Hand-Eye Coordination for Robotic Grasping using Deep Learning for Frame Coordinate Transform

by

Tejas Seshadri Sarma
B.E., University of Mumbai, 2017

Thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Chicago, 2019

Chicago, Illinois

This thesis is dedicated to Prof. Andrew Y. Ng, for introducing me to the world of Machine Learning.

ACKNOWLEDGMENTS

I would like to express my gratitude towards my advisor - Professor Brian D. Ziebart - for the help and guidance he has provided and helped develop my interest in the area of Artificial Intelligence, Machine Learning and Robotics, and has greatly influenced my skills, helping me bring the best of myself out, for past year and a half. I would also like to thank Professor Milos Zefran and Professor Natalie Parde, for providing me with their time and gracious efforts to be a part of my Masters thesis committee.

I would also like to express my gratitude towards Sanket Gaurav, for his immense help for introducing me to the important concepts of ROS, and some of the other prerequisites that were essential to the completion of the project. I would also like to thank all the member students in AI and Robotics lab for their continual support and help throughout the duration of the project.

TS

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES (Continued)

LIST OF ABBREVIATIONS

AR	Autonomous Robot
ROS	Robot Operating System
DNN	Deep Neural Network
DOF	Degrees of Freedom
UIC	University of Illinois at Chicago
FFN	Feed-forward Networ
FOV	Field Of View

SUMMARY

Autonomous Robotics involves the independent operation of a robot in a work environment. The Baxter Robot head camera, unlike some other external camera devices such as the Microsoft Kinect, lacks a depth sensor, which leads to its inability to gauge any distance-based information only from its field of vision. This makes Baxter-centered vision-only pick and place tasks quite challenging.

In this project, we see how Deep Learning has been used to automatically learn vision-based depth information for a sample plane, in the Baxter's Cartesian Frame of reference. Using Deep Neural Networks(DNNs), we can see how this vision-based depth information can be learnt quite accurately, even for a highly nonlinear fish-eye distortion that is present in Baxter's head camera's field of vision.

Using only a handful of collected points as seeds,we efficiently generated large synthetic data sets, which were then used to train Deep Models, ultimately mapping a Cartesian plane to Baxter's field of view, for fast pick and place operations of test targets on that plane. The goal of this approach was to design a generalized pipeline, that can be used for other robots, which fall into a similar category as the Baxter, virtually seamlessly.

CHAPTER 1

INTRODUCTION

1.1 Problem Statement

When we consider object manipulation by organisms, like humans, there exists a quick-response feedback mechanism intermediate to the perceptive stage and the final action that is taken (1). Complex manipulation tasks, including the extraction of an object from a bin containing many other objects, is easily performed without a lot of advance planning, relying only on haptic and optical feedback (1). On the other hand, object manipulation by robots is more heavily dependent on pre-planning, analysis, and complex decision making, along with relatively simple feedback, including trajectory traversal, to ensure a stable execution of the task. (1)

A humanoid robot, with the use of two robotic arms, can move and engage with objects, much like us humans. This feature of such robots has made them get attention from organizations and researchers in the field of robotics. They have been useful for many tasks, such as product assembly, and object manipulation. In the past few years, there has been an advent in the development of bipedal robots, also including the Baxter robot(2)(3) have been widely applied in the industrial field, especially for the assembly task. More often than not, such tasks involve the use of *Computer Vision* and *Mechatronics*, both of which have gone extensive development over the past few decades. (3)

The Baxter camera system, on its own, is not equipped with depth sensors, making it difficult to evaluate three-dimensional Cartesian coordinates of the objects within its field of vision. External camera devices, such as the relatively inexpensive Microsoft Kinect depth camera, bridge this gap quite efficiently, though at the cost of extending the dependence of the operation beyond the physical limits of the Baxter robot itself.

1.2 Thesis Statement

Coordinate Frame Transformation, or Reference Frame Transformation, is a process that is used to effectively locate a target, detected in a robot's field of view, in three-dimensional Cartesian space. Being able to evaluate the three-dimensional Cartesian coordinates of a target object helps eliminate the usage of error based proximity detection of the robotic arm, in relation to the target. This greatly reduces the grasp time, which in-turn reduces the overall time for completion of a complex task involving multiple grasps. Here, the grasp approach mechanism begins with detecting, and isolating the target object within the task frame, which is a subset of the field of view of the robot's camera.

Post the detection of the target by the robot, the coordinates of the centroid of the target are now fed into a trained Deep Learning model, which effectively outputs the corresponding three-dimensional Cartesian coordinates of this target in the robot's own frame of reference. Subsequently, the usage of an Inverse Kinematics solver, on the calculated Cartesian coordinates, directs the motion of the robot arm towards the target, effectively attempting a grasp. This mechanism attempts a grasp without the usage of dynamic calibration of the robotic arm, during the traversal, which leads to significant reduction in latency between planning and the grasp action itself.

Dynamic trajectory planning can sometimes lead the robot arm to take a very convoluted path from its initial position, to ultimately reach the target object. Alternate approaches, that use external cameras equipped with depth sensors, extend the dependence of the robot beyond its own resources. The presented Frame Transform model can reduce this dependence, by using only the robot's resources, and Deep Learning to bridge the gap created by the absence of advanced features(in this case, lack of depth sensors on the built-in head Camera), in the hardware present on the Baxter Robot.

The goal of the experiment is to determine the extent to which a Deep Learning Model can be used to replace a more conventional semi-analytical multi-stage pipeline for location estimation. We aim to explore the capabilities of Deeper Models and their ability to learn complex relations which would otherwise be concluded by means of a multi-stage pipeline of execution. While this experiment is constrained to a sample Cartesian Plane in the Baxter's frame of reference, it is intended to be a starting point to using Deep Learning to locate targets in Baxter's proximity efficiently.

1.3 Outline of the Document

This thesis is organized into five parts: Introduction; Conceptual Foundation; Computer Vision Subsystem; Frame Transformation Subsystem; Experiments and Results.

The first chapter, ***Introduction***, gives an overview of the problem space that is being explored in this project, and the solution that is being proposed to combat the stated problem.

Followed by this, in the next chapter, ***Conceptual Foundation***, gives a brief introduction to the major concepts and theoretical knowledge that was used in the project's various components. This is important to thoroughly understand the further chapters in the thesis.

The third chapter, ***Computer Vision Subsystem***, details the entire process of enabling the robot to "see" and effectively perceive the given environment. In this chapter, we see the various steps that are taken from obtaining the image of the work environment, to the various image manipulation techniques that have been applied to the obtained image. This is followed by the details of the object recognition algorithms that are applied on the processed image, which then enables us to evaluate the visual pixel coordinate position of the concerned target.

In the fourth chapter, ***Frame Transformation Subsystem***, we see how the location information obtained from the Computer Vision subsystem is put to use. Here, we showcase how deep learning is applied to use the visual location information to determine the actual location of the target in Cartesian space. It includes the design, implementation, training, and testing of a variety of different models, to obtain this result.

In the fifth chapter, ***Experiments and Results***, we see the performance of the different models that were created and tested. We compare all the models to get an empirical and quantitative view on the performance and effectiveness of the different models and the approach, as a whole.

Finally, in the last chapter, we conclude the thesis and explain the further research that can be carried out in the future, in this area and related sub-areas.

CHAPTER 2

CONCEPTUAL FOUNDATION

2.1 Autonomous Robotic Control

Autonomy, according to Oxford [1], is the ability for an entity to make an informed decision, or choose a particular course of action, without interference from the environment around it. Wikipedia [2] defines an autonomous robot (AR), as one that operates and performs tasks with a high level of autonomy. While some autonomous robots behave autonomously on when certain rigid environmental requirements are met, others are able to function as fully autonomous, completely independent of most environmental constraints.

In general, a robot is considered completely autonomous when it meets the following crucial requirements:

1. The robot should be able to observe and obtain information about the environment that it is operating within
2. The robot should have the capacity to perform tasks unsupervised, for extended periods of time satisfactorily.

[1]https://en.oxforddictionaries.com/definition/autonomy

[2]https://en.wikipedia.org/wiki/Autonomous_robot

3. For performing the assigned tasks, the robot should be able to manage to traverse the work environment either completely without, or partially with assistance from an operator.
4. The robot should know how to steer clear of actions and modes of operation that can be considered potentially dangerous to the other entities both within, and outside the work environment.

As we can see in the Figure 1, the Baxter Robot has a work surface placed in front of it. The work surface is centered and perpendicular to the Baxter's origin, and the lower center placed at $x = 0.74m$. The entire work surface an elevation of $z = -0.24m$ with respect to Baxter's origin. The Baxter head camera is held at a pan of $P = 0$ at the default inclination. The output of the head camera is fed to the Deep Learning Subsystem for location prediction based on visual coordinates. Unlike dynamic methods for calculating coordinates of target based on relative position of a movable camera (4), this approach looks at a more direct, absolute value based static estimation of a target's position.

2.2 The Baxter Robot

The Baxter Robot was introduced by a company by the name Rethink Robotics. It is an humanoid industrial robot, which is made to be extremely safe, and is designed to work in collaboration with human users.(4) It is mounted on a moving pedestal connected to its torso. The robot has two arms on either side of the robot's body. Each of the two arms have seven joints and subsequently seven Degrees of Freedom (DOF) provided by the joints. The complete robot can be seen in Figure 2 (The Microsoft Kinect unit visible above the LCD screen is part of a different experiment).(2)

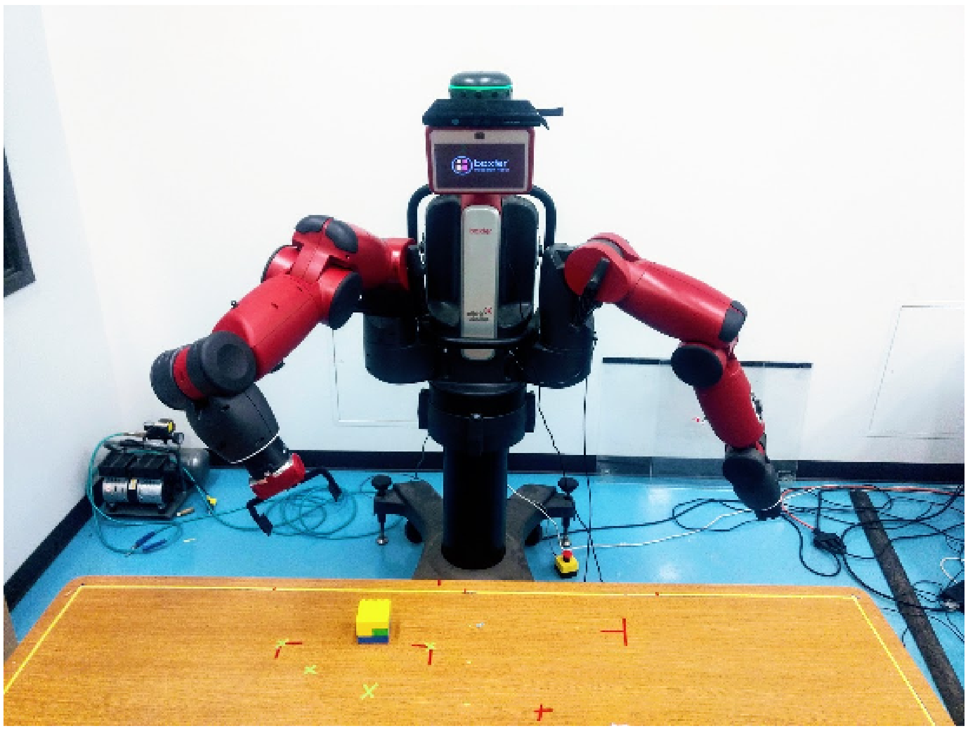


Figure 1. Hand-Eye Coordination Experimental Setup

Each of the two arms can be customized with different attachments such as two-pronged grippers, and robotic hands with fingers. These attachments can be fixed to the ends of each arm when required. The *head* of the the Baxter robot is an LCD screen which also acts as a display and a *face*, when necessary. The *head* has a pan joint, which can be controlled programmatically. The Baxter also has three cameras. The primary camera is positioned on top of the head, at the top bezel portion of the LCD screen. The two secondary cameras are present on the ends of each of the two arms. Each of the three cameras have a focal length

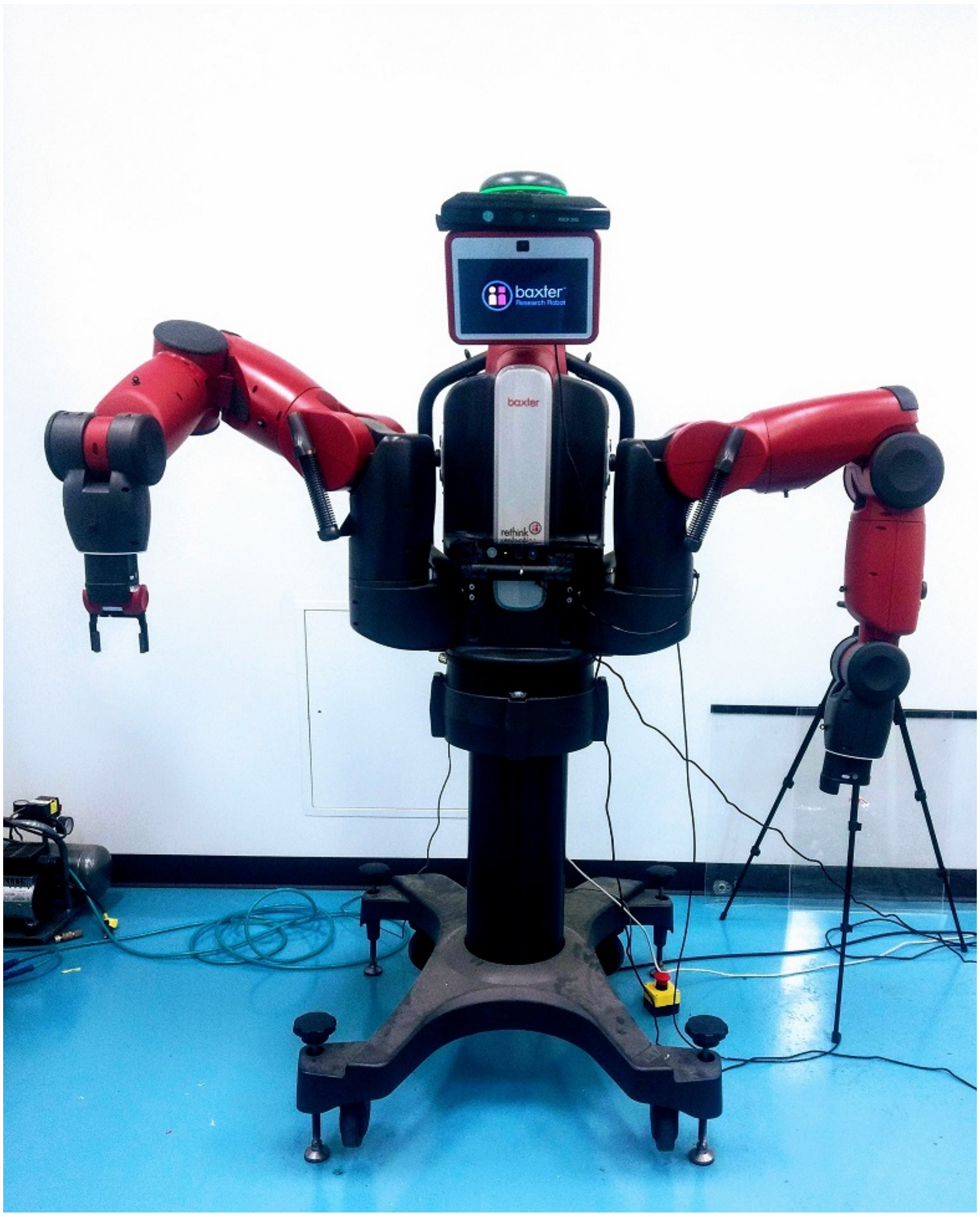


Figure 2. The Baxter Robot

of 1.2mm, with a maximum resolution of 1280 x 800 and an effective resolution of 640x400, operating with a frame rate of 30fps.(2)

The seven joints present on each of the arms of the Baxter Robot are labeled as:

$$J_{baxter} = [S0, S1, E0, E1, W0, W1, W2] \tag{2.1}$$

The total connections, consisting of these seven joints, accompanied by eight corresponding links, provide the seven DOF to the robot. There are three basic types of joints on the robot:

1. Shoulder Joints:

$$S_{baxter} = [S0, S1] \tag{2.2}$$

2. Elbow Joints:

$$E_{baxter} = [E0, E1] \tag{2.3}$$

3. Wrist Joints:

$$W_{baxter} = [W0, W1, W2] \tag{2.4}$$

The joint $S0$ is directly attached to the robot's *arm mount*, which in turn is fixed to the torso of the robot. $S0$ and $S1$ contain the *Upper Shoulder* section of the arm between them, while $S0$ and $E0$ contain the *Lower Shoulder* section. $E0$ and $E1$ contain the *Upper Elbow* section of the arm between them, while $E0$ and $W0$ contain the *Lower Elbow* section. $W0$ and $W1$ contain the *Upper Forearm* section of the arm between them, while $W1$ and $W2$ contain the

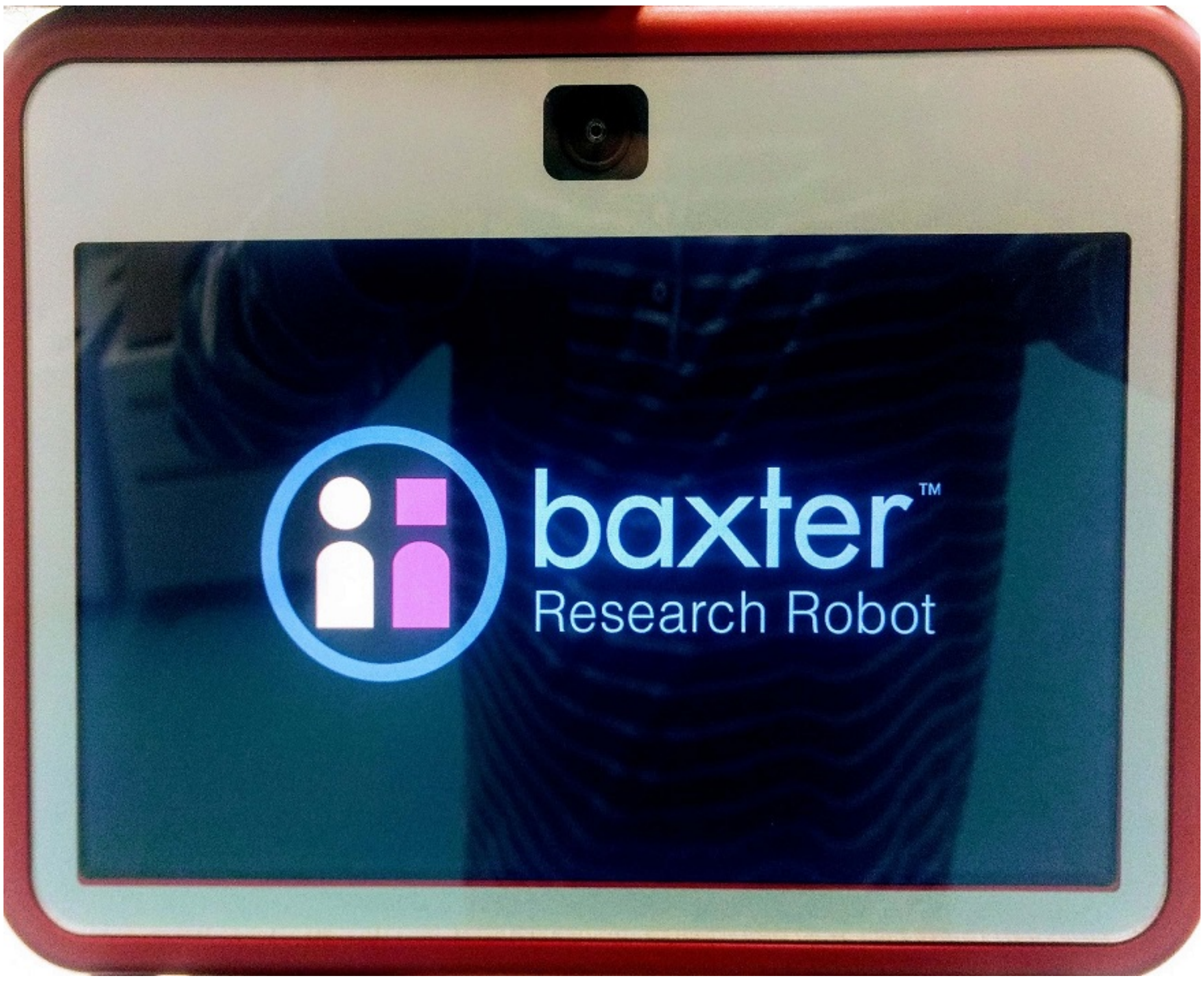


Figure 3. The Baxter Head Camera and LCD Screen.

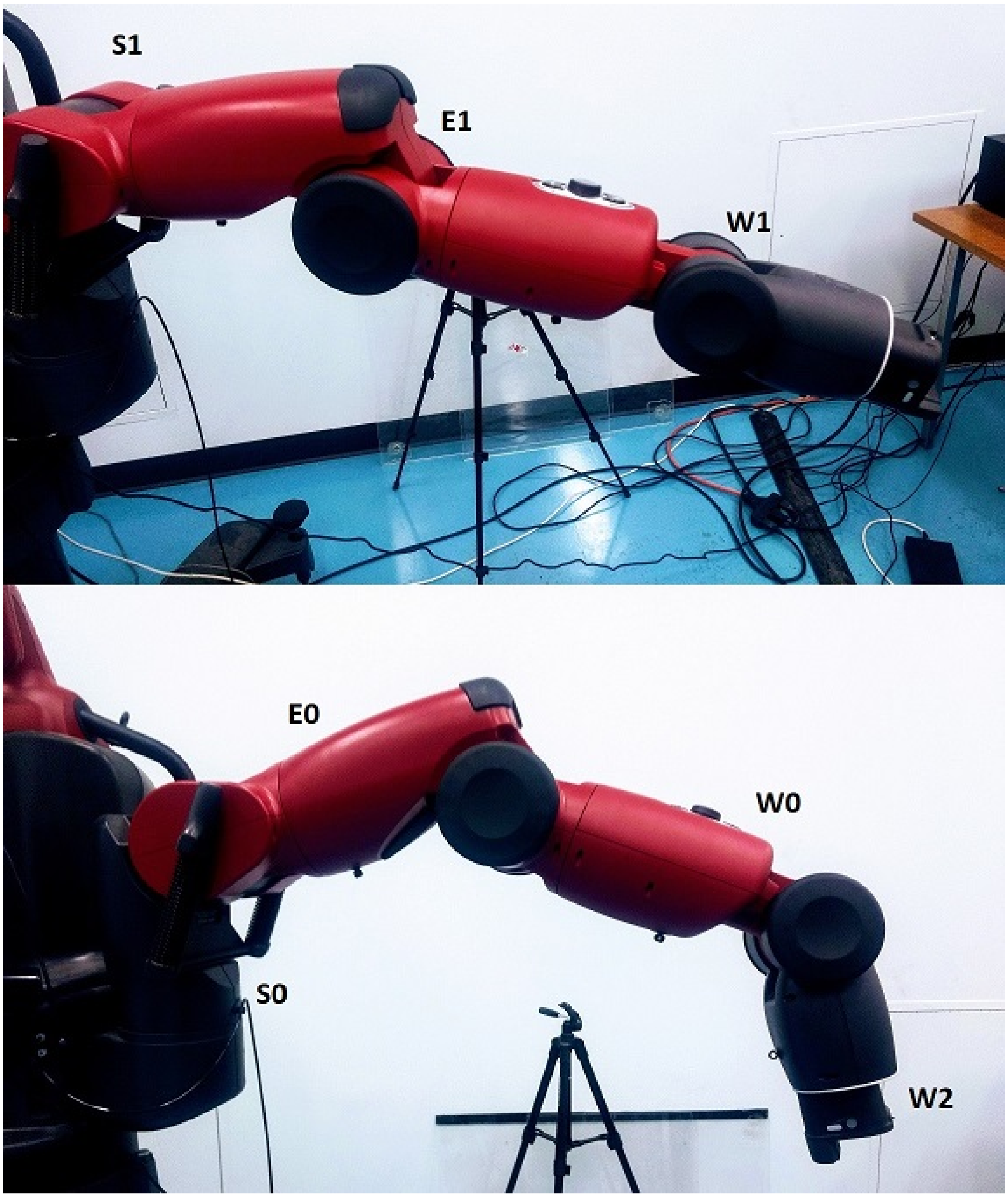


Figure 4. The Baxter Robotic Arm and Joints.

Lower Forearm section. The robot's wrist, also known as the end-effector, is located beyond the wrist joint $W2$.(5) The seven joints are also divided into two categories, based on their movement-type:

1. Bend Joints: (These joints are flexible to bend about a hinged pivot point.)

$$J_{baxter_bend} = [S1, E1, W1] \tag{2.5}$$

2. Rotational Joints: (These joints rotate about an axis.)

$$J_{baxter_rotational} = [S0, E0, W0, W2] \tag{2.6}$$

These joints will be referenced in later sections, when calculating Cartesian coordinates using Inverse Kinematics service that is available in the Baxter Robot SDK.

2.3 ROS(The Robot operating System)

The Robot Operating System (ROS) is a very popular and widely used robotics mid-level framework, that is flexible, and is used for writing robot-specific software programs. ROS provides a variety of different tools and libraries that users can use to create extremely complex and very robust robot behavior with relative ease. ROS also extends to a large variety of different popular robotic platforms. As Wikipedia [1] says, ROS is technically a middleware, as it helps to provide a mode of communication between different processes at the lowest level. Even

[1]https://en.wikipedia.org/wiki/Robot_Operating_System

as ROS is a middleware rather than a conventional operating system, there are still services that it provides including, but not limited to as hardware abstraction, passing of messages between different processes, device control at low-levels and package management.(6)(7)

Processes that are ROS based are generally represented in the form of a graph architecture, where nodes are denoted wherever the different ROS processes take place. These nodes can effectively send and receive various information, including ones related to sensor output data, control related messages, among others. In this experiment, the nodes are the control programs written and the services present within the Baxter code infrastructure. In particular, the major nodes used are the Baxter *head* camera, and the Inverse Kinematics node, for complex mathematical transformation based calculations throughout the execution of the primary task and the course of action related to it. As a sample case, a Baxter Node publishes the image that is viewed by the head camera. An OpenCV bridge then subscribes to that node, and receives that image for further processing. Some of the important ROS client libraries are written in C++, Python, and Lisp, which are designed to operate in Unix based systems only. Packages in ROS are used for a variety of applications such as action-planning, perceptive recognition, SLAM(simultaneous localization and mapping), robot models and various other general algorithms.(6)(7)

2.4 Baxter Forward and Inverse Kinematics

Robot kinematics refers to the area of analysis of the motion of multi-DOF robot mechanisms, primarily using geometry. This focus on geometric analysis is based primarily on the assumption, that the robotic body links are immovable in nature, which are intended to be solely rotary or bend links. Robot kinematics helps develop a bidirectional relation between the absolute Cartesian location of the end-effector(the end of the robot's arm, here), and the various joint parameters that dictate the motion of the robot's arms. It enables researchers formulate and modify frame based transforms and motion path planning in a very direct manner. Baxter has a lot of very helpful modules containing functions for both forward and inverse kinematic operations. Robot kinematics is crucial for designing a simulation model.(8)

In forward kinematics, the various link and joint parameters are used to calculate the position of a robot's end-effectors analytically. These positional coordinate parameters are calculated with respect to the base/origin contained within the robot (the lower center point in the torso, in the case of the Baxter robot). The Baxter robot SDK it uses a coordinate transformation system representation, called as the URDF (Unified Robot Description Format) format, which is used to define the coordinate transformation relation between the robot's joints and it's links. The URDF file is written in XML describing a robot, and explaining in detail about its various parts, joints, and dimensions, among other parameters.(8)

The graph representation of Baxter robot's internal mechanism can be seen in Figure 5. This graph can be obtained by using the *rosgraph* command in the appropriate ROS terminal. The obtained graphical representation is a good visual map about the relation between the various mechanisms between the joints, links, as well as the kinematics operations within the Baxter.

Arm control for the Baxter robot is normally carried out using the seven joint angles. As an example, position joint control allows the user to be able to fix the position value in

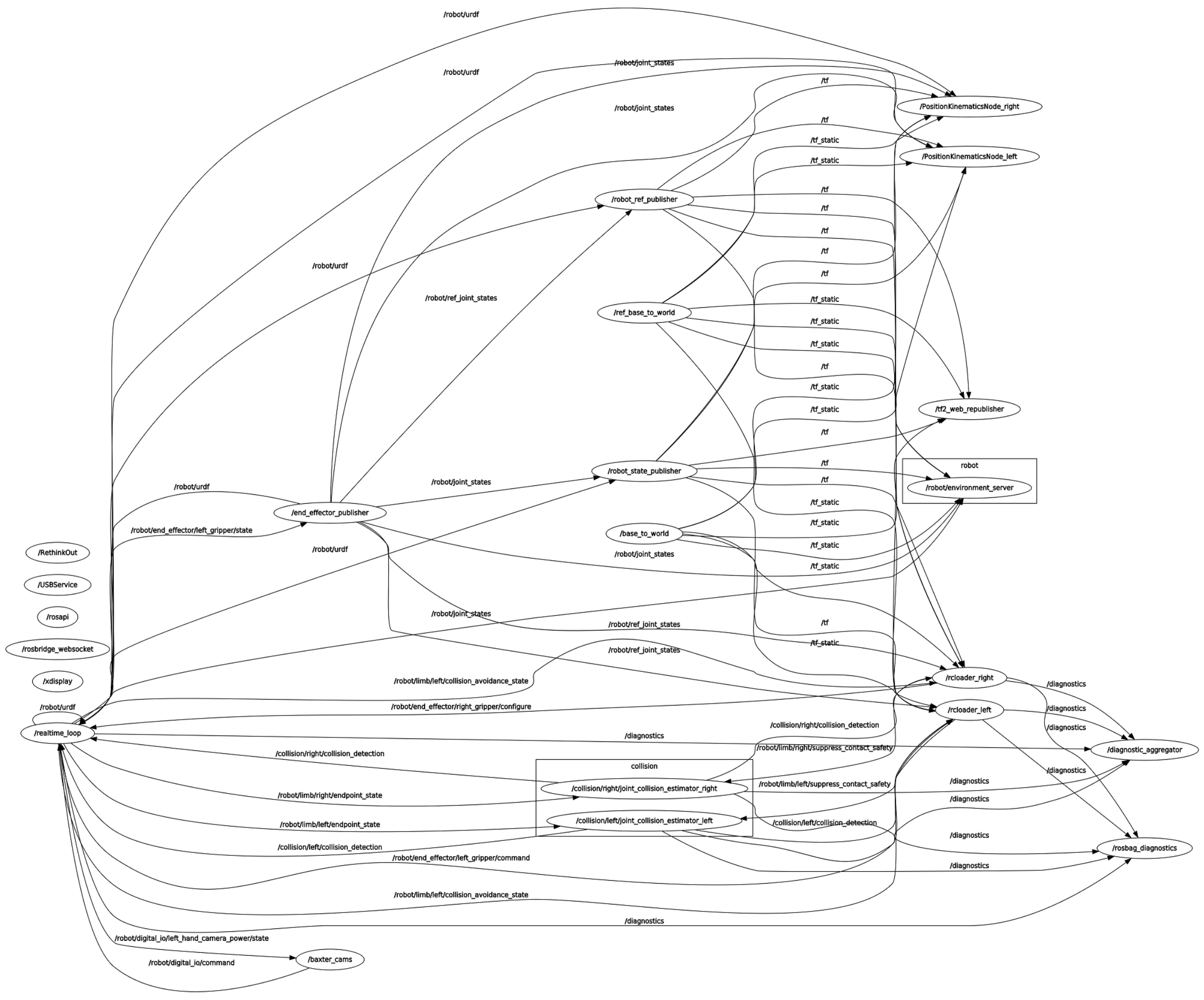

/robot/urdf
/robot/joint_states
/PositionKinematicsNode_right
/PositionKinematicsNode_left
/robot_ref_publisher
/ref_base_to_world
/tf2_web_republisher
robot
/robot/environment_server
/robot_state_publisher
/end_effector_publisher
/base_to_world
/RethinkOut
/USBService
/rosapi
/rosbridge_websocket
/xdisplay
/realtime_loop
/rcloader_right
/rcloader_left
/diagnostic_aggregator
collision
/collision/right/joint_collision_estimator_right
/collision/left/joint_collision_estimator_left
/rosbag_diagnostics
/baxter_cams
/tf
/tf_static
/diagnostics
/robot/ref_joint_states
/robot/end_effector/left_gripper/state
/robot/limb/left/collision_avoidance_state
/robot/end_effector/right_gripper/configure
/collision/right/collision_detection
/robot/limb/right/suppress_contact_safety
/robot/limb/left/suppress_contact_safety
/collision/left/collision_detection
/robot/limb/right/endpoint_state
/robot/limb/left/endpoint_state
/robot/end_effector/left_gripper/command
/robot/digital_io/left_hand_camera_power/state
/robot/digital_io/command

Figure 5. Diagram of Baxter’s Internal Mechanism

radians for each of the two shoulders, elbows, and wrist joints. In robotics manipulation, the Cartesian space of the arm's endpoint, is much more easier to work with. In the Baxter too, the Inverse Kinematics package called as ***IKService*** is used for the transformation from the Cartesian space representation of the position, to the seven joint angle values, which can be used to directly control the robot.(2) The Cartesian representation for the position of Baxter's end-effectors is as follows(5):

$$P_{end-effector} = [x, y, z, r_i, r_j, r_k, r_w] \tag{2.7}$$

Here, $[x, y, z]$ represent the three values determining the Cartesian coordinate position of the end-effector in Baxter's frame of reference, and $[r_i, r_j, r_k, r_w]$ represent the four quaternion values representing the orientation (roll, pitch, and yaw) of the end-effector with respect to Baxter.

2.5 Overview of OpenCV

Computer Vision is the area of Computer Science which focuses on creating programs to enable a machine to interpret and analyze visual data, such as images and video, enabling it to "see". OpenCV (Open Source Computer Vision Library) is a readily available open source computer vision package that was created to provide libraries containing functions written as optimized and portable code. It is primarily geared toward real-time computer vision problems.(9)

In OpenCV, images are represented as matrices, (numpy arrays in python), consisting of three channels of pixels: BGR (Blue, Green and Red). Each pixel itself is represented as intensity values, in the form of real numbers, in each of the three channel matrices. The images that are captured are generally known as raw images. These raw images undergo a series of transformations and manipulations before Computer Vision Algorithms can be applied on them. Using these transformations can tremendously help to concentrate the important information from the image, while leaving the remaining data out. These transformations also help process the images more effectively, for useful evaluations using computer vision algorithms. Some of the transformations that were used in this project were

1. Geometric Transformations
2. BGR $\rightarrow$ Grayscale Transformation
3. Gaussian Filtering
4. Otsu's Thresholding
5. Canny Edge Detection

2.5.1 Geometric Transformations

Sometimes, the captured images are very large in size, with the most important information being concentrated to one small region in the image. To make the images easier to work with, they are reduced to a much more manageable size, which both reduces space and computation time. This can be done with the simple command ***cv2.resize(img,(int(newX),int(newY)))***, where ***img*** is the image matrix, and ***newX*** and ***newY*** are the desired resized dimensions.

OpenCV "squeezes" the original image to fit within the new dimensions, although, not maintaining the original aspect ratio. In our experiment, we choose the scaling factor to maintain the original aspect ratio of the image. Cropping the image is a much simpler operation, as it is exactly the same as slicing a matrix. (10) We crop the original image, so as to only keep the work surface in the view of the Baxter, and remove the other objects within its view.

2.5.2 BGR $\rightarrow$ Grayscale Transformation

This transformation essentially reduces a colored image to a grayscale image. What it also does, is reduced the number of channels from three to one. This greatly reduces the size of the image matrix as well. Detecting intensity based gradients becomes much simpler, and isolation of features can be done with much less filtering and thresholding. Grayscale images are also useful for applying edge detection algorithms. This can be done in OpenCV by simply using the flag ***cv2.COLOR_BGR2GRAY*** (10)

2.5.3 Gaussian Filtering

Image blurring or Image Smoothing is obtained by performing a convolution operation of the image with a low-pass kernel. This technique is useful for removing noises present in the image. This method in fact eliminates high frequency content including noise and fine edges, from the image. Thus, the edges in the new image are blurred after applying this filtering operation. Gaussian Filtering is performed by convolution operation of the image with a Gaussian kernel. This operation can be executed by using the function, ***cv2.GaussianBlur()***. Here, we mention the width and height values for the Gaussian kernel, which are conventionally chosen to be positive odd integers. In addition, the standard deviation in X and Y direction, which are denoted as σ_X and σ_Y respectively, are also mentioned. In the case that only σ_X is given, σ_Y is considered equal to σ_X. If $\sigma_X = \sigma_Y = 0$, then they are both calculated from the vale of the kernel size. Gaussian blurring is a very effective technique for in removing the Gaussian noise present in an image.(10)

2.5.4 Canny Edge Detection

Edge detection is a very useful image processing operation that helps identify the boundaries of various objects and surfaces on the image. It is especially used for isolating objects within an image and find their boundary limits. ***Canny Edge Detection*** is a popular multi-stage edge detection algorithm, that takes place in four different stages.

1. Noise Reduction
2. Intensity Gradient Determination
3. Non-max Suppression
4. Hysteresis Thresholding

Noise Reduction: The first stage in this algorithm is to minimize the amount of noise that is present in the image with the help of applying a Gaussian Filter (conventionally with a 5x5 kernel size) on the original image.

Intensity Gradient Determination: This filtered and noise-free image that is obtained after this stage, is then convolved directions, by using a ***Sobel kernel*** in each of the two image coordinate to obtain a first order differential coefficient in both the directions. The derivatives obtained are ***G_x*** and ***G_y*** in the x axis and the y axis pixel coordinate directions in that order. The obtained two differential coefficient values then are further used to calculate the edge gradient and direction every pixel in the image. The direction of the Gradient is always perpendicular to edges.

Non-max Suppression: After calculating the magnitude and direction values for the gradient, the image is then analyzed to eliminate any pixels which may not contribute to an edge. This is done by determining if every pixel is locally a peak value in its vicinity, in the direction of the differential coefficient.

Hysteresis Thresholding: In this step, the true edges are filtered out of the pseudo-edges in the image. This is done by using two threshold values, namely ***minVal*** and ***maxVal***. Edges that had an intensity gradient value, that was greater than maxVal are definite edges and those edges with the value less than that of minVal are not true edges, and are discarded. The ones that lie between the two threshold values are classified as edges or non-edges based on the manner of connectivity. They are considered to be part of true edges only if they are directly joined to a definite edge, and discarded otherwise. All this is carried out in OpenCV by means of a single function, ***cv2.Canny()***

2.5.5 Contours

Contours, in Computer Vision, are defined at the very high level as a curve that connects all continuous points, along a set margin boundary, which either are of the same color or same level of intensity. The detected contours, in an image are very useful for analyzing shapes, and also for the tasks of detecting objects and subsequently recognizing those objects. It is common practice to use binary images to find contours. Thus, before applying the contour detection algorithm, we apply a binary threshold or canny edge detection to the image. This is followed by using ***findContours*** function to determine the contours from an image. (10)

2.6 Deep Feedforward Neural Networks

Deep Feedforward Neural Networks (DNNs), are primarily considered as the baseline for most deep learning models. The main aim of a DNN is to be able to approximate some arbitrary function f^*. These DNN models are referred to as feedforward, because the path taken by the information is first through the function that is being evaluated from $\boldsymbol{x}$, followed by the transitional stage analysis and calculations that define the function $\boldsymbol{f}$, ultimately reaching the desired output $\boldsymbol{y}$.(11). A generalized architecture of a Deep Neural Network used in this experiment is shown in Figure 6.

2.6.1 Hidden Units

After the input layer, a series of many hidden layers further extract much more abstract features from the input features. The layers are called as “hidden as their values are not directly given as input data. Instead, the neural network model must evaluate the information which is useful for explaining the patterns in the training data.(11) Rectified linear units, or ***ReLU*** units are a very good option, when it comes to hidden units. The challenging task is to decide when to use which kind of hidden units. The process of formulating an optimal hidden layer comprises of a lot of trial and error, testing various types of units based on their predicted performance, after which, it is used to then train a neural network model, after which we test the effectiveness of the hidden unit on a validation dataset. (11)

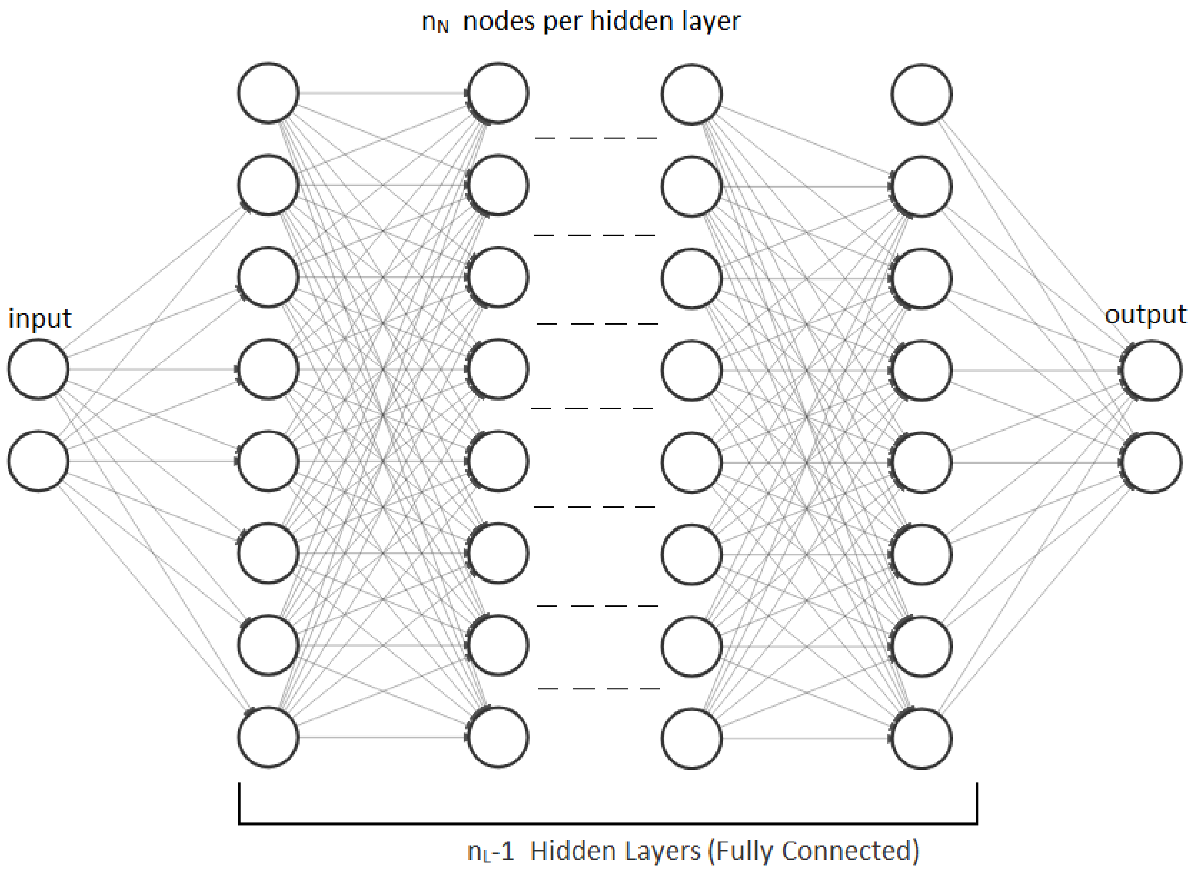

n_N nodes per hidden layer
input
output
n_L-1 Hidden Layers (Fully Connected)

Figure 6. Deep Neural Network General Architecture

2.6.2 Leaky Rectified Linear Units (LeakyReLU)

Rectified linear units, or ReLU units, as we saw in the previous section, follow the activation function $g(z) = max0, z$. ReLU units are very similar to linear units and thus, are easy to optimize. The gradient direction is much more useful for learning, compared to activation functions with the potential to have second-order effects. ReLU units are conventionally applied in $h = g(W^\top x + b)$. A generalization of ReLU, called as Leaky ReLU(12), is based on using a non-zero slope α_i when:

$$z_i < 0 : h_i = g(z, \alpha)_i = max(0, z_i) + \alpha_i min(0, z_i) \tag{2.8}$$

Leaky ReLU fixes α_i to a small value like 0.02.

CHAPTER 3

COMPUTER VISION SUBSYSTEM: TARGET DETECTION AND LOCALIZATION

The Computer Vision Subsystem experimental setup is the same as described in Chapter 2, and can be seen in Figure 1. Computer Vision Subsystem comprises of three stages mainly, which we shall see in detail, in this chapter:

1. Image Extraction and Processing
2. Image Analysis and Object Detection
3. Object Localization and Position Mapping

3.1 Baxter Area of Operation

The Baxter robot is quite flexible spatially, in most controlled autonomous operative environments. The robot in itself, however has various inevitable hardware limitations, due to the design and build of its various components. These limitations invariably include the robot's two arms. Each of the two Baxter's robotic arms have a near hemispherical three-dimensional Cartesian reach, inside which they can operate at highest efficiency. The current project, however has an area of operation spread out over a fixed work surface, which contains the intended target objects.

The effective area of operation of the Baxter robot is given by the area covered by the intersection of the Baxter's reach limits, with the horizontal work surface. This area of operation can be viewed in Figure 7. The area in gray represents an approximate reach boundary for Baxter, and the overlap with the work surface represents the effective area of operation.

3.2 Image Extraction from Baxter camera using CVBridge

As we have seen earlier, Baxter has three camera units (one on each arm, and one on the head). Each of these three cameras have their images that are Published at dedicated ROS nodes, as soon as they are enabled. The operations and specifications of the camera units can be controlled by the help of the Baxter Camera Control Modules, which contains all the functions necessary to adjust the camera resolution, and opening/closing. The image can be easily obtained by Subscribing to the corresponding ROS node.

This image can be extracted using cv_bridge package as follows(13):

```
raw_image = bridge.imgmsg_to_cv2(head_image, desired_encoding="passthrough")
```

This extracted image is now ready to be analyzed using OpenCV. The transformation method can be seen in Figure 8.

3.3 Color-Space Transformation and Smoothing

The head camera images were subscribed to at a resolution of 1280x800px. The head was fixed at its default position (zero pan and default tilt angle). The raw image obtained was then cropped manually, so as to isolate the work surface. The cropped image was at a resolution of 840x270px. This resolution was chosen so as to make the work surface entirely visible, while leaving out any other background view that may obstruct focus from the target object. The raw image of the work environment as observed from Baxter's head camera can be seen in Figure 9

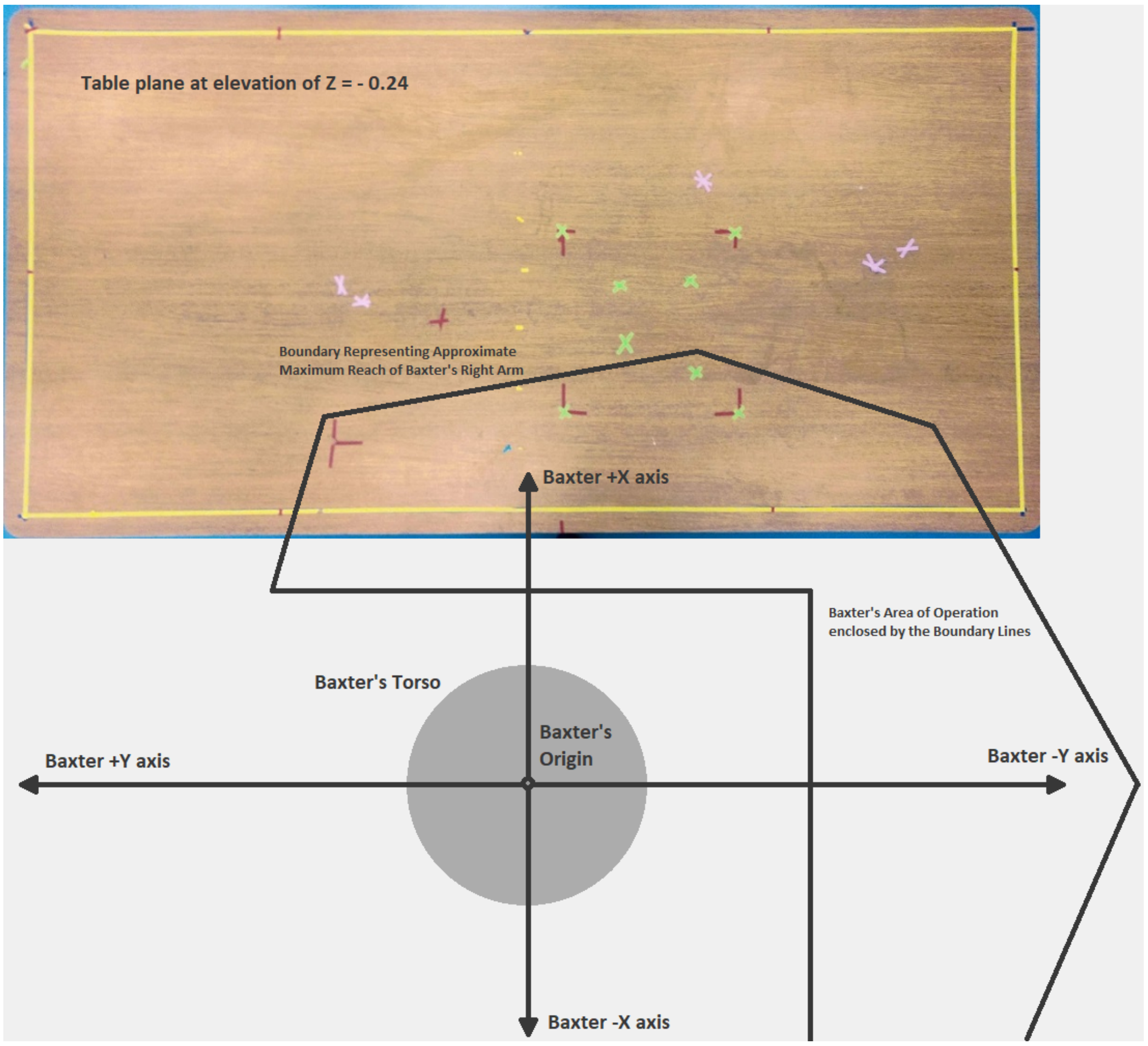


Table plane at elevation of Z = - 0.24
Boundary Representing Approximate Maximum Reach of Baxter's Right Arm
Baxter +X axis
Baxter's Area of Operation enclosed by the Boundary Lines
Baxter's Torso
Baxter's Origin
Baxter +Y axis
Baxter -Y axis
Baxter -X axis

Figure 7. Baxter's Effective Area of Operation

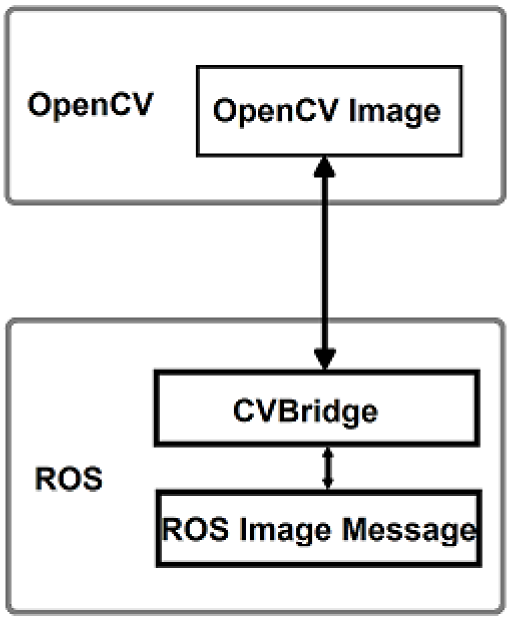


Figure 8. CvBridge

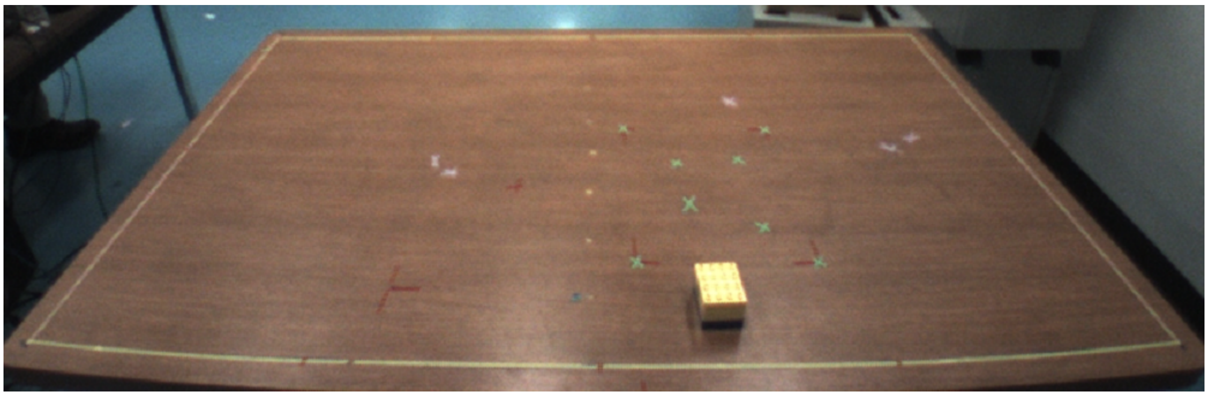


Figure 9. Raw Image of Work Surface

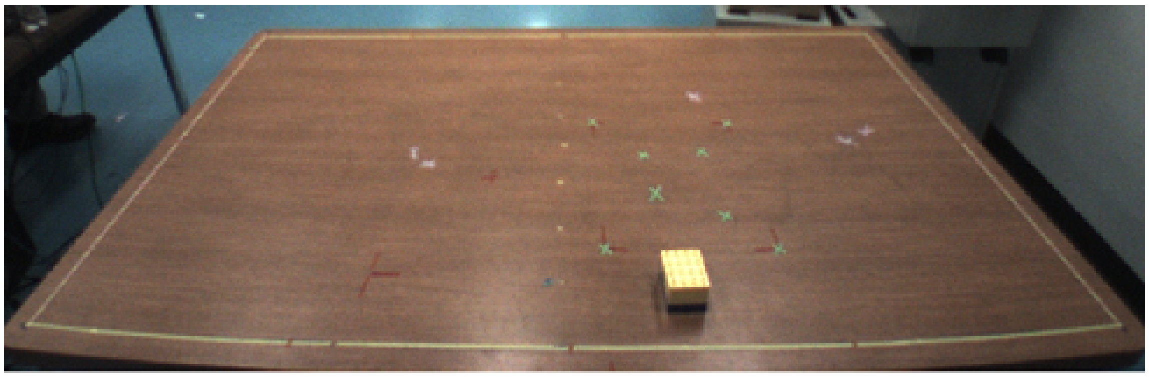
As we can see, the image has been cropped to isolate the work surface for easier analysis for target objects.

The next stage would be to apply a color space transformation to this image that has been obtained from the head camera. As discussed in Chapter 2, we apply a $BGR \rightarrow GRAY$ color space transformation using ***cv2.COLOR_BGR2GRAY***. This transformation can be seen in Figure 10 (b). The reason we apply this transformation is to convert the three-color channel image into a monochrome color channel image. This makes it easier for us to apply further filters and transformations to the image.

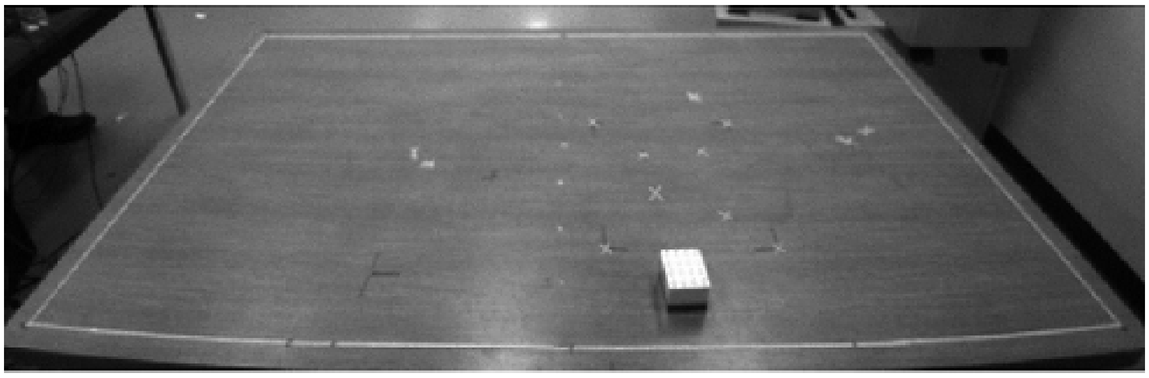
After the color-space transformation, we now have a monochromatic image. The next stage is to eliminate any unwanted noise from this image. This can be done by performing a smoothing (also known as blurring) operation on the image. As we saw in Chapter 2, smoothing of an image is done by a 2D convolution operation of the image with an existing, or custom-made low pass filter. In this experiment, we use a Gaussian Kernel. This is a preferred kernel for this project, as it is very effective in eliminating the Gaussian Noise that is present in the monochromatic image. This stage, as previously seen, can be carried out by using the function ***cv2.GaussianBlur()***, as follows:

```
    guass_image = cv2.GaussianBlur(gray_space_image,(5,5),0)
```

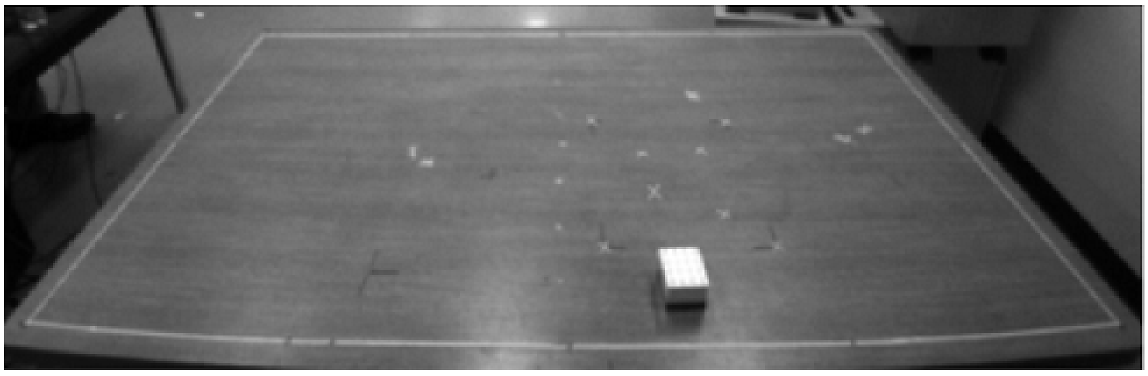
In the above example, as in our experiment, we have used a $(5, 5)$ kernel for the convolution with the monochromatic image. The effect of this operation is a slight blur in the image, as can be observed in the Figure 10 (c). This image is now relatively free of Gaussian noise, and ready for object detection algorithms to take over.



(a) Original Image



(b) Image after applying BGR to GRAY color-space transformation



(c) Image after convolution with Guassian Filter with (5, 5) kernel

Figure 10. Image Transformation and Smoothing

3.4 Object Detection from Image

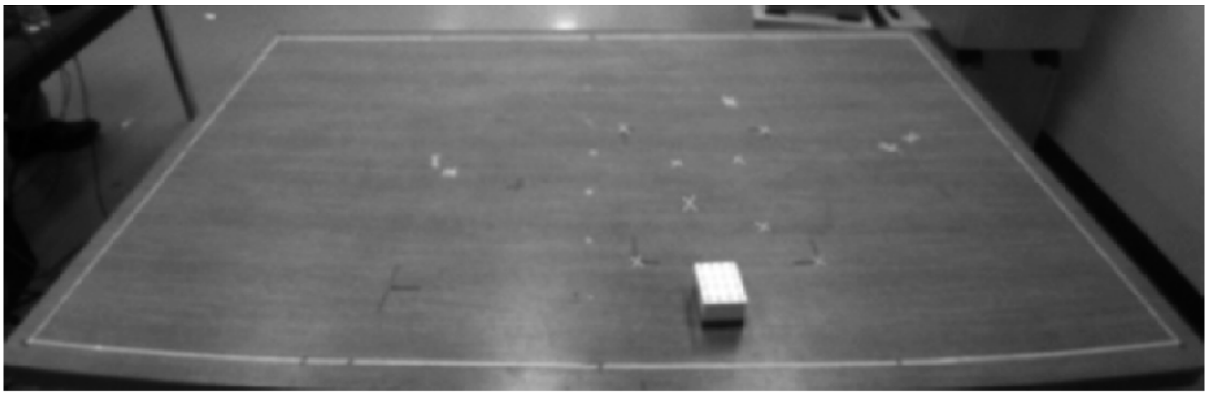
As we saw in the previous section, the transformed, and smoothed image is now ready for applying Object detection algorithms. As we can see in Figure 10 (c), the target block that is intended to be grasped, is present a little offset from the lower centre of the work surface (within the area of operation for the Baxter Robot). We shall consider a cube block case for explanation of the next stages in this experiment. The same logic can be extended to other shapes as well, with equivalent modification. There are two steps to detecting a regular target object (a cube block here):

1. Detecting the approximate shape of the target
2. Isolating the possible target contour based on area of all potential targets

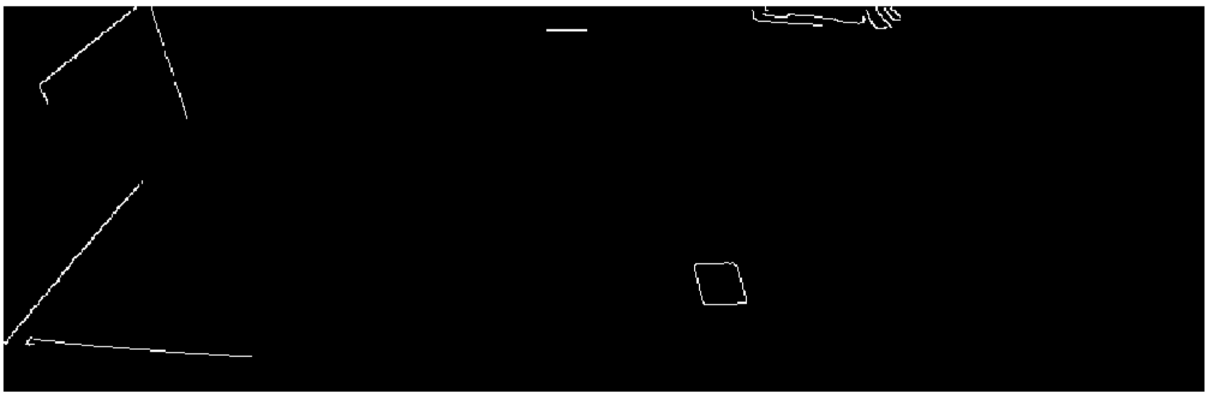
3.4.1 Shape Detection

From the previous section, we have obtained a filtered and noise-free image. We apply Canny Edge detection algorithm on this image, to detect the target object. The edge detected image can be seen in Figure 11 (b).

The algorithm for shape detection revolves around the concept of approximate contour vertex count. We can determine the locations of the contours in the processed image. The function ***cv2.findContours*** is used to obtain a collection of contour outline curves for every white blob that is present on the edge-detected image. For us to identify the shape of the target object, we shall be applying the technique of approximating contours. This method of approximating contours is an algorithm that helps to reduce total count of points present in the



(a) Gaussian Filtered Image



(b) Canny Edge Detected Image

Figure 11. Edge Detected Image

contour plot by replacing it with a smaller array of points. This algorithm of approximating a curve is also known as the ***Ramer-Douglas-Peucker***(14) algorithm, or the split-and-merge method. The process of Contour estimation is based assuming that a given curve can be reduced to a continuous series of short length line segments. This step give us an approximated curve that effectively consists of a subset of coordinates that were part of the original curve. For performing an approximation of a contour, we first calculate the perimeter of the given contour-plot, after which we construct the final contour approximation. Contour approximation can be implemented using OpenCV by applying the function ***cv2.approxPolyDP***.

After approximating the contour, we can perform a shape detection operation on the image. What we need to understand here, is that a given approximated contour contains a list of vertex coordinates. We can effectively identify the shape of an object by observing the total number of vertices present in this list. As an example, if a contour-plot contains four vertices, then it can be identified as either a square or a rectangle. To distinguish between the two, we calculate the value of aspect ratio of the identified shape, which is the ratio of width of the contour bounding box with its height. For a square, as in our case, the contour aspect ratio value is approximately 1. The angles of the square cross section would not be 90 degrees, however, and the square would be closer to a rhombus due to fish-eye distortion.

3.5 Object Localization using Contour plot

In the field of Computer Vision and Image Processing, the values of moments of an image are quite useful to describe the shape of a particular object present in an image. These moment values can record inherent statistically important properties of the given shape contour, which

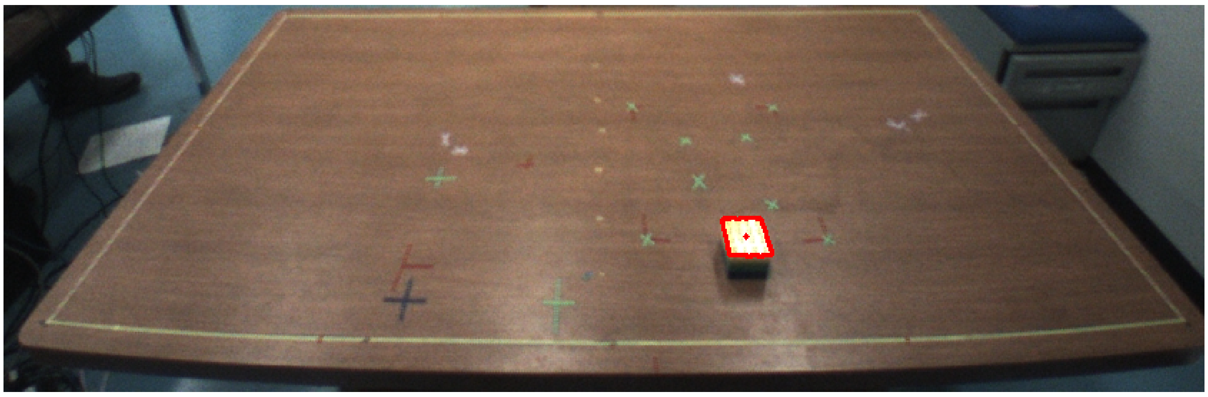


Figure 12. Object Localization and Centroid Calculation

include area of the object, its centroid, the relative orientation value, in addition to other desirable properties of the object. The centroid coordinates ***C(x, y)*** of a given contour can be calculated in the following manner. We apply the function ***cv2.moments()*** on the contour and receive a dictionary M containing the moment values. The coordinates are given by:

$$X = \frac{M_{10}}{M_{00}}; Y = \frac{M_{01}}{M_{00}} \tag{3.1}$$

The localization of the object and centroid calculation can be seen in Figure 12. The pixel coordinates of the object are now obtained and can be used as inputs to the Frame Transformation Subsystem.

CHAPTER 4

FRAME TRANSFORMATION SUBSYSTEM: TARGET CARTESIAN POSITIONING

From the Computer Vision Subsystem, we have obtained the ***(x, y)*** coordinates for the centroid of the target object placed in the work space (a cube, in our case). Now, the next stages in the process will be to use these visual coordinates, and determine the three-dimensional Cartesian coordinates of the target object on the work surface, in the frame of reference of the Baxter's torso. After obtaining the Cartesian coordinates, we can use Inverse Kinematics to solve for the corresponding joint angle values for each of the seven robot arm joints. After all these stages, the robot can attempt a grasp, to potentially pick up the target cubic block.

4.1 Seed Data Collection

The primary approach taken by us to designing a Machine Learning model to obtain Cartesian Coordinates from visual coordinates, in the most ideal case, would involve large scale collection of training data: both visual coordinates, and their corresponding three-dimensional Cartesian coordinates. This procedure will be very time-consuming, thus reducing the practicality of a trained model significantly. Instead, automating the collection of seed training points and generating a large dataset from those points, to train a model, is a much more convenient alternative.

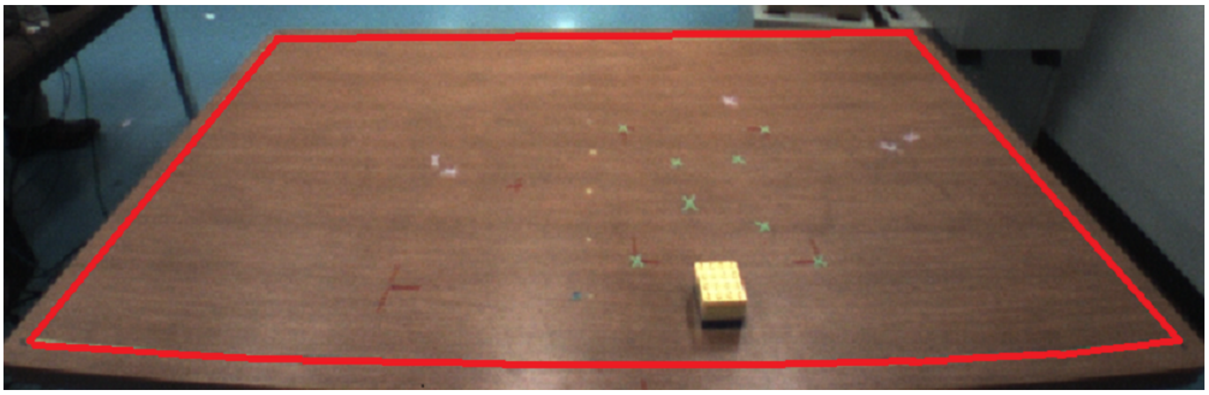


Figure 13. Contour Plot of Reference Rectangle

For collecting the seed points, we have a reference rectangle drawn on the work surface, as seen in Figure 9. We first identify this rectangle as a contour in the image of the environment as seen by the Baxter Robot. As an example, you can see the contour of the rectangular box in red in the Figure 13. From this contour, we obtain the contour plot, which is the array of all coordinate points that the contour-plot comprises of. The coordinates are saved as a numpy array, which can be used to plot and extract seed points along the perimeter of the rectangular contour. For this project, we experiment with two seed sizes: 6 point seed array, and 12 point seed array. Here, $\boldsymbol{s}$, is the number of seeds, and we can see training data and targets for $s = 6$ and $s = 12$ in Figure 14 and Figure 15 respectively.

4.2 Synthetic Data Creation using Point-Web Generation

After collecting the seed points, we then use these seed points to generate our synthetic dataset to train the Machine Learning Models. This is done by collecting training points

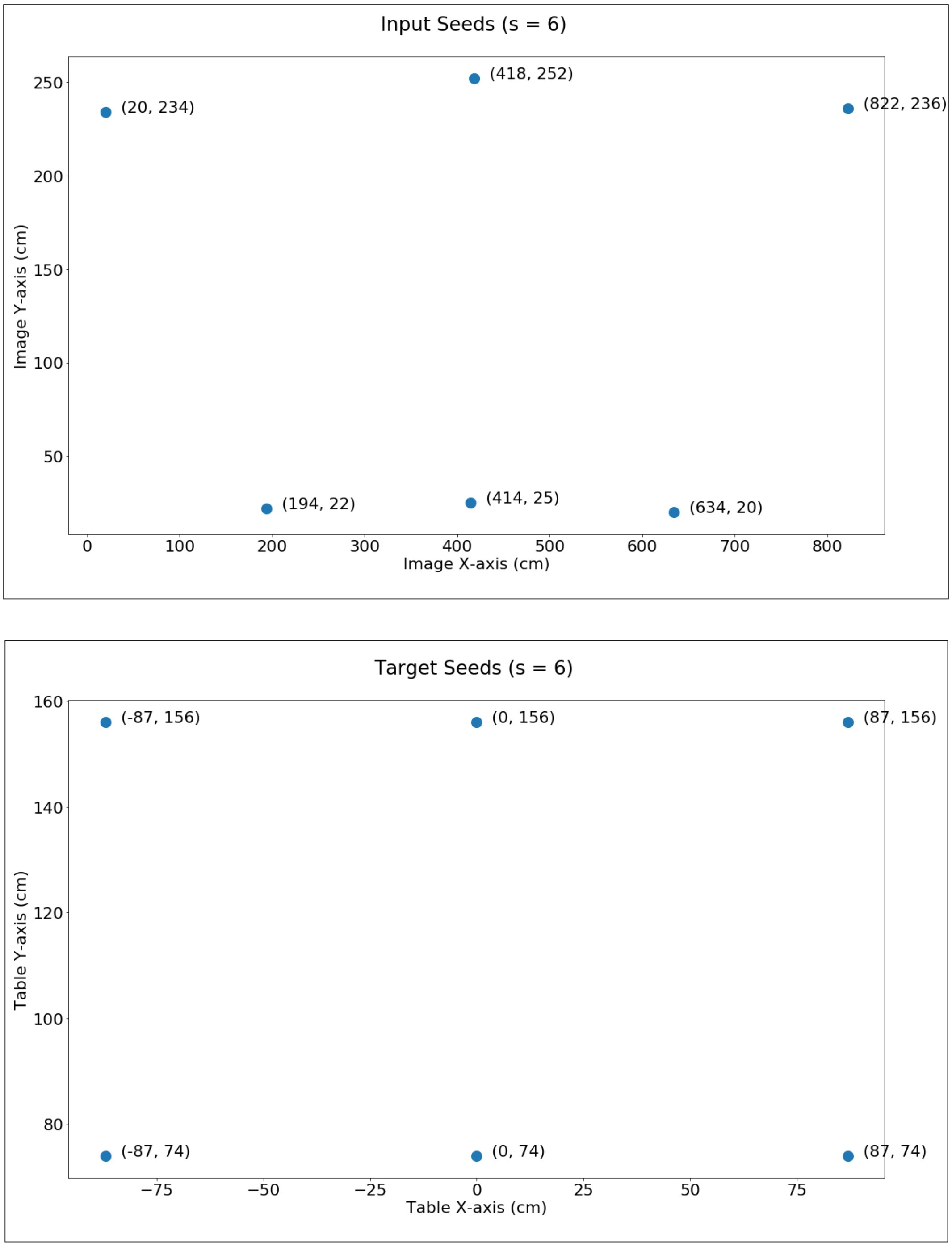
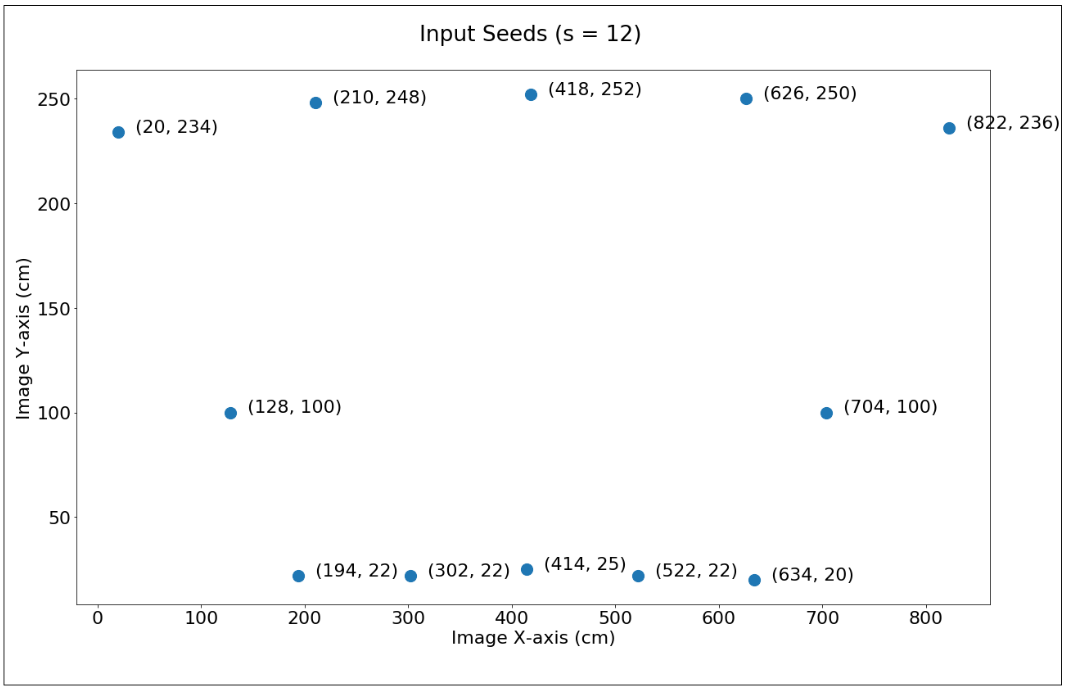


Figure 14. Input and Target Seeds (s = 6)



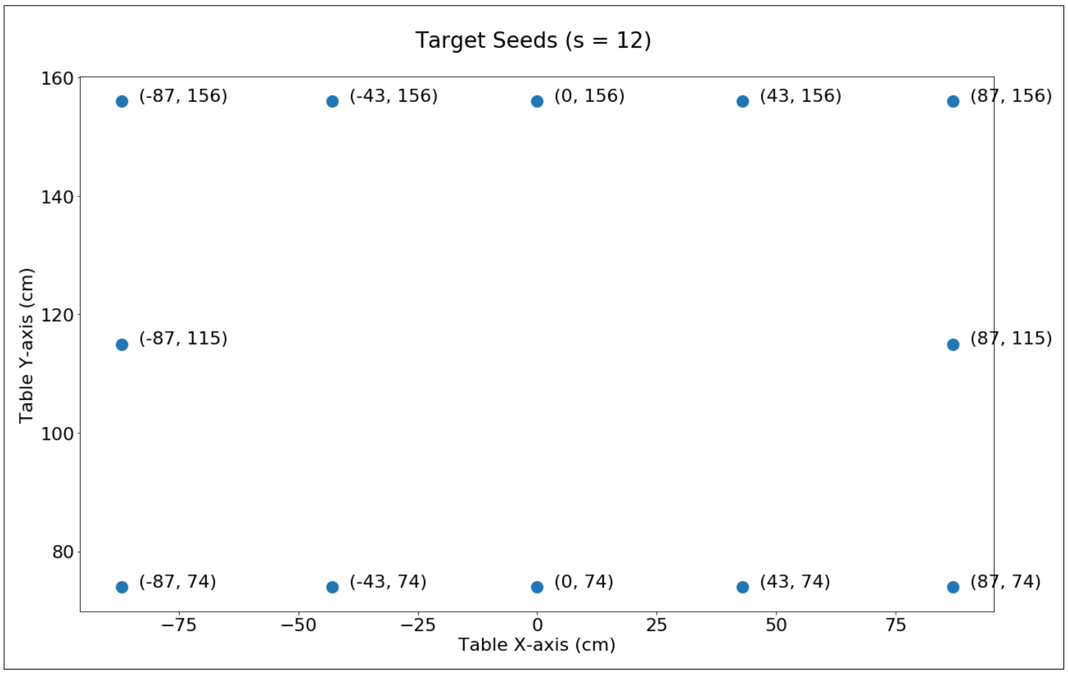


Figure 15. Input and Target Seeds (s = 12)

generated along the web of line segments created by taking pairs of seed points as the endpoints for each line segment. In other words, we take all possible pairs of seed points, and determine the equation of the line segment with those points are the endpoints. The general slope-intercept form of an equation of a line, having slope m and y-intercept c is given by:

$$y = m.x + c \tag{4.1}$$

The equation can be determined using the two point form of the equation of a line in a 2D plane. The two point form for the equation of a line with points (x_1, y_1) and (x_2, y_2) as the endpoints is given as:

$$(y - y_1) = \left[\frac{(y_2 - y_1)}{(x_2 - x_1)}\right](x - x_1) \tag{4.2}$$

The above equation can be re-written as

$$y = \left[\frac{(y_2 - y_1)}{(x_2 - x_1)}\right](x) + \left[y_1 - \frac{(y_2 - y_1)}{(x_2 - x_1)}(x_1)\right] \tag{4.3}$$

From equations 4.1, and 4.3, we can get the slope m and y-intercept c as

$$m = \left[\frac{(y_2 - y_1)}{(x_2 - x_1)}\right] \tag{4.4}$$

$$c = \left[y_1 - \frac{(y_2 - y_1)}{(x_2 - x_1)}(x_1)\right] \tag{4.5}$$

After we obtain the equations for each of the line segments in the web, we can use them to generate N points along each of the lines, with ends $P_1(x_1, y_1)$, and $P_2(x_2, y_2)$ as follows:

$$x_{min} = min(x_1, x_2) \tag{4.6}$$

$$x_{max} = max(x_1, x_2) \tag{4.7}$$

$$d = \frac{|x_1 - x_2|}{N - 1} \tag{4.8}$$

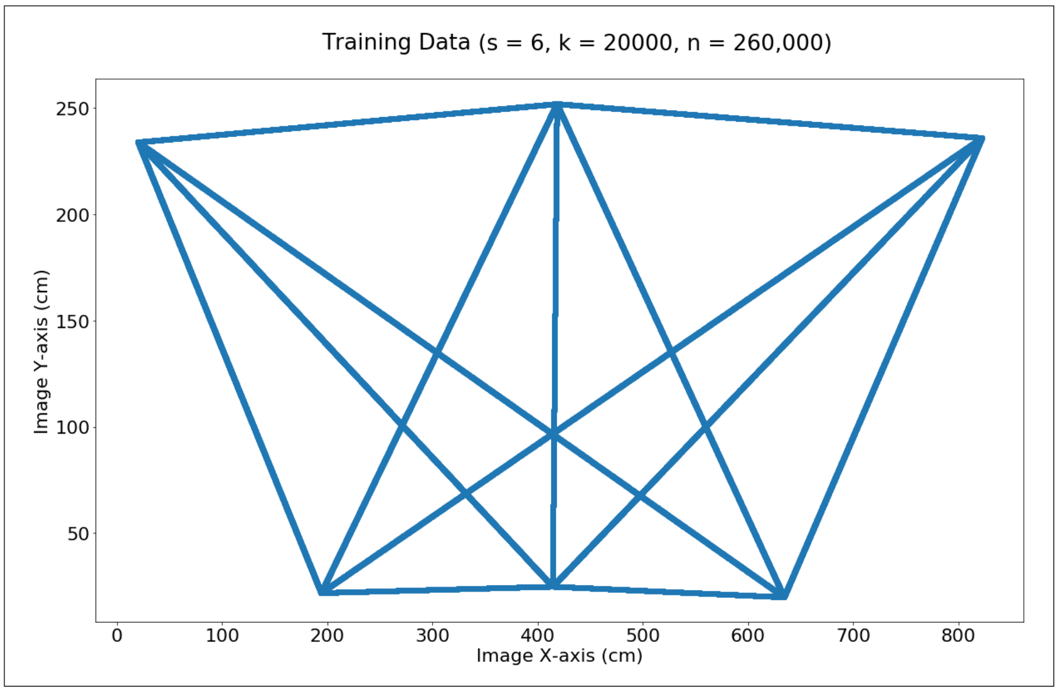
The N points can be generated using these values as:

$$X = [(x_{min}), (x_{min} + d), (x_{min} + 2d), ..., (x_{min} + (N - 2)d), (x_{max})] \tag{4.9}$$

Once we have the x-coordinate value vector generated, we can use vector algebra to generate the corresponding ordinate vector as:

$$Y = mX + C \tag{4.10}$$

Here, m is the slope calculated from equation 4.4, and C is the vector broadcasted form of c calculated in equation 4.5. The same set of equations can be used for both the seed values. The web of points generated for $s = 6$ is shown in Figure 16, and he points generated for $s = 12$ is shown in Figure 17. In this project we have used $N = 2000$, to generate 260,000 training examples for $s = 6$ and $1,080,000$ training examples for $s = 12$.



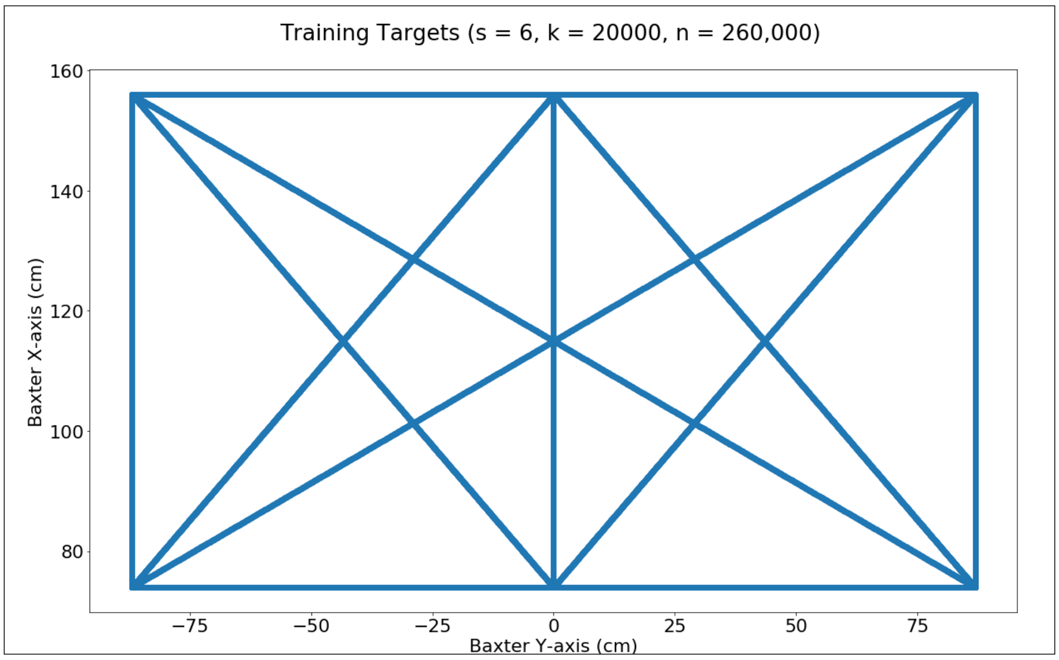
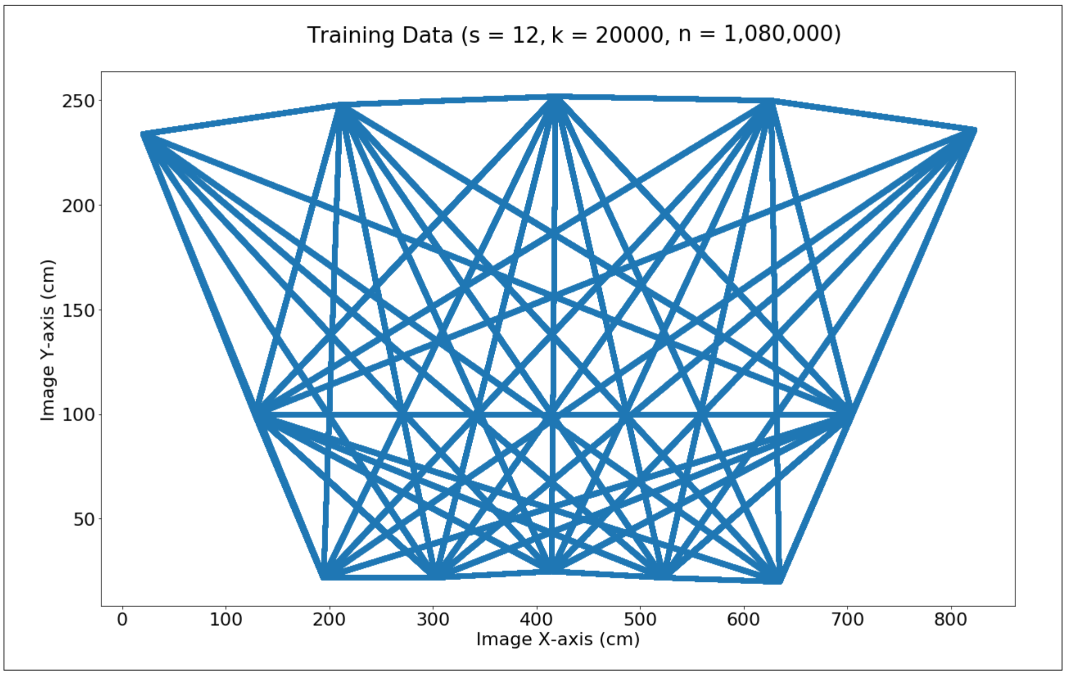


Figure 16. Generated Synthetic Training Data and Targets (s = 6)



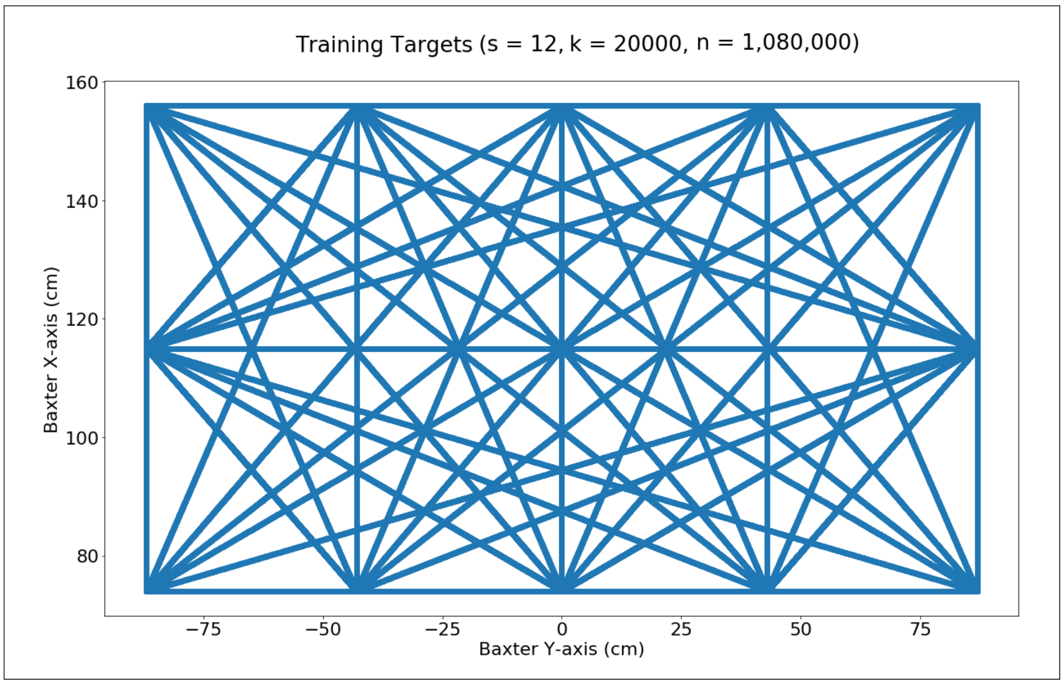


Figure 17. Generated Synthetic Training Data and Targets (s = 12)

TABLE I

ARCHITECTURES OF DNN MODELS

Model	Layers (n_L)	Nodes (n_N)	Seed Size s
DNN_100_5	5	100	6
DNN_100_10	10	100	6
DNN_100_20	20	100	6
DNN_100_30	30	100	6
DNN_200_5	5	200	6
DNN_200_10	10	200	6
DNN_500_2	2	500	6
DNN_100_5	5	100	12
DNN_100_10	10	100	12
DNN_100_20	20	100	12
DNN_200_5	5	200	12

4.3 Deep Neural Network Models - Architectures

The architecture of the Deep Learning Models that has been used here is Deep Multi Layer Perceptron Neural Networks, or simple Deep Neural Networks (DNNs). As a part of laying down notation references used for the models, the number of layers shall be given by n_L, and the number of nodes present in each layer shall be given by n_N. Activation functions for each neuron shall be given by $g()$ and the output of each neuron is given by z. We have evaluated eleven different models for the project. The architecture specifications can be seen in Table I. All neurons use a LeakyReLU activation function with gradient $r = 0.02$. Using LeakyReLU activation will help combat the vanishing gradient problem often faced in very Deep Models.

4.4 Deep Neural Network Models - Training and Evaluation

4.4.1 Training

The models were all built using the Keras framework over TensorFlow. The optimizer used for the models was the Adam Optimizer (15). This was because it adaptively decays the learning rate while training to help efficient convergence of the model. The hyper-parameter specifications can be seen in. All models were trained in batches of size $b = 2048$, for $k = 600$ epochs. The models were trained on a CUDA GPU. The training data was divided onto a 90% training set and a 10% test set for all the models. For the models, we used the decay parameters $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

4.4.2 Evaluation

The loss function used for training the model was the mean Euclidean Distance Loss function, for the predicted coordinate with respect to the ground truth coordinates. This loss function, for m training examples can be defined as:

$$J(W) = \frac{1}{m}\sum_{i=1}^{m} E(Y_i, \hat{Y}_i) = \frac{1}{m}\sum_{i=1}^{m} \sqrt{[Y_i - \hat{Y}_i]^2} \tag{4.11}$$

The training progress of the models trained on data generated from seed $s = 6$ can be seen in Figure 18 and Figure 19, and the models trained on data generated from seed $s = 12$ can be seen in Figure 20.

The final testing losses can also be seen in Table II and in Figure 21. From here, it can be observed that interestingly, a denser point web proved to be counter-intuitive to the effectiveness in prediction. Generally, the point webs generated by $s = 6$ seed size proved to be very effective in letting the models generalize to the test set.

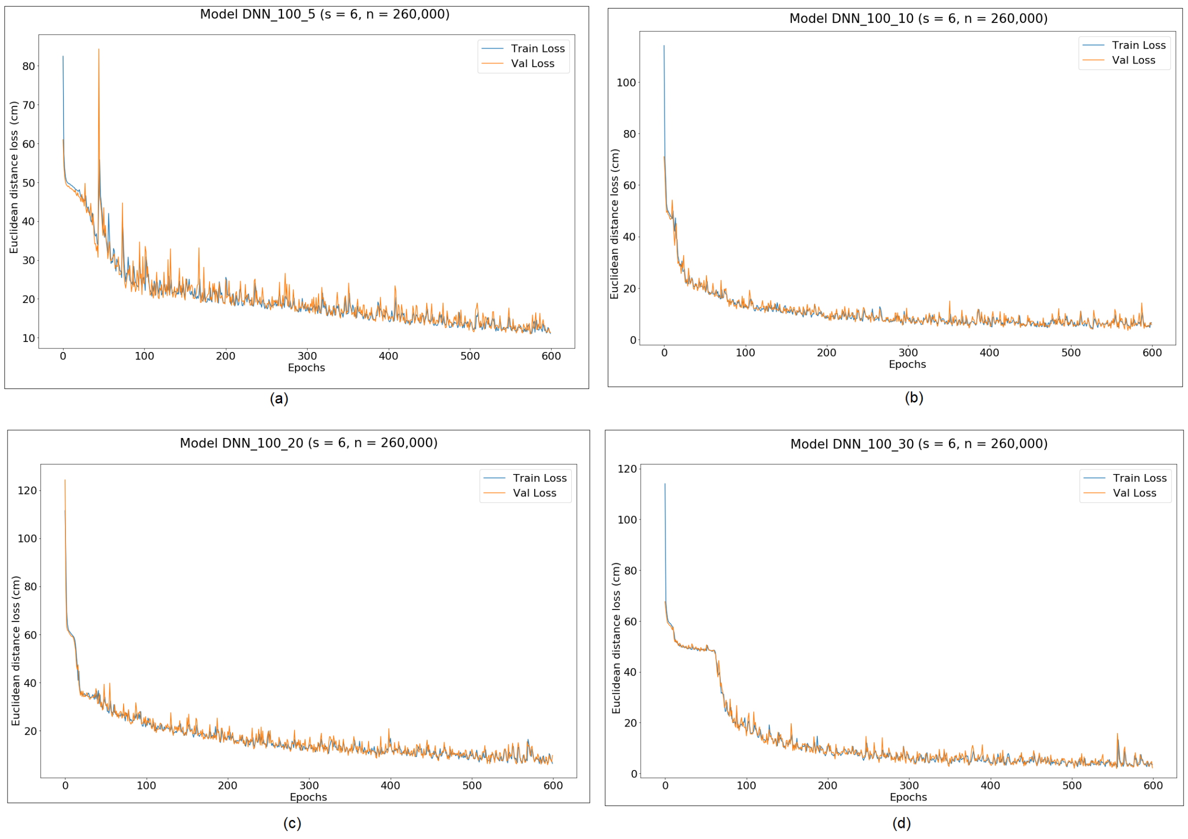
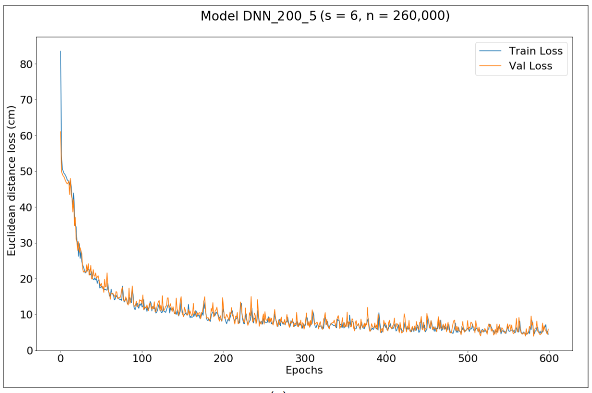
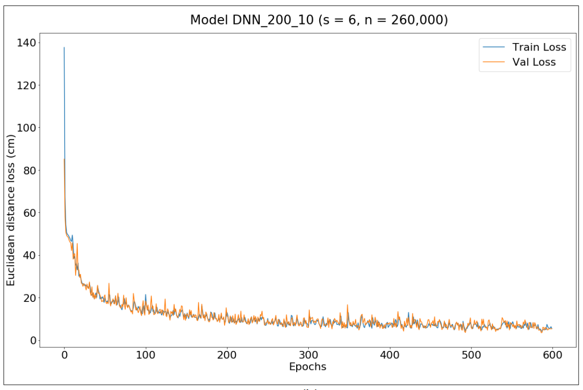


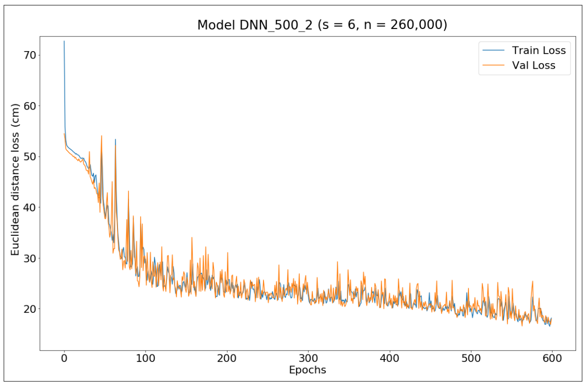
Figure 18. Training of Deep Learning Models (s = 6)



(a)



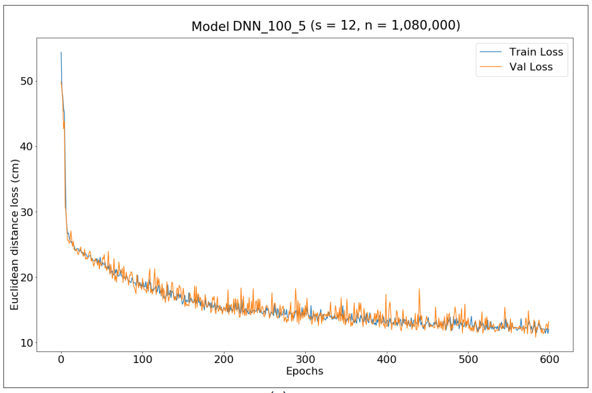
(b)



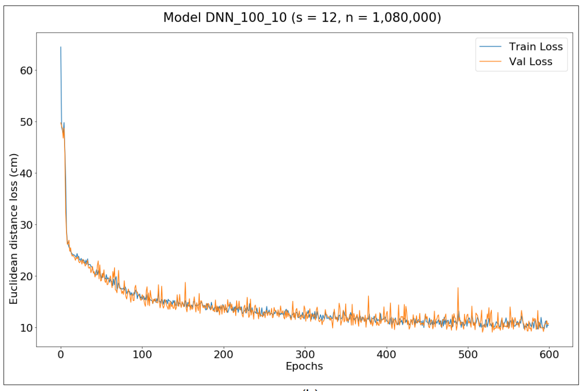
(c)

Figure 19. Training of Larger Deep Learning Models (s = 6)

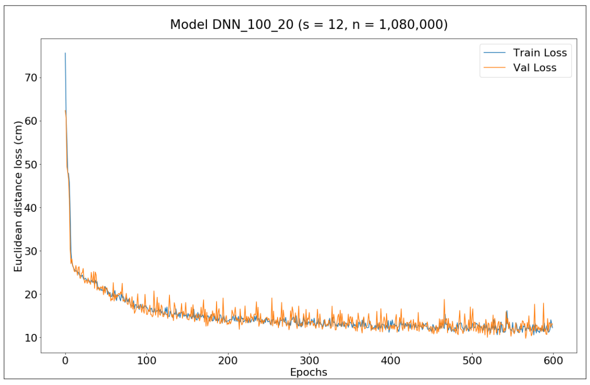
The error on the test set was not uniformly distributed across the plane. The points further away from Baxter experienced higher error prediction rates than the points closer to it. This is quite possibly due to increasing fish-eye convergence away from the Baxter. Model DNN_100_30 performed the best, reaching a test error of $3.103cm$ after the 600 epochs of training, for seed $s = 6$. This model took approximately 2 hours to train on the GPU.



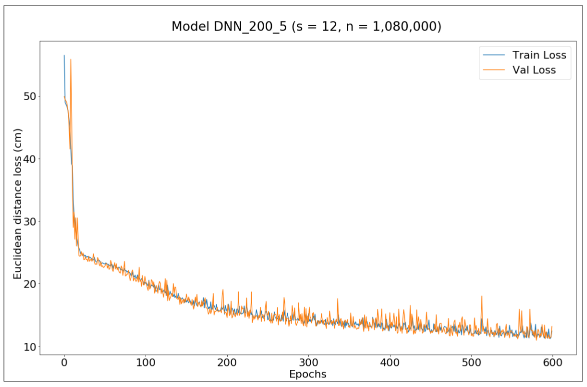
(a)



(b)



(c)



(d)

Figure 20. Training of Deep Learning Models (s = 12)

In total, about 100 different model architectures were tested and tuned during the course of the research. Out of those, the eleven that have been showcased here, were chosen to represent a sample of all the models tested. The models chosen were chosen as they best represented the relation between model architecture and the performance on the two data sets. The criteria for choosing the models included effectiveness, and training time feasibility, which made them

TABLE II

TRAINING AND EVALUATION RESULTS

Model	Learning Rate (α)	Euclidean Distance Error (cm)
Seed size s = 6		
DNN_100_5	50e-4	10.16
DNN_100_10	20e-4	7.07
DNN_100_20	15e-4	5.97
DNN_100_30	**10e-4**	**3.10**
DNN_200_5	15e-4	6.13
DNN_200_10	10e-4	4.71
DNN_500_2	50e-4	14.51
Seed size s = 12		
DNN_100_5	50e-4	11.73
DNN_100_10	50e-4	9.31
DNN_100_20	20e-4	11.89
DNN_200_5	10e-4	10.97

useful for the transformation problem. It was also observed that tuning of the various hyper-parameters and choosing the appropriate architecture of the models was extremely challenging, and finalizing a model took several training sessions before it could be deemed feasible. The training data was split into two sets; 90% in the training set, and 10% in the validation set. The validation set was used to tune the hyper-parameters to their optimal values.

4.5 Inverse Kinematics using IKService

Once we have obtained the Cartesian coordinates, we need to extend the robot arm to that point in space, in the Baxter's Frame of reference. For this, we shall use Inverse Kinematics. Baxter SDK has on-robot Inverse-Kinematics (IK) Service which can be used to obtain a joint angles solution, if valid, for a given endpoint Cartesian point and Orientation. We can thus obtain the seven joint angles from Cartesian coordinates as follows:

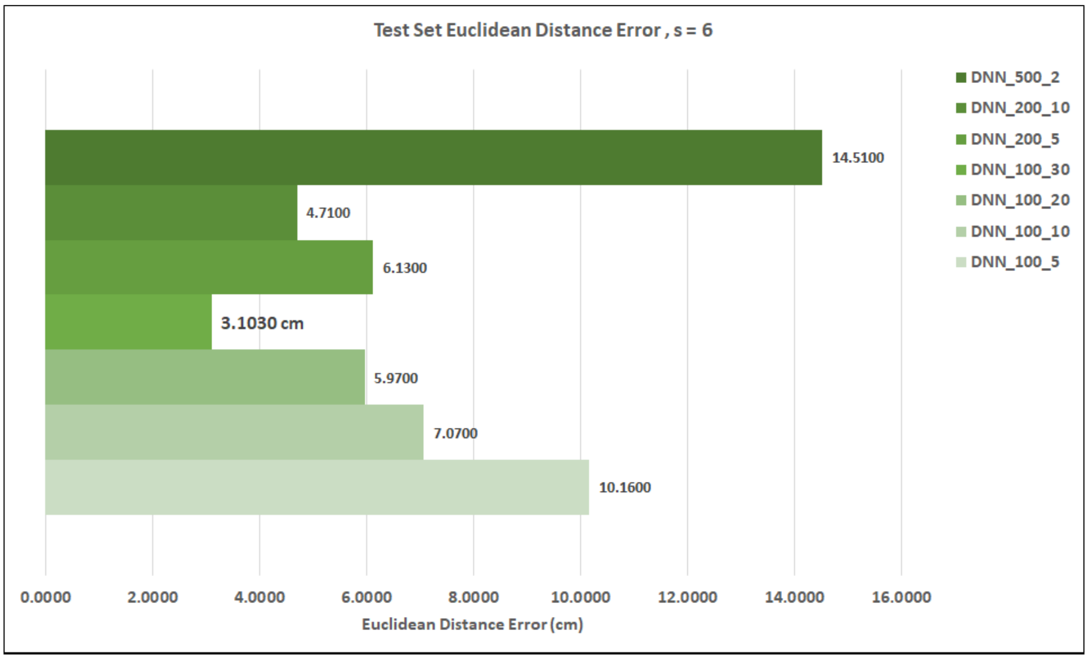
$$IK([x, y, z, r_i, r_j, r_k, r_w]) = [S0, S1, E0, E1, W0, W1, W2] \tag{4.12}$$

For this experiment, we have tested with a top approach grasp constraint, for the Inverse Kinematics calculations. For this, we have a constant values for $[r_i, r_j, r_k, r_w]$. as:

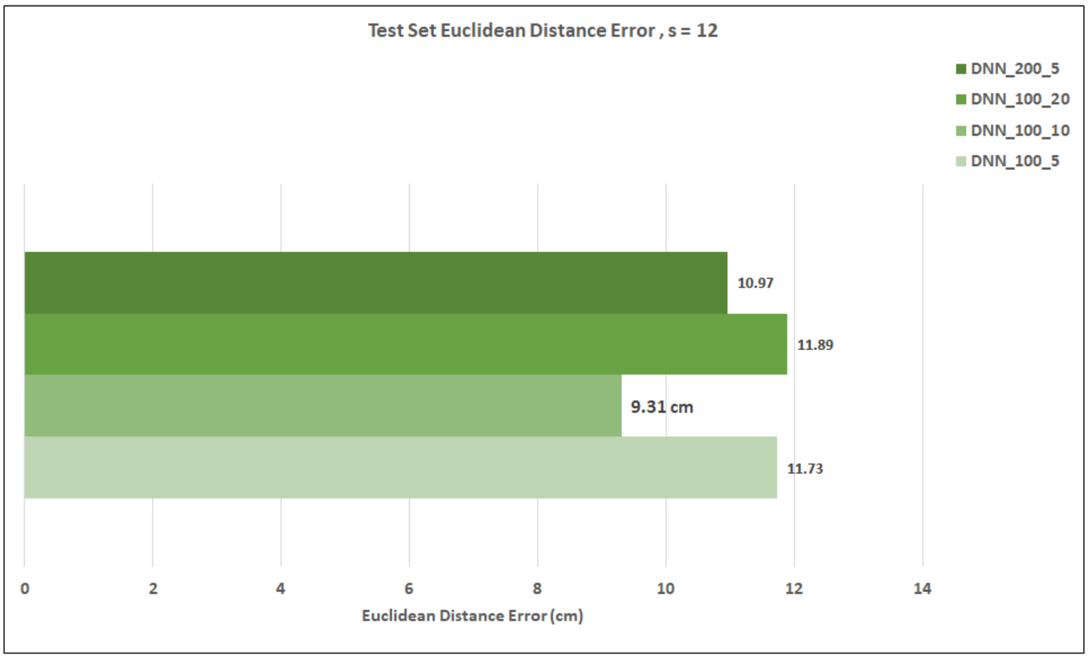
$$r_i = 0, r_j = 0, r_k = 0, r_w = 1 \tag{4.13}$$

Thus, $[x, y]$ for the final Cartesian position for the Baxter end-effector is obtained from the Deep Learning Model, $[z] = 0.24$ is constant for the experimental setup, and the orientation values $[r_i, r_j, r_k, r_w] = [0, 0, 0, 1]$ are constraints that are set on the Inverse-Kinematics solver, to plan and attempt a top grasp for the target.

In this chapter, we have seen how the models fared in the training and validation phase of the experiment. In the next chapter, we shall see how these models fared in the live experiments that were carried out to confirm their performance.



(a)



(b)

Figure 21. Training and Evaluation Results

CHAPTER 5

EXPERIMENTS AND RESULTS

In this section, we shall discuss the grasps that were attempted using the various models and present evaluation metrics for those experiments.

5.1 Test grasp reach attempts

For each of the eleven models, we tried five grasp attempts as part of the testing process. For the experiment, we used an $8cm x 8cm x 6cm$ cuboidal block as a target for the robot to pick-up. Of the five attempts, four were within the area of operation of the robot arm, and one test attempt beyond the area of operation for evaluation purposes. The test grasp positions can be seenTable III. A total of 55 grasp attempts were recorded in the testing process, 44 being actual grasps, and 11 being theoretical grasps. An actual grasp was one, where the target was placed in a region within the FOV of Baxter, and inside the area of operation, as seen in Figure 7. An actual grasp was one, where the target was placed in a region within the FOV of Baxter, but outside the area of operation.

A sample grasp attempt can be seen in Figure 22. Here, in (a), we see that the arm is at an arbitrary neutral position on the right side of the Baxter's torso (not constant). In (b), it reads the environment in front of it through the head camera, and detects and locates the target block. In (c), it approaches the target from a top-based approach path. Finally, in (d), we can pick the block off the work space, for further actions.

TABLE III

TEST GRASP POSITIONS

Position	x	y
1	75.0000	21.0000
2	75.0000	-23.0000
3	79.0000	12.0000
4	85.0000	-30.0000
5	89.0000	-32.0000

5.2 Grasp attempt evaluation metrics

As we saw in the previous section, each model was tested on five grasps. During the training of the models, it was also observed that the errors for the validation set examples increased, further away from the Baxter. Hence it was fair to expect a similar trend in the experimental grasps too. For the grasps, we obtain the grasp-wise Euclidean distance error for each model on each grasp, and compare them in Figure 23. From the figure, we can see that for $s = 6$, DNN_100_30 performs the best with extremely low errors on all grasps. Meanwhile for $s = 12$, DNN_200_5 performs the best. We can also observe that for the last grasp, which is the virtual grasp point, the error is higher in all models. This follows the trend of increasing non-linearity in the fish-eye distortion, as we go farther from the Baxter.

5.3 Comparison of DNN models

Finally, we can compare the overall performance of all models for both the data-sets, and view the average error of each model on all five grasps, for both $s = 6$ and $s = 12$ in Figure 25.

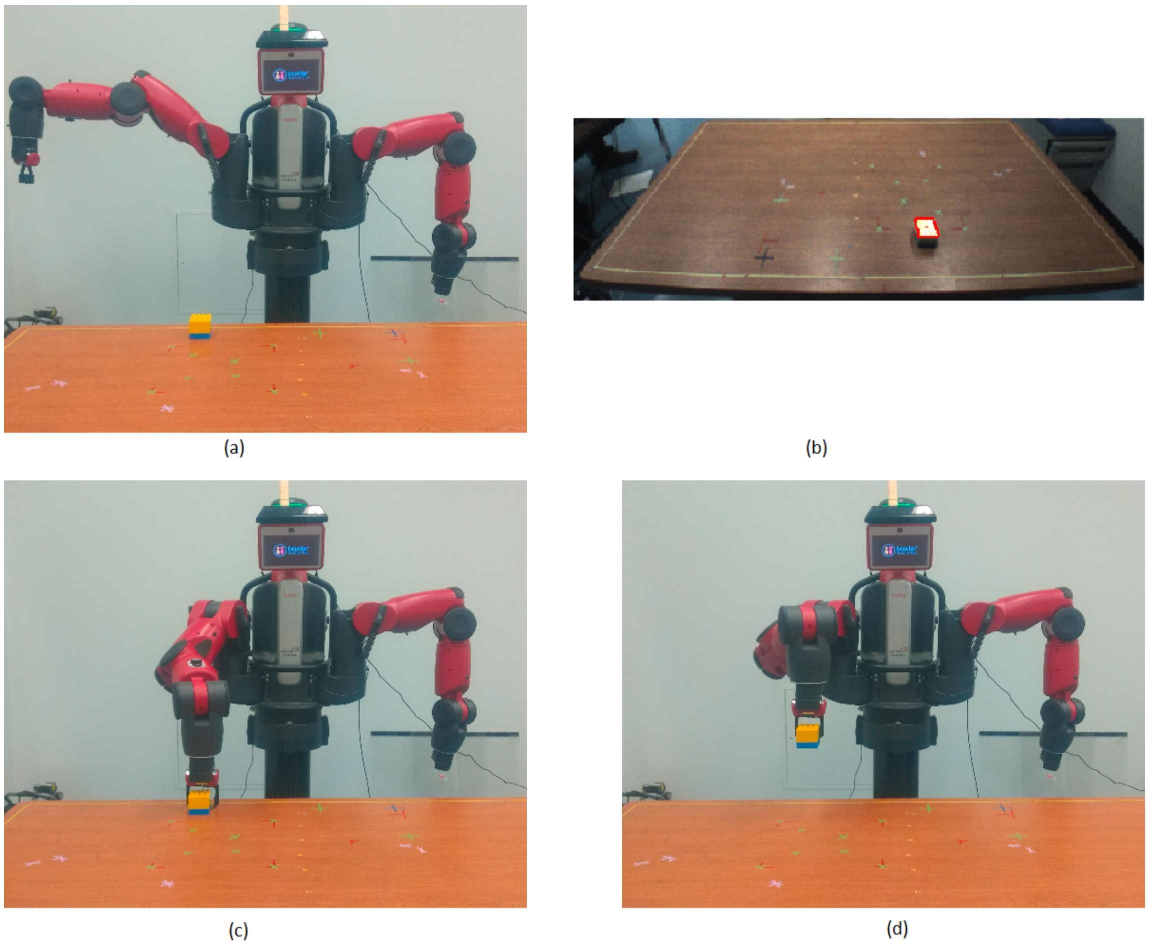
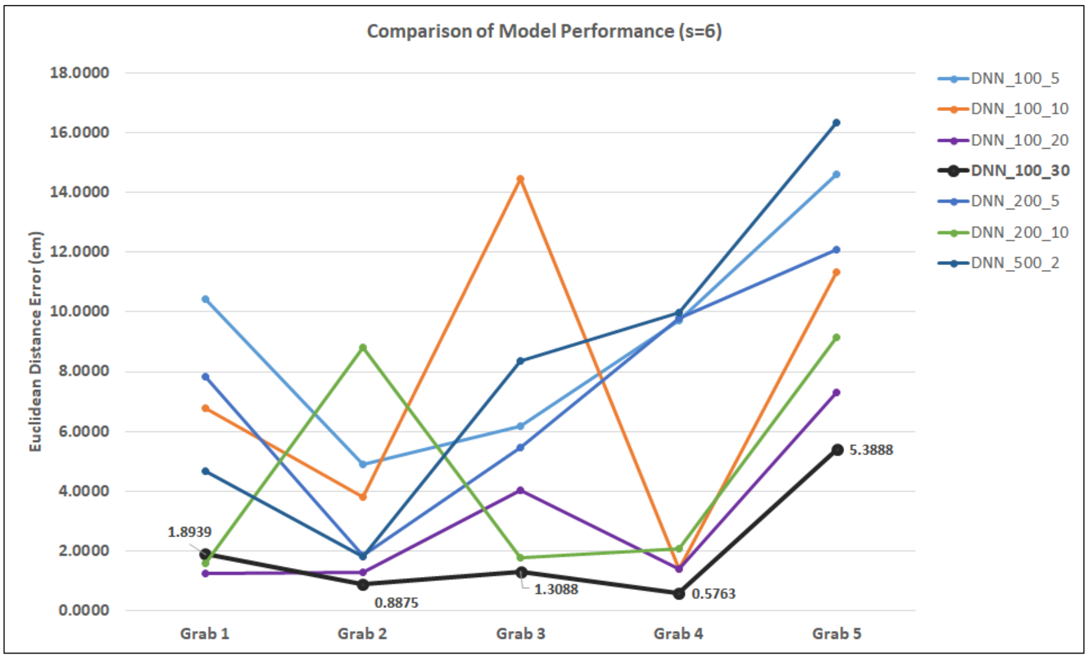
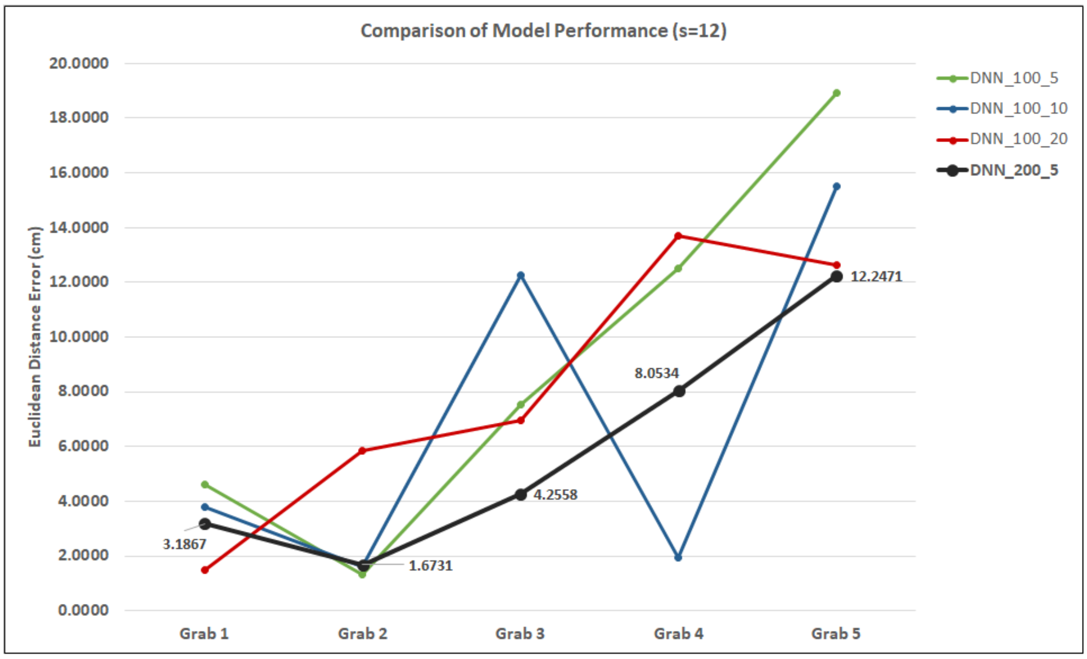


Figure 22. Grasp Attempt for Cuboidal Target

Here we see that overall for $s = 6$, DNN_100_30 performed the best with only 2.0111 cm error in grasping, while DNN_200_5 performed the best with 5.832 cm of error for $s = 12$.

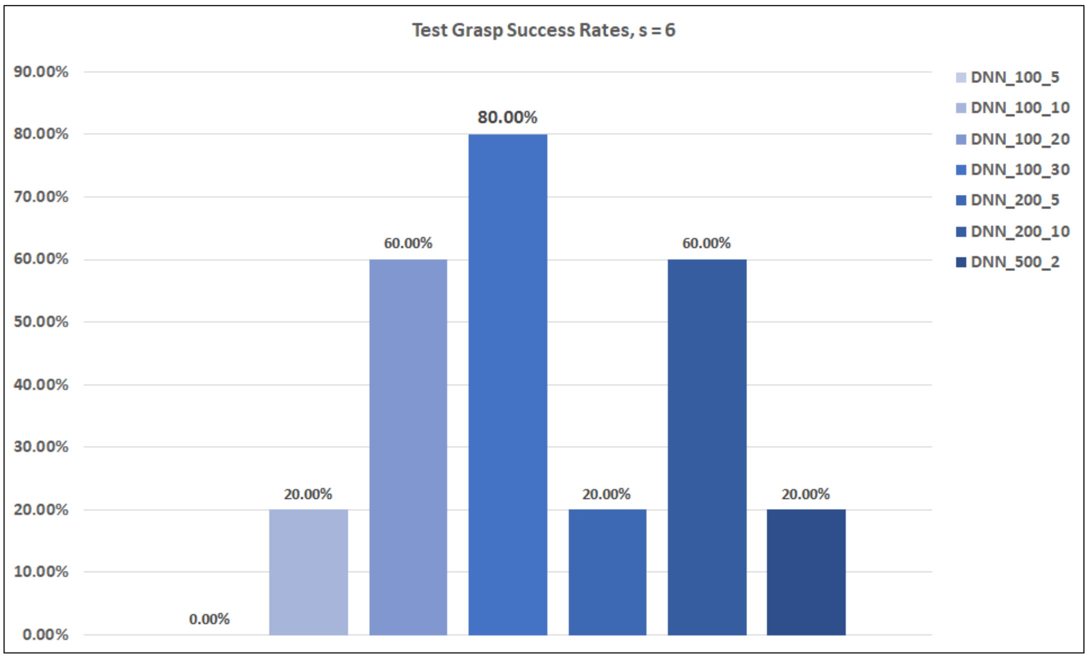


(a)

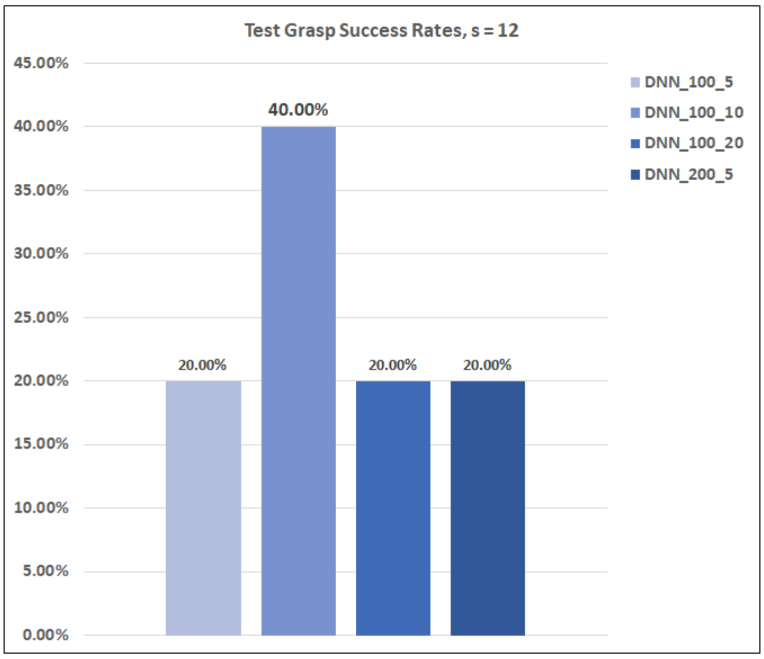


(b)

Figure 23. Performance of Models for s = 6 and s = 12

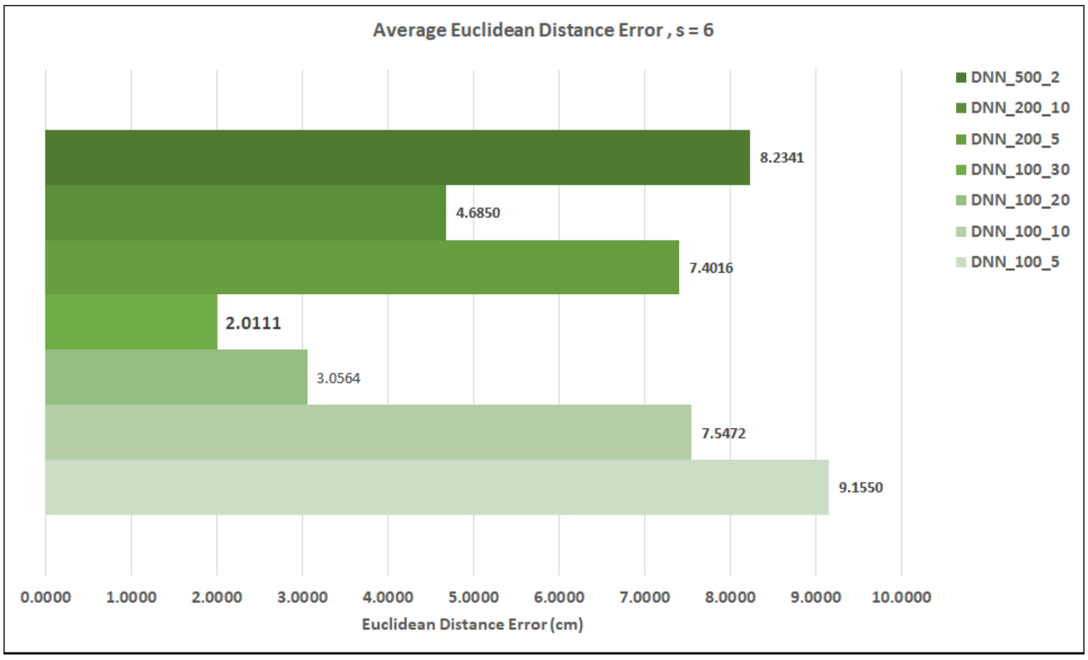


(a)

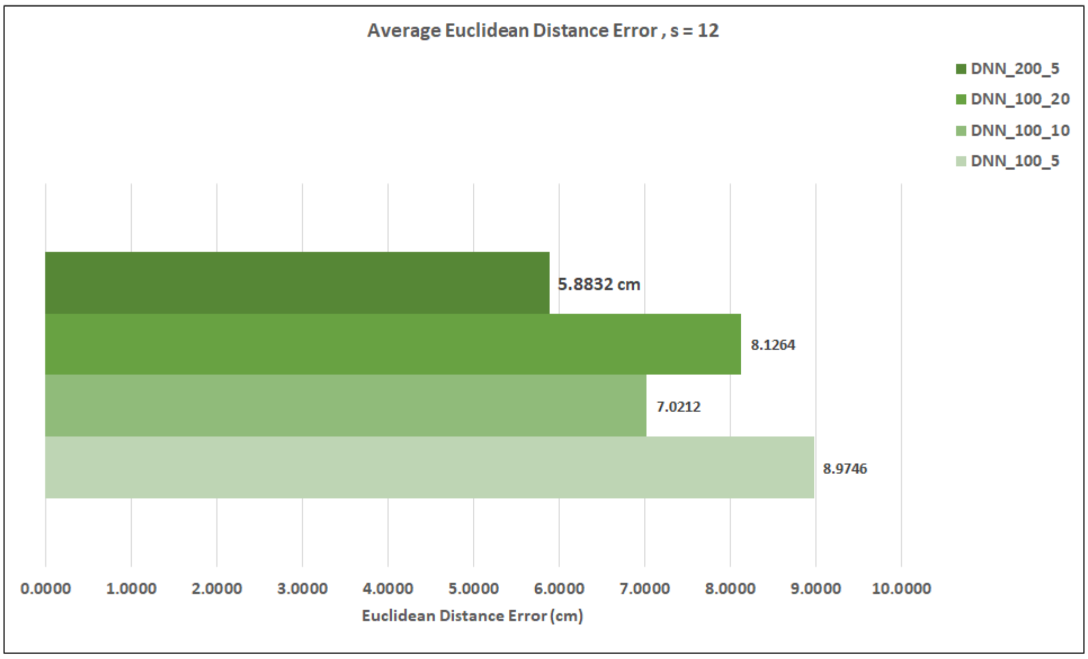


(b)

Figure 24. Success Rates for Models on Test Grasps



(a)



(b)

Figure 25. Average Euclidean Distance Error

CHAPTER 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In Chapter 3, we have discussed how we can enable the robot to detect, and localize the target object placed on the work surface using OpenCV. This information can be then used to calculate the position of the target in the field of view of the Baxter.

In Chapter 4, we saw how the visual position of the target was fed into various Deep Learning Models, which were then used to locate the target in Cartesian Frame of the Baxter. The results obtained from this section show us that Deep Neural Networks are indeed quite efficient in mapping highly nonlinear transformations with a high level of generalization. These models, coupled with an Inverse Kinematics solver enabled a predetermined approach to the target, rather than a dynamic grasp approach.

The approach discussed here was also intended to be expandable to similar robotic systems, with very little modification in the execution pipeline. The computer vision subsystem, along with the Frame Transformation subsystem help us achieve successful grasps by the Baxter.

6.2 Future Work

This experiment was a means to explore how Deep Learning can be used to replace traditional semi-analytical based execution pipelines for segments of robotic tasks, such as pick-and-place. Future work in this domain will enable us to use more complex approaches, similar to the one used here, to map three-dimensional Cartesian spaces using Deep Learning, instead of planes, which will open up a range of tasks that can be accomplished with increased efficiency. We could also look into approaches using Convolutional Neural Network based models, to further extend the capabilities of the Baxter.

CITED LITERATURE

1. Levine, S., Pastor, P., Krizhevsky, A., and Quillen, D.: Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. Google, 2015.

2. Rethink robotics Inc: Baxter robot. `http://sdk.rethinkrobotics.com/wiki/`.

3. Huang, Y., Zhang, X., C. X., and Ota, J.: Vision-guided peg-in-hole assembly by baxter robot. Advances in Mechanical Engineering, 9(12):1–9, 2017.

4. Chen, A. and Wang, K.: Computer vision based chess playing capabilities for the baxter humanoid robot. Proceedings of International Conference on Control, Automation and Robotics, IEEE, 2016.

5. Gaurav, S.: Goal-Predictive Robotic Teleoperation using Predictive Filtering and Goal Change Modeling. Master's thesis, University of Illinois at Chicago, Chicago, 2017.

6. ROS.org: About ROS. `http://www.ros.org/about-ros/`.

7. Martinez, A. and Fernandez, E.: Learning ROS for Robotics Programming. Packt Publishing Ltd, 2013.

8. Ju, Z., Yang, C., and Ma, H.: Kinematics modeling and experimental verification of baxter robot. Proceedings of Chinese Control Conference, IEEE, 2004.

9. Culjak, I., Abram, A., Pribanic, T., Dzapo, H., and Cifrek, M.: A brief introduction to opencv. Proceeding of International Convention on Information and Communication Technology, Electronics and Microelectronics, ICT, 2012.

10. OpenCV: OpenCV Docs. `https://docs.opencv.org`.

11. Goodfellow, I., B. Y. and Courville, A.: Deep Learning. MIT Press, 2016. `http://www.deeplearningbook.org`.

12. Maas, L. A., Hannun, Y. N., and Ng, A. Y.: Rectifier nonlinearities improve neural network acoustic models. Proceedings of International Conference on Machine Learning, 2013.

13. ROS: CV Bridge for ROS. http://wiki.ros.org/cv_bridge.

14. Wikipedia: RamerDouglasPeucker algorithm. https://en.wikipedia.org/wiki/RamerDouglasPeucker_algorithm.

15. Kingma, P. D. and Ba, J. L.: Adam: A method for stochastic optimization. Proceeding of International Conference on Learning Representations, 2015.

16. Harada, K., Kajita, S., Saito, H., Morisawa, M., Kanehiro, F., Fujiwara, K., Kaneko, K., and Hirukawa, H.: A humanoid robot carrying a heavy object. Proceedings of International Conference on Robotics and Automation, IEEE, 2005.

17. Loy, G. and Barnes, N.: Fast shape-based road sign detection for a driver assistance system. Proceedings of International Conference on Intelligent Robots and Systems, IEEE, 2004.

18. Cashbaugh, J. and Kitts, C.: Automatic calculation of a transformation matrix between two frames. IEEEAccess, 6:9614–9622, 2004.

19. Bradski, G. and Kaehler, A.: Learning OpenCV. OReilly, 2008.

20. Wkikipedia: Autonomous Robots. https://en.wikipedia.org/wiki/Autonomous_robot.

VITA

NAME:	Tejas Seshadri Sarma
EDUCATION:	M.S., Computer Science, University of Illinois at Chicago, Chicago, IL, 2019
	B.E., Electronics Engineering, University of Mumbai, Mumbai, India, 2017
EXPERIENCE:	University of Illinois at Chicago, Chicago, IL, Graduate Teaching Assistant, Feb 2018 - August 2018
	Daemo, Stanford CrowdResearch, Stanford University, Undergraduate Student Researcher, May 2015 - Dec 2017
	Wegilant, Android Developer Intern, Dec 2014 - Jan 2015