step 5: Adding other objects to the simulation scene and saving the scene it in *.ttt* format

Step 6: Using any of the 6 API connection to tele operate the robot

Creating a robot model (.ttm model file)

There are three ways of using model file in V-Rep. One way is to use the existing models in models library. another way is to build a model from scratch using CAD models including all the mesh and dae files, applying joints/links, setting up parameter values, initializing suitable physics parameters. Third way is to use the URDF Plugin to import the model.

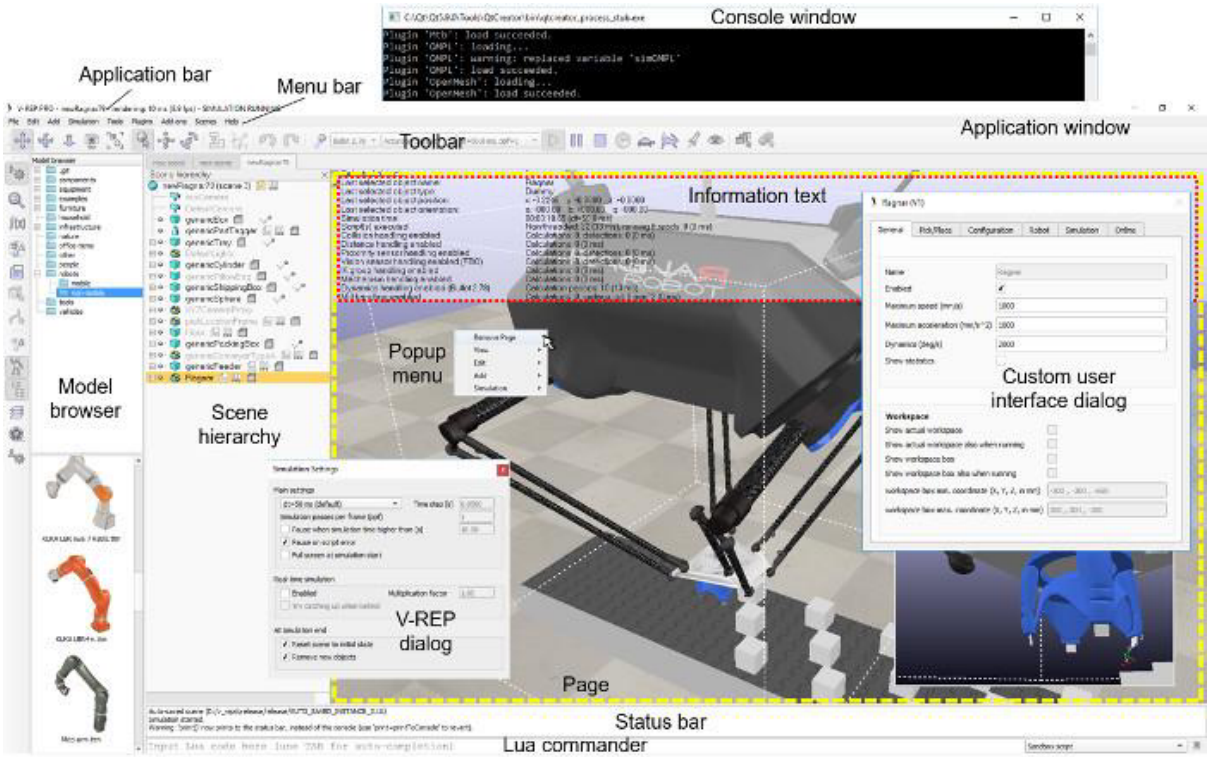


Fig. 2.16 Vrep toolbars

(Ref. www. coppeliarobotics.com)

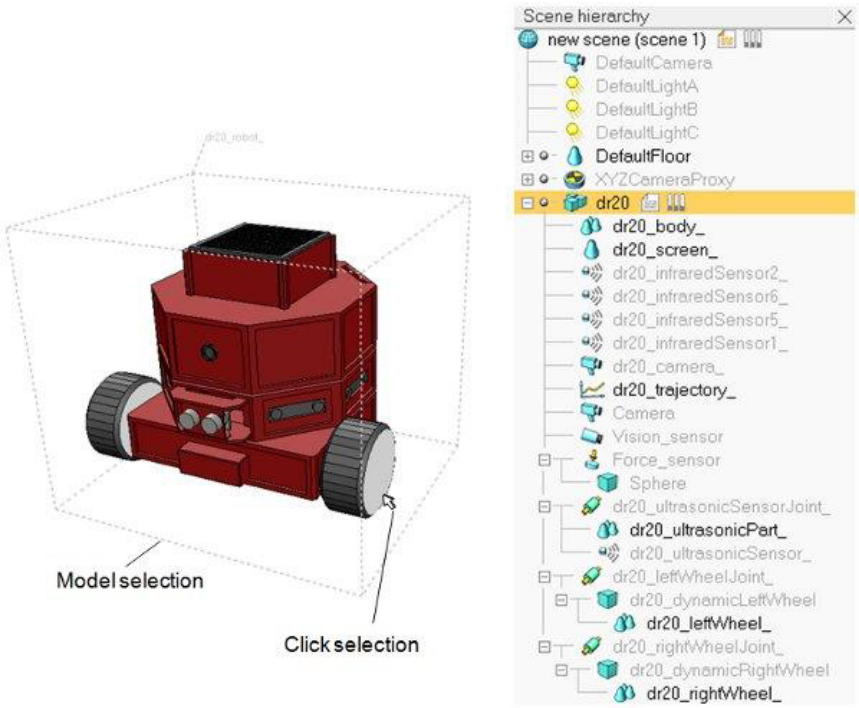


Fig. 2.17 Robot model Vrep

(Ref. www. coppeliarobotics.com)

Setting joint/links parameters and setting up dynamics properties

After loading the URDF model of the robot or you build CAD model adding meshes and DAE files, the various components of the robot can be viewed on scene hierarchy browser, one advantage of Vrep over gazebo or any other simulation software is that the robot model can be edited inside the Vrep simulation environment. The first step is to set properties of all movable joints to dynamic by enabling the motor and control loop.

Main script and child scripts

The default programming language used in Vrep is Lua. The main script and child scripts are written in Lua**.** Main script consists of code that runs the simulation, child scripts consists of code for each part of the robot. These two scripts are used to initialize the simulation environment and robot model parameters.

2.3 MACHINE LEARNING AND NEURAL NETWORKS

2.3.1 Machine Learning

It is an application of artificial intelligence whose main target is to make computer learn by themselves using some algorithms. There are 4 different types of machine learning:

Supervised learning, Semi-supervised learning, Unsupervised learning and Reinforcement learning. Supervised learning: in this type of machine learning we train on labeled datasets and use algorithms to arrive at the target. In this type of machine learning, the targets are well known before starting training and the main objective is to use dataset and train appropriately to arrive at the target.

Example: Classification, Regression

Unsupervised learning: In this type of machine learning, unlabeled dataset is trained by suitable algorithms to predict some relation among the data. The main difference between this type of machine learning and supervised learning is that here the target is not fixed and not known to us.

Example: Data segmentation

Semi-supervised learning: This type of machine learning is the combination of both supervised and unsupervised learning.

Reinforcement Learning: This type of machine learning differs from both supervised and unsupervised learning in the sense that here there is a goal to be achieved and set of possible actions to be taken are given and according to the type of actions taken a certain reward value is assigned. The main objective is to maximize the cumulative reward function. The algorithms work on a

feedback loop where at each state the reward value is assigned for each action performed taking into considerations of some constraints.

Example: Trajectory following robot, Alpha GO

2.3.2 NEURAL NETWORKS

Neural networks are an inspiration from biological neural networks. Although the basic structure of computational neural network is analogous to biological neuron, still the computational complexity of biological neuron is far too high to reach or replicate.

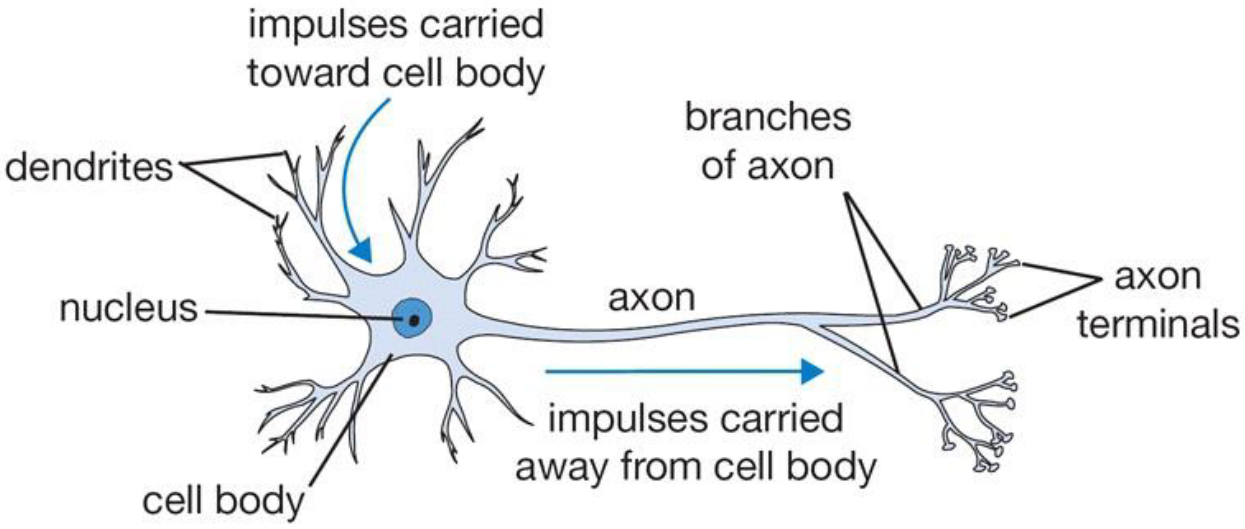


Fig. 2.18 Biological Neurons
(Ref. cs231n.github.io)

A biological neuron consists of a cell body with nucleus, dendrites, axon with axon terminals. The impulses are transferred in-between the neurons to pass the messages. The impulses from previous neurons are carried to the cell body through dendrites and passed on to next neuron through the axon. At cell body the impulses are processed. The cell body has a complex activation function.

A computational neural network model, signals x_0 are carried through axon interact multiplicatively with the dendrites $woxo$ where w_0 denote the synaptic strength which are

learnable parameters. The $woxo$ is summed at the cell body and if the sum is greater than a threshold value, the cell body fires the neuron and is sent to another neuron through axon. The threshold function is called activation function. In computational neural network model, the exact timing of firing is not that important but the frequency of firing is computed.

Neural networks have ability to detect patterns and derive information from complex data. Neural network works very differently from conventional computers. Conventional computers follow a set of instructions. It just acts as an arbitrary machine that follows instructions from humans. Neural network works similarly like human brain to process information.

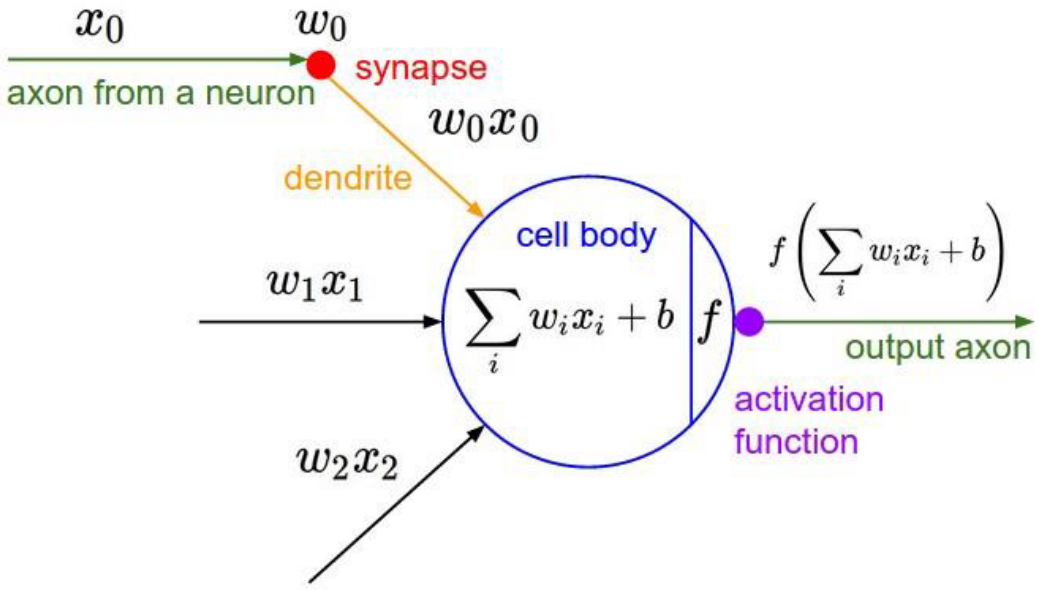


Fig. 2.19 Conventional model of Neurons

(Ref. cs231n.github.io)

Activation functions:

Activations are used to define the thresholding value above which the neurons are activated and signals are sent to next neurons. Thresholding values compares the sum of product of parametrized weights and input.

There are 5 types of activation functions mainly used.

1.Sigmoid:

It is a non-linear function which takes a real valued number and squashes it in between 0 and 1

Function: $\sigma(x) = 1/(1 + e - x)$ (2.1)

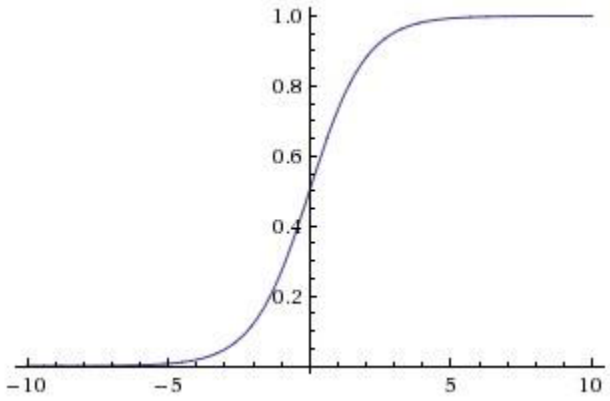


Fig. 2.20 Sigmoid Function

(Ref. cs231n.github.io)

2. Tanh:

It is a non-linear function which takes a real valued number and squashes it in between -1 and 1

Function: $tanh(x) = 2\sigma(2x) - 1$ (2.2)

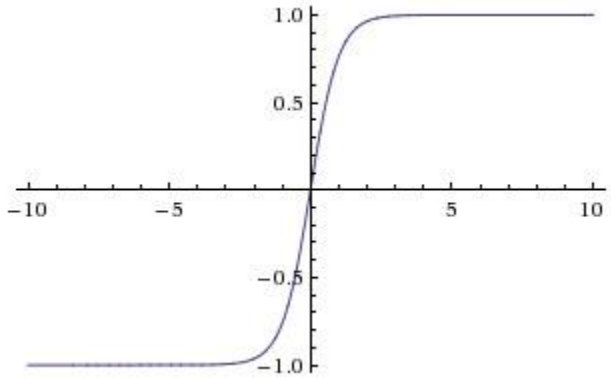


Fig. 2.21 Tanh Function

(Ref. cs231n.github.io)

3. ReLU:

This activation function is thresholded at zero

Function: $f(x) = max(0, x)$ (2.3)

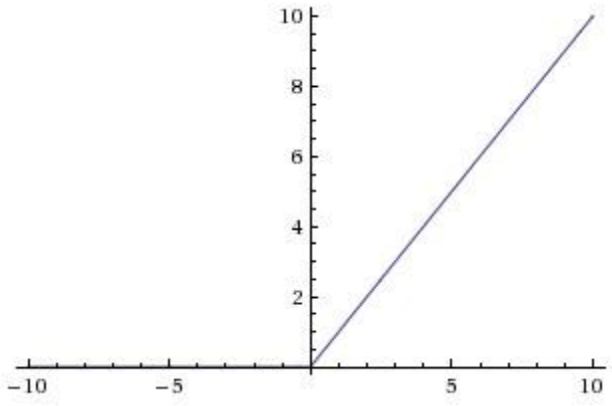


Fig. 2.22 ReLU Function

(Ref. cs231n.github.io)

4.Leaky ReLU:

It is very similar to ReLU but when x<0 this function has a small dip (negative slope).

Function: $f(x) = 1(x < 0)(\alpha x) + 1(x >= 0)(x)$ (2.4)

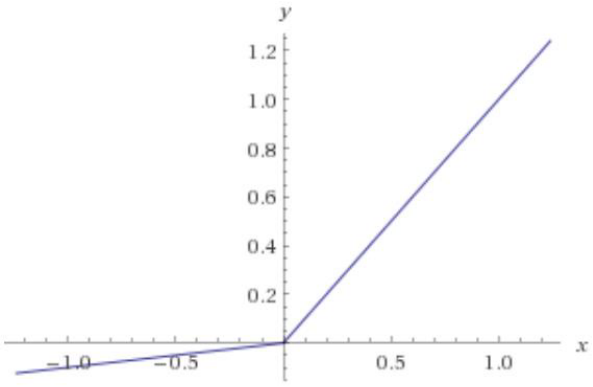


Fig. 2.23 Leaky ReLU Function

(Ref. cs231n.github.io)

5.Maxout:

It is a nonlinear function on dot product of weights and inputs plus additional biases. Both ReLU and Leaky ReLU are special cases of this function.

Function: $max(wT1x + b1, wT2x + b2)$ (2.5)

ReLU non-linearity function is mostly used as activation function. The next most probable choice is Leaky ReLU or Maxout, these two activation functions prevent the case of dying ReLU networks.

History of Neural networks:

The idea of neural networks was inspired from biological neural networks, when a scientist in 1960s found out the possible working of a neuron and tried to build neurons in electrical circuits. Through the evolution of computers, scientists were able to apply the idea of neural networks. The main idea for this inspiration was based on the notion of creating thinking machines which led to development of artificial intelligent.

The framework of neural network can be thought to be carried on from the linear function model of image classification problem. Image classification problem was aimed at classifying images based on class labels. In simple terms it aims at taking a random image and classifying it based on class labels defined. The initial strategies used were using Nearest Neighbor classifier and K Nearest Neighbor classifiers. Later, the model was shifted to a linear function mapping which consists of a function

$f(x_i, W, b) = \mathrm{W}x_i + \mathrm{b}$ (2.6)

where W denotes the weight parameters, x_i denotes input images and b denotes the bias

This linear function maps the images to a confident score function. The parameters W and b are learnable parameters and can be adjusted to obtain a good result. The function is called a score function in general.

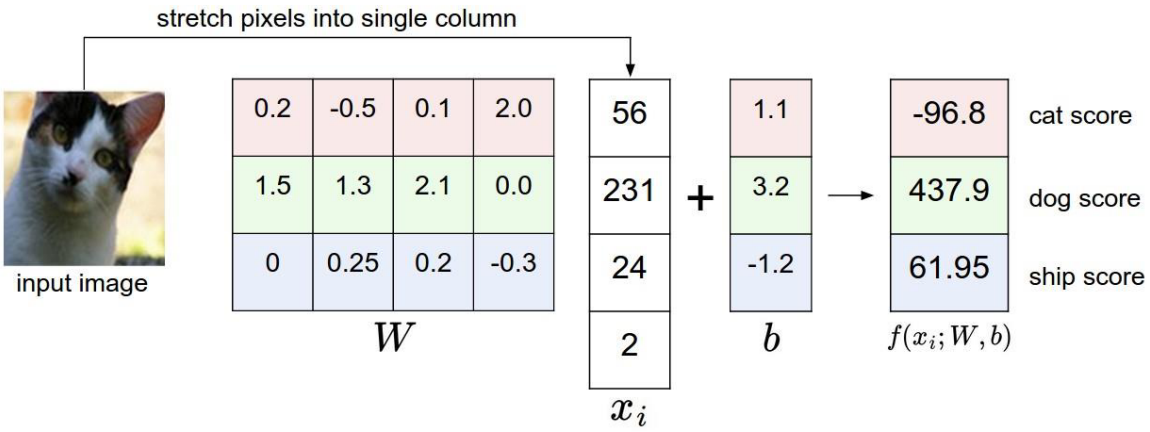


Fig. 2.24 Score Function Mapping

(Ref. cs231n.github.io)

Loss functions

After the score function is defined, the next step is to control the weight and bias parameters so as to get a class score consistent with the class labels. In order to achieve this, we want to find the outcomes which are more likely and which are least likely. This can be done by computing a loss function to compute the value of loss.

Types of Loss functions: Mean absolute loss, Mean squared loss, Root mean squared loss, SVM/Hinge loss, Softmax/Cross entropy loss. Loss functions can be classified into either Regression loss or Classification loss.

The two types of Loss functions used commonly:

1.Multiclass Support vector machine Loss (SVM Loss):

The SVM loss function computes the value of Loss such that the score of correct class is above a certain margin of all other scores.

The loss function is $L_i = \sum_{j \neq y_i} max(0, w_j^T x_i - w_{y_i}^T x_i + \Delta)$ (2.7)

Where Li denotes the sum of loss values of all incorrect classes for a score function, y_i denotes the correct class score and y_j denotes the current class score, Δ denotes the margin value, w_j is the j-th row of W reshaped as a column

The SVM loss is also called as the Hinge loss.

2. Softmax Loss:

Softmax Loss function is another popular and most commonly used loss function. It represents loss in a more understandable intuitive way and use probabilistic interpretation.

The loss function is defined as

$$L_i = -log(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}}) \tag{2.8}$$

or

$$L_i = -f_{y_i} + log\sum_j e^{f_j} \tag{2.9}$$

Where L_i denotes the sum of loss value of all incorrect classes , f_{y_i} denotes the score function for correct class and f_j denotes the score function for current class

The Softmax loss function is also called as Cross entropy loss function.

Regularization loss:

The one glitch that can be found while computing the loss function is that value of loss is same for two distinct values of weight parameters. In order to avoid this condition, L_2 regularization loss term is added to the loss function, The L_2 distance computes the square of difference between the values, the L_2 regularization loss terms consists of only weight parameters and doesn't include any input term.

Optimization of loss

After Loss is computed, the next step is to optimize the value of loss obtained. Optimization of loss can be achieved by computing gradient descent function. The gradient can be computed either analytically or by using calculus. The main idea behind using gradient descent is that it computes the value of slope of gradient that moves towards the below of the valley. The direction in which we move towards the least value also matters and can be set by tuning gradient descent step size. The batch size of neural networks can be huge and hence computing gradient descents can be difficult. Hence, the gradient descents are computed for mini-batch. Stochastic gradient descent is a special case of gradient descent which contains a single value.

Back propagation: learning gradients and weights

The main aim is to learn the controllable weight parameters. After optimizing the loss functions, we have to update the value of weights by back-propagating through the network. In other words, gradients back-propagate through the network and help us learn weights parameters. In order to understand the concept of back-propagation, we need to understand how gradients work and rules followed while we work with gradients.

There are three main rules/gates followed while computing gradients:

Add gate: Add gate takes the gradients of the outputs and distributes them equally to the inputs

Max gate: Max gate distributes the gradients of outputs to exactly one of the inputs which had the highest value during forward pass

Multiply gate: Multiply gate distributes the gradients of outputs to inputs in cross-distributed multiplication, like giving more gradients values to the inputs which had low values during forward pass and vice-versa.

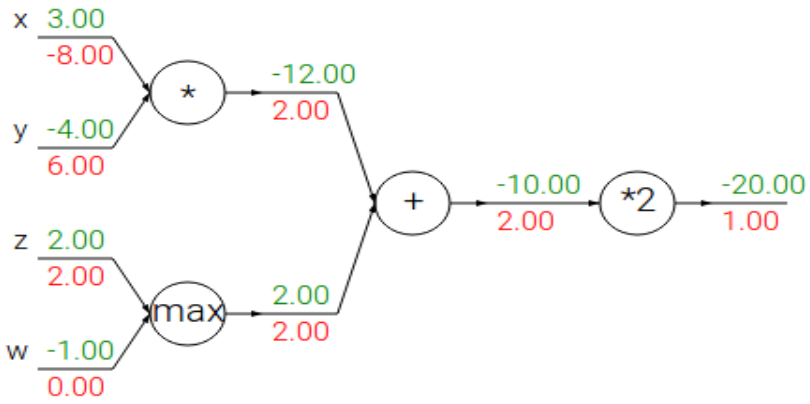


Fig. 2.25 Back propagation

(Ref. cs231n.github.io)

Gradient updates

While back-propagating through the network, gradients are computed. Gradient Descent can be very slow to run on large datasets. Because in one iteration of gradient descent, the algorithm requires a prediction of each instance in the training set and this can take a long time when there are millions of instances in the training set. In this situation, a variation of gradient descent called stochastic gradient descent is used.

The procedure of stochastic gradient descent is same as gradient descent but the update to the weights is performed after each training instance or after a mini batch rather than the whole batch of training set as in gradient descent. The learning can be much faster using this algorithm for very large training datasets and often only a small number of passes through the dataset to reach a good accuracy.

Gradient Descent with Momentum

Gradient descent with momentum converges faster than the standard gradient descent algorithm. It can be seen that standard gradient descent takes larger steps in the y- direction and smaller steps in the x-direction. This algorithm will be able to reduce the steps taken in the y-direction and concentrate the direction of the step in the x-direction, our algorithm would converge faster. This is what momentum does, it restricts the oscillation in one direction so that the algorithm can converge faster. Also, since the number of steps taken in the y-direction is restricted, a higher learning rate can be set.

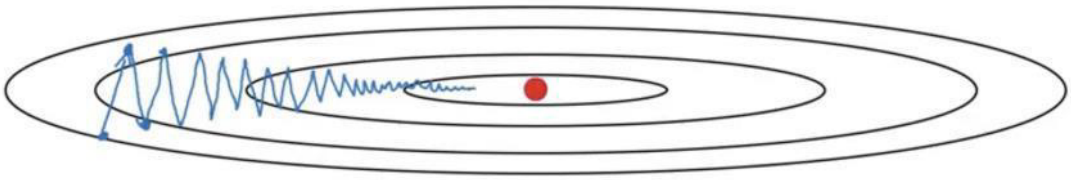


Fig. 2.26 Gradient Descent

(Ref. cs231n.github.io)

RMSprop

RMSprop was developed by Professor Geoffrey Hinton. It is similar to Gradient Descent with momentum. This algorithm restricts the oscillations in the vertical direction. Therefore, learning rate can be set to higher value. The difference between RMSprop and gradient descent is on how the gradients are calculated. The value of momentum is denoted by beta and is usually set to 0.9.

Adam

Adaptive Moment Estimation (Adam) combines ideas from both RMSprop and Momentum. The exponential average of the gradient as well as the squares of the gradient for each parameters is calculated. To decide our learning step, we multiply our learning rate by average of gradient and

divide it by the root mean square of the exponential average of square of gradients and then the weights are updated. The hyper parameter beta1 is generally 0.9 while beta2 is generally 0.99. Epsilon value is generally 1e-10.

Architecture of Neural Networks

Types of Neural networks: Feedforward and Feedback networks

Feedforward networks are straight networks that carry input in one direction. They associate the inputs directly with the output. Feedback networks are complex networks which contain intermediate states that vary throughout the process. These networks are very powerful. Feedback networks are also called as Recurrent Neural Networks.

Layers/Groups of Neural networks:

Input, Hidden and output

Input layers contains the input units containing raw information

Hidden layers contain hidden units connecting the input and output layers

Output layers contain units which are the result

Neural networks can also be classified based on single-layer or multi-layer neural networks. Neural networks are called as either artificial neural networks or single-layer/multi-layer perceptron interchangeably.

Fully Connected Neural Network

A full connected neural network is the otherwise called simply neural network is a network in which all the layers are connected in entirety with each other. It consists of an input layer, no of hidden layers and an output layer. In general, the number of hidden layers is 2 or 3, increasing the

number of hidden layers beyond 3 can bring about voluminous amount of parameters which can be difficult to handle. One drawback of fully connected layers is that when input is too big the number of parameters at each layer would proportionately be too high to deal with, this restricts the size of inputs of fully connected layers.

Convolutional Neural Networks

A Convolutional neural network differs from a fully connected neural network. In this type of neural network, the layers are not entirely connected with each other as in a fully connected neural network. Each layer is convolved with another layer using an appropriate filter size, pooling layers and stride of filter. Convolutional neural network also consists of input layer, hidden convolution layer, ReLU activation layers, Maxpool layers and fully connected output layers. Neurons in Convolutional neural networks differ from that of fully connected neural networks. Neurons are described by a 3 dimensional $h * w * d$ value (height width and depth).

Recurrent Neural Networks

A recurrent neural network is different from the feedforward neural network in the sense it has a feedback loop. The recurrent neural networks are otherwise called as feedback neural networks, which has a complex structure and can be used for powerful computation with respect to memory considerations. The memory part of recurrent networks plays an important role, which is achieved by monitoring each intermediate state in each loop. This attribute of recurrent neural network well distinguishes it from feedforward neural network. The RNNs can easily learn and predict time series of numbers, any patterns. The RNNs have wide range of application when it comes to predicting time series sequence as it gives importance to the intermediate state.

The greatest disadvantage of Recurrent neural networks is that the gradients computed during back propagation can be either over exponentially increasing or vanishing. When the gradients are exceedingly increasing, it is very hard to compute the number of parameters that can be involves and similarly when the value of gradients are vanishing beyond a certain value, the neural nitrogen wouldn't be able to detect any concrete changes. It is easier to deal with exploding gradients then vanishing gradients.

Long short term memory neural networks

Long short term memory network is also a recurrent neural network. It was found in the year 1997. It gives the solution for the problem of increasing/vanishing gradients. It has four gates in its structure: Input gate, gate gate, output gate, flag gate. These gates facilitate the amount of information to be stored, deleted, retrieved. LSTMs have proven to be very useful in predicting time series prediction. One drawback of LSTM would be that it only takes into account the probabilities of state rather than the actual value of state.

Training Neural network

1.Preprocessing

The most important step in training neural network is to preprocessing the input, it saves a lot of noise occurring in the network and cut down the possible errors. Some of the methods used in Preprocessing are Mean subtraction, Normalization, PCA and whitening.

2.Weight Initialization

Another most important to take care before training is to set the initial value to weights. Setting weights equal to zero is not ideal as it can lead to easier fading out of networks, also uniform initialization of weights is not good, since all the inputs parameters will be equally affected. The most ideal way to initialize weights

is to have a very low random value. The most common method used for initializing weights is Xavier Initialization.

3.Learning and evaluation

After initialization and preprocessing, next step is to start learning. The dataset is used as input to train and test neural network. The dataset is split into two parts, namely train and test. The train data contains data with known output, the model learns on this data to generalize. The test data is a subset of the entire dataset which is used to find model's prediction and efficiency for the test subset. Generally, the model is first trained on train dataset and later is tested on test dataset.

4.Hyper-parameter optimization

One important step about training neural network model is hyper-parameter optimization. Hyper-parameters are variables which change over the course of training neural networks. These parameters are learnable and play an important role in shaping the neural network model.

Architecture of Neural network and specifications

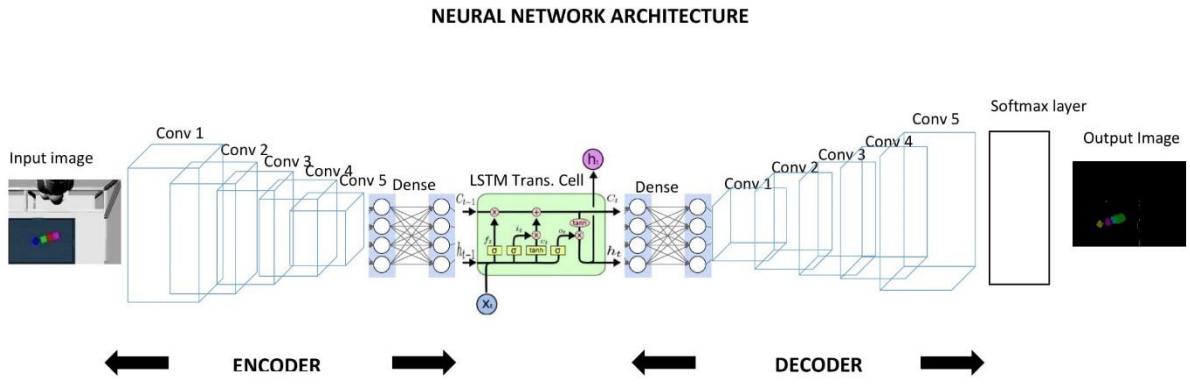


Fig. 2.27 Neural network architecture

Weight Initialization: Xavier Initialization

Optimizer: ADAM Optimizer

Encoder: 5 convolutional layers and 1 dense layer

Decoder: 5 convolutional layers and 1 dense layer

Loss function: Maray cross entropy

Activation Function: ReLU non linearity

Train dataset: 400

Test dataset: 100

No of images per folder: 15

CHAPTER 3

SCOPE OF PRESENT WORK

In this section, the present scope of the project will be elaborated. The detailed description of applications where and how the project can be implemented is explained.

The main objectives of the project are

1. To understand the interaction of a Robotic arm with its environment by understanding the underlying physics
2. To manipulate the Robotic arm to build and break a tower of blocks by grasping the boxes
3. To record images containing the pose and orientation of the boxes during the process of building and breaking the tower
4. To train and test a Recurrent neural network to predict the future possible state of the blocks using the images recorded

The project aims in establishing an interaction of the Robot with its surrounding environment by understanding the physics. Various position, orientation and depth map of objects are captured as images. Images are processed to obtain a concrete dataset for further training with a neural network to predict states at future sequence through machine learning algorithms. The project is amalgamation of Robotics, Machine learning and Computer vision put together to achieve Self Learning of the robot. The concept behind Self learning is self-explanatory as the name suggests. It forms the first step of any learning. Human babies try to learn to walk or pick any object by various trial and errors through self-learning. The analogy between human babies and robots can be brought about in this way. The project is based on Human like robots or Humanoid Robots. The

core principle of Humanoid robotics is to make robots mimic and understand the environment like human beings do. The various primary tasks that can be achieved by Humanoid robots are moving, picking, placing, grasping, lifting, rotating. They can be used to do routine tasks in factory, tasks in a nuclear research environment that are harmful for human being like disposal of nuclear waste, complex tasks. The quality that differentiates humanoid robot from other type of robots is the intelligence or the ability to learn by understanding the environment.

CHAPTER 4

EXPERIMENTAL PROCEDURE

The project consists of two parts:

1. Simulating a 7 Degrees of Freedom Panda arm of Franka Emika Robot in a Gazebo/V-rep Simulation environment, to make it repeatedly build and break a tower of blocks.
2. Recording color images, depth map containing pose and orientation of blocks to generate a dataset of images which is used in training a Recurring Neural Network for predicting the next sequence denoting the position and orientation of the blocks.

4.1 Building and breaking a tower of blocks

4.1.1 Pre-requisites

The most important need for simulation is a very good simulation environment with good physics engine built-in. The simulation software must be able to replicate the physics properties of the real world. It must also be able to provide with huge variety of models starting from simple primitive shape to complex robots. The simulation environment must also be equipped with easy navigation, controlling, editing, tuning parameters, adjusting kinematic and dynamic properties of objects. It must lay down a framework for setting up controllers to control various parts of the robot model. It is also essential to integrate the simulation environment with ROS by adding suitable services and messages. Gazebo simulation and V-rep simulation are good simulation environments with all the above pre-requisites. They both have 4 built in physics engine and huge model browser and ability to import/export models in SDF/URDF format. Both Gazebo and V-rep can be integrated with ROS. Gazebo has separate gazeboros packages and gazeboros control which can be downloaded and built

with catkin. In Vrep, ROS integration can be achieved by using downloading vrep_ros_inteface package and building with catkin by again adding suitable services and messages.

4.1.2 Pre-Processing

The steps involved in pre-processing are briefly explained below:

Step 1: Importing Franka model

Importing Franka Emika model with Gazebo/Vrep involves importing the associated URDF script

```
<?xml version="1.0" ?>
<robot name="panda" xmlns:xacro="http://www.ros.org/wiki/xacro">
  <link name="panda_link0">
    <visual>
      <geometry>
        <mesh filename="package://Panda/meshes/collision/link0.obj"/>
      </geometry>
      <material name="panda_white"/>
    </visual>
    <collision>
      <geometry>
        <mesh filename="package://Panda/meshes/collision/link0.obj"/>
      </geometry>
      <material name="panda_white"/>
    </collision>
  </link>
  <link name="panda_link1">
```

```
    <visual>
      <geometry>
        <mesh filename="package://Panda/meshes/collision/link1.obj"/>
      </geometry>
      <material name="panda_white"/>
    </visual>
    <collision>
      <geometry>
        <mesh filename="package://Panda/meshes/collision/link1.obj"/>
      </geometry>
      <material name="panda_white"/>
    </collision>
  </link>
  <joint name="panda_joint1" type="revolute">
    <safety_controller k_position="100.0" k_velocity="40.0" soft_lower_limit="-2.8973"
soft_upper_limit="2.8973"/>
    <origin rpy="0 0 0" xyz="0 0 0.333"/>
    <parent link="panda_link0"/>
    <child link="panda_link1"/>
    <axis xyz="0 0 1"/>
    <limit effort="87" lower="-2.9671" upper="2.9671" velocity="2.1750"/>
  </joint>
  <link name="panda_link2">
    <visual>
```

```
<geometry>
```

Fig. 4.1 Franka Panda URDF file

The above URDF file is a part of Panda.urdf file used for panda model.

The panda robot consists of an arm with 7 joints and links attached with a hand having a left and right joint and links

Step 2: Setting up the Work table, Camera, cardboard box

```
?xml version='1.0'?>
sdf version="1.4">
model name="workcell-assembly-v2">
static>true</static>

 <!-- workcell-assembly-v2 -->
 <link name="workcell-assembly-v2">
   <pose>0 0 0.07 0 0 0</pose>
   <inertial>
     <mass>13.55</mass>
     <inertia><ixx>0.294</ixx><iyy>0.294</iyy><izz>0.023</izz><ixy>0</ixy><ixz>0</ixz><iyz>0</iyz></inertia>
   </inertial>
   <collision name="collision">
     <geometry>
         <mesh>
           <uri>model://workcell-assembly-v2/meshes/workcell-assembly-v2.stl</uri>
         </mesh>
     </geometry>
   </collision>
   <visual name="visual">
     <geometry>
         <mesh>
           <uri>model://workcell-assembly-v2/meshes/workcell-assembly-v2.stl</uri>
         </mesh>
     </geometry>
   </visual>
 </link>

<!-- robot-base-fixation1 -->
 <link name="robot-base-fixation1">
   <pose>0.62 0.138 0.97 0 0 0</pose>                <!-- to move the arm-base edit the x-coordinate -->
   <inertial>
     <mass>1.84</mass>
     <inertia><ixx>0.294</ixx><iyy>0.294</iyy><izz>0.023</izz><ixy>0</ixy><ixz>0</ixz><iyz>0</iyz></inertia>
   </inertial>
   <collision name="collision">
     <geometry>
         <mesh>
```

Fig. 4.2 Work cell assembly with kinect camera SDF file

Step 3: Adding the models to world/scene

In gazebo, after models are imported they are added into world file. The gazebo server parses the world file and gazebo graphical enables users to visualize the elements.

world file:

```
<?xml version="1.0" ?>

<sdf version="1.4">

        <world name="default">

        <scene>

     <ambient>0.0 0.0 0.0 1.0</ambient>
```

```
    <shadows>0</shadows>
  </scene>
  <include>
    <uri>model://ground_plane</uri>
  </include>
  <include>
    <uri>model://sun</uri>
    <pose>0 0 0 0 0 0</pose>
  </include>
  <include>
    <uri>model://open-cardboard-box</uri>
    <pose>0.820397 0.656998 0.972500 0 0 0</pose>
  </include>
  <include>
    <uri>model://box</uri>
    <pose>0.84 0.67 1.02 0 0 0</pose>
  </include>
  <include>
    <uri>model://box1</uri>
    <pose>0.84 0.67 1.14 0 0 0</pose>
  </include>
  <include>
    <uri>model://box2</uri>
    <pose>0.84 0.67 1.26 0 0 0</pose>
  </include>
```

```
    <include>
      <uri>model://box3</uri>
      <pose>0.84 0.67 1.38 0 0 0</pose>
    </include>
    <!--include>
      <uri>model://Brick</uri>
      <pose>19 0.5 1.2 0 0 0</pose>
    </include>
    <include>
      <uri>model://Concrete_Roadblock</uri>
      <pose>20 0.5 1.2 0 0 0</pose>
    </include-->
    <include>
          <uri>model://workcell-assembly-v2-kinect</uri>
      <name>workcell-assembly-v2-kinnect</name>
      <pose>0 0 0 0 0 0</pose>
    </include>
  </world>
</sdf>
```

Fig. 4.3 Gazebo World file

In Vrep, all models are added into the scene hierarchy browser dialog box manually which also enables us to view /edit properties of each sub-element.

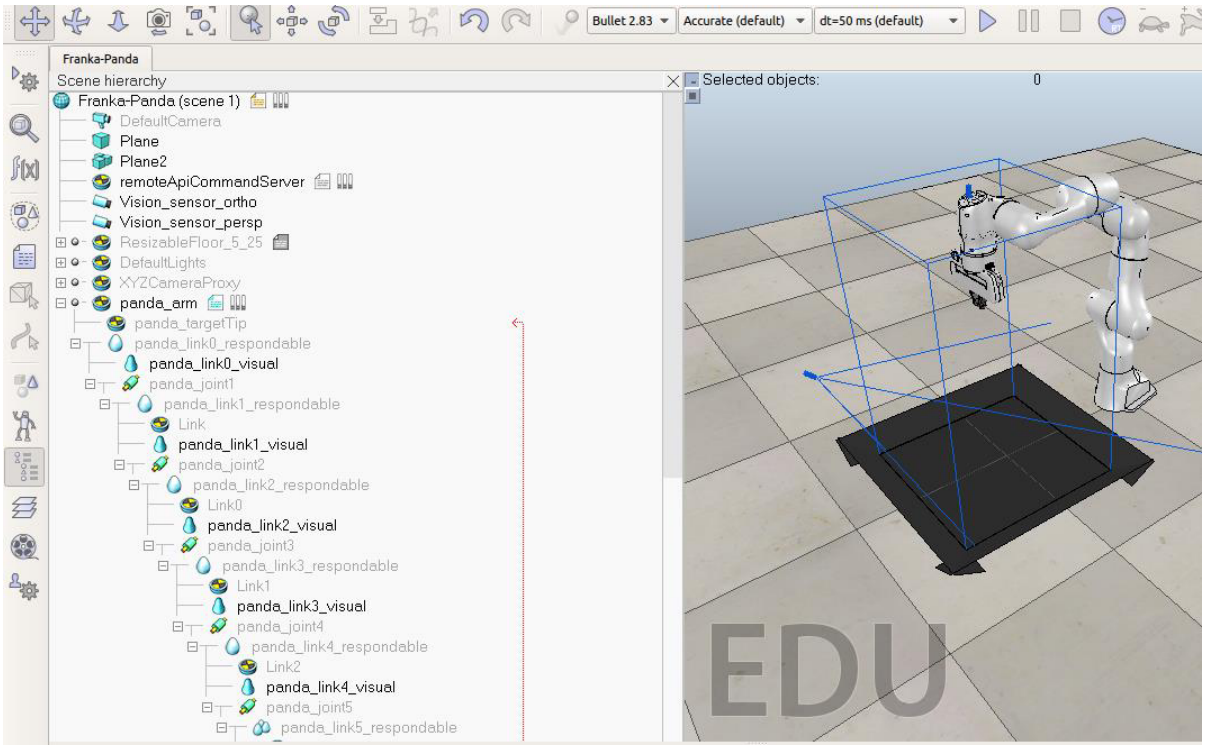


Fig 4.4 Scene Hierarchy Browser

Step 4: Setting up Dynamic properties and controllers in Vrep/ Gazebo

In Gazebo, the dynamic and controller's configuration are setup using MoveIt setup assistant

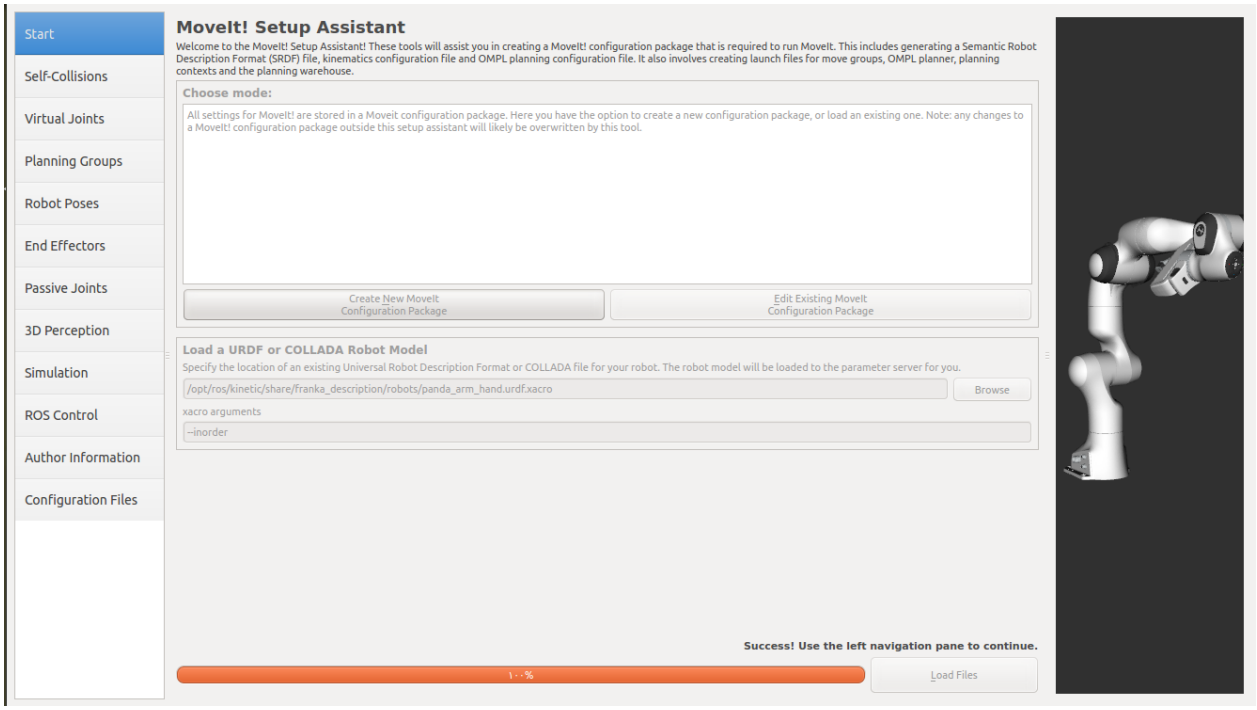


Fig. 4.5 MoveIt! Setup Assistant

(Ref. www. moveit.ros.org)

In Vrep, after a robot model is imported, it is required to setup each joint as dynamic and selecting suitable controller and enabling the control loop. One important step before this would be to check the scene hierarchy.

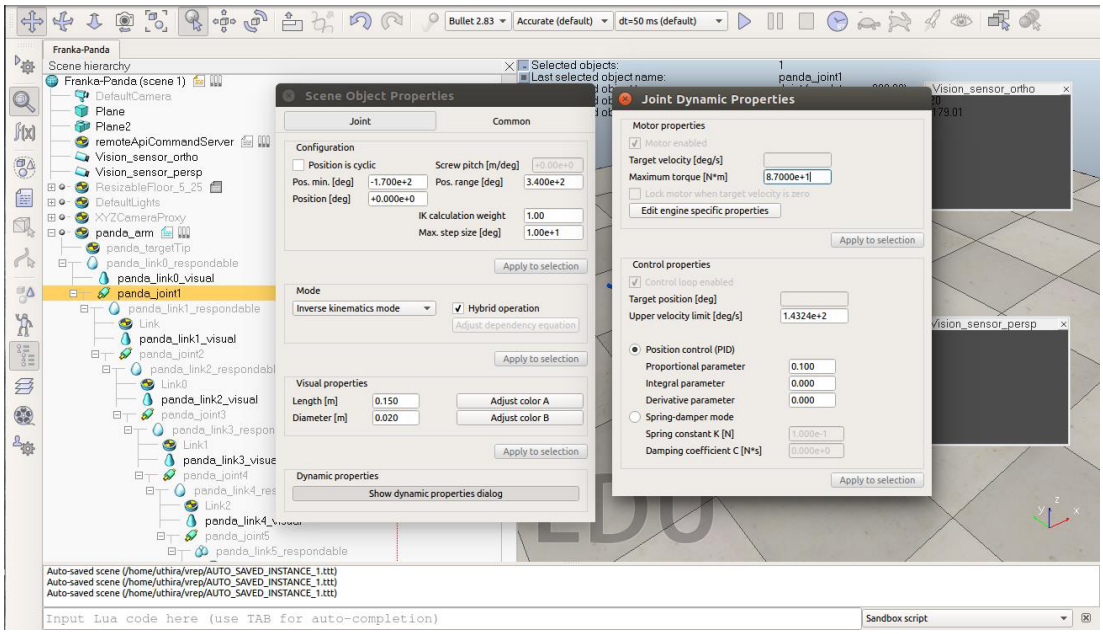


Fig. 4.6 Dynamics Properties Dialog box

Step 5: Attaching script

In Vrep, it is required to add a main script to specify to initialize parameters and setting up control loop signals and a threaded/non threaded child script for each part of the robot specifying its position, orientation, object name, object handle.

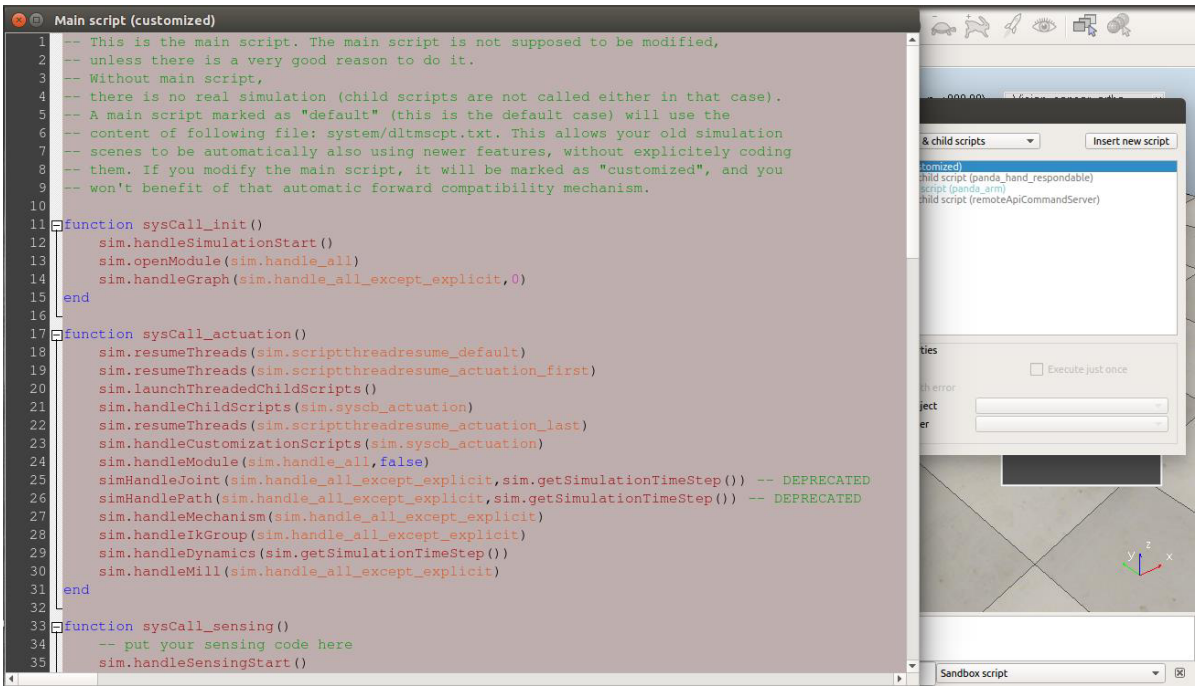


Fig. 4.7 Main Script

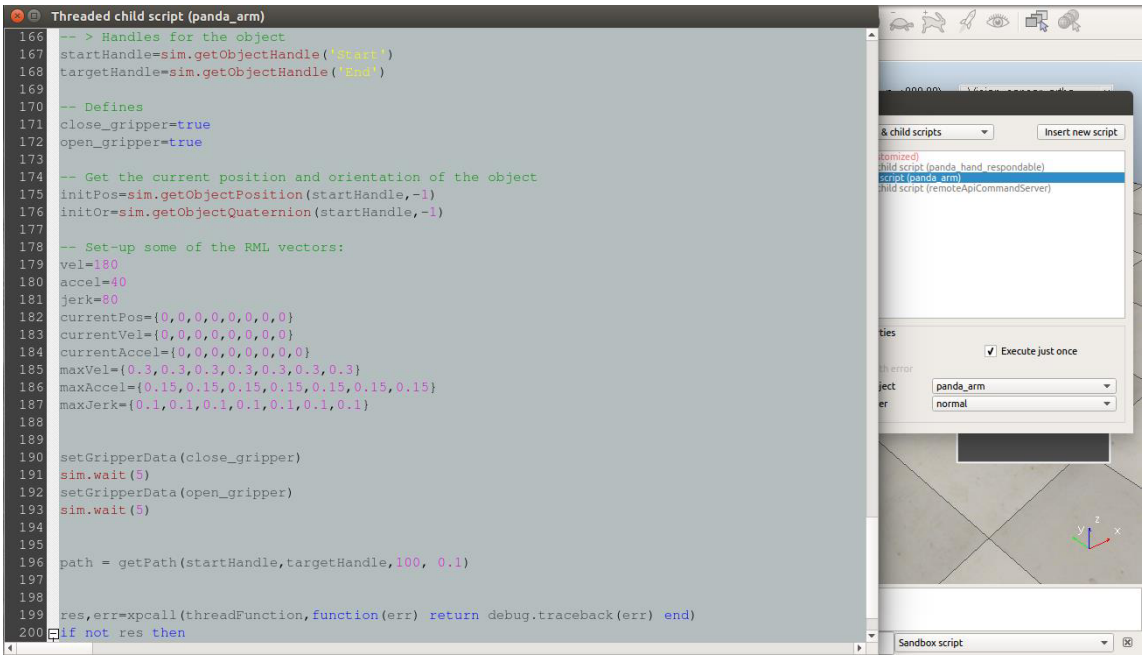


Fig. 4.8 Panda_arm child script

In Gazebo, the URDF file along with suitable scripts are added to launch file along with description file. On launching the launch file, the gazebo simulation environment is opened along with the world file containing the models.

Launch file:

```
<launch>
<env name="GAZEBO_MODEL_PATH" value="$(find
franka_arm_gripper_realsense_gazebo_description)/model:$(find
franka_arm_gripper_realsense_gazebo_description)/props_models:$(optenv
GAZEBO_MODEL_PATH)" />
   <!-- Launch Gazebo -->
   <include file="$(find gazebo_ros)/launch/empty_world.launch">
      <arg name="world_name" value="$(find
franka_arm_gripper_realsense_gazebo_description)/worlds/sim_scenario#3.world"/>
      <arg name="paused" value="false"/>
      <arg name="use_sim_time" value="true"/>
      <arg name="gui" value="true"/>
      <arg name="recording" value="false"/>
      <arg name="debug" value="false"/>
      <arg name="physics" value="ode" />
   </include>
   <!-- Generate/Load robot description file -->
   <include
file="$(findfranka_arm_gripper_realsense_gazebo_description)/launch/description.launch"/>
   <!-- Spawn urdf into Gazebo -->
   <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model" args="-param
robot_description -urdf -model arm gripper" />
```

```
    <node name="record_topics_for_verification" pkg="rosbag" type="play"    args="
/home/uthira/catkin_ws/src/franka_active_sensing/franka_arm_gripper_realsense_gazebo_descript
ion/scripts/pointclouds.bag" output="screen">
  <param name="use_sim_time" value="true" />
  <remap from="/kinect_camera/depth/points" to="/kinect_camera/depth/points2"/>
    </node>
    <!-- Start moveit node -->
  <include file="$(find arm_control)/launch/moveit.launch"/>
      <!-- start rviz node -->
  <include file="$(find arm_control)/launch/rviz.launch"/>
    <!-- run init object models script -->
  <!--node pkg="franka_arm_gripper_realsense_gazebo_description" name="init_models"
type="init_models.py" output="screen">
    </node-->
  </launch>
```

Fig. 4.9 Launch file

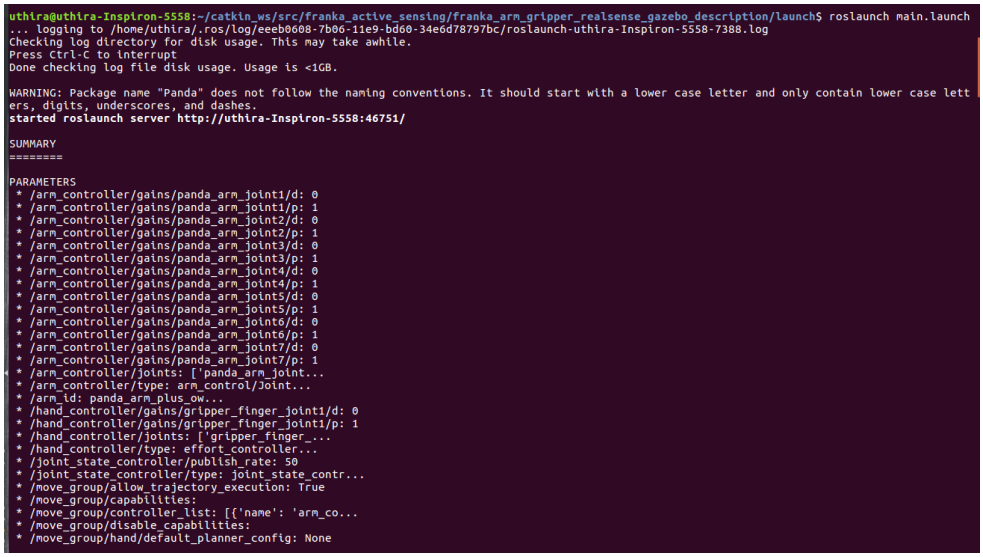


Fig. 4.10 Roslaunch main launch

Step 6: Adding boxes

The most important step involved would be to define the properties of box which is used for making and breaking the tower. In gazebo, we define SDF file of each box which is used in simulation and boxes are included along with the world file.

SDF file:

```
<?xml version='1.0'?>
<sdf version ='1.6'>
  <model name ='box'>
    <pose>0 0 0 0 0 0</pose>
    <static>false</static>
    <link name ='link'>
    <inertial>
      <mass>5.0</mass>
      <inertia> <!-- inertias are tricky to compute -->
        <!-- http://gazebosim.org/tutorials?tut=inertia&cat=build_robot -->
        <ixx>0.083</ixx>       <!-- for a box: ixx = 0.083 * mass * (y*y + z*z) -->
        <ixy>0.0</ixy>         <!-- for a box: ixy = 0 -->
        <ixz>0.0</ixz>         <!-- for a box: ixz = 0 -->
        <iyy>0.083</iyy>       <!-- for a box: iyy = 0.083 * mass * (x*x + z*z) -->
        <iyz>0.0</iyz>         <!-- for a box: iyz = 0 -->
        <izz>0.083</izz>       <!-- for a box: izz = 0.083 * mass * (x*x + y*y) -->
      </inertia>
    </inertial>
    <collision name="collision">
      <geometry>
```

```
        <box>
          <size>0.07 0.07 0.07</size>
        </box>
      </geometry>
    </collision>
    <visual name="visual">
      <geometry>
        <box>
          <size>0.07 0.07 0.07</size>
        </box>
      </geometry>
      <physics type="ode">
       <real_time_update_rate>1000</real_time_update_rate>
      </physics>
      <material>
        <ambient>0 0 0 0</ambient>
        <diffuse>1 0 0 1</diffuse>
        <specular>0 0 0 0</specular>
        <emissive>0 0 0 0</emissive>
      </material>
    </visual>
    </link>
  </model>
</sdf>
```

Fig. 4.11 Box SDF file

In Vrep, we initialize the boxes by setting up their properties in run-time by calling import shape script using remote API server.

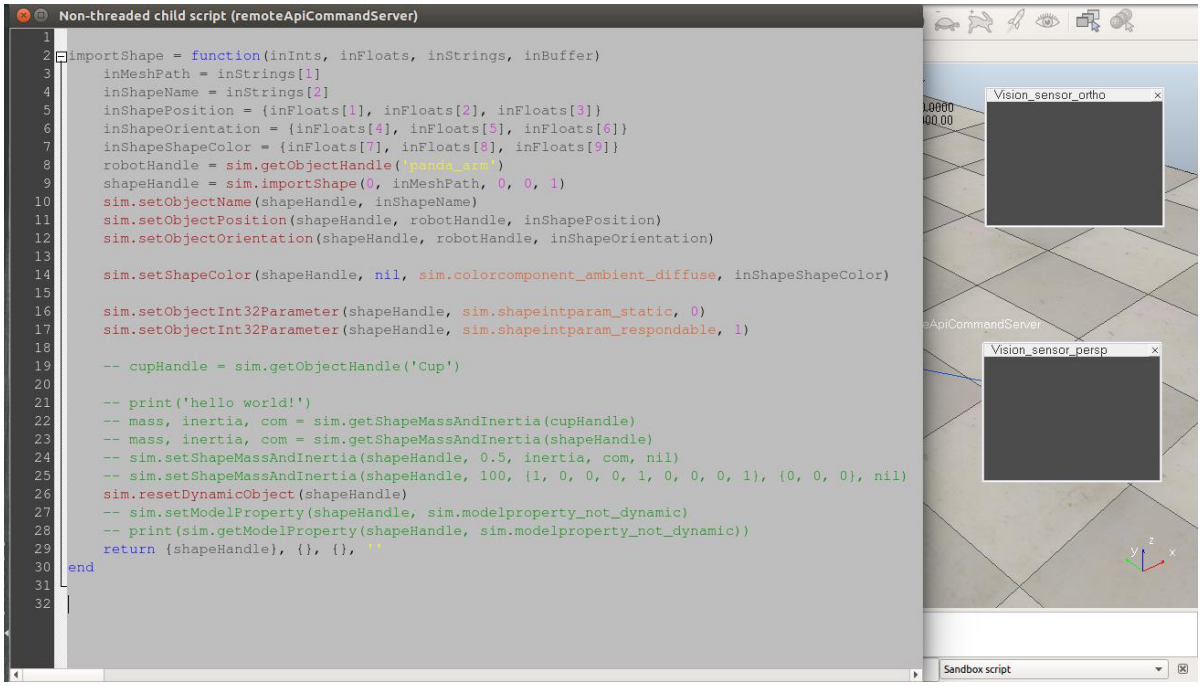


Fig. 4.12 Remote API Server script

```
            self.add_objects()

    def add_objects(self):

        # Add each object to robot workspace at x,y location and orientation (random or pre-loaded)
        self.object_handles = []
        sim_obj_handles = []
        for object_idx in range(len(self.obj_mesh_ind)):
            #print(object_idx)
            print("Adding objects")
            curr_mesh_file = os.path.join(self.obj_mesh_dir, self.mesh_list[self.obj_mesh_ind[object_idx]])
            if self.is_testing and self.test_preset_cases:
                curr_mesh_file = self.test_obj_mesh_files[object_idx]
            curr_shape_name = 'shape_%02d' % object_idx
            drop_x = (self.workspace_limits[0][1] - self.workspace_limits[0][0] - 0.2) * np.random.random_sample() + self.workspace_limits[0][0]
+ 0.1+(0.05*object_idx)
            drop_y = (self.workspace_limits[1][1] - self.workspace_limits[1][0] - 0.2) * np.random.random_sample() + self.workspace_limits[1][0]
+ 0.1+(0.05*object_idx)
            object_position = [drop_x, drop_y, 0.15]
            object_orientation = [2*np.pi*np.random.random_sample(), 2*np.pi*np.random.random_sample(), 2*np.pi*np.random.random_sample()]
            if self.is_testing and self.test_preset_cases:
                object_position = [self.test_obj_positions[object_idx][0], self.test_obj_positions[object_idx][1], self.test_obj_positions
[object_idx][2]]
                object_orientation = [self.test_obj_orientations[object_idx][0], self.test_obj_orientations[object_idx][1],
self.test_obj_orientations[object_idx][2]]
            object_color = [self.obj_mesh_color[object_idx][0], self.obj_mesh_color[object_idx][1], self.obj_mesh_color[object_idx][2]]
            ret_resp,ret_ints,ret_floats,ret_strings,ret_buffer = vrep.simxCallScriptFunction(self.sim_client,
'remoteApiCommandServer',vrep.sim_scripttype_childscript,'importShape',[0,0,255,0], object_position + object_orientation + object_color,
[curr_mesh_file, curr_shape_name], bytearray(), vrep.simx_opmode_blocking)
            if ret_resp == 8:
                print('Failed to add new objects to simulation. Please restart.')
                exit()
            curr_shape_handle = ret_ints[0]
            self.object_handles.append(curr_shape_handle)
            if not (self.is_testing and self.test_preset_cases):
                time.sleep(2)
        self.prev_obj_positions = []
        self.obj_positions = []
```

Fig. 4.13 Adding Objects Script

4.1.3 Simulation

Python script is called for enabling connection to Vrep API server. When this script is executed, the connection to the remote server is established, the robot gripper is initialized, the boxes are instantiated into the Vrep environment. Using two functions grasp and push, the boxes are grasped to build a tower of blocks and pushed to break them by gripper by properly configuring the position and orientation of gripper.

```
    def grasp(self, position,heightmap_rotation_angle, workspace_limits,set_position,orientation,set_flag=0):
        print('Executing: grasp at (%0.16f, %0.16f, %0.16f)' % (position[0], position[1], position[2]))

        if self.is_sim:

            # Compute tool orientation from heightmap rotation angle
            tool_rotation_angle = (heightmap_rotation_angle % np.pi) - np.pi/2

            # Avoid collision with floor
            position = np.asarray(position).copy()
            #position[2] = max(position[2]-0.04, workspace_limits[2][0] + 0.002)

            # Move gripper to location above grasp target
            grasp_location_margin = 0.09
            # sim_ret, UR5_target_handle = vrep.simxGetObjectHandle(self.sim_client,'UR5_target',vrep.simx_opmode_blocking)
            location_above_grasp_target = (position[0], position[1], position[2] + grasp_location_margin)

            # Compute gripper position and linear movement increments
            tool_position = location_above_grasp_target
            sim_ret, UR5_target_position = vrep.simxGetObjectPosition(self.sim_client, self.UR5_target_handle,-1,vrep.simx_opmode_blocking)
            move_direction = np.asarray([tool_position[0] - UR5_target_position[0], tool_position[1] - UR5_target_position[1], tool_position[2]
- UR5_target_position[2]])
            move_magnitude = np.linalg.norm(move_direction)
            move_step = 0.05*move_direction/move_magnitude
            asd=np.floor(move_direction[0]/move_step[0])
            if not math.isnan(asd):
             num_move_steps = int(asd)
            else:
             num_move_steps = 3
            # Compute gripper orientation and rotation increments
            sim_ret, gripper_orientation = vrep.simxGetObjectOrientation(self.sim_client, self.UR5_target_handle, -1, vrep.simx_opmode_blocking)
            rotation_step = 0.3 if (tool_rotation_angle - gripper_orientation[1] > 0) else -0.3
            num_rotation_steps = int(np.floor((tool_rotation_angle - gripper_orientation[1])/rotation_step))


            # Simultaneously move and rotate gripper
            for step_iter in range(max(num_move_steps, num_rotation_steps)):
                #print(step_iter)
                vrep.simxSetObjectPosition(self.sim_client,self.UR5_target_handle,-1,(UR5_target_position[0] + move_step[0]*min
(step_iter,num_move_steps), UR5_target_position[1] + move_step[1]*min(step_iter,num_move_steps), UR5_target_position[2] + move_step[2]*min
```

Fig. 4.14 Grasp function

```
    def push(self, position, heightmap_rotation_angle, workspace_limits):
        position = np.asarray(position)
        print('Executing: push at (%f, %f, %f)' % (position[0], position[1], position[2]))

        if self.is_sim:

            # Compute tool orientation from heightmap rotation angle
            tool_rotation_angle = (heightmap_rotation_angle % np.pi) - np.pi/2

            # Adjust pushing point to be on tip of finger
            position[2] = position[2] + 0.026

            #push_val = [[1.0,0.0],[0.9,0.1],[0.7,0.5]]
            # Compute pushing direction
            direction_val=['right','left','top','down']
            direc=random.choice(direction_val)
            print('pushing towrds',direc)
            if(direc=='right'):
              position[0]=position[0]-0.09
              push_orientation=[1.0,0.0]
            elif(direc=='left'):
              position[0]=position[0]+0.09
              push_orientation=[-1.0,0.0]
            elif(direc=='top'):
              position[1]=position[1]+0.09
              push_orientation=[0.0,-1.0]
            elif(direc=='down'):
              position[1]=position[1]-0.09
              push_orientation=[0.0,1.0]
            push_direction = np.asarray([push_orientation[0]*np.cos(heightmap_rotation_angle) - push_orientation[1]*np.sin
(heightmap_rotation_angle), push_orientation[0]*np.sin(heightmap_rotation_angle) + push_orientation[1]*np.cos(heightmap_rotation_angle)])
            print('push direction',push_direction)
            # Move gripper to location above pushing point
            pushing_point_margin = 0.1
            location_above_pushing_point = (position[0], position[1], position[2] + pushing_point_margin)

            # Compute gripper position and linear movement increments
            tool_position = location_above_pushing_point
            sim_ret, UR5_target_position = vrep.simxGetObjectPosition(self.sim_client, self.UR5_target_handle,-1,vrep.simx_opmode_blocking)
```

Fig. 4.15 Push function

4.1.4 Post Processing/Fine-Tuning

The grasp action is fine-tuned by changing the pose of the gripper suitably by considering both position and orientation.

```
    def push_to(self,pose, move_step_size=0.01, single_step=False):

        # Get current position and orientation of UR5 target
        sim_ret, UR5_target_position = vrep.simxGetObjectPosition(self.sim_client, self.UR5_target_handle,-1,vrep.simx_opmode_blocking)
        sim_ret, UR5_target_orientation = vrep.simxGetObjectOrientation(self.sim_client, self.UR5_target_handle, -1, vrep.simx_opmode_blocking)

        # Calculate the movement increments
        move_direction = pose[:3,-1] - UR5_target_position
        move_magnitude = np.linalg.norm(move_direction)
        move_step = move_step_size * move_direction / move_magnitude
        num_move_steps = int(np.ceil(move_magnitude / move_step_size))

        # Calculate the rotation increments
        rotation = np.asarray(transformations.euler_from_matrix(pose))
        rotation_step = rotation - UR5_target_orientation
        rotation_step[rotation >= 0] = 0.1
        rotation_step[rotation < 0] = -0.1
        num_rotation_steps = np.ceil((rotation - UR5_target_orientation) / rotation_step).astype(np.int)

        # Move and rotate to the target pose
        if not single_step:
          for i in range(max(num_move_steps, np.max(num_rotation_steps))):
            pos = UR5_target_position + move_step*min(i, num_move_steps)
            rot = [np.pi/2,UR5_target_orientation[1]+rotation_step[1]*min(i, num_rotation_steps[1]),np.pi/2]
            vrep.simxSetObjectPosition(self.sim_client,self.UR5_target_handle,-1,pos,vrep.simx_opmode_blocking)
            vrep.simxSetObjectOrientation(self.sim_client, self.UR5_target_handle, -1, rot, vrep.simx_opmode_blocking)

        vrep.simxSetObjectPosition(self.sim_client,self.UR5_target_handle,-1,pose[:3,-1],vrep.simx_opmode_blocking)
        vrep.simxSetObjectOrientation(self.sim_client, self.UR5_target_handle, -1, (np.pi/2,rotation[1],np.pi/2), vrep.simx_opmode_blocking)

    def move_to(self, tool_position,tool):

        if self.is_sim:
            #print('moving grippper to target')
            # sim_ret, UR5_target_handle = vrep.simxGetObjectHandle(self.sim_client,'UR5_target',vrep.simx_opmode_blocking)
            sim_ret, UR5_target_position = vrep.simxGetObjectPosition(self.sim_client, self.UR5_target_handle,-1,vrep.simx_opmode_blocking)

            move_direction = np.asarray([tool_position[0] - UR5_target_position[0], tool_position[1] - UR5_target_position[1], tool_position[2]
- UR5_target_position[2]])
```

Fig. 4.16 Push to function

The push action is fine-tuned by defining the push direction in all directions and push length and orientation.

```
def push(self, position, heightmap_rotation_angle, workspace_limits):
    position = np.asarray(position)
    print('Executing: push at (%f, %f, %f)' % (position[0], position[1], position[2]))

    if self.is_sim:

        # Compute tool orientation from heightmap rotation angle
        tool_rotation_angle = (heightmap_rotation_angle % np.pi) - np.pi/2

        # Adjust pushing point to be on tip of finger
        position[2] = position[2] + 0.026

        #push_val = [[1.0,0.0],[0.9,0.1],[0.7,0.5]]
        # Compute pushing direction
        direction_val=['right','left','top','down']
        direc=random.choice(direction_val)
        print('pushing towrds',direc)
        if(direc=='right'):
          position[0]=position[0]-0.09
          push_orientation=[1.0,0.0]
        elif(direc=='left'):
          position[0]=position[0]+0.09
          push_orientation=[-1.0,0.0]
        elif(direc=='top'):
          position[1]=position[1]+0.09
          push_orientation=[0.0,-1.0]
        elif(direc=='down'):
          position[1]=position[1]-0.09
          push_orientation=[0.0,1.0]
        push_direction = np.asarray([push_orientation[0]*np.cos(heightmap_rotation_angle) - push_orientation[1]*np.sin
ightmap_rotation_angle), push_orientation[0]*np.sin(heightmap_rotation_angle) + push_orientation[1]*np.cos(heightmap_rotation_angle)])
        print('push direction',push_direction)
        # Move gripper to location above pushing point
        pushing_point_margin = 0.1
        location_above_pushing_point = (position[0], position[1], position[2] + pushing_point_margin)

        # Compute gripper position and linear movement increments
        tool_position = location_above_pushing_point
```

Fig 4.17 Push direction script part

4.2 Training with Recurrent Neural Network

4.2.1 Pre-requisites

The most important pre-requisite for training any neural network is the dataset on which it will train. There is huge requirement of good datasets for testing many machine learning algorithms. A good dataset should have characteristics that are matching with the requirement. There are many datasets The datasets sometimes come with ground truth which signifies the absolute correct value. There are some scenarios for which generating datasets with labels is difficult. In such cases existing dataset models are combined to give a custom made dataset model like flying chairs dataset. In this project the dataset consists of images generated from Gazebo simulation environment. These images are obtained when random force are applied to tower of 4 different blocks and the images contain different position and orientation of the blocks.

4.2.2 Pre-processing

The steps involved in Preprocessing are briefly explained below:

Step 1: Getting the Image Dataset from Gazebo Simulation

Image datasets are obtained from Gazebo simulation from Kinect Camera installed in the Franka robot work-cell assembly. The Kinect camera is located at the top of the work-cell assembly and thus all images are top-view images. A Kinect camera has sensor_msgs/raw_image topic publishes image which are subscribed by a subscriber, the subscribed images are then converted to cv format and stored in folders.

```
(0.0,0.0,0.0)),'world'))
            self.reset_state(ModelState('box1',Pose(Point(x1,y1,1.14),Quaternion(0.0,0.0,0.0,1.0)),Twist(Vector3(0.0,0.0,0.0),Vector3
(0.0,0.0,0.0)),'world'))
            self.reset_state(ModelState('box2',Pose(Point(x1,y1,1.26),Quaternion(0.0,0.0,0.0,1.0)),Twist(Vector3(0.0,0.0,0.0),Vector3
(0.0,0.0,0.0)),'world'))
            self.reset_state(ModelState('box3',Pose(Point(x1,y1,1.38),Quaternion(0.0,0.0,0.0,1.0)),Twist(Vector3(0.0,0.0,0.0),Vector3
(0.0,0.0,0.0)),'world'))
        except rospy.ServiceException, e:
            print("Service call failed: %s"%e)

    def applyImpulse(self):
        try:
            self.apply_impulse_force(
                                    body_name = body,
                                    reference_frame = "ground_plane::link",
                                    wrench = geometry_msgs.msg.Wrench(force = self.impulse_force),start_time=rospy.Time
(0),duration=rospy.Duration(.05))

            print("Applied impulse force : " + str(self.impulse_force) + " Newtons.")
        except rospy.ServiceException, e:
            print("Service call failed: %s"%e)

    def startDataRecord(self):
        self.img_process = subprocess.Popen('python im.py ', stdin=subprocess.PIPE, shell=True)
        #time.sleep(100.0)
        #self.rosbag_process.terminate()

    def terminate_ros_node(self, s):

        list_cmd = subprocess.Popen("rosnode list", shell=True, stdout=subprocess.PIPE)
        list_output = list_cmd.stdout.read()
        retcode = list_cmd.wait()
        assert retcode == 0, "List command returned %d" % retcode
        for str in list_output.split("\n"):
            if (str.startswith(s)):
                os.system("rosnode kill " + str)
```

Fig. 4.18 Apply random force function 1

```
    def terminate_ros_node(self, s):

        list_cmd = subprocess.Popen("rosnode list", shell=True, stdout=subprocess.PIPE)
        list_output = list_cmd.stdout.read()
        retcode = list_cmd.wait()
        assert retcode == 0, "List command returned %d" % retcode
        for str in list_output.split("\n"):
            if (str.startswith(s)):
                os.system("rosnode kill " + str)

    def stop_recording_handler(self):
        self.img_process.terminate()
        self.terminate_ros_node("/ws")
        self.terminate_ros_node("/im")

    def run(self):
        print("Starting..")
        self.unpause_physics_client()
        time.sleep(1.5)
        self.logger.warn("Starting data recording.")
        self.startDataRecord()
        time.sleep(1.0)
        self.applyImpulse()
        print(" on box-link: ",self.body)
        print("File updated ")
        time.sleep(10.0)
        time.sleep(self.run_length_s)
        self.stop_recording_handler()
        print( "close")
        time.sleep(7.0)
        self.resetting()
        print("state reset")

# Main function.
if __name__ == '__main__':
    # Initialize the node and name it.
    rospy.init_node('force')
```

Fig. 4.19 Apply random force function 2

```
class im(object):

    def __init__(self):
        self.count=0
        self.viz=1
        self.image_pub1 = rospy.Publisher("/kinect_camera/depth/image_raw",Image, queue_size=10)
        #self.image_pub2 = rospy.Publisher("/kinect_camera/depth/depth_image_raw",Image, queue_size=10)
        self.bridge = CvBridge()
        self.sub_image = rospy.Subscriber("/kinect_camera/depth/image_raw",Image, self.processImage, queue_size=1)


    def processImage(self,imgmsg):
        try:
         encoding='bgr8'
         #cv2.namedWindow('win')
         image_cv = self.bridge.imgmsg_to_cv2(imgmsg)
         image=cv2.normalize(image_cv,None, 0,255, cv2.NORM_MINMAX,cv2.CV_8U)
        except CvBridgeError as e:
          print(e)

        time.sleep(0.2)
        fc = open("iter.pkl")
        data = pickle.load(fc)
        s=data
        fc.close()
        path="/home/uthira/catkin_ws/src/franka_active_sensing/franka_arm_gripper_realsense_gazebo_description/scripts/im"
        k='c'+str(s)
        createFolder(path,k)
        newpath=os.path.join(path,k)
        cv2.imwrite(os.path.join(newpath, "colorimage frame %d.jpg" % self.count), image)
        print(" created folder")
        self.count=self.count+1
        #cv2.imwrite(os.path.join(newpath, "colorimage frame %d.jpg" % count), Image1)
```

Fig. 4.20 Recording Images

Step 2: Resizing/preprocessing the images

The images obtained from the Gazebo simulation are resized or preprocessed to fit the requirements of the training framework.

```
#!/usr/bin/python
from PIL import Image
import os, sys
import cv2
import glob

#sys.path.append('/usr/local/lib/python3.5/site-packages')
rootdir  = os.getcwd()
newpath='home/uthiralakshmi/RKN/rkn/data/train'
#dirs = os.listdir( path )
def createFolder(pi,folder):
    try:
        root_path = pi
        os.mkdir(os.path.join(root_path,folder))
    except OSError:
        print (' Creating folder. ')

def resize():
  for i in range(1,400):
   dirs='c'+str(i)
   path1=os.path.join(rootdir,dirs)
   path=os.path.join(path1, '*.jpg')
   count=0
   for file1 in glob.glob(path):
    if file1.endswith('jpg'):
      print("Processing %s" % file1)
      oriimg = cv2.imread(file1,cv2.IMREAD_COLOR)
      img=cv2.normalize(oriimg,None, 0,255, cv2.NORM_MINMAX,cv2.CV_8U)
      newimg = cv2.resize(img,(60,80))
      k='r'+str(i)
      createFolder(newpath,k)
      newpath=os.path.join(newpath,k)
      cv2.imwrite(os.path.join(newpath, "image %d.jpg" % count), newimg)
      count=count+1

resize()
```

Fig.4.21 code for resizing

4.2.3 Annotating input images to obtain label images

The dataset of images was annotated to obtain ground truth image labels. The label images were obtained by image segmentation by using color. The concept of image segmentation implemented gets the color channel indices of images which are distinct and assigned a suitable color class label.

```
pre_obs = train_observations
pre = (predictions)  # Converting float32 to uint8
# saving test targets images and prediction images in folders
acc_px = np.zeros([labels.shape[0],labels.shape[1]],dtype=np.float32)
for i in range(labels.shape[0]):
    for j in range(seq_length):

        image1 = pre[i, j, :, :, :]  # prediction image
        image2 = labels[i, j, :, :, :]  # test_targets image
        rootdir = os.getcwd()


        ppath = os.path.join(rootdir, 'pre_images')
        l = 'p'+str(i)
        createFolder(ppath, l)
        ppath1 = os.path.join(ppath, l)

        color_high_box=[[0,0, 0], [260,50, 50],  [50,260, 50],  [50,50,260],  [260,50,260]]# in the order red, green, blue, violet
        pre_rgb = np.zeros([image1.shape[0],image1.shape[1], 3],dtype=np.uint8)
        lab_rgb = np.zeros([image2.shape[0],image2.shape[1], 3],dtype=np.uint8)
        pre_max = np.argmax(image1,axis = 2)
        lab_max = np.argmax(image2,axis = 2)
        #pre_max = image1[:,:,0]
        #lab_max = image2[:,:,0]
        for r in range(0,image1.shape[0]):
             for c in range(0,image1.shape[1]):

                 #pre_rgb[r,c,:] = color_high_box[int(image1[r,c])]
                 #lab_rgb[r,c,:] = color_high_box[int(image2[r,c])]
                 pre_rgb[r,c,:] = color_high_box[int(pre_max[r,c])]
                 lab_rgb[r,c,:] = color_high_box[int(lab_max[r,c])]
        #print(pre_max)

        cv2.imwrite(os.path.join(ppath1, "pre image %d.jpg" % j), pre_rgb)

        #tpath = os.path.join(rootdir, 'test_images')
        #f = 't'+str(i)
        #createFolder(tpath, f)
```

Fig.4.22 code for labelling images

4.2.4 Training, Testing and Evaluation

The dataset of images was split into train dataset and test dataset. The train dataset consisted of 400 iterations of images stored in train folder. The test dataset consisted of 100 iterations of images stored in test folder. The neural network was trained initially with few iterations ad epoch lengths and slowly the number of iterations were increased. In each epoch, the value of loss was calculated and at the end of each iteration the average loss was computed. The loss value must converge as we increase the number of iterations. This indicates that the neural network is learning some parameters. After getting a good accuracy after training, the neural network is tested on test dataset and final value of accuracy was calculated. Train dataset and test data test consists of observations and targets. The observations denoted the input image and targets denote the ground truth label images. The loss function is calculated by comparing the observations and targets during training and by comparing test targets and prediction values during evaluation.

```
"""train"""
train_observations, train_targets, train_obs_valid = prepare_data(
    data_fn=data.get_train_data)
test_observations, test_targets, test_obs_valid = prepare_data(
    data_fn=data.get_test_data)

#train_observations = train_observations.astype(float)
#test_observations = test_observations.astype(float)
#train_observations -= (train_observations[0,0,...])
#test_observations -= (train_observations[0,0,...])

print('train_targets.shape',train_targets.shape)
print('test_targets.shape',test_targets.shape)
print('test_targets.dtype',test_targets.dtype)
saver = tf.train.Saver()
for i in range(iterations):

    model_runner.train(observations=train_observations,
                       targets=train_targets,
                       training_epochs=epochs_per_iteration,
                       observations_valid=train_obs_valid)
    sess = model_runner.tf_session
    export_dir = 'home/uthiralakshmi/RKN/rkn/saved_models'
    #saved_path=saver.save(sess,os.path.join(os.getcwd(), 'model1.ckpt'))
    print('saved')
```

Fig. 4.23 Training

4.2.5 Tweaking parameters

After preprocessing the dataset and annotating the dataset to obtain labels, the neural network is trained and subsequently tested. The most important part of training neural networks is to optimize the loss functions. The loss functions are optimized by using ADAM optimizer function by computing stochastic gradient descent. The weight parameters are updated by gradient values through back

propagation. The loss function can be reduced by increasing the parameters in each layer of convolutional layer, by tuning the learning rate and step size, by increasing the number of layers in encoder or decoder, by changing the loss function type, by introducing a Softmax layer at the end of the network to obtain the probability of class labels. Hyper parameter optimization is very interesting and challenging of training any neural network, it involves engineering the most suitable way to reduce the loss function and obtain maximum accuracy of prediction. There are lot of methods available for tweaking the neural network and most of times it is done by trial and error methods. The efficiency of computer scientist lies in the way they engineer the aspects of tweaking the neural network to obtain the optimum result.

4.2.6 Calculating prediction accuracy

The pixel wise prediction accuracy is obtained by comparing the maximum value of color channel indices corresponding to test target labels and prediction labels. The accuracy is computed as the mean of sum of absolute difference of maximum value of prediction and test target labels for all the labels greater than 0 divided by the sum of maximum value of labels.

CHAPTER 5

RESULTS AND DISCUSSION

5.1 Simulation Results

The Franka Emika Panda arm was simulated with Gazebo simulation environment by creating a world file consisting of Franka panda model, work cell-assembly, Kinect camera, 4 boxes of different color. URDF file of panda arm and gripper was integrated to form a single URDF Xacro file. The SDF file of all other components were included through the world file. All other scripts were attached; launch file was created. The gazebo was interfaced with ROS, RViz and MoveIt. The motion planning was achieves using OMPL with a FAST IK solver. The controllers were setup using MoveIt setup assistant. There are 3 controllers for panda, namely arm controller, hand controller, joint space controller. The move group node is interfaced with ROS and gazebo for setting up controllers.

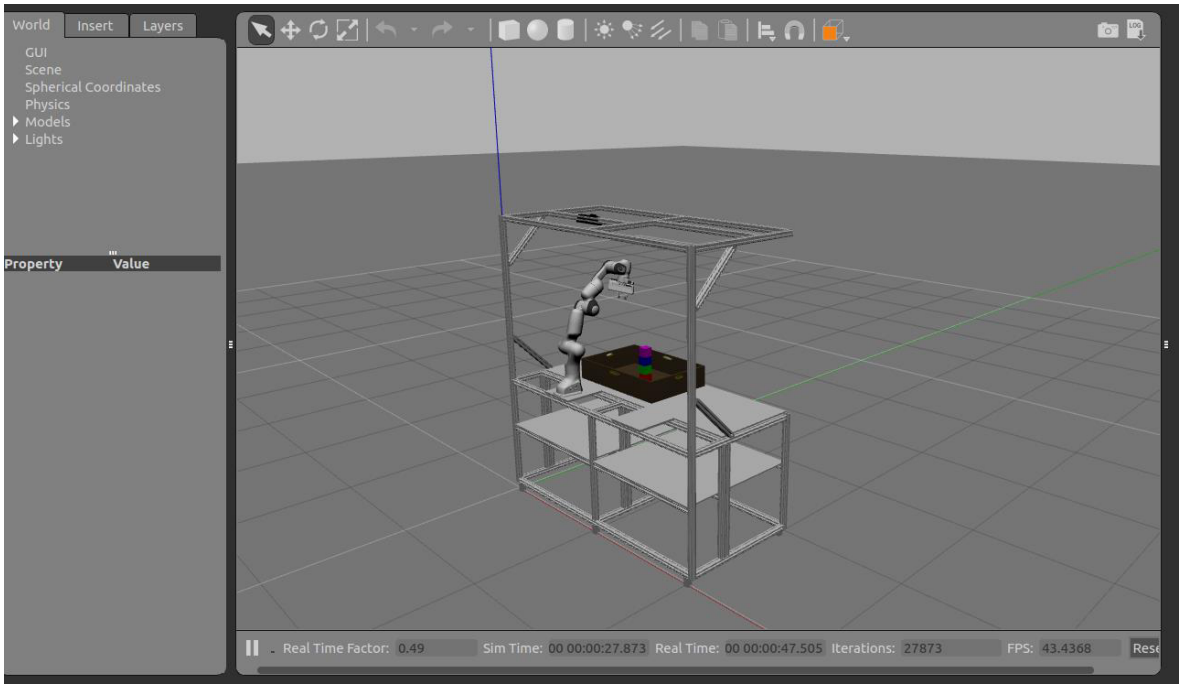


Fig. 5.1 Franka Panda Setup in Gazebo

There was some error when initializing hand controller. The action command was unable to get connected to the hand controller. Hence, it was difficult to control the gripper of Panda robot in Gazebo to manipulate the boxes to build and break a tower of breaks.

Following the issue with hand controller in Gazebo, another simulation environment V-Rep was used for simulating UR5 to build and break a tower of blocks. The code for simulation was inspired from Andy Zeng Visual grasping and pushing toolbox[7]. The idea was modified to make the UR5 robot grasp and push blocks. Using grasp and push functions, the UR5 robot was made to build and break a tower of blocks.

In order to build a tower of blocks, the robot has to first identify the position and orientation of blocks. The tower consists of 4 blocks, out of which one block is assigned as a base block. All the other blocks have to be placed on the top of the base block one by one. Building a tower involves first opening the gripper, moving to the position of the object, closing the gripper when the block is in between the gripper, move the gripper above the location of target and moving the gripper to the position above the base block. One important thing is to make the gripper align the orientation of block it is taking to the base block in order to make the tower more stable. The breaking of tower is very simple and only involves push action of the gripper. The gripper needs to be fully closed and be placed near to the tower and move in the indirection of the tower. The push action is performed in 4 directions: top, down, right, left.

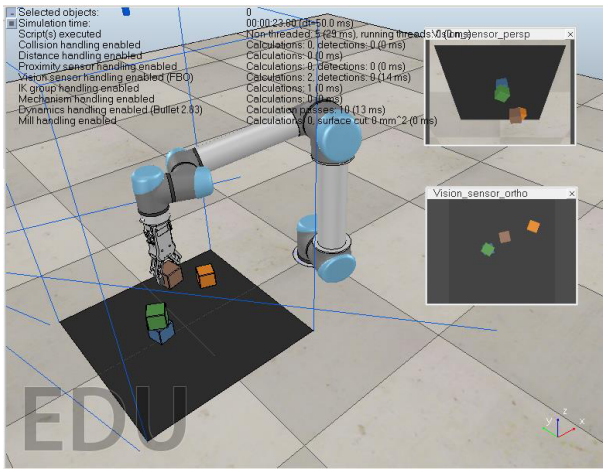


Fig.5.2 UR5 building a tower 1

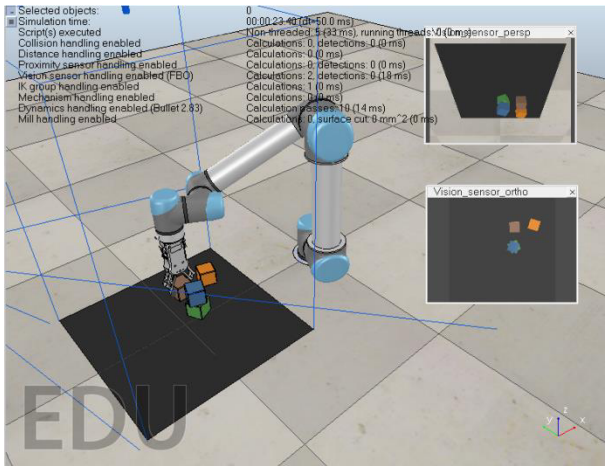


Fig.5.3 UR5 building a tower 2

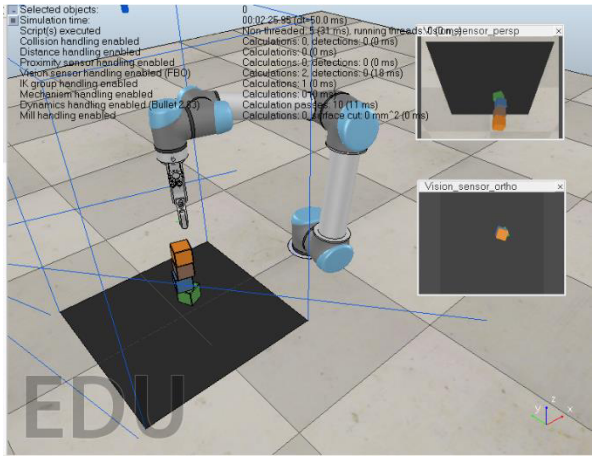


Fig.5.4 tower of blocks

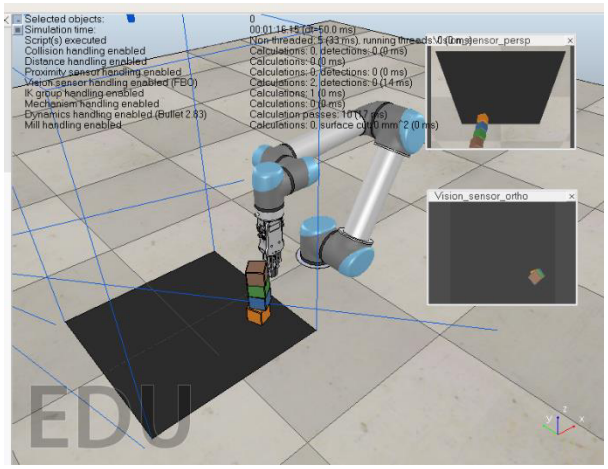


Fig.5.5 UR5 breaking a tower

Franka panda robot was also simulated in V-rep simulation environment. The Franka robot URDF file was imported and joints/links parameters were initialized. The dynamics properties of the various joints and other objects were setup. The workspace of the Franka Panda was setup. The boxes were initialized using Remote API server using import shape function. The Franka panda was manipulated to perform Push actions and Grasp actions. The idea of pushing and grasping implemented for UR5 was used for Franka Panda. The Franka Panda was successful in pushing action with some issues with grasping action.

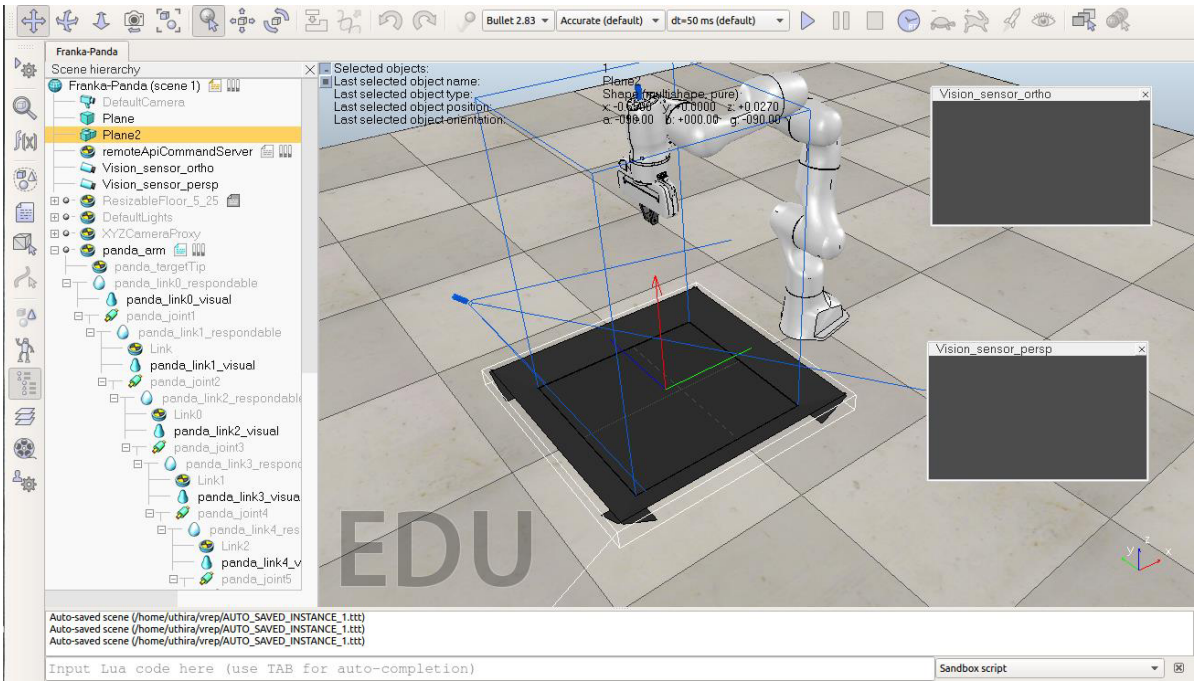


Fig. 5.6 Franka Panda Setup in V-Rep

5.2 Sequence Prediction Results

Images of blocks where recorded when the tower of blocks was being broken by applying random force in X and Y directions. The images recording started at the moment of application of force and recorded for a couple of seconds for 500 iterations and stored in folders. These images were used as a dataset for training a recurrent neural network namely long short term memory neural network for predicting subsequent sequences of blocks position and orientation.

The images were preprocessed by resizing to fit the requirements of neural network. The architecture of neural network consists of a down sampling convolutional encoder layers, a LSTM transition cell followed by an up sampling convolutional decoder layers and a Softmax layer. The input images were annotated by creating label images. Image segmentation was used to annotate the input image to create image labels. The input images dataset was split into train and test dataset. The train dataset was trained for number of iterations and the neural network model was tested on test dataset. The output image is also a label image which maps the color classes of the 4 different

blocks. The model was trained for 10 iterations containing 30 epochs each. The loss function used was maray cross entropy loss. The optimizer used was ADAM optimizer. The activation used is ReLU non-linearity function. The accuracy of prediction was calculated by comparing the test target labels and prediction results.

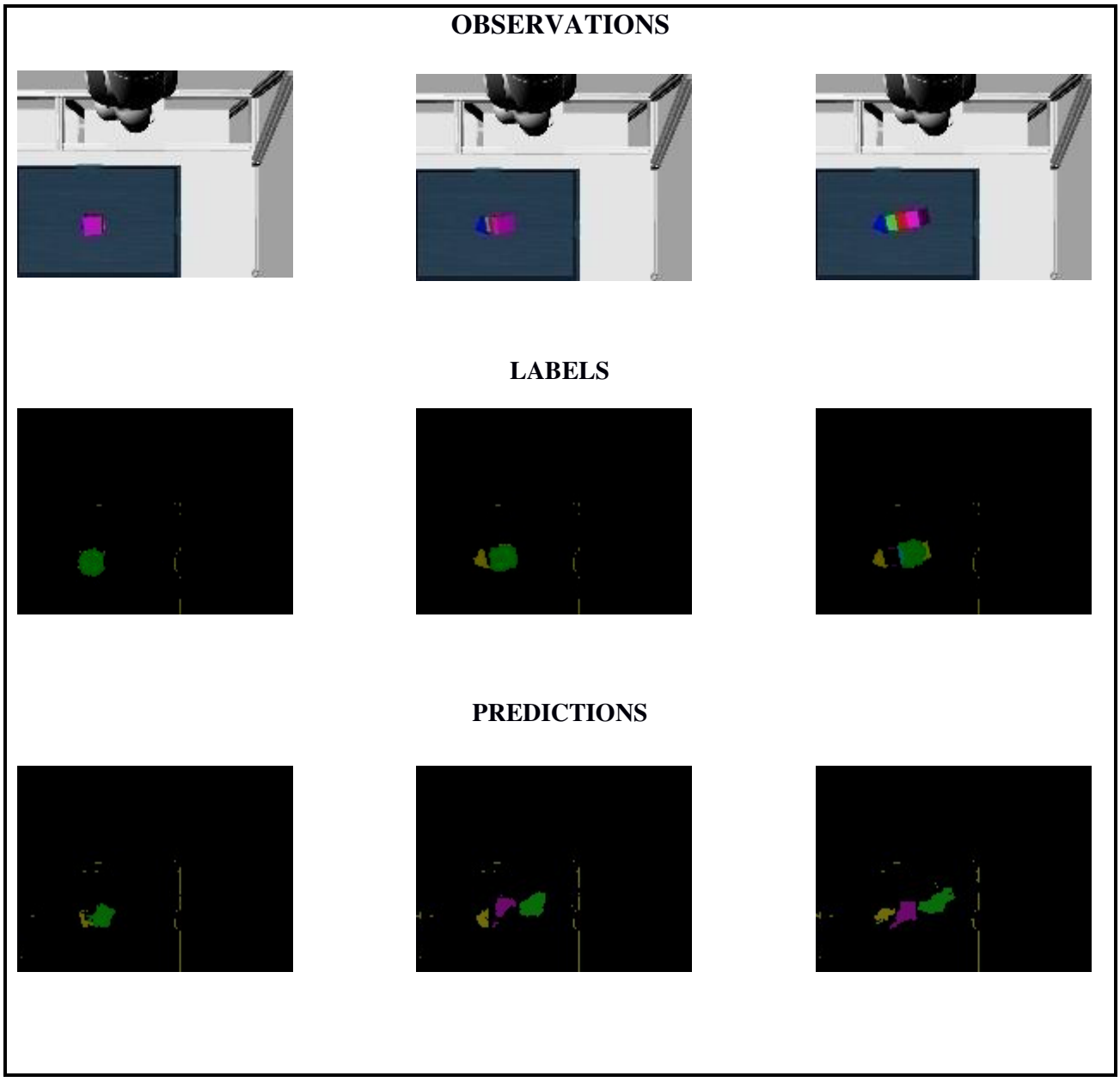


Fig. 5.7 Prediction results

In the above image, the observations are 3 image samples from the input dataset, the labels are annotated images corresponding to the 3 sample input images, the predictions are output images for

each of the 3 sample label. The predicted image is the next image in sequence for a given labelled image.

ITERATIONS	TRAIN ACCURACY	TEST ACCURACY
1	40.378292	42.90274
2	41.350796	43.704166
3	41.197277	43.371964
4	41.123024	42.86047
5	40.305782	42.35727
6	39.995853	42.213898
7	41.725925	43.460625
8	40.223236	42.110325
9	40.98449	42.345097
10	41.82094	43.3392

Table 5.1 Prediction accuracy

The table above shows the value of train and test accuracies for 10 iterations containing 32 epochs each. The value of test and train accuracy are almost similar, yet the performance of the model is not fully efficient. This may be due to noise variations present in the input image.

CHAPTER 6

CONCLUSION

The project is an interdisciplinary project covering basic physics, robotics, machine learning, python programming, computer vision, neural networks. The project tests the basic understanding of concepts and application skills of a person. It is very interesting and challenging to learn, understand all the concepts and implement combining all the disciplinarians. The project is a prior to anyone who wants to pursue a carrier in robotics, machine learning. The project paves a road for aspirants trying to design and simulate humanoid robots. Various problems of manipulation were taken into account and corresponding logic were programmed to manipulate the robot. The hyper-parameters of neural network architecture were tweaked and engineered to train and test the dataset to obtain predictions to the maximum accuracy possible.

FUTURE SCOPE

- Implementing the Franka panda robot full work cell assembly in V-Rep
- Predicting the sequences using advanced versions of Long short term memory network
- Learning push, grasp actions using Reinforcement learning algorithms
- Implementing building and breaking of tower in real time

REFERENCES

1. ROS official website documentation and tutorials - http://wiki.ros.org

2. Gazebo official website documentation and tutorials - http://gazebosim.org/

3. V-Rep official website documentation and tutorials - http://www.coppeliarobotics.com/

4. MoveIt! official website documentation and tutorials - http://docs.ros.org/kinetic/api/moveit_tutorials/html/index.html#

5. CS231n Stanford university computational neural networks courses and documentation

6. Andy Zeng, Shuran Song, Stefan Welker, Johnny Lee, Alberto Rodriguez, Thomas Funkhouser- "Learning Synergies between Pushing and Grasping with Self-Supervised Deep Reinforcement Learning", Princeton University, Google, Massachusetts Institute of Technology

7. Andy Zeng .et.al Visual pushing and grasping Toolbox, GitHub - https://github.com/andyzeng/visual-pushing-grasping

8. Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." Neural computation 9.8 (1997): 1735-1780.