Imperial College London

MEng Individual Project

Department of Computing

Imperial College of Science, Technology and Medicine

# Grasp Quality Deep Neural Networks for Robotic Object Grasping

*Author:*
Silvia Sapora

*Supervisor:*
Edward Johns

*Second Marker:*
Antoine Cully

June 17, 2019

Submitted in partial fulfillment of the requirements for the MEng Computing of Imperial College London

**Abstract**

In the last few years, researchers have started exploring the potential of deep learning methods applied to robotic grasping and manipulation tasks. Despite the many advantages of this learning technique, the generation of vast amounts of domain-specific data remains one of the main challenges for this field of research. To help reduce the data collection time, Dex-Net was designed to generate synthetic depth images, robot parallel-jaw grasps and metrics of grasp robustness based on physics for thousands of 3D object models. The resulting dataset was used to learn Grasp Quality Convolutional Neural Networks (GQ-CNN) models able to predict the probability of success of candidate parallel-jaw grasps on objects from depth images. Building on top of the Dex-Net project, we generated our own datasets to train multiple GQ-CNNs, exploring new architectures and data generation methods.

Although extremely accurate, the GQ-CNN models trained by Dex-Net still rely on discrete sampling of grasp candidates. We present a new Grasp Quality Network able to predict the probability of success and grasp angle for each pixel of a depth image, avoiding the need for grasp sampling. To the best of our knowledge, this is the first network of this kind trained on a synthetic dataset. Thanks to our novel training methods, the model was able to extrapolate the necessary information from incomplete grasp quality images and correctly learn how to predict grasp quality from a depth image. Additionally, our Grasp Quality Network is able to locate a robust grasp 100 times faster than the GQ-CNN proposed in the Dex-Net 2.0 research paper. Finally, we compared and analyzed the performances of our Grasp Quality Neural Networks both in simulation, using our own testing and evaluation framework, and in the real world.

**Acknowledgements**

I would like to thank:

- My supervisor, Dr. Edward Johns, for all the time he spent on this project, and his invaluable guidance and advice.
- My family for their unconditional love and support.
- My best friends Betta, Bianca, Eleonora, Francesca and Sofia for their constant support and encouragement.
- All my friends, for sharing with me the most fun and challenging moments of the degree.

# Contents


**1 Introduction** **1**
- 1.1 Objectives . . . 2
- 1.2 Challenges . . . 3
- 1.3 Contributions . . . 4
- 1.4 Outline of Contents . . . 4

**2 Preliminaries** **6**
- 2.1 Deep Learning . . . 6
- 2.2 Neural Networks . . . 7
- 2.3 Convolutional Neural Networks . . . 8
  - 2.3.1 The Convolution Operation . . . 9
  - 2.3.2 Pooling . . . 10
- 2.4 Training Neural Networks . . . 11
  - 2.4.1 Gradient Descent . . . 11
  - 2.4.2 Back-Propagation . . . 12
- 2.5 Coordinate Frames and Notations . . . 12
- 2.6 Cameras . . . 13
- 2.7 Inverse Kinematics . . . 14

**3 Background** **15**
- 3.1 Grasp Planning . . . 15
- 3.2 Grasps Evaluation . . . 16
- 3.3 Grasp Representation . . . 16
- 3.4 Motion Planning and Grasp Execution . . . 17
  - 3.4.1 End-to-End Motion Planning . . . 17
- 3.5 Datasets . . . 18
- 3.6 Domain Adaptation and Simulated Data . . . 19
- 3.7 CNNs for Grasp Detection . . . 19
- 3.8 Bridging the Reality Gap . . . 20
- 3.9 Dex-Net . . . 21
  - 3.9.1 Intro . . . 21
  - 3.9.2 Dex-Net Research . . . 22
  - 3.9.3 Dex-Net Codebase . . . 26
  - 3.9.4 GQCNN Codebase . . . 28

**4 Extending Dex-Net** **32**
- 4.1 Installation Issues . . . 32
- 4.2 Calculating Grasps . . . 33
  - 4.2.1 Rescaling meshes . . . 35

4.2.2 Calculating Grasps Quality . . . 36
4.2.3 Obtaining Grasps Parallel to Surface . . . 37
4.2.4 Checking Grasp Validity . . . 38
4.3 The Database . . . 39
4.3.1 Grasp Calculation Database . . . 40
4.3.2 GQ-CNN Dataset . . . 40
4.4 Generating Depth Images . . . 43
4.4.1 Sampling the camera pose . . . 45
4.4.2 Rendering Images with Meshrender . . . 46
4.4.3 Rendering Images with V-REP . . . 47
4.4.4 Aligning Images with Grasps . . . 47
4.5 Generating RGB images . . . 49
4.6 Procedurally Generated objects . . . 50
4.7 GQ-CNNs Training . . . 52

**5 Testing and Evaluation on Dex-Net 54**
5.1 Evaluation metrics . . . 55
5.2 V-REP . . . 55
5.3 Environment Set-Up . . . 56
5.4 Remote API . . . 56
5.5 Physics Simulator Choice . . . 56
5.6 Vision Sensors . . . 57
5.7 Robotic Gripper . . . 58
5.8 Collision Checking . . . 58
5.9 Inverse Kinematics . . . 60
5.10 Evaluation Data . . . 60
5.11 Evaluation Script . . . 62

**6 Optimizing Dex-Net 64**
6.1 Training on subsets of Dex-Net . . . 64
6.2 Different quality metrics . . . 65
6.3 Higher Resolution Images . . . 67
6.4 GQ-CNNs Architecture . . . 67
6.4.1 Original Dex-Net 2.0 . . . 67
6.4.2 96×96 Dex-Net 2.0 . . . 68
6.5 Hyperparameter Search Results . . . 69

**7 Beyond Dex-Net 74**
7.1 Grasp Quality Auto-Encoder(GQ-AE) . . . 74
7.2 Data Generation . . . 76
7.2.1 Challenges . . . 76
7.2.2 Possible Solutions . . . 76
7.3 GQ-AE Loss Function . . . 78
7.4 Pytorch . . . 79
7.5 GQ-AE architecture . . . 79
7.6 Training . . . 79

7.6.1 Grasp Quality Images Datasets . . . . . 79
7.6.2 Grasp Angle Images Datasets . . . . . 84
7.7 Finetuning . . . . . 84
7.8 Evaluation . . . . . 85

**8 From Simulation to Real World** **88**
8.1 Intera SDK . . . . . 88
8.2 Cameras and Lighting . . . . . 89
8.3 Coordinate Frames . . . . . 90
8.4 Gripper . . . . . 91
8.5 Objects . . . . . 92
8.6 Evaluation . . . . . 92

**9 Evaluation** **96**
9.1 Original GQ-CNN . . . . . 96
9.2 GQ-CNN increased resolution . . . . . 96
9.3 GQ-CNN 3 million parameters . . . . . 97
9.4 GQ-AE . . . . . 97

**10 Conclusions and Future Work** **98**
10.1 Conclusions . . . . . 98
10.1.1 Lessons Learned . . . . . 98
10.2 Future Work . . . . . 99

**Bibliography** **100**

**A Pseudocode** **106**
A.1 Camera Randomization Parameters . . . . . 106
A.2 Obtaining Camera Image and Info with ROS . . . . . 106
A.3 Antipodal Grasp Sampling . . . . . 107

# 1 Introduction

Grasping is something that comes naturally to humans, people instinctively know how to grasp an object, even if they have never seen it before. But for robots, this is a very challenging task that involves perception, planning, and control[23, 53]. Even a simple task such as picking up a bottle requires multiple steps in order to be completed successfully. The robot needs to use their perception abilities(such as a camera or laser sensors) to identify the location of the bottle, calculate the best position to grasp it, plan a trajectory for the robotic arm to get into the desired position, close the gripper, lift, and then verify if the grasp was successful.

As robots are taking on a more important role in factories as well as our homes, the ability to grasp objects fast, accurately and reliably has become a fundamental skill for them to have. Particularly, the demand for more general-purpose grasping has also increased. Traditionally, these task-specific algorithms have been hard-coded to fit very specific needs and situations. While this approach is usually fast, it has significant limitations when we try to generalize it to fit other circumstances: it cannot be reused in different environments and cannot react to unexpected situations(such as novel objects). The successful outcome of the task is heavily restricted to situations predicted and accounted for by the programmer. These algorithms are also time-consuming to write, and any error in one of the steps could cascade to the others, causing the grasp to fail. If, for example, the object is not located correctly, or our robot collides with the object while trying to grasp it, we will probably fail to obtain a successful grasp(or at least obtain a less stable one). Robotic grasping currently performs well below human object grasping benchmarks[23], but it keeps being improved.

Recently, Deep Learning techniques have enabled significant advancements in robotic vision, natural language processing, and automated driving applications. Thanks to the successful results in these fields, researchers have started exploring the potential of deep learning methods in robotic applications. Deep Learning is a branch of machine learning, and it involves the use of an artificial neural network inspired by the biological nervous system. Once the data is fed to this network, a series of parallel and simultaneous mathematical operations is applied. The goal is to learn a set of rules that will later be used for decision making. When these techniques are applied to a robotic system, they enable robots to autonomously perform tasks that come naturally to humans. Thanks to deep learning, we would be able to replace the pipeline of steps the robot has to follow, enabling it to learn how to perform actions based purely on sensor data, such as depth images coming from cameras or gripper joints positions. This helps in moving towards a more general approach to grasping, rather than many hard-coded, task-specific components tailored to a certain environment. Deep learning models are commonly used in classification and detection problems, but there is a strong interest to apply them to new domains.

The downside of Deep Learning is that it requires vast amounts of training data

to achieve good performance. One possible way of obtaining such data is through human labelling or months of execution time on physical robots. In both cases, there are considerable costs in both time and money, and recent studies suggest that the performance of grasping systems might be strongly influenced by the amount of data available [24]. Cheaper alternatives include: plan grasps using physics-based analysis such as caging, grasp wrench space (GWS) analysis, robust GWS analysis or simulation, where thousands of robots can work in parallel. Moreover, all these methods can be computed using Cloud Computing, making them much faster than a real-time robot, which further speeds up the data gathering process. However, these methods use a different perception system that estimates properties such as object shape or pose either perfectly or according to known Gaussian distributions. This is prone to errors, may not generalize well to new objects, and can be slow to match point clouds to known objects during execution.

Recent results suggest that it is possible to grasp a variety of isolated objects with high precision using Convolutional Neural Networks (CNNs) trained on synthetic data. Even though using synthetic data offers many advantages, as explained above, one more thing we have to consider the difficulty of transferring simulated experience into the real world. This is often referred to as the "reality gap". The reality gap is a subtle but important discrepancy between reality and simulation that prevents simulated experience from directly transferring into effective real world performance[8]. Recent research[20] shows it is possible to train a robot exclusively in simulation and then apply the model in the real world without the need for further training or adaptation, successfully crossing the reality gap. In this project, we will implement a grasping system using Dex-Net(Dexterity Network). Dex-Net is a research project including code, datasets, and algorithms for generating datasets of synthetic point clouds, robot parallel-jaw grasps and metrics of grasp robustness based on physics, for thousands of 3D object models to train machine learning-based methods to plan robot grasps. The data generated by Dex-Net can be used to train Grasp Quality Convolutional Neural Networks, networks trained to be able to rank grasps and be able to distinguish stable from less stable grasps. We aim to replicate the results achieved by Dex-Net and described in the Dex-Net research papers, as well as gaining a deep understanding of their codebase. We plan to recreate the Dex-Net pipeline: use the codebase to calculate possible grasps on a 3D mesh, simulate both RGB and depth images for each possible grasp, and then train a Grasp Quality Convolutional Neural Network that we will evaluate both in simulation and on a real robot. After this, we will explore different methods to improve Dex-Net's performance on known and unknown objects, as well as exploring new approaches to building faster, more reliable Grasp Quality Networks.

## 1.1 Objectives

Our goal was to use Dex-Net's findings and codebase to train a Grasp Quality Convolutional Neural Network(GQ-CNN) to be able to reliably recognize robust grasps. After replicating Dex-Net results, we would use the gained knowledge and experience to improve on Dex-Net's findings. Our main goals for this project are summarized

here:

- Understand the Dex-Net research paper and codebase while looking into other similar methods of data gathering for GQ-CNN training.
- Use the Dex-Net codebase to generate robust grasps for any given mesh, then evaluate those grasps in simulation to confirm their validity.
- Recreate the Dex-Net pipeline to generate stable grasps, generate depth images for each grasps and train a GQ-CNN with the gathered data, trying to replicate Dex-Net's results.
- Develop a testing framework to reliably evaluate different GQ-CNNs and grasping policies.
- Explore the Dex-Net and GQ-CNN codebase to find ways of improving the network's performance.
- Explore entirely new architectures, methods for data generation and GQ-CNN training.
- Evaluate and compare the different GQ-CNNs, data creation methods and policies both in simulation and in the real world.

## 1.2 Challenges

Starting out, we expected that working with the Dex-Net codebase was going to be quite straightforward. Unfortunately, due to many bugs and installation issues, this was not the case. A number of inconsistencies between the research papers and the codebase further contributed to slowing down our progress, especially in the beginning. Overall, the biggest challenges we encountered during the course of the project were:

- Inconsistencies between the Dex-Net research paper and the codebase.
- Lack of documentation for the codebase, coupled with misleading, confusing or outdated comments and variable names.
- Incomplete installation script, outdated dependencies and library conflicts. Bugs not only in the Dex-Net codebase but in its library dependencies as well.
- Long data generation and network training times.
- Huge amounts of data necessary for training. Some datasets measured over 80GB. With such large amounts of data, even transferring data from one machine to another became a challenge.
- Constantly changing codebase. Since we were working with the current state-of-the-art of GQ-CNN training methods, it meant we had to deal with huge updates to the codebase at multiple times through our project.

## 1.3 Contributions

- Recreate Dex-Net's pipeline and replicate their results in both simulation and real world.
- Confirm the validity of grasps generated through Dex-Net to train a Grasp Quality Neural Network.
- Contribute to the Dex-Net codebase with bug fixes, installation script updates and solving library clashes.
- Create a testing framework for evaluating and comparing GQ-CNNs and policies. This was done in simulation, using V-REP and the model of a Sawyer robotic arm.
- Create many datasets for grasp quality training. The datasets published by the Dex-Net researchers are limited to $32 \times 32$ pixels depth images, with grasp collisions calculated using a Yumi Robotic gripper. We generated more datasets using a Baxter parallel gripper for collision checking. We also added RGB images to our training data and explored different image resolutions.
- Explore different variations of the original Dex-Net architecture and trained a network with 84% fewer parameters and similar performance.
- Develop a novel technique to train Neural Networks on incomplete data, allowing them to extrapolate and only learn from relevant data. The network we trained was able to extrapolate relevant information from thousands of partial data images and correctly merge the knowledge gained from each one to accurately predict complete grasp quality images.
- Generate datasets to train a novel Grasp Quality Auto-Encoder network, able to locate reliable grasps on a depth image 100 times faster(0.04 seconds vs 4 seconds) than the original GQ-CNN proposed by the Dex-Net 2.0 publication.

## 1.4 Outline of Contents

This section will give a brief overview of the structure of the report together with a brief summary. Given the many components of our project, the evaluation of each one usually follows in the same chapter. For all our components, we use the same testing framework introduced and explained in detail in Chapter 5.

- **Chapter 2** gives an introduction to Neural Networks, convolutions, coordinate frames, camera models and Inverse Kinematics.
- **Chapter 3** explains the challenges of robotic grasping and discusses how other researchers have tried to achieve fast and reliable grasping. This chapter also includes a brief overview of some of the most popular datasets and methods for domain adaptation, as one of our goals is to evaluate our grasp quality

neural networks in real-world tasks. We will also discuss Dex-Net's approach to robotic grasping, their dataset creation pipeline, methodologies, and the physics behind their grasp creation algorithm. Finally, we will briefly summarize the structure of the Dex-Net codebase.

- **Chapter 4**. From here on, we begin talking about our work, starting with the improvements and extensions we made to Dex-Net. This includes bug fixes, support for RGB images and more in-depth explanations of some parts of the codebase either overlooked or not explained clearly in the original research paper, while also highlighting any inconsistencies we found.
- **Chapter 5** introduces our testing framework to evaluate GQ-CNNs. In this chapter, we also explain what metrics we decided to use to compare our models and the data we evaluated our networks on.
- **Chapter 6** talks about the optimizations we made to the Grasp Quality Neural Network proposed by Dex-Net, whilst creating different networks trained on different amounts and types of data, and comparing between them.
- **Chapter 7** introduces our new Grasp Quality Auto-Encoder, talks about our data creation process, and compares it to the Dex-Net Grasp Quality Convolutional Neural Network, highlighting the pros and cons of each approach.
- **Chapter 8** talks about how we used the GQ-CNN we trained to execute grasps on a real robot, a Sawyer robotic arm. We also discuss the strengths and weaknesses of our application.
- **Chapter 9** summarizes strengths and weaknesses of evaluations explained in detail in Chapters 6, 7 and 8 using the testing framework and evaluation strategies introduced in 5.
- **Chapter 10** includes a conclusion, summarizes our findings, shares what we learned and discusses some ideas we would like to explore in the future.

# 2 Preliminaries

## 2.1 Deep Learning

Deep structured learning, or more commonly called deep learning, has emerged as a new area of machine learning research. During the past several years, the techniques developed from deep learning research have had a profound impact on a wide range of signal and information processing work, especially in the field of image and object recognition, or more recently speech recognition. In 2011, a fast implementation of CNNs with max-pooling achieved superhuman performance in a visual pattern recognition contest for the first time[18].

There exist several high-level description and definitions of deep learning, but they all share the following characteristics:

- Deep learning is part of a broader family of machine learning methods based on learning representations.
- It is based on algorithms for learning multiple levels of representation in order to model complex relationship among data.
- It exploits many layers of non-linear information processing for supervised or unsupervised feature extraction and transformation, and for pattern analysis and classification
- High-level features and concepts are thus defined in terms of lower-level ones, and such a hierarchy of feature is called a deep architecture.
- Typically uses artificial neural networks. The levels in these learned models correspond to distinct levels of concepts.

Deep learning is in the intersections among the research areas of neural networks, artificial intelligence, graphical modeling, optimization, pattern recognition, and signal processing[15].

The main reasons for the popularity of deep learning today are the drastically increased chip processing abilities(e.g. general purpose graphical processing units or GPGPUs), the significantly increased size of data used for training, and the recent advances in machine learning[14].

These advances have enabled deep learning methods to effectively exploit complex, non-linear functions, to learn distributed and hierarchical feature representations, and to make effective use of both labeled and unlabeled data[15].

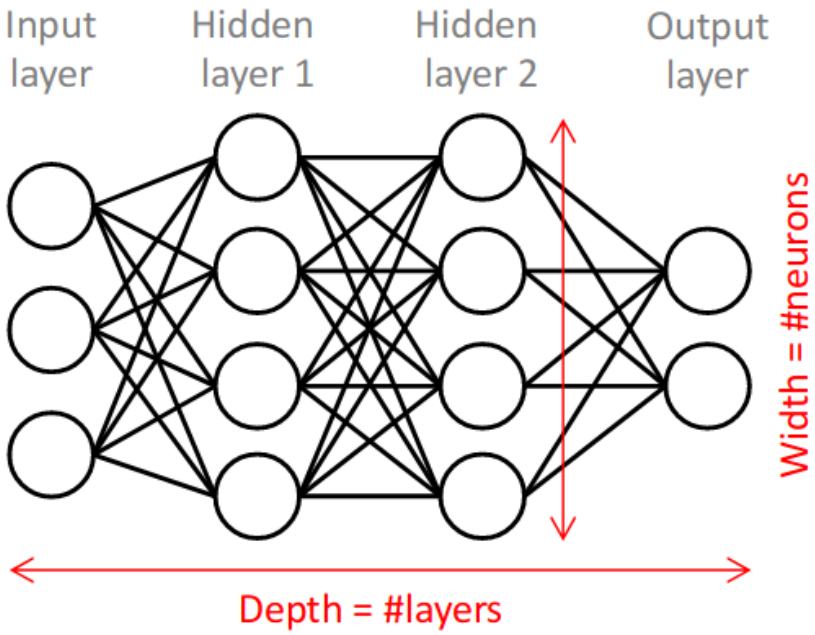


**Figure 2.1:** A simple neural network

## 2.2 Neural Networks

A standard artificial neural network (ANN or NN) consists of many simple, connected units called artificial neurons, which vaguely model the neurons in a biological brain. Each connection between two neurons(a synapse in a biological brain, an edge in the graph of an ANN) can transmit a signal from one artificial neuron to another.

In the most common ANN implementation, the signal transmitted between neurons is a real number, and the output of each artificial neuron is computed by some non-linear function(called activation function) of the sum of the weights of its inputs. The activation function can be implemented so that the signal is only sent from one neuron to the next if it crosses a given threshold[19]. Learning is about finding weights that make the ANN manifest desired behavior, such as, in our specific case, grasping an object. The weight increases or decreases the strength of the signal at a connection. Typically, artificial neurons are aggregated into layers. Different layers may perform different kinds of transformations on their input. Input neurons receive complex data inputs, other neurons get activated through weighted connections from previously active neurons. The output neurons may influence the environment by triggering actions. Depending on the problem and how the neurons are connected, long chains of computational stages(layers) may be required, where each stage(layer) transforms (often in a non-linear way) its input.

Let's take the example of face recognition from an image: the data will be transformed by each layer of the ANN into a more and more abstract representation of the previous layer(or the raw input, as a matrix of pixels, in the first layer). The first layer may abstract the pixels to encode edges, the second layer may indicate and

encode positions of these edges, the third layer may recognize facial features such as nose and eyes, and the fourth layer may recognize that the image contains a face. It's important to note that a deep learning process can learn which features to optimally place in which level on its own. Also, there is no need to somehow hard-code the knowledge that, for example, faces have eyes, as the ANN would learn so on its own based on the data used during training.

**Activation Function** The activation function of a node defines the function that is applied to the input of the neuron in order to obtain a certain output. In modern neural networks,the default recommendation is to use Rectified Linear Unit(**ReLU**) activation function[19]. This function returns 0 if it receives any negative input, but for any positive value x, it returns that value back. So it can be written as $f(x) = max(0, x)$.

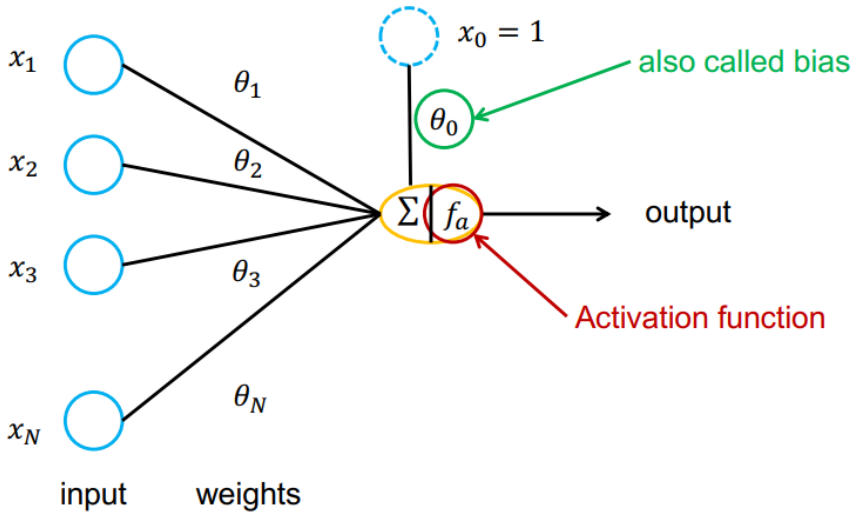


**Figure 2.2:** Artificial Neuron Model

## 2.3 Convolutional Neural Networks

Convolutional networks, also known as convolutional neural networks, or CNNs, are a specialized kind of neural network for processing data that has a known grid-like topology. An example is image data, which can be thought of as a 2D grid of pixels. This type of network uses a mathematical operation called convolution, a specialized kind of linear operation. Convolutional networks are simply neural networks that use convolution instead of general matrix multiplication in at least one of their layers[19].

### 2.3.1 The Convolution Operation

In the most general form, convolution is an operation on two functions of a real-valued argument. Suppose we are tracking the location of a robotic gripper with a laser sensor. Our laser sensor provides a single output $x(t)$, the position of the gripper at time $t$. Now, suppose the laser sensor is noisy. To obtain a less noisy estimate of our gripper's position, we would like to average several measurements. Of course, more recent measurements will be more relevant, so we will want this to be a weighted average that gives more weight to recent measurements. We can do this with a weighting function $w(a)$, where $a$ is the age of a measurement. If we apply such weighted average operation at every moment, we obtain a new function $s$ providing a smoothed estimate of the position of our gripper:

$$s(t) = \int x(a)w(t-a)da$$

This operation is called convolution. The convolution operation is typically denoted with an asterisk:

$$s(t) = (x * w)da$$

In general, convolution is defined for any functions for which the above integral is defined and may be used for other purposes besides taking weighted averages. In convolutional networks we have a specific terminology:

- Input: The first argument(in our example, the function x)
- Kernel: The second argument(in our example, the function w)
- Feature map: The output

In the previous example, the idea of a laser sensor that can provide measurements at every instant is not realistic. Usually, data used in a program will be discretized. Therefore, we will assume our laser provides data once per second. We can also define a discrete convolution:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

In machine learning applications, the input is usually a multidimensional array of data, and the kernel is usually a multidimensional array of parameters that are adapted by the learning algorithm. As we assume the input and kernel arrays will be zero everywhere but for the finite set of points which we specify, we can implement the infinite summation as a summation over a finite number of array elements. Finally, we often use convolutions over more than one axis at a time. For example, if we use a two-dimensional image as our input, we probably also want to use a two-dimensional kernel K:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Discrete convolution can be viewed as multiplication by a matrix, but the matrix has several entries constrained to be equal to other entries.

Traditional neural network layers use matrix multiplication by a matrix of parameters with a separate parameter describing the interaction between each input unit and each output unit. This means that every output unit interacts with every input unit. Convolutional networks, however, typically have sparse interactions (also referred to as sparse connectivity or sparse weights). This is accomplished by making the kernel smaller than the input. For example, when processing an image, the input image might have thousands or millions of pixels, but we can detect small, meaningful features such as edges with kernels that occupy only tens or hundreds of pixels. This means that we need to store fewer parameters, which both reduces the memory requirements of the model and improves its statistical efficiency. It also means computing the output requires fewer operations. This results in a large improvement in the efficiency of the algorithm and training of the network.

Convolutional networks provide a way to specialize neural networks to work with data that has a clear grid-structured topology and to scale such models to very large size. This approach has been the most successful on a two-dimensional image topology. [19]

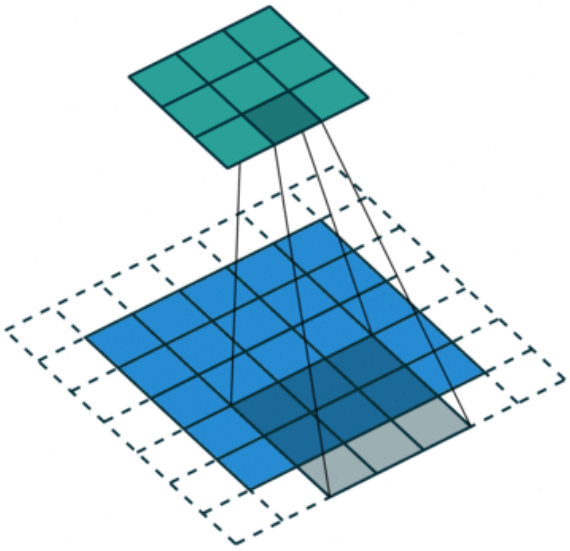

**Figure 2.3:** Example of convolution operation with stride equal to 2[52]

### 2.3.2 Pooling

Convolutional networks may include local or global pooling layers, which combine the outputs of groups of neurons in one layer into a single neuron in the next layer. For example, the max-pooling operation reports the maximum output within a neighborhood. Other popular pooling functions include the average, the $L^2$ norm, or a weighted average based on the distance from the central pixel. In all cases, pooling

helps to make the representation approximately invariant to small translations of the input. For example, when determining whether an image contains a face, we do not need to know the location of the eyes with high precision, we only care about the fact that they are present. Because pooling summarizes the responses over a whole neighborhood, it is possible to have fewer pooling units than detector units. This improves the computational efficiency of the network because the layer has roughly $k$ times fewer inputs to process. When the number of parameters in the next layer is a function of its input size(such as when the layer is fully connected and based on matrix multiplication), this reduction in the input size can also result in reduced memory requirements for storing parameters[19]. Reduction in the number of parameters also reduced the search space of the optimization. Fewer parameters help prevent overfitting and result in faster and more efficient training.

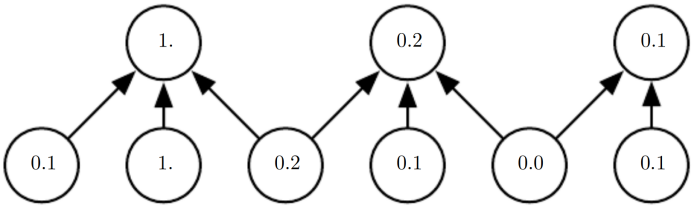


**Figure 2.4:** Pooling with down-sampling. Here max pooling is used with a pool width of three. This reduces the representation size by a factor of two, which reduces the computational on the next layer[19]

## 2.4 Training Neural Networks

Given a target function $y = f^*(x)$, our model represents a function $y = f(x;\theta)$, and our learning algorithm will adapt the parameters $\theta$ to make $f$ as similar as possible to $f$. Training a Neural Network aims to minimize the loss function $\mathcal{L}$, a representation of the difference between the current function of our model $y = f(x;\theta)$ and the target function $y = f^*(x)$. A commonly used loss function is Mean Squared Error(MSE), defined as follows:

$$\mathcal{L}(\theta) = \frac{1}{n}\sum_{i=0}^{n}(f^*(x_i) - f(x_i;\theta))$$

### 2.4.1 Gradient Descent

Unfortunately, the non-linearity of a neural network causes most loss functions to become non-convex. This means Neural Networks are usually trained using iterative, gradient-based optimizers that drive the cost function to a minimum[19]. Intuitively, the gradient descent method works similarly to a lost climber trying to go back to the bottom of a mountain. He starts from a random point(random guess of parameters), then he looks around and takes a step in the direction with the steepest descent. After each step, he reevaluates the direction of steepest descent and takes another

step. He repeats this process until he reaches the bottom of the mountain. In this analogy, the mountain represents our loss function, that we aim to minimize, and at each step we reevaluate it, calculate its gradient, identify in which direction it steps down the most and tweaks our parameters in that direction. We repeat this process over and over until convergence. The parameters update of one gradient descent step is applied using the following formula:

$$\Theta^{k+1} := \Theta^k - \tau \nabla \mathcal{L}(\Theta)$$

### 2.4.2 Back-Propagation

In multi-layer neural networks, the implementation of gradient descent is achieved through back-propagation[11]. Back-propagation is a method of updating the weights of connections between neurons proportionally to the gradient of the networks loss function. A generic optimization algorithm can be described as follows:

1. Forward pass: Make a prediction and measure the error
2. Backward pass: Go through each layer in reverse to measure the error contribution from each connection (gradient calculation step)
3. Adjust connection weights to reduce the error (gradient descent step)

## 2.5 Coordinate Frames and Notations

A position vector denoted as ${}_W\mathbf{r}_S$ indicates the origin of $\underrightarrow{\mathcal{F}}_S$(coordinate frame S) represented in $\underrightarrow{\mathcal{F}}_W$(coordinate frame W). Example: ${}_W\mathbf{r}_S$ could indicate the position of a sensor S with respect to $\underrightarrow{\mathcal{F}}_W$ (the world coordinate frame).

A rotation matrix from $\underrightarrow{\mathcal{F}}_S$ to $\underrightarrow{\mathcal{F}}_W$ will be denoted as $\mathbf{C}_{WS}$. A rotation matrix transforms a vector's coordinate frame representation as follows: ${}_W\mathbf{a} = \mathbf{C}_{WS}\,{}_S\mathbf{a}$. In the 2D case, we have only 1 angle we can change(rotation around the z-axis). A rotation matrix from $\underrightarrow{\mathcal{F}}_A$ to $\underrightarrow{\mathcal{F}}_B$ where $\underrightarrow{\mathcal{F}}_B$ is rotated by $\gamma$ anti-clockwise with respect to $\underrightarrow{\mathcal{F}}_A$ can be written as:

$$\mathbf{C}_{AB} = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) \\ \sin(\gamma) & \cos(\gamma) \end{bmatrix}$$

In the 3D case, we have 3 Degrees of Freedom(DoF)(i.e. rotation around x, y and z axes). Below, the rotation matrices around the x, y and z-axis respectively by angle $\gamma$ are shown.

$$\mathbf{C}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\gamma) & -\sin(\gamma) \\ 0 & \sin(\gamma) & \cos(\gamma) \end{bmatrix} \mathbf{C}_y = \begin{bmatrix} \cos(\gamma) & 0 & \sin(\gamma) \\ 0 & 1 & 0 \\ -\sin(\gamma) & 0 & \cos(\gamma) \end{bmatrix} \mathbf{C}_z = \begin{bmatrix} \cos(\gamma) & -\sin(\gamma) & 0 \\ \sin(\gamma) & \cos(\gamma) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

These rotation matrices are orthonormal, this means that:

$$\mathbf{C}_{WS} = \mathbf{C}_{SW}^{-1} = \mathbf{C}_{SW}^{T}$$

To obtain a rotation around multiple axes is enough to multiply these matrices. A 6 DoF pose consists of position(3 DoF) and orientation(3 DoF). We can write the transformation of a position vector ${}_B\mathbf{r}_P$ (position of P in $\underrightarrow{\mathcal{F}}_B$) to ${}_A\mathbf{r}_P$ (position of P in $\underrightarrow{\mathcal{F}}_A$) as:

$$ {}_A\mathbf{r}_P = \mathbf{C}_{AB}\,{}_B\mathbf{r}_P + {}_A\mathbf{r}_B $$

Or, we can use the homogeneous transformation matrix:

$$ \mathbf{T}_{AB} = \begin{bmatrix} \mathbf{C}_{AB} & {}_A\mathbf{r}_B \\ 0_{1\times 3} & 1 \end{bmatrix} $$

In this case we have to turn our position vector ${}_B\mathbf{r}_P$ into homogeneous coordinates to be able to apply the homogeneous transformation:

$$ {}_A r_P := \begin{bmatrix} {}_A\mathbf{r}_P \\ 1 \end{bmatrix} = \mathbf{T}_{AB} \begin{bmatrix} {}_B\mathbf{r}_P \\ 1 \end{bmatrix} $$

## 2.6 Cameras

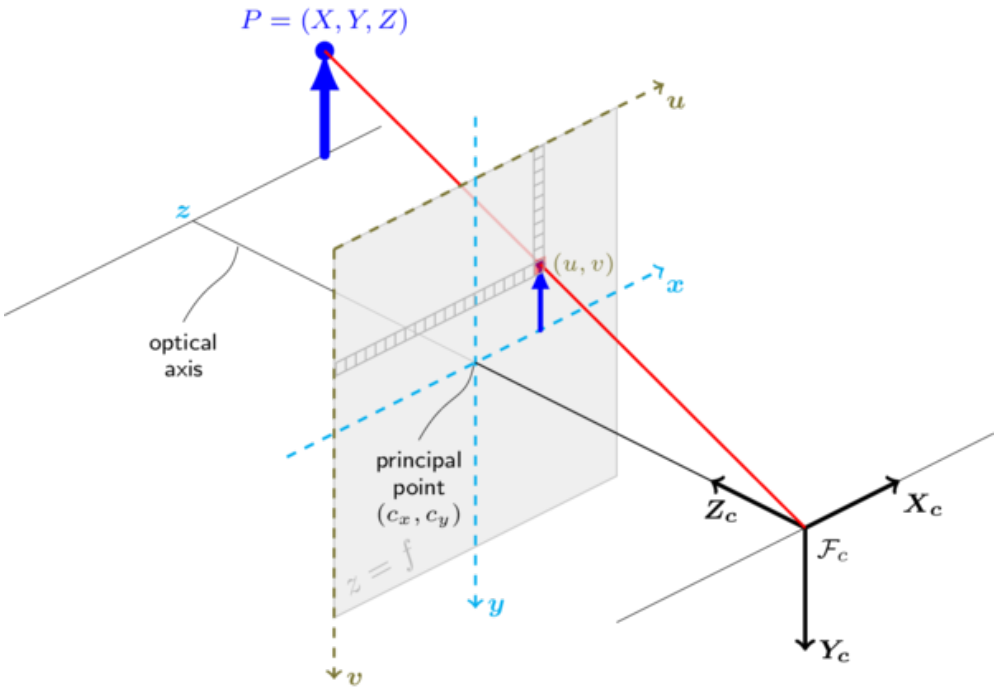


**Figure 2.5:** The pinhole camera model[51]

For the purposes of our project, we don't need to explain in detail the physics of the pinhole camera model, as explaining the mathematics behind it is going to be enough. In this model, the 3D world is projected onto a plane(called projection plane) in front of the camera center. The distance between the camera center and the projection plane is called focal length. The projection of any point in the 3D world onto the projection plane of the camera is obtained applying the formula below:

$$ m' = A[R|t]M' $$

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Where $(X, Y, Z)$ indicate the coordinates of the point in the 3D world coordinate frame and $(u, v)$ are the coordinates of the projection point in pixels. $A$ is the camera matrix, or a matrix of intrinsic parameters(internal and fixed to a particular camera/digitization setup, they define the location and orientation of the camera with respect to the world frame), while $[R|t]$ is the matrix of extrinsic parameters(external to the camera and may change with respect to the world frame, they allow a mapping between camera coordinates and pixel coordinates in the image frame). $(c_x, c_y)$ indicates the center of the image(in pixels) and is sometimes called the principal point. $(f_x, f_y)$ are the focal lengths also expressed in pixel units. It's also important to note that the camera coordinate frame is oriented so that the z-axis points towards the projection place, the x-axis is parallel to it and indicates the horizontal axis of the image, while the y-axis indicates the vertical axis[51].

## 2.7 Inverse Kinematics

Inverse Kinematics(IK) is defined as the use of kinematic equations to determine the joint parameters of a robotic arm so that the end-effector moves to a desired position. While kinematics is related to the motion of points, objects, and systems of objects, it has no consideration for what causes the motion and doesn't consider mass, force or torques. Inverse Kinematics, on the other hand, was initiated in order to solve the problem of moving a redundant kinematic arm with specific degrees of freedom(DoF) to a pre-defined target[2]. In the case of a 2D environment and one joint, the problem is reduced to simple trigonometry. However, as the number of joints and dimensions increase, Inverse Kinematics can become a challenging problem to solve.

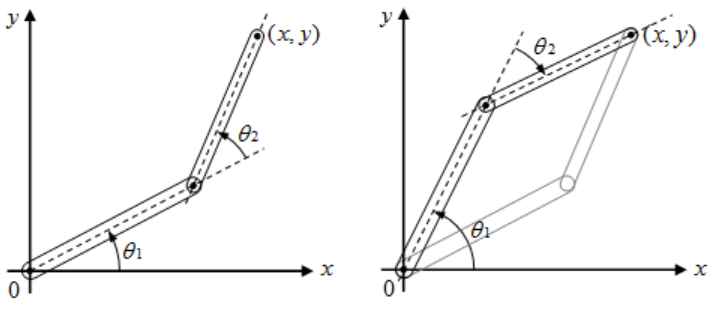


**Figure 2.6:** Inverse Kinematics 2D problem applied to a manipulator with two joints. This particular problem as two possible solutions[47]

# 3 Background

Reliable robotic grasping is challenging due to imprecision in sensing and actuation, which leads to uncertainty about properties such as object shape, pose, material properties, and mass[26]. Successful robotic grasping systems should be able to overcome these obstacles to produce useful results. A robotic grasping implementation has the following phases [23]:

- **Grasp planning**: A visual recognition problem in which the robot uses its sensors to detect graspable objects in its environment. The sensors used for perceiving the robots environment are typically 3D vision systems or RGB-D cameras. The goal is to predict potential grasps from sensor information and map the pixel values to real world coordinates. This is a fundamental step in performing a grasp as the subsequent steps are dependent on the coordinates calculated in this step.
- **Trajectory planning**: An optimal trajectory for the robotic arm is then planned to reach the target grasp position. The calculated trajectory should avoid collision with the object or the surface where the object is resting.
- **Execution**: The planned trajectory for the robotic arm is executed using either an open-loop or a closed loop controller.

## 3.1 Grasp Planning

Grasp planning consists in finding a gripper configuration that maximizes a success(or quality) metric, taking into consideration the object shape and specific environmental factors, such as the pose of the object on a table or an obstacle between the robot and the object. *Robust* grasp planning tackles the same problem but takes into consideration the presence perturbations in the object's properties(e.g. object shape, pose, or mechanical properties such as friction), caused by the imprecision in perception and control[29]. In more detail, grasp planning includes three key tasks:

- **Object Localization**
- **Pose Estimation**
- **Grasp Detection**

Object localization can be achieved using object detection and segmentation methods, while pose estimation includes RGB-based and RGB-D-based methods. Grasp detection includes traditional task-specific, hard-coded methods, and deep learning-based methods, but in general two main categories exist, and they are based on success criteria[27]:

- **analytic methods**[39]: evaluate performance according to physical models such as the ability to resist external wrenches[36] or the ability to constrain the object's motion[48]. These methods typically require knowing the object shape and location exactly. They usually involve pre-computing a database of known 3D objects labeled with grasps and quality metrics. Point clouds are then matched to known 3D objects using visual and geometric similarity and the highest quality grasp is then executed.
- **empirical (or data-driven) methods**[7]: typically use human labels[5] or the ability to lift the object in physical trials[35]. They usually use machine learning techniques to develop models that map from robotic sensor readings to success labels from humans or physical trials.

## 3.2 Grasps Evaluation

The analysis of the Grasp Wrench Space (GWS) has been traditionally used to define different quality measures [40]. GWS describes the force and momentum applied at the point of contact. Among the different quality measures, the computation of the largest minimum resisted wrench (Ferrari-Canny, $Q_\epsilon$) [17] is one of the most widely used. This metric represents the maximum perturbation wrench that a grasp can resist in any direction[38].

According to [40] grasp synthesis algorithms take into account the following basic properties:

- **Disturbance resistance**: a grasp can resist disturbances in any direction when object immobility is ensured, either by finger positions (form closure) or, up to a certain magnitude, by the forces applied by the fingers (force closure[22]). Main problem: determination of contact points on the object boundary.
- **Dexterity**: a grasp is dexterous if the hand can successfully move the object according to the task it has to complete. Main problem: determination of hand configuration.
- **Equilibrium**: a grasp is in equilibrium when the resultant of forces and torques applied on the object (by the fingers and external disturbances) is null. Main problem: determination and control of the proper contact forces.
- **Stability**: a grasp is stable if any error in the object position caused by a disturbance disappears in time after the disturbance vanishes. Main problem: control of restitution forces when the grasp is moved away from equilibrium.

## 3.3 Grasp Representation

Multiple grasp representations have been used throughout the analyzed literature. In earlier works, grasps where simply represented as points on images of real examples or 3D meshes based on simulations. The grasp point was therefore given by

$g = (x, y, z)$. In other words, the grasp is simply defined as a point on a 2D image plane. A major drawback of this method is that it only determines where to grasp an object and it does not specify how wide the gripper has to be open or its orientation. As these parameters can severely affect the successful outcome of a grasp, another popular representation has been proposed: the oriented rectangle representation. This is a seven-dimensional representation containing information about grasping point, grasping orientation, and gripper opening width. In world coordinates, their grasp representation, $G$, is defined as $G = (x, y, z, \alpha, \beta, \gamma, l)$ . Another grasp representation introduced in more recent research is the combined location and orientation representation. In [35], the authors used the simple $G = (x, y, \theta)$ representation and dropped the dimensional parameters of height and width. The literature suggests that any preference between representations is application specific.

## 3.4 Motion Planning and Grasp Execution

Motion planning can be achieved using analytical methods, imitating learning methods(i.e. imitate human grasping behaviour), and reinforcement learning methods. These methods design the path from the robot hand to the grasp points on the target object. Here motion representation is the key problem. Although there exist an infinite number of trajectories from the robotic hand to the target grasp points, many areas could not be reached due to the limitations of the robotic arm. Therefore, the trajectories need to be planned.

One of the most significant challenges in grasp execution is precise robot control. Ju et al [21] have recommended the use of closed-loop control algorithms for accurate grasping. In contrast to an open-loop controller, a closed-loop controller receives continuous feedback from the vision system during the entire grasping task, allowing it to take into account and potentially fix inaccuracies and unexpected movements while moving its robotic arm. The downside of a closed-loop controller is that it can be much slower, drastically affecting the speed of the task. This is because of the additional processing power needed to handle the feedback. [23]

### 3.4.1 End-to-End Motion Planning

End-to-end motion planning is a collection of closed-loop methods where grasping points are not given[16]. A typical functional flow-chart of end-to-end motion planning is illustrated in Figure 3.1. These methods directly accomplish the grasping task after being given an original RGB-D image by using reinforcement learning. The reward function is defined in relation to the state of grasping. Levine et al. [24] proposed a learning-based method for hand-eye coordination in robotic grasping from monocular images. They utilized the grasp attempts as the reward function and trained a large convolutional neural network to predict the probability that task-space motion of the gripper will result in successful grasps.

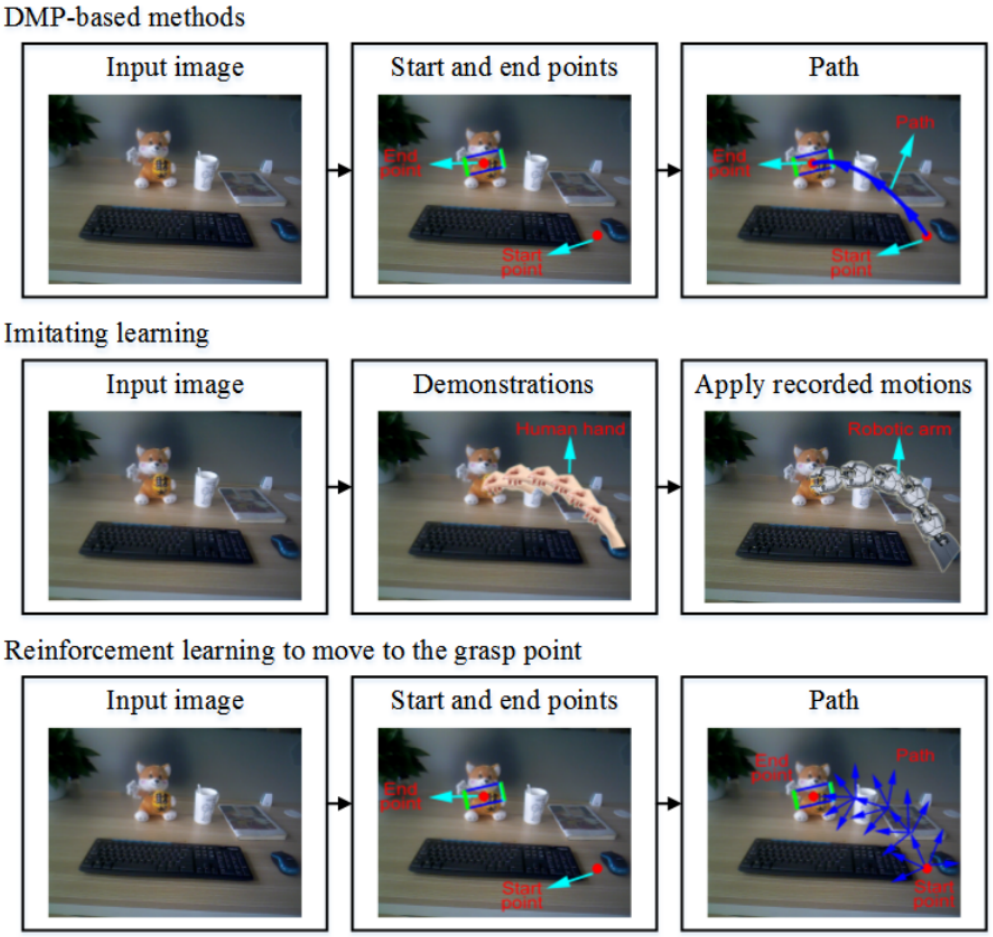


**Figure 3.1:** A typical functional flow-chart of end-to-end motion planning [16]

## 3.5 Datasets

The performance of a simple machine learning algorithm relies on the amount of training data as well as the availability of domain-specific data. Recent publications suggest that the availability of training data is one of the main challenges for this learning method. Some researchers combined datasets to create a larger dataset while others collected and annotated their own data. The Cornell Grasp Dataset(CGD)[13] is a popular grasp dataset that appeared in a number of research studies, suggesting it has a reasonable diversity of examples for generalised grasps. The CGD was created with grasp rectangle information for 240 different object types and it contained 885 images, 885 point clouds and 8019 labelled grasps including valid and invalid grasp rectangles. The grasps are defined in parallel plate gripper, a common type of gripper found in many robots, and the type we will be using in this project as well. By including point clouds, the CGD dataset also allows to create RGB-D images for learning purposes.

## 3.6 Domain Adaptation and Simulated Data

By using synthetic data and domain adaptation, Bousmalis et al [8] were able to reduce the number of real-world samples needed to achieve a certain performance level by up to 50 times, using only randomly generated simulated objects. The authors used two different sources of objects for their experiments:

- **Procedurally generated** random geometric shapes: created by attaching rectangular prisms at random locations and orientations. The prisms were then converted into meshes using Blender and applying random levels of smoothing.
- **Realistic objects** obtained from the publicly available ShapeNet 3D model repository: each object was re-scaled to a random graspable size and assigned a reasonable random mass.

They found that simulated data always aids in improving vision-based real world grasping performance, regardless of the number of real world samples used during training. They also observed there was no need for realistic 3D models to significantly improve performance. They compared the performance of two grasping systems: one trained using procedurally generated shapes and another trained using objects from the ShapeNet[10] repository, both combined with 10% real world data and under all randomization scenarios. They found that using procedural objects was the better choice in all cases. These findings have interesting ramifications, as it could indicate that the noise intrinsic of real world objects might just confuse the model and that using simpler, more primitive shapes might aid the learning process.

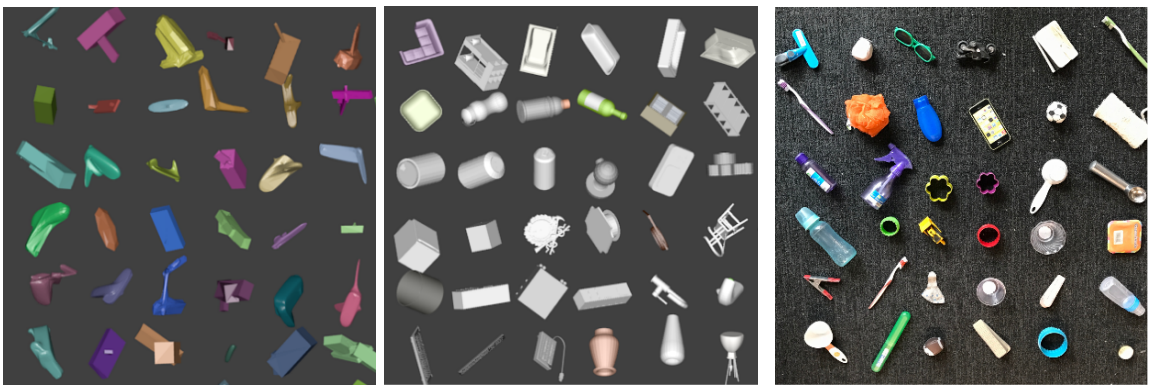

**Figure 3.2:** (Left) Procedurally generated objects. (Center) ShapeNet objects. (Right) Real objects[8]

## 3.7 CNNs for Grasp Detection

The state-of-the-art work in robotic grasp detection involves using different variations of CNNs to learn the optimal gripper configuration for different object shapes

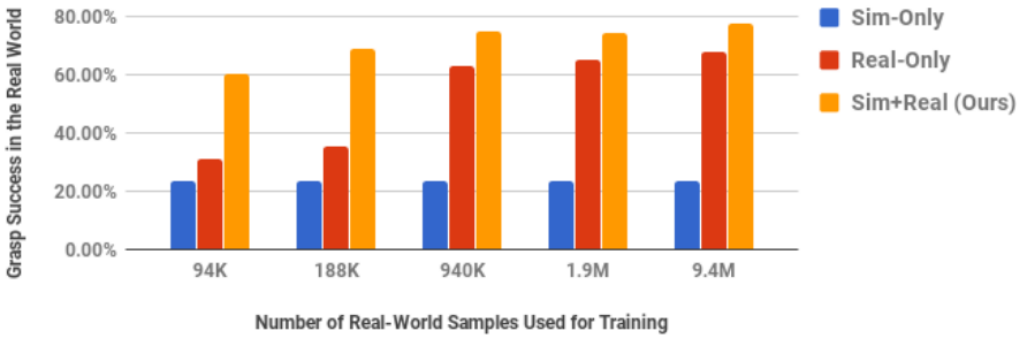


**Figure 3.3:** The effect of using 8 million simulated samples of procedural objects with no randomization and various amounts of real data.[8]

and poses. They do so by ranking multiple grasp configurations predicted for each object image. This is achieved by ranking multiple grasp configurations predicted for an image of the object we intend to grasp. The ranking of configuration is done using learnt parameters from the representation learning capability of deep learning.

Analytic and model-based grasping methods can achieve excellent generalization to situations that satisfy the assumptions they were created on. However, the complexity and unpredictability of the real world usually defy these assumptions. Many of the grasping systems that have shown the best generalization in recent years incorporate CNNs into the grasp selection process.[8]

## 3.8 Bridging the Reality Gap

In this project, we aim to train an accurate grasping model using no real-world images. Although learning from simulated data has significant advantages due to the scalable, rapid, and low-cost of data collection, the model would be of little use if the knowledge gained from simulation didn't apply to the real world.

Discrepancies between physics simulators and the real world make transferring behaviours from simulation challenging[12]. System identification, the process of tuning the parameters of the simulation to match the behaviour of the physical system, is time-consuming and error-prone. Even with strong system identification, the real world has unmodeled physical effects(e.g. non-rigidity and wear-and-tear) that are not captured by current physics simulators. Furthermore, low fidelity simulated sensors like image renderers are often unable to reproduce the noise present in their real-world counterparts. All these differences, known collectively as the reality gap, form the barrier to using simulated data on real robots.

A simple but promising method for addressing the reality gap is called domain randomization. Instead of training the model on a single simulated environment, the simulator is randomized to expose the model to a wide range of environments at training time. If the variability of the simulation is significant enough, models trained in simulation will generalize to the real world with no additional training. Josh et al[53] have applied this method to object localization, successfully training a

real-world detector that is accurate to 1.5cm and robust to distractors and partial occlusions using only data from a simulator with non-realistic random textures. James et al[20] have used this method successfully to train models that not only succeed in the real world but are also able to accomplish the task with variations in the positions of the object, camera, and joint angles. Moreover, the model shows robustness to distractors, lighting conditions, changes in the scene and moving objects(including people).


**Figure 3.4:** Illustration of the domain randomization approach. An object detector is trained on hundreds of thousands of low-fidelity rendered images with random camera positions, lighting conditions, object positions, and non-realistic textures. At test time, the same detector is used in the real world with no additional training.

## 3.9 Dex-Net

In this Section, we will introduce and summarize the most interesting aspects of the Dex-Net publications. This will be done in order to give a general overview of the methodologies, algorithms, and architectures used by Dex-Net. We will also try to gain an understanding of some of the physics and statistics involved in the generative process of synthetic grasps, as we will be working closely with code related to these aspects.

### 3.9.1 Intro

The Dexterity Network (Dex-Net) [29, 27] is a research project including code, datasets, and algorithms for generating datasets of synthetic point clouds, robot parallel-jaw grasps and metrics of grasp robustness based on physics for thousands of 3D object models to train machine learning-based methods to plan robot grasps.

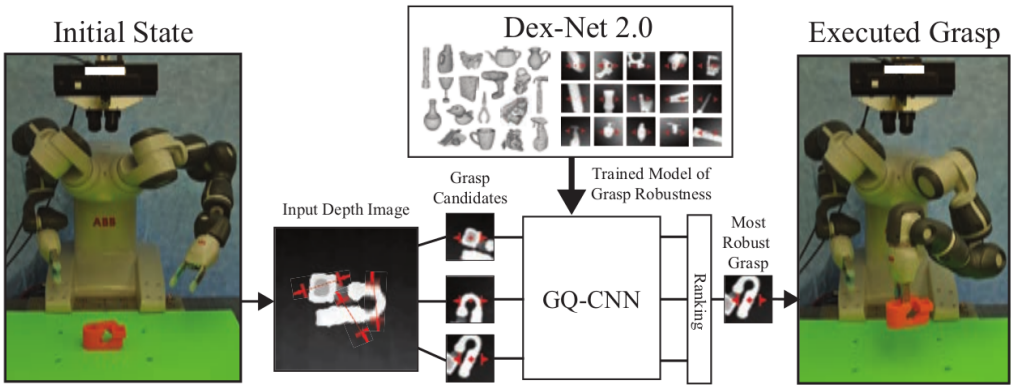


**Figure 3.5:** Dex-Net architecture. First, the Grasp Quality Convolutional Neural Network (GQ-CNN) is trained in order to predict the robustness candidate grasps. Then, an object is presented to the robot, though a depth camera, together with some grasp candidates. The GQ-CNN rapidly determines the most robust grasp candidate, which is executed by the robot.

The broader goal of the Dex-Net project is to develop highly reliable robot grasping across a wide variety of rigid objects such as tools, household items, packaged goods, and industrial parts.

Thanks to Dex-Net, a synthetic dataset was created associating 6.7 million point clouds and analytic grasp quality metrics with parallel-jaw grasp quality metrics planned using robust quasi-static GWS analysis on a dataset of 1500 3D object models in randomized poses on a table. This dataset attempts to reduce data collection time for deep learning of robust robotic grasp plans[3]. The dataset is then used to train a Grasp Quality Convolutional Neural Network(GQ-CNN), which can be used to predict the robustness of grasp candidates. A summary of the Dex-Net data creation process, GQ-CNN training, and grasp execution is shown in Figure 3.5.

### 3.9.2 Dex-Net Research

**Problem Statement** Dex-Net considers the robust grasp planning problem for a given 3D object model and parallel-jaw grasp grippers using probability of force closure($P_F$) under uncertainty of object pose, and friction coefficient as a grasp quality metric. Object shape is given as a signed distance field(SDF) $f : \mathbb{R}^3 \rightarrow \mathbb{R}$ [25]. For each point in the object's bounding box, the SDF associated the distance from the closest surface. Therefore, the values will be equal to zero on the object's surface, positive on the outside and negative on the inside. The object is specified in units of meters with given center of mass $\mathbf{z} \in \mathbb{R}^3$. It assumes soft-finger contacts with a Coulomb friction model[57] (the soft finger model allows the application of the same forces as the hard contact plus a torque around the direction normal to the contact boundary) and that the gripper jaws are opened to their maximum width $w \in \mathbb{R}$ before closing them to attempt the grasp.

**Sources of Uncertainty** Dex-Net assumes a Gaussian distribution on object pose, and friction coefficient to model errors in registration and robot calibration.

- $\mu_\xi \in SE(3)$ represents the mean object pose
- $\xi$ is an object pose random variable defined as $\xi = exp(\mathbf{v}^\Lambda)\mu_\xi$
  - where $\Lambda$ is a function of the type $\Lambda : \mathbb{R}^6 \rightarrow SE(3)$
  - where **v**, defined as $\mathbf{v} \sim \mathcal{N}(0, \Sigma_\mathbf{v})$ and $\mathbf{v} \in \mathbb{R}^6$, is the gripper pose uncertainty with mean $\mu_\mathbf{v} \in G$
- The random variable $\gamma$, defined as $\gamma \sim \mathcal{N}(\mu_\gamma, \Sigma_\gamma)$, is a Gaussian distribution on the friction coefficient with mean $\mu_\gamma \in \mathbb{R}$

$\widehat{\gamma}, \widehat{\xi}, \widehat{\mathbf{v}}$ are samples of the random variables defined above.

**Contact Model** Given a grasp **g** and an object $\mathcal{O}$ and samples $\widehat{\gamma}, \widehat{\xi}, \widehat{\mathbf{v}}$, let $\mathbf{c}_i \in \mathbb{R}^3$ for $i \in 1, 2$ denote the 3D contact location between the i-th gripper jaw and the surface. Each contact $\mathbf{c}_i = \mathbf{x} + (-1)^i(w/2 - t * i)\mathbf{v}$ where[25]:

$$t * i = \arg\min_{t \geq 0} \text{ t such that } f(\mathbf{x} + (-1)^i(w/2 - t)\mathbf{v}) = 0$$

The surface normal at contact $\mathbf{c}_i$ with tangent vectors $\mathbf{t}_{i,1}, \mathbf{t}_{i,2} \in S^2$ is defined as

$$n_i = \frac{\bigtriangledown f(\mathbf{c}_i)}{\| \bigtriangledown f(\mathbf{c}_i) \|_2}$$

To compute the forces that each contact can apply to the object for friction coefficient $\widehat{\gamma}$, the friction code is discretized at $\mathbf{c}_i$[37] into a set of $l$ facets with vertices

$$\mathcal{F}_i = \{n_i + \widehat{\gamma} cos(\frac{2\pi j}{l})\mathbf{t}_{i,1} + \widehat{\gamma} sin(\frac{2\pi j}{l})\mathbf{t}_{i,2} | j = 1, ..., l\}$$

Each force $f_{i,j} \in \mathcal{F}_i$ can exert a corresponding torque $\tau_{i,j} = \mathbf{f}_{i,j} \times \rho_i$ where $\rho_i = (\mathbf{c}_i - \mathbf{z})$ is the moment arm at $\mathbf{c}_i$. Under the soft contact model, each contact $\mathbf{c}_i$ exerts an additional wrench $\mathbf{w}_{i,l+1} = (\mathbf{0}, \mathbf{n}_i)$. Thus the set of all contact wrenches that can be applied by a grasp **g** under the model is

$$\mathcal{W} = \{\mathbf{w}_{i,j} = (\mathbf{f}_{i,j}, \tau_{i,j}) | i = 1.2 \text{ and } j = 1, .., l + 1)\}$$

**Grasp Representation** Dex-Net 1.0 uses a grasp representation such that $g = (\mathbf{x}, \mathbf{v})$, where $\mathbf{x} \in R^3$ represents the centroid of the parallel-jaw grasp in 3D space and $\mathbf{v} \in S^2$ represents an approach direction, or axis, of the grasp. The surface of the object is represented by SDF(or singed distance field) $f$ and includes all points that satisfy the following equation: $S = \{\mathbf{y} \in R^3 | f(\mathbf{y}) = 0\}$. An SDF assigns to every point its distance from the closest surface of the object. This means the SDF will have value 0 for points on the surface of the object, positive value for points external to the object and negative value for points inside it. Dex-Net 2.0 uses a slight variation of the above representation: $g = (\mathbf{p}, \varphi)$, where $\mathbf{p} = x, y, z$ is the centroid of the grasp and $\varphi$ is the approaching angle.

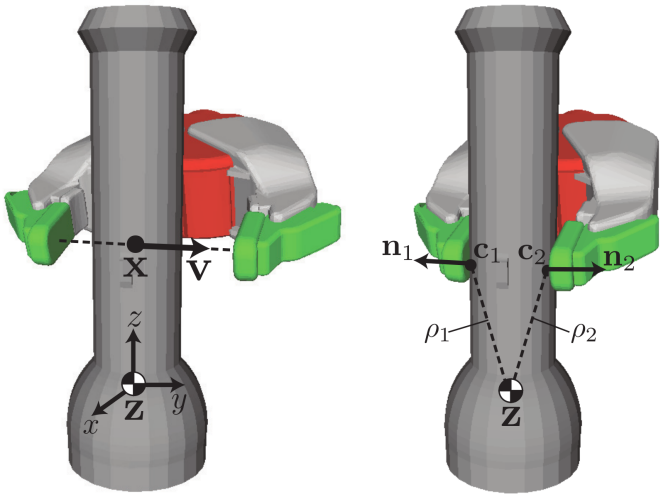


**Figure 3.6:** Grasp parameterization and contact model used in Dex-Net.[29]

**Quality Metric** Force closure[22] is a binary quantity that measures whether or not a grasp can resist external wrenches (forces and torques) applied to a grasped object in arbitrary directions when an object shape is known precisely. The Ferrari-Canny[17] grasp metric $Q_F$ measures the strength of force closure by the relative magnitude of wrenches that the gripper would have to exert to resist external wrenches. A successful force closure corresponds to a positive Ferrari-Canny value.

**Grasp sampling** Because we aim to use a parallel-jaw gripper, we need to sample antipodal grasps. A grasp is defined as antipodal when its contact points are placed on opposite sides of the object, as shown in Figure 3.6. In order to find these grasps, Dex-Net uses a modification of the 2D algorithm presented by Smith et al[5, 25]. Given a 3D object $\mathcal{O}_i$, a maximal opening of the gripper $w$, a sampled friction coefficient $\widehat{\gamma}$ and a set of points on the surface S of an SDF $f$, Dex-Net generates $N_g$ grasps. To sample a single grasp, it first generates a contact point $\mathbf{c}_1$, by sampling uniformly from S. Next, a random direction $\mathbf{v} \in \mathbb{S}^2$ is sampled uniformly from the friction cone and $\mathbf{c}_2$ is computed using $\mathbf{c}_2 = \mathbf{c}_1 + (w - t_2^*)\mathbf{v}$, where $t_2^*$ is defined above, and $\mathbf{x} = 0.5(\mathbf{c}_1 + \mathbf{c}_2)$ is the center of the grasp. This yields a grasp $\mathbf{g}_{i,k} = (\mathbf{x}, \mathbf{v})$. This grasp is then added to the candidate set if the contacts are antipodal[25].

**Dataset Generation**

- **3D Models**: each mesh is aligned with the standard frame of reference, rescaled to fit within a gripper of 5cm and assigned a mass of 1kg, centered in the object bounding box. For each object, a series of stable poses is computed. Each stable pose has a probability of occurrence, all stable poses with a probability below a certain threshold are ignored and not added to the database.
- **Parallel-Jaw Grasps**: each object is labelled with a set of parallel-jaw grasps. A quality metric $E_Q$ is calculated for a combination of grasp, object pose, gripper

pose, and friction coefficient uncertainty using Monte-Carlo sampling.

- **Stable poses**: For each stable pose of the object a set of grasps perpendicular to the table surface and collision-free for a given gripper model are added.
- **Rendered Depth Images**: Each object is then paired with a set of depth images for each object's stable pose. Each image is rotated, translated, cropped, and scaled to align the grasp pixel location with the image center and the grasp axis with the middle row of the image, creating a 32×32 pixels image.

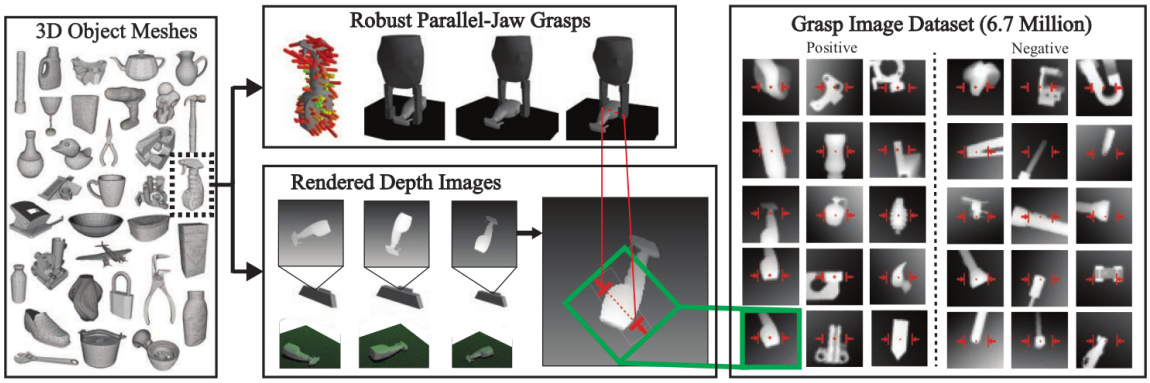


**Figure 3.7:** Dex-Net database creation pipeline.

**Grasp Quality Convolutional Neural Network(GQ-CNN)** This network estimates the probability of grasp success(robustness), which can be used to rank grasp candidates. The most robust grasp is checked for kinematical reachability and possible collision with the table if both checks pass then the grasp is executed. The GQ-CNN takes as input the gripper depth from the camera $z$ and a depth image centered at the grasp center $\mathbf{v} = (i, j)$ and aligned with the grasp axis orientation $\varphi$. This is done so the network doesn't need to learn rotational invariances and allows it to evaluate any grasp orientation in the image rather than a predefined set.

The input is normalized by subtracting the mean and dividing by the standard deviation of the training data, following standard preprocessing conventions. The filters of the first layer appear to compute oriented image gradients at various scales, which may be useful for inferring contact normals and collisions between the gripper and object. The GQ-CNN has approximately 18 million parameters. The architecture, shown in Figure 3.8, contains four convolutional layers in pairs of two separated by ReLU(described in 2.2) non-linearities followed by 3 fully connected layers and a separate input layer for the $z$, the distance of the gripper from the camera. The GQ-CNN is optimized using back-propagation with stochastic gradient descent and momentum. The weights are initialized by sampling from a zero-mean Gaussian with variance $\frac{2}{n_i}$, where $n_i$ is the number of inputs to the $i$-th network layer. In order to expand the dataset, each image is reflected around its vertical and horizontal axes and rotated by 180 degrees, since every one of these modifications results in a grasp equivalent to the one we started with.

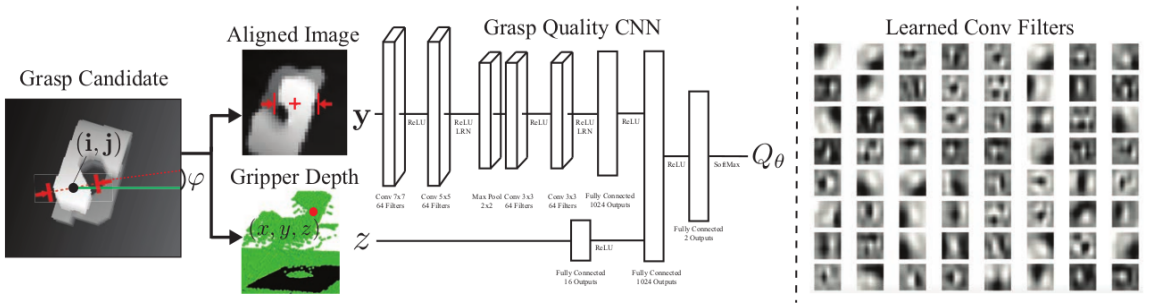


**Figure 3.8:** GQ-CNN architecture.

### 3.9.3 Dex-Net Codebase

The Dex-Net codebase is organized in four main libraries: database, grasping, learning, and visualization. We will briefly summarize the structure of the database and grasping libraries.

#### Database

The database part includes classes for access and storage of 3D models, images, grasps, and grasp quality metrics from Dex-Net. It includes a wrapper for reading and writing `hdf5` databases(currently the only type supported) using the `h5py` Python library. It provides functions to add, read and delete graspables(they include name, SDF, mesh, stable poses, and mass), grasps, and images.

It also includes a Mesh Processor class, that creates a graspable object given a mesh and some configuration parameters. This class creates the SDF corresponding to the given mesh and a list of stable poses for the object. To create the SDF, Dex-Net uses a library called `SDFGen`, that creates a file with extension `.sdf`. Another library, `meshpy`, is then used to read the `.sdf` file and create a `SDF3D` Python object that will then be used by Dex-Net to calculate grasps. It also provides a class to re-scale meshes.

#### Grasping

**Contact** Each contact point is defined as a point on a graspable object's surface and a direction. This class can provide the surface normal at the contact point and calculate a friction cone(takes the number of facets and friction coefficient as parameters).

**Grasp** In Dex-Net, the class `ParallelJawPtGrasp3D` represents a grasp in 3D space. Each grasp has two contact points. From these two contact points we can then calculate the center of the grasp, the axis, the approaching angle, the width of the grasp and so on. Each grasp includes a transformation matrix from the grasp frame to the object frame. The translation vector is represented by the grasp center. The class also provides functions to calculate the distance between grasps, surface information, and other useful features.

**Grasp Sampling** Dex-Net provides three different samplers: Uniform Gaussian Sampler, Gaussian Grasp Sampler, and Antipodal Grasp Sampler. The Uniform Gaussian Sampler samples grasps by sampling pairs of points on the object surface uniformly at random.

The Gaussian Sampler samples grasps by sampling a center from a Gaussian with mean at the object center of mass and grasp axis by sampling the spherical angles uniformly at random.

The Antipodal sampler samples antipodal pairs using rejection sampling. First, it chooses a random point on the object surface, then samples random directions within the friction cone, then forms a grasp axis along the direction, closes the gripper's fingers, and keeps the grasp if the second contact point is also in the friction cone.

**Grasp Quality** Dex-Net provides multiple algorithms for evaluating grasp quality, but the two main ones are Force Closure, and Ferrari-Canny L1. Force closure returns a binary value, while Ferrari-Canny returns the strength of force closure by the relative magnitude of wrenches that the gripper would have to exert to resist external wrenches. Both metrics can be calculated in either Robust Static or Robust Quasi-Static fashion. Using Robust Static means we know with certainty both the position of the gripper and the position of the object.

The Robust Quasi-Static method takes into consideration some level of uncertainty, the degree of which can be specified in a configuration file, over gripper and object position. In particular, for grasp uncertainty we can define:

- Uncertainty over x, y, z position
- Uncertainty over x, y, z rotation

For uncertainty in object position we can define:

- Uncertainty over x, y, z position
- Uncertainty over x, y, z rotation
- Uncertainty in scale

We can also define uncertainty in friction coefficient. The Robust Quasi-Static method is implemented by sampling $n$ samples from a Gaussian distribution for each source of uncertainty(e.g. gripper position on $x$-axis, object rotation around $y$-axis ...). For each one of the $n$ samples, the Force Closure or Ferrari-Canny metrics are calculated. Finally, the quality metrics of each of the $n$ samples are averaged to obtain the final result.

If Force Closure is calculated using a Robust Quasi-Static method, it can be called Probability of Force Closure, as grasp quality would be one if the grasp is certain considering the given uncertainty parameters or 0 if the grasp is never going to be successful.

#### Overview

To generate grasps for a given mesh, we would first call the `MeshProcessor` class in the Database library and create the corresponding `GraspableObject`. Once the object and its corresponding files have been computed, we can create an Antipodal Grasp Sampler and generate the grasps. Once we obtain the grasps, we can create a Grasp Quality Configuration, including our quality method(e.g. force closure, Ferrari-Canny) and quality type(e.g. quasi-static, robust quasi-static). Then we can use the resulting Grasp Quality Function to evaluate our grasps. When we have our grasps and quality metrics, we can use the visualization library to display the grasps on the object, using different colors to indicate the quality of each grasp.

We can also use the database library to save our calculated grasps and corresponding quality metrics in a `hdf5` file, so that we can display them again in the future without the need to compute them again. If we wanted to use this data to train a GQ-CNN, we could use the database library to generate 2.5D(depth images) point clouds from our computed grasps.

### 3.9.4 GQCNN Codebase

The GQCNN(Grasp Quality Convolution Neural Network) repository is part of the Dex-Net project and it is developed and maintained by the same researchers at the Berkeley Automation Lab. GQ-CNNs are neural network architectures that take as input a depth image representing a grasp, and output the predicted probability that the grasp will successfully hold the object while lifting, transporting, and shaking the object. The GQ-CNN weights are trained on the datasets of synthetic point clouds, parallel jaw grasps, and grasp metrics generated using Dex-Net. As the Github repository is named GQCNN, we will use this name to refer to the codebase, and GQ-CNN to refer to the neural network architectures.

The GQCNN repository provides tools to train, finetune and analyze GQ-CNNs, together with a variety of policies that can be used to successfully plan grasps starting from a depth image. This repository was updated to its 1.0 version in April 2019, adding support for grasp planning in object clusters, a suction point and Fully-Convolutional CNNs(the latest Dex-Net publication).

#### Training

GQCNN uses TensorFlow[1] for training. The repository allows the user to define the GQ-CNN architecture and hyperparameters in a configuration file so that there is no need to write any code to train new GQ-CNNs. In the configuration files it is possible to define the GQ-CNN layers, batch size, epochs, frequency of evaluation(using validation set), train/validation split, type of loss function(l2, sparse and weighted cross entropy are supported), training mode(classification or regression) and threshold for positive examples, among others.

GQCNN also offers the possibility of training starting from an already trained network such as Dex-Net 2.0. This process is called fine-tuning. Because training from scratch can be time-consuming, the most efficient way to train a new network

is to fine-tune the weights of a pre-trained GQ-CNN model, which has already been trained on millions of images.

Dex-Net 2.0 is trained using a batch size of 128, a momentum term of 0.9, and an exponentially decaying learning rate step size of 0.95. The first layer of $7\times7$ convolution filters suggests that the network learned fine-grained vertical edge detectors and coarse oriented gradients. The Dex-Net authors hypothesize that vertical filters help to detect antipodal contact normals while the coarse oriented gradients estimate collisions.

#### Analysis

A tool for analyzing trained GQ-CNNs. Calculates statistics such as training/validation errors and losses. It plots Precision-Recall Curve and ROC, and also saves True Positives, True Negatives, False Positives and False Negatives examples from both the training and validation sets to help with training.

#### Policies

All GQ-CNN grasping policies are child classes of the base `GraspingPolicy`. They operate on `RgbdImageStates` and return a `GraspAction`, the grasp with the highest probability of success according to the given GQ-CNN. The `GraspAction` includes a `Grasp2D` object and its q-value. The q-value is the result obtained from the GQ-CNN we are running the policy with, it is an indicator of grasp quality(i.e. probability of success). The `Grasp2D` defines a grasp in terms of its center(in image coordinates), grasp axis and depth(from camera). This class also provides a method for transforming the grasp's 2D image coordinates in 3D coordinates that can be used to position the gripper. This is done by deprojecting the 2D grasps using the camera intrinsic and extrinsic parameters. Multiple grasps can be obtained from the same contact point by discretizing the height starting from the object surface to the table surface(h=0). The grasp height(or depth) is calculated by:

1. Finding the minimum value in a window around the center pixel of the grasp
2. Uniformly sampling a depth value between a minimum and a maximum offset

There are multiple types of policies and the primary ones are listed below.

**Antipodal Grasp Sampling Algorithm** The Antipodal Grasp Sampling method is designed to sample antipodal grasps specified as a center, angle, and height with respect to the table. First, edge detection is performed by applying a Gaussian filter with a threshold over the image, the threshold value can be tuned to detect pixel areas with a smaller or higher gradient magnitude. Each edge pixel is also assigned a direction, normal to the surface found by the edge detection step. After this, pairs of pixels are uniformly sampled from the edges to generate antipodal contact points on the object. The pairs need to be parallel to the table, their distance cannot be greater than the maximum gripper width and the depth in the center pixel between the two contact points must be within a given range. If any of these condition fails,

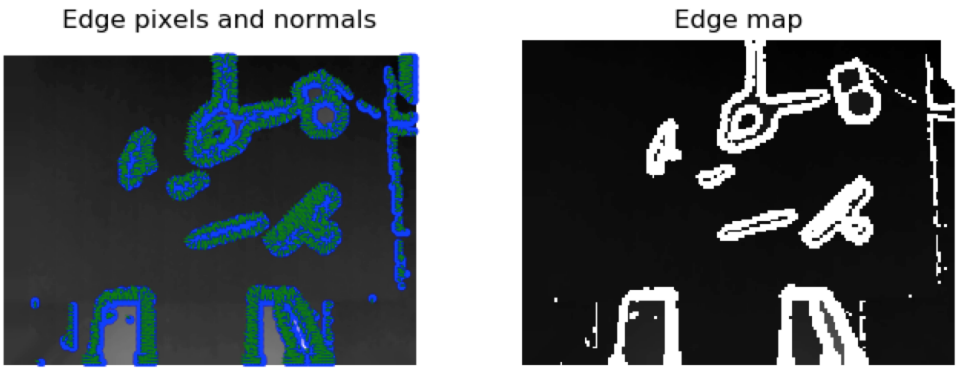


**Figure 3.9:** Edge detection during antipodal grasp planning[54]

the grasp is pruned from the total set of grasps. Finally, the 2D antipodal grasps in image space are deprojected to 3D. The algorithm is described in more details in Listing 11 in the Appendix.

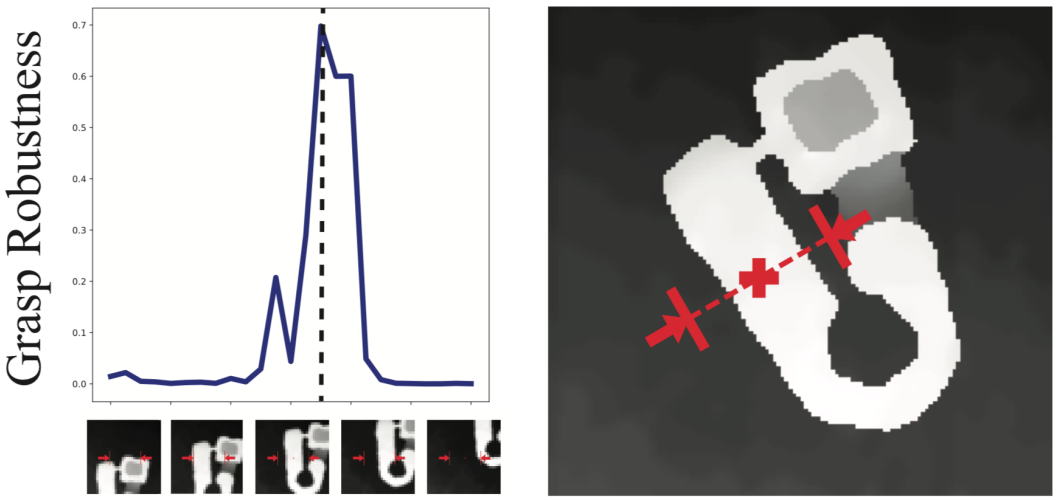


**Figure 3.10:** Edge detection during antipodal grasp planning[54]

**Cross Entropy Robust Grasping Policy (CEM)** Randomly choosing a grasp from a set of candidates does not work well in cases where the grasping regions are small and require very precise gripper configurations. Taking a look at the image above, we can see that as we sweep candidate grasps from top to bottom, grasp robustness stays near zero and spikes momentarily when we reach the good, yet narrow grasping area. Thus, uniform sampling of grasp candidates is inefficient especially since were trying to perform real-time grasp planning.

We can modify our sampling strategy such that at every iteration, we refit the candidate distribution to the grasps with the highest predicted robustness. The algorithm to perform this fitting is the cross-entropy method (CEM) which tries to minimize the cross-entropy between a mixture of Gaussians and the top-k percentile of grasps ranked by GQ-CNN. The result is that at every iteration, we are more likely

to sample grasps with high-robustness values (grasps in the spike area) and converge to an optimal grasp candidate. More formally, the algorithm can be described as follows:

1. Sample an initial set of candidates
2. Sort the candidates
3. Fit a Gaussian Mixture Model to the top P%
4. Re-sample grasps from the distribution we obtained in step 3
5. Repeat steps 2-4 for K iterations
6. Return the best candidate from the final sample set

# 4 Extending Dex-Net

In this Chapter, we will be discussing the improvements and extensions we made to the Dex-Net and the GQCNN repositories, together with the bugs we found, reported and fixed. We will also occasionally explain in details how some of the algorithms work, as we often found discrepancies between what was described in the Dex-Net publications and the codebase(although this is probably because the publicly available code is not identical to the one the authors used in their research). To be able to improve and extend the codebase, we had to understand how most of the algorithms worked and how the various components interacted with each other.

We hoped reading through the research papers and the documentation would have been enough, but since we started working with the code and found installation issues, bugs on what we thought were basic features, incompatibility issues between libraries and outdated or misleading comments, We had to adapt our plan. We started diving deeper and deeper not only into the Dex-Net and GQCNN codebases but also all of their dependencies. This took up a substantial part of our time, especially at the beginning of the project, and we think it will be worthwhile to describe the codebase functioning for two main reasons:

- Build a solid foundation for the reader to understand our improvements, extensions and bug fixes.
- Provide accurate documentation that could prove useful to people that might be interested in working with this codebase in the future or plan on extending it.

## 4.1 Installation Issues

As proven by the long list of open issues in the Dex-Net Github repository, installing and getting the program to work was not as straightforward as we initially expected. This took a couple of weeks, as the installation script was not complete and the codebase required a number of libraries that were not listed in the requirements.

The installation environment caused further complications. we tried installing it on Mac OS X but failed because of incompatibilities between OpenGL versions. We also tried installing it in a Docker container, but this didn't seem to solve the OpenGL incompatibilities. Then, we tried the lab machines but gave up as it is impossible to use `sudo` and there were lots of libraries missing. Finally, I managed to dual-boot my own laptop to have Ubuntu 18.04 installed on it. Even this caused some problems, as the Dex-Net codebase was developed and tested on Ubuntu 14.04 and 16.04. A couple of libraries required (e.g. `vtk`) were not available on Ubuntu 18.04, but luckily the updated libraries were backward compatible with their previous version.

It took another week to make sure the code was working as expected. The program kept returning errors and warnings every time it was run, but it looked like this was normal behaviour and didn't cause any actual problems. However, the grasps were not rendered correctly, so we had to make sure this was not caused by any of the initial warnings. After some debugging, we found a mistake in the visualization library and solved it. We also suggested our fixes on the Dex-Net Github repository, as shown in Image 4.1.

## 4.2 Calculating Grasps

Shortly after we got Dex-Net up and running, we realized that the code didn't provide any direct way of calculating grasps for new meshes, but was only able to visualize those already computed from a database. We implemented a Python script that would call Dex-Net functions to calculate antipodal grasps for a mesh.

Unfortunately, the script wouldn't run because of a bug in the `MeshProcessing` class in the database library. This bug was related to reading and parsing the mesh file and was easy to solve. Although the script was now running, the results didn't look correct. When attempting to display the grasps on the object, the rendering window would be blank. We first tried to display the grasps alone, then the object alone. Both renderings looked accurate, but when we tried to display them together the visualization would return nothing.

After a bit of debugging, we noticed the coordinates of the grasps seemed to be displaced from the mesh they were being calculated from. While the mesh was centered at $[0, 0, 0]$, the grasps seemed to be centered somewhere around $[-50, -50, -50]$. We then attempted to draw a line between these two points in the visualization, and realized it was actually displaying both the mesh and the grasps, they were just too small to see and very far away from each other. The next step was to figure out why this was happening. According to the Dex-Net publications and codebase, grasps where calculated based on an SDF(signed distance field) that was generated from the initial mesh provided. The SDF created would be centered somewhere around $[-50, -50, -50]$ and this was why we were getting the displaced grasps as a result. Each SDF file provides its center at the beginning of the file, and the center provided was not $[-50, -50, -50]$, therefore something was wrong with the way the surface points of the SDF were calculated.

After careful reading and analysis of the Dex-Net codebase and its dependencies, we realized there was a bug in the way the SDF Python object was created when reading from the file, causing the resulting points to be displaced somewhat randomly w.r.t. the original mesh. This mistake seemed to be caused by an incompatibility between the way the SDF file was created and the `meshpy` library reading it to create a Python object from it. Once this was fixed, the grasps appeared to be perfectly aligned with the original mesh. The Dex-Net maintainers were not aware of this problem and we suggested my fix on the Dex-Net Github repository.

install.sh — View file @ 1ef75272

```
@@ -34,12 +34,22 @@ ros)
        exit 1
 esac

+# install pip
+curl "https://bootstrap.pypa.io/get-pip.py" -o "get-pip.py"
+sudo python get-pip.py
+
+# install additional deps
+sudo apt-get install -y build-essential python-dev libboost-all-dev assimp-utils libassimp-dev freeglut3-dev libxmu-dev libxi-dev libopenimageio-dev mesa-utils python-tk pkg-config catkin libassimp-dev
+
 # install apt deps
 sudo apt-get install cmake libvtk5-dev python-vtk python-sip python-qt4 libosmesa6-dev meshlab libhdf5-dev

 # install pip deps
 pip install numpy scipy scikit-learn scikit-image opencv-python pyassimp tensorflow h5py mayavi matplotlib catkin_pkg multiprocess dill cvxopt ipython pillow pyhull setproctitle trimesh

+# install additional pip deps
+sudo pip install visalization python-fcl openni2 primesense triangle
+
 # install deps from source
 mkdir deps
 cd deps
```

**SilviaSapora** commented 25 days ago • edited

I had the same problem, but I didn't have your error "Display grasps failed".
But you could try what I did and see if it solves yours as well.
I solved mine by changing the lines of code in dex-net/src/dexnet/visualization/visualizer3d.py in the function to visualize grasps from:

```
mlab.points3d(g1_tf.data[0], g1_tf.data[1], g1_tf.data[2], color=endpoint_color, scale_factor=endpoint_scale)
mlab.points3d(g2_tf.data[0], g2_tf.data[1], g2_tf.data[2], color=endpoint_color, scale_factor=endpoint_scale)
mlab.plot3d(grasp_axis_tf[:,0], grasp_axis_tf[:,1], grasp_axis_tf[:,2], color=grasp_axis_color, tube_radius=tube_radius)
```

to ->

```
points = [(x[0], x[1], x[2]) for x in grasp_axis_tf]
Visualizer3D.plot3d(points, color=grasp_axis_color, tube_radius=tube_radius)
```

**pyni** commented 24 days ago

It works!
Thanks for your suggestion!

👍 2

**pyni** closed this 16 days ago

**Figure 4.1:** Fixes suggestions on Dex-Net Github repository. Above, installation script fixes. Below, visualization library fixes.

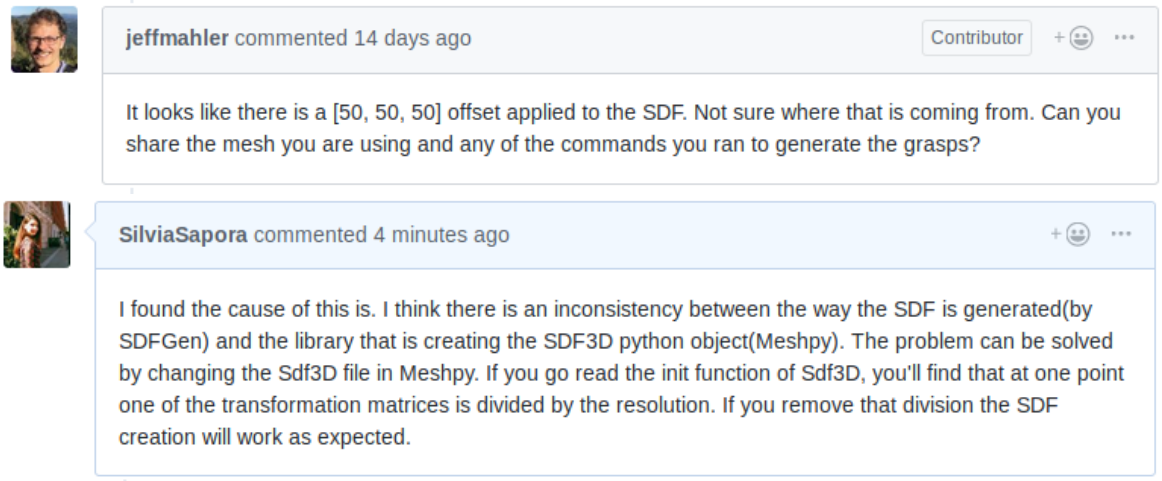


**Figure 4.2:** Extracts from SDF generation bug discussion on the Dex-Net Github repository

### 4.2.1 Rescaling meshes

The Dex-Net codebase contains functions to rescale meshes in case they are too big to fit in the maximum width of the robotic gripper. According to the Dex-Net 2.0 publication, each mesh is aligned to a standard frame of reference using the principal axes and rescaled to fit within a given gripper width. Looking at the codebase, this rescaling can be performed in different ways according to how the configuration is set, and the paper doesn't make clear which one of these methods is used. In the Dex-Net codebase, in order to save a mesh into the database or calculate grasps from it, a `GraspableObject` must be generated. This class take a mesh file as an input, transforms it according to the given parameters and creates an SDF file based on the transformed mesh.

A `GraspableObject` requires parameters for preprocessing the given mesh, including a preprocessing meshlab script, object density, object scale, object rescaling type and the amount of SDF padding to use just to mention a few. The `GraspableObject` class loads the mesh and tries to clean it by removing any unreferenced vertices and incomplete triangles. After this, the pose is standardised so that all objects are aligned with a standard frame of reference. If rescaling is enabled in the configuration than Dex-Net will check the rescaling type parameter to know which dimension to scale along.

The rescaling process begins by going through all the vertices of the matrix and calculating the minimum and maximum coordinates for each axis. The mesh size is defined as the distance between the points defined by the 3 minimum and maximum coordinates. It is important to note that these two points are not actual vertices present in the mesh, but just the minimum and maximum value of each dimension combined, this is mostly done just to simplify the process of calculating the relative scale. The rescaling mode is then used to define in with what method the relative scale will be calculated. The scale of the object(another parameter defined by the user, normally this is set to be equal to the maximum width of the robotic gripper

we are planning to use) is divided by the relative scale to obtain the scale factor. All vertices in the mesh are then multiplied by the scale factor to obtain the rescaled mesh. In total, there are 5 different rescaling modes, listed and explained below:

- **Fit minimum dimension**: The relative scale is equal to the dimension of the minimum dimension of the original mesh
- **Fit median dimension**: The relative scale is equal to the median of the three dimensions of the original mesh
- **Fit maximum dimension**: The relative scale is equal to the dimension of the maximum dimension of the original mesh
- **Relative**: The relative scale is equal to 1.0
- **Fit diagonal**: The relative scale is set to the distance between the minimum and maximum coordinates of the original mesh divided by 3, effectively making the gripper size exactly one-third of the diagonal.

After the mesh is rescaled, the center of the mesh bounding box and the centroid(mean of all vertices) are recalculated. It is important to note that during this process, the `MeshProcessor` class will create a temporary file with the new object mesh and it will use this to generate the SDF. The file has a standard name based on the name of the object, therefore the class will attempt to save two objects with the same name in the same file. Even though this shouldn't be a problem in the case of a single database, as each object needs to have a unique key, we encountered problems when working with multiple databases, or deleting a database and creating a new one, as all these files were being saved in the same directory, and files where not being deleted together with the database. Even when a database was erased, using the specific functions provided by Dex-Net, these files were not deleted. Moreover, if the `MeshProcessor` class found a mesh file for a given object, it will not recompute the mesh but it will use the file already present to calculate the SDF.

This caused a few problems with my meshes and SDFs being different before the cause of the problem was identified. We then simply updated the database API code so that it would delete the mesh files corresponding to the objects we wanted to delete in the database.

### 4.2.2 Calculating Grasps Quality

Even obtaining the quality of each grasp proved to be a challenge. The repository didn't provide any example code on how to do this, so we had to implement this functionality. The Dex-Net paper states probability of force closure was used to train their GQ-CNN, but the configuration provided in the Dex-Net repository was defined so that force closure was calculated using the Quasi-Static method, so that the grasp quality function returned an integer: 1 if the force closure was successful or 0 otherwise, not taking into account any probability. We decided we wanted to use Ferrari-Canny rather than force closure anyway, so we tried to use the corresponding function provided. Unfortunately, because of a bug, the function always returned

0. After some debugging, we found out the library used to calculate the Convex Hull(`pyhull`) of the mesh would fail when called. Luckily, we found an equivalent library(`scipy`) that worked and returned plausible results for the quality of each grasp.

After a while, we figured out how to set up the Grasp Quality configuration so that both Force Closure and Ferrari-Canny could be calculated in a probabilistic fashion, using the Robust Quasi-Static method.

### 4.2.3 Obtaining Grasps Parallel to Surface

The next step was to obtain grasps parallel to the table surface where the object was resting. Any object could be resting in different poses on a given surface, and Dex-Net, through the `trimesh` library, provided functions to calculate stable poses for each mesh. Given a stable pose, we wanted to filter grasps that were parallel to the table surface. To do this, we had to calculate the angle between the grasp axis and the surface normal of the table. Dex-Net didn't provide a function to achieve this, so we had to provide our own implementation. Dex-Net also didn't provide any way of visualizing multiple grasps on an object in a given stable pose, so we added the functionality, shown in Figure 4.3. To obtain grasps parallel to the surface of the

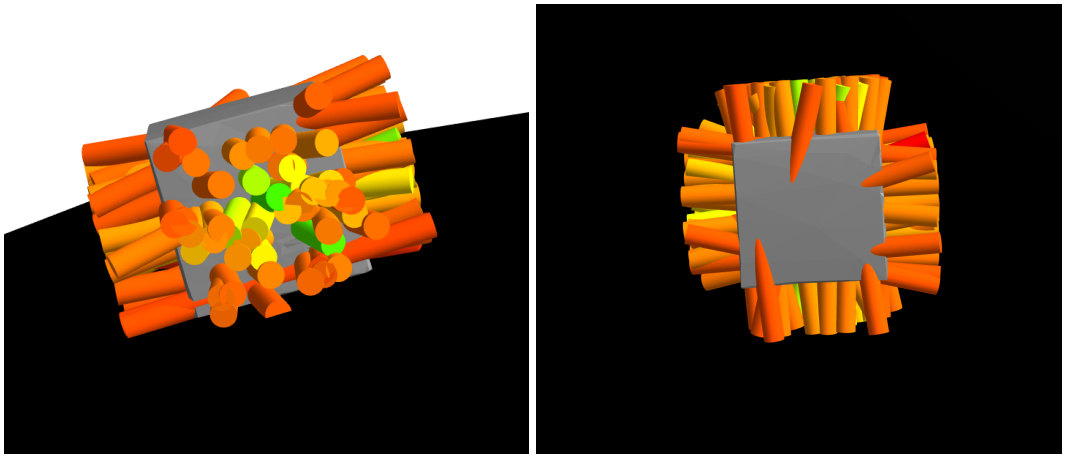

**Figure 4.3:** Antipodal grasps parallel to the table surface sampled from a cube. Grasp quality is indicated by color, where green indicates a more stable grasp. Grasp qualities in this image were calculated using the Ferrari-Canny metric, in a Quasi-Static fashion(no uncertainty)

table for a given stable pose, the first step is to retrieve all the grasps(or calculate them aren't stored already) for a certain object from the database. Then, for each grasp, its 3D axis(the normalized line that connects the 2 3D contact points that define the grasp) is obtained in object coordinates. Following the notation introduced in Section 2.5 we will denote this grasp axis as $\mathbf{g}_{OBJ}$. If the object is in a stable pose defined by a homogeneous transformation matrix $\mathbf{T}_{W,OBJ}$ than the object mesh and all its grasps will be rotated by $\mathbf{C}_{W,OBJ}$. We can leave the translation part of the

transformation aside in this case as we are checking angles. By multiplying the axis of the grasp by the rotation matrix we obtain the axis in world coordinate frame:

$$\mathbf{g}_W = \mathbf{C}_{W,OBJ}\mathbf{g}_{OBJ}$$

We know the $z$-axis in the world coordinate frame is $[0, 0, 1]$, therefore it will be enough to calculate the dot product between $\mathbf{g}_W$ and $\mathbf{z}_W$ to obtain the projection of the axis onto the $z$-axis(normal to the table surface). Since the both the grasp axis and the $z$-axis are normalized our result will be between 0 and 1. If the result is 0, it means the grasp axis is perfectly parallel to the table, if the result it's 1, it means the grasp axis is perpendicular. Once we obtain this result we can simply compare it to a threshold and discard all grasps with a projection above a set value(5 or 10 degrees for example).

### 4.2.4 Checking Grasp Validity

Before moving on into the next part of the project, which consists in generating images from the grasps to train the GQ-CNN, we wanted to make sure the generated grasps and their quality metrics were actually accurate and a good indicator of grasp success. This is necessary to investigate as the grasp quality metrics used in the Dex-Net codebase(i.e. Robust Ferrari-Canny, Ferrari-Canny, Probability of Force Closure and Force Closure) use a physics-based model that is based on strong assumptions of the real world, object shapes and friction model of the object and gripper. A possibility considered by [5, 7] was that these grasp quality metrics could represent only a necessary, not sufficient, condition for a grasp to be executed successfully. As no GQ-CNN could ever perform better than the data it was trained with, we wanted to test the quality of our grasps in simulation, before proceeding to generate the depth images corresponding to each grasp, object, and stable pose.

To do this we set up a simulation environment in V-REP(described in detail in Chapter 5). For each combination of object and stable pose, we checked the Dex-Net database for all the object's grasps that were parallel to the table in the given pose, following the method explained in Subsection 4.2.3. After the grasps are gathered, the object is set to the given stable pose in the middle of the workspace.

Unfortunately, some of the poses turned out to not be very stable once the object was loaded in the simulation. The objects would often wobble and move into another pose before stabilizing. This meant the grasps for those specific poses could no longer be checked accurately, as the object had moved. Consequently, we had to discard all the poses that didn't appear to be stable in the simulation. Luckily we had many examples of sufficiently stable poses, so this didn't affect our ability to make an accurate evaluation of our grasp quality metrics. The next step was to position our gripper in the position indicated by the grasp. This meant translating the grasp transform from object($\underrightarrow{\mathcal{F}}_{OBJ}$) to world($\underrightarrow{\mathcal{F}}_{W}$) frame, as the grasp coordinates and orientation were give in the object frame($\underrightarrow{\mathcal{F}}_{OBJ}$). This could be easily done by multiplying the grasp transform by the pose transform $\mathbf{T}_{W,OBJ}$ as follows:

$${}_W\mathbf{g} = \mathbf{T}_{W,OBJ}\,{}_{OBJ}\mathbf{g}$$

Once the grasp position is obtained in world coordinates, we can translate it to gripper position. For 2D, projected grasps, the grasp's coordinate frame is defined so that the center of the coordinate frame is the center of the grasp(mid-point between the two fingers of the gripper), the $y$-axis is parallel to the line connecting the two (projected) endpoints of the grasp, the $x$-axis points down towards the table and the $z$-axis is parallel to the table. As the grasp is represented this way, it means the gripper will need to be positioned on the object with its fingers aligned with the grasp's $y$-axis and with the end of its fingers pointing towards the grasp $x$-axis. This is coordinate frame notation is represented in Figure 4.4. Unfortunately, while this is the convention for 2D, projected planar grasps(4 DoF), the situation is more complicated for 3D, 6 DoF grasps. For these grasps, the $x$-axis doesn't have to be perpendicular to the table and the $z$-axis doesn't need to be parallel to it. These grasps are defined independently of the object's pose and the only defined element is the $y$-axis that connects the two (not projected) endpoints. This means the grasp coordinate frame needs to be rotated around the $y$-axis so that the $x$-axis points towards the table, otherwise it would be impossible to grasp the object. The Dex-Net codebase doesn't implement this feature, since only planar, 2D grasps would be tested by the gripper. This meant we had to implement a function that, given the coordinate frame of a 3D, 6 DoF grasp, would be able to transform it to a 5 DoF grasp, effectively stopping the gripper from "rotating" around the y-axis of the grasp. To achieve this, each grasp was rotated so that the $x$-axis would always be pointing down towards the table where the object was resting. This meant maximizing the projection of the $x$-axis of the grasp coordinate system onto the normal of the table surface($z$-axis of the world coordinate frame).

The results of our experiments in simulation confirmed how analytic Quasi-Static and Robust Quasi-Static grasp quality metrics can be used as an accurate reward function for learning antipodal grasping policies. If domain randomization is added then these policies will also likely be robust to sensor noise and imprecision.

## 4.3 The Database

Dex-Net relies on 2 types of databases: one for grasp calculation and one for the GQ-CNN training. In the research paper, this distinction is not made very clear, and we initially thought my aim was to create one database containing object meshes, grasps, grasp metrics(i.e. quality) and the rendered images. But after looking at the code we found this not to be the case, as the Dex-Net database does not contain the rendered images necessary of training, and the GQ-CNN doesn't support the Dex-Net hdf5 structure as input. The GQ-CNN code is built to accept a `TensorDataset` class, which mostly consists of a directory full of `.npz` files(compressed numpy files) containing the rendered images that will be used during training, together with grasp metrics and information about grasp configuration(i.e. depth).

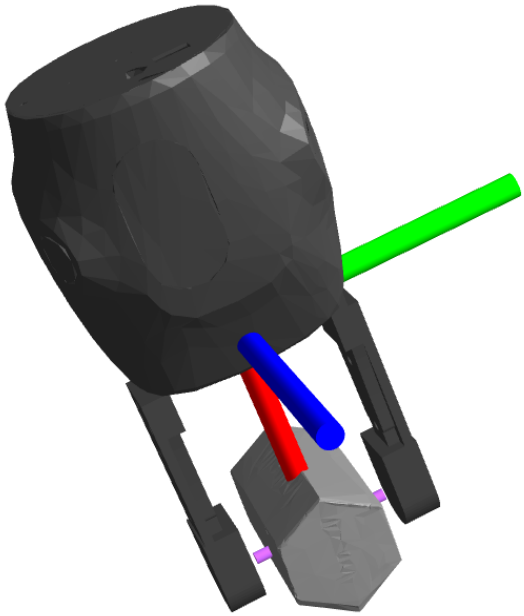

**Figure 4.4:** Grasp-Gripper coordinate frame orientation. The grasp axis is shown in pink. The $y$-axis(green) is parallel to the grasp axis and the table, the $x$-axis(red) points down and is perpendicular to the table and the $z$-axis(blue) is perpendicular to the grasp axis and parallel to the table. The gripper shown in this Figure is a Yumi gripper.

### 4.3.1 Grasp Calculation Database

The grasp calculation database is a `hdf5` database containing object meshes, SDFs, grasps, stable poses and grasp metrics, it can be extended as new objects and grasps are calculated. The database's structure is illustrated in Figure 4.5. It is important to note that at this stage grasps are not specific to any stable pose(we don't check if they are parallel to the table) but just associated with an object. In the GQ-CNN dataset, this will change, as a set of grasps is associated with a specific stable pose for each object. A grasp is only associated with a given stable pose if the grasp is parallel to the table in that pose. This means that the same grasp can be associated to multiple stable poses.

### 4.3.2 GQ-CNN Dataset

The GQ-CNN dataset is a `TensorDataset`, a class defined in `perception`, a library developed and maintained by the same researchers that created Dex-Net. The GQ-CNN dataset normally contains grasps, grasp quality metrics and depth images corresponding to each grasp, but can also contain binary images, information about camera intrinsics and camera pose and other info that it's not strictly necessary for training purposes but can be useful to verify the validity of the data. We managed to find a script to generate the second dataset using the data from the first one only

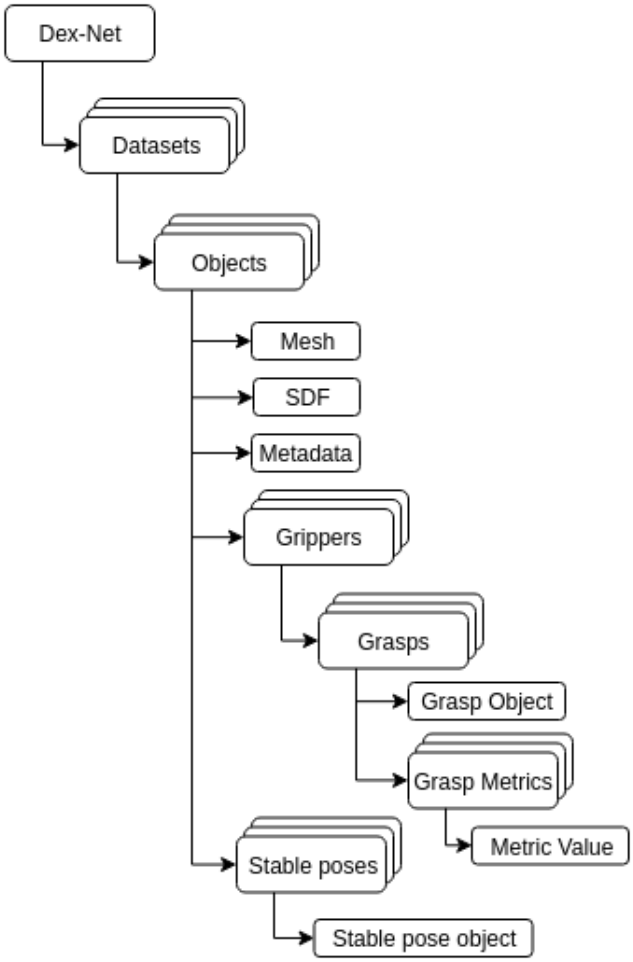


**Figure 4.5:** Dex-Net grasp calculation database's structure

around March. At this time we had already spent some time implementing my own version of it. This, however, turned out to be useful: the original Dex-Net script was using `OpenRave` for collision checking, currently not supported on Ubuntu 18.04. The Dex-Net developers stated many months ago they were planning on moving away from `OpenRave`, as it is quite complicated to install even on older Ubuntu versions. Also, Dex-Net is only compatible with a forked version of an older `OpenRave` release. Even then, no version of Dex-Net as been officially released where the `OpenRave` dependency was being removed, so we had to use V-REP for collision checking(more details on this in Chapter 5). The original script also makes use of `meshrender`, a rendering library to generate the depth images to use during training. At the time we managed to find the script we had already implemented my own code that made use of V-REP and its vision sensors to generate depth images starting from an object, a stable pose, and a given grasp. This gave me the opportunity to compare the two implementations: even if the resulting images were almost identical, the `meshrender` library took significantly less time in gathering the images. This is why we decided to use `meshrender` to generate our images.

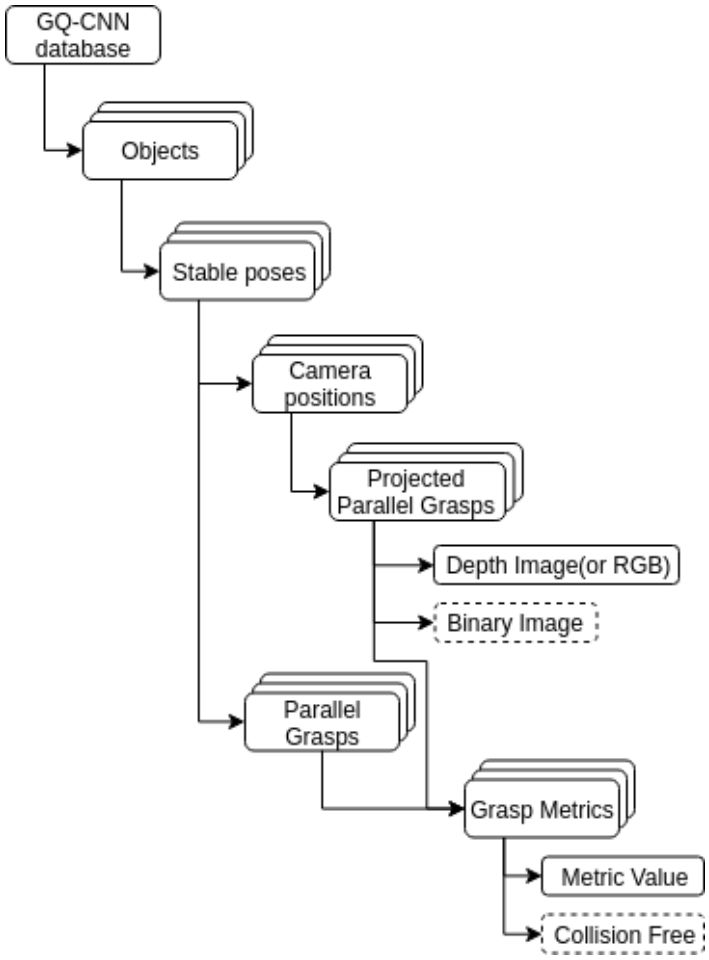


**Figure 4.6:** GQ-CNN dataset structure. Dotted components are optional. The collision free element for each grasp can be saved explicitly or alternatively the metric value can be set to 0.

```
# 1. Precompute the set of valid grasps for each stable pose:
#    i) Parallel to the table
#   ii) Collision-free

for dataset in datasets:
  for obj in dataset:
    stable_poses = dataset.stable_poses(obj.key)
    for stable_pose in stable_poses:
      grasps = dataset.grasps(obj.key, gripper=gripper.name)
      candidate_grasps = []
      for grasp in grasps:
        # ignore grasp if not parallel to the table
        if parallel_table(grasp, stable_pose):
```

```
        collision_free = not simulation.check_collision(grasp)
        candidate_grasps.append((grasp, collision_free))

    # 2. Render images
    for grasp, coll_free in candidate_grasps:
      # sample images from random variable
      random_variable = RandomizedCameraPosition(obj, stable_pose)
      render_samples = random_variable.sample(img_samples_per_stable_pose)
      for render_sample, camera_position in render_samples:
        depth_im = render_sample.depth

        # project grasp based on randomized camera position
        grasp_2d = grasp.project_camera(camera_position)

        # center image at grasp center and align with grasp axis
        depth_im = depth_im.transform(grasp_2d.center, grasp_2d.angle)
        # crop and resize image
        depth_im = depth_im.crop(96, 96).resize(32,32)

        tensor_datapoint['depth_im'] = depth_im
        tensor_datapoint['grasp_info'] = grasp_2d
        tensor_datapoint['metric'] = grasp.metric_value * coll_free
        tensor_dataset.add(tensor_datapoint)
```

**Listing 1:** Psudocode of the script used for the creation of the GQ-CNN dataset

## 4.4 Generating Depth Images

As mentioned before, the GQ-CNNs are trained using depth images as input and grasp metrics as labels. Each object is paired with a set of depth images for each object's stable pose. Each image is then rotated, translated, cropped, and scaled to align the grasp pixel location with the image center and the grasp axis with the middle row of the image. For our datasets, we decided to generate 50 images for each combination of object, stable pose, and grasp.

For each image, the camera position and rotation sampled from a uniform distribution to make the GQ-CNN more robust to uncertainties in the position of the object and the camera and facilitate domain transfer.

In Dex-Net 2.0, 50 images are generated for each grasp and stable pose. Initially, due to time and computational power constraints, we had to limit the number of samples calculated for our datasets of procedurally generated objects. But we eventually managed to get to the 50 images per grasp that Dex-Net used.

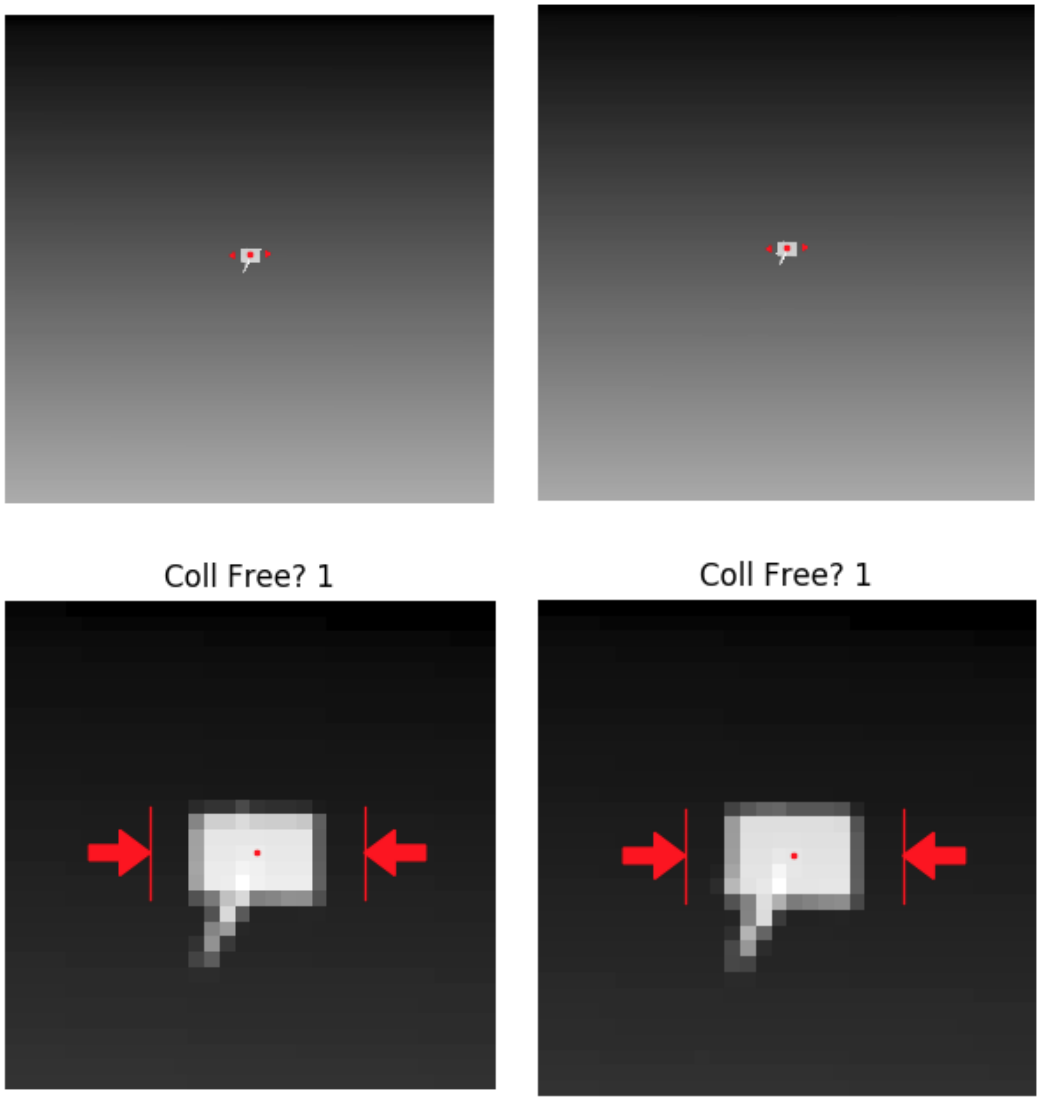


**Figure 4.7:** Comparison between depth images generated using V-REP(left) and `meshrender`(right). At the top, we can see the rendered images(96×96 pixels). At the bottom, the images are rotated, centered and cropped(32×32 pixels) to represent the grasp. It is easy to see the two images are almost identical, and given V-REP takes much more time than `meshrender` to render an image, we decided to use the latter for our dataset generation.

The parameters we used for camera randomization can be found in Listing 9 in the Appendix.

To fully understand the camera configuration parameters it is useful to remember the camera axis are defined as shown in Figure 4.8.

### 4.4.1 Sampling the camera pose

Using a configuration similar to the one presented above, we can sample a camera pose and obtain a `camera_to_world_pose` ($\mathbf{T}_{W,C}$). This 4×4 homogeneous matrix defines the desired position of the camera in world coordinates. To calculate this matrix, we use the algorithm in Listing 2.

```
# sample rotation around each axis
radius = random.uniform(min_radius, max_radius)
elev = random.uniform(min_elev, max_elev)
az = random.uniform(min_az, max_az)
roll = random.uniform(min_roll, max_roll)
# sample plane translation
tx = random.uniform(min_x, max_x)
ty = random.uniform(min_y, max_y)

# calculate camera_to_world_pose
T_world_camera = camera_to_world_pose(radius, elev, az, roll, tx, ty)

def camera_to_world_pose(radius, elev, az, roll, tx, ty):
  # generate camera center and z-axis from spherical coordinates
  translation = [x, y, 0]
  t_world_camera = [sph2cart(radius, az, elev)] + translation
  z_axis = -[sph2cart(radius, az, elev)]
  z_axis = normalize(z_axis)

  # find the canonical camera x and y axes
  x_axis = [z_axis[1], -z_axis[0], 0]
  x_axis = normalize(x_axis)
  # find y-axis as cross product of z-axis and x-axis
  y_axis = cross_product(z_axis, x_axis)
  y_axis = normalize(y_axis)

  # rotate by the roll
  R_world_camera = [[cos(roll), -sin(roll), 0],
                    [sin(roll),  cos(roll), 0],
                    [0,          0,         1]]
  # create final transform
  T_world_camera = RigidTransform(R_world_camera,
                                  t_world_camera,
                                  from_frame='camera',
```

```
                                    to_frame='world')
  return T_world_camera
```

**Listing 2:** Pseudocode of camera pose randomization script

After the `camera_to_world_pose` is calculated, we can use it to render the images. Depending on which framework we use, we will have to use it in different ways.

### 4.4.2 Rendering Images with Meshrender

#### Installation

Getting the `meshrender` library to work correctly with Dex-Net was quite a challenge. If Dex-Net is installed following the instructions and scripts provided calling `meshrender` will return an `AttributeError`. This error is caused by a name conflict, In Dex-Net, there are two different libraries with the same name.

Both are necessary for the correct functioning of the Dex-Net repository, but one of them was not building correctly. This meant Dex-Net was calling the same library(the only one working) even where the other one was needed. This was a known issue on the Dex-Net Github repository, but nobody had published a solution or suggested any fixes.

The first thing we did to address the problem was renaming the library that couldn't build to avoid the name conflict. This was a C++ library and the building was failing because some other required libraries were missing or couldn't be found. After some research and debugging, we managed to track down, install and compile all the necessary libraries to obtain a working version of the library that would link correctly with my project.

The solution was published in the Dex-Net issues section and the other users confirmed they managed to get the library working correctly on their machines as well.

#### Implementation

First, the table is set at position at `object_to_table_pose` $\mathbf{T}_{T,O}$. Then, we calculate a transform `object_to_world_pose` $\mathbf{T}_{W,O}$. When applied to the object mesh, it will set the object resting at coordinates $[0, 0, 0]$ in the given stable pose. To obtain the camera pose in object coordinates(object_to_camera_pose $\mathbf{T}_{C,O}$), the `camera_to_world_pose` transform $\mathbf{T}_{W,C}$ is inverted and multiplied by the `object_to_world_pose` transform $\mathbf{T}_{W,O}$. In pseudo-code:

```
world_to_camera_pose = camera_to_world_pose.inverse()
object_to_camera_pose = world_to_camera_pose.dot(object_to_world_pose)
```

Using the frame notation introduced in Section 2.5(W represents the world frame, C represents the camera frame and O represents the object frame):

$$\mathbf{T}_{C,W} = \mathbf{T}_{W,C}^{-1}$$

$$\mathbf{T}_{C,O} = \mathbf{T}_{C,W}\mathbf{T}_{W,O}$$

Meshrender also needs the projection matrix of the camera, commonly defined as:

```
projection_matrix = [[      fx,       skew,       cx],
                     [       0,         fy,       cy],
                     [       0,          0,        1]]
```

Where `fx` represents the $x$-axis focal length of the camera in pixels, `fy` the $y$-axis focal length and `cx` and `cy` define the optical center of the camera in pixels. `meshrender` gives the option to set different types of lighting and material properties for the objects in the scene. Summarizing, in this framework the object's mesh remains still, therefore what we aim to obtain is a transformation from the object's coordinates to the image coordinates, and our computations reflect this.

### 4.4.3 Rendering Images with V-REP

In V-REP, the object is set to the given stable pose(using the `object_to_world_pose` transform $\mathbf{T}_{W,O}$) and all we have to do is set the camera position using the `camera_to_world_pose` transform $\mathbf{T}_{W,C}$. V-REP also doesn't need a projection matrix, as the camera parameters can be set directly using the V-REP API or GUI. After gathering some images we realized V-REP encodes images as (rows, columns, channels), and when retrieved from the simulator the rows are inverted. We undo these transformations and return a tensor suitable for training.

### 4.4.4 Aligning Images with Grasps

Once we have obtained our 2.5D images of the object in a given stable pose, we need to rotate and crop each image for each one of our grasps and resize the images according to our GQ-CNN parameters. The Dex-Net grasps are 6-DoF, but we can only represent planar grasps(4-DoF) with our images, therefore we will need to project our grasps.

Each one of the rendered images for a given object and stable pose will have a slightly different camera position, orientation, and possibly intrinsic parameters. Each grasp will be projected according to these parameters specific to each image. We will then look at the angle and center of the grasp to transform the original image(translate and rotate). Once the aligned image is obtained, we can crop it and resize according to our parameters. In the original Dex-Net GQ-CNN these parameters are set to 96×96 pixels(crop size) and 32×32 pixels(final image size).

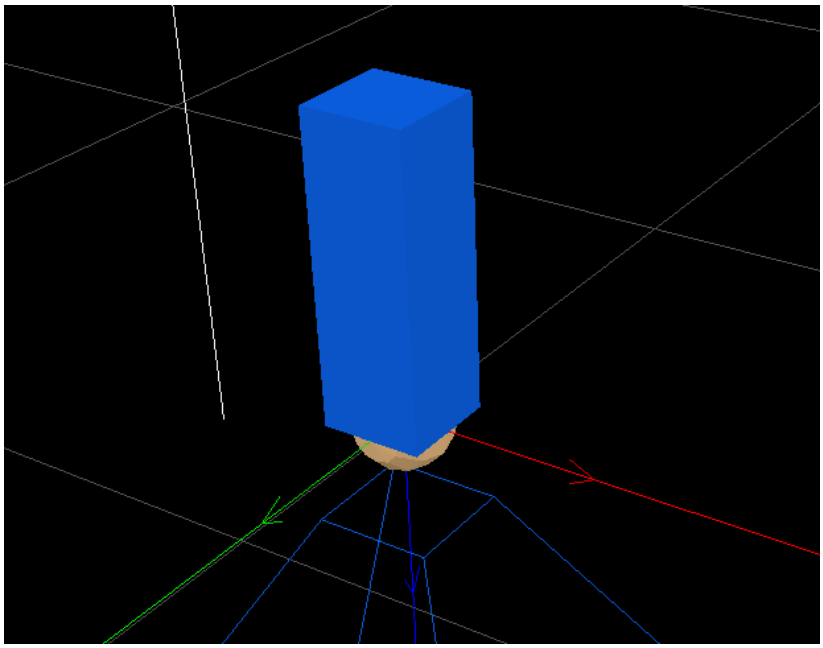

**Figure 4.8:** A representation of a camera's axis. The blue arrow represents the $z$-axis, and it points downwards towards the work-space. The red arrow is the $x$-axis, and it points in the image's horizontal direction. The green arrow is the $y$-axis and it points in the image's vertical direction.

The pseudocode in Listing 3 shows how to project a grasp from 3D to 2D using the camera parameters.

```
# Project a grasp for a given gripper
# into the camera specified by a set of intrinsics.
def project_camera(self, T_obj_camera, camera_intr):
  # T_obj_camera: rigid transform from the obj frame to the camera frame
  # compute pose of grasp in camera frame
  T_grasp_camera = T_obj_camera * self.T_grasp_obj
  y_axis_camera = normalize(T_grasp_camera.y_axis[:2])

  # compute grasp axis rotation in image space
  grasp_angle = arccos(y_axis_camera[0])
  if y_axis_camera[1] < 0:
    grasp_angle = -grasp_angle
  while grasp_angle < 0:
    grasp_angle += 2 * PI
  while grasp_angle > 2 * PI:
    grasp_angle -= 2 * PI

  # compute grasp center in image space
```

```
t_grasp_camera = T_grasp_camera.translation
p_grasp_camera = Point(t_grasp_camera)
u_grasp_camera = camera_intr.project(p_grasp_camera)
depth_grasp_camera = t_grasp_camera[2]
return Grasp2D(u_grasp_camera,  grasp_angle, depth_grasp_camera)
```

**Listing 3:** Pseducode for projecting grasp from 3D to 2D

## 4.5 Generating RGB images

The Dex-Net and GQCNN codebases rely entirely on depth images, as RGB images are not even generated in the original GQ-CNN dataset generation script. Unfortunately, the robot we were planning to use for real-world testing, a Sawyer robotic arm, was not equipped with a depth camera. This meant we had to extend the codebase to work with RGB images as well. First, we generated a new dataset of more than 6 million data points, this time with RGB images.

Then, we had to adapt our Grasping Policy to find antipodal grasps using an RGB image instead of a depth image. The first step was to locate all the edge pixels in our image. The traditional edge detection algorithms are done through detecting the maximum value of the first derivative or zero crossing of the second derivative [56]. Even though first order differential operators (i.e. Roberts operator, Prewitt operator, Sobel operator) and second order differential operators (i.e. Laplace operator, LOG operator) have many advantages such as simple computation, rapid speed and easy to implement, they are more sensitive to noise and therefore not ideal in engineering application. In 1986, Canny proposed three criteria to judge edge detection operator performance: SNR criterion, localization precision criterion, and single edge response criterion, and deduced the best Canny edge detection operator [9, 33]. Compared with common edge detection algorithm, in most cases, the Canny algorithm has the best performance [34, 4, 49]. This is why we decided to use the Canny edge detection algorithm for our specific application. After finding an implementation of the algorithm in Python on GitHub[6], we proceeded to change it slightly to adapt it to our needs and application. The edge detection capabilities of the algorithm seemed satisfactory, at least in the case of an object of mostly uniform color and the table color was significantly different from that of the object.

The next step was implementing a way of calculating normals from the surface of the object for each edge pixel. After all the algorithms were set up and working accurately on sample images, it was time to implement them in the GQ-CNN repository. This was quite a challenge, as the initial aim was to include it as seamlessly as possible by making use of the `ColorImage` class provided by the `perception` library. The goal was creating an alternative policy class to the `AntipodalDepthImageGraspSampler` one, which could take a `ColorImage` object as input as well as a `DepthImage` object, and could produce accurate results for both by calling the same functions on the image variable, regardless of it being an RGB or depth image. Unfortunately, it was

impossible to implement it this way without changing the `perception` library, which we decided was better to avoid, as it would be hard to track changes and extensions. This meant we had to use simple `numpy` arrays for image representations and transformations, making my antipodal grasp sampling policy quite different from the original from an implementation perspective, even though it was still very similar in terms of logic. We then created a `AntipodalColorImageGraspSampler`, that would be called automatically in case no depth image was passed to the sampler or in the case it was explicitly specified in the configuration file.

## 4.6 Procedurally Generated objects

While considering ideas to improve the Dex-Net performance, we decided to try and include procedurally generated shapes into our training dataset to hopefully improve the performance of our network. This was inspired by the findings of Bousmalis et al[8] briefly summarized in Section 3.6. We hope this addition will help with the sim-to-real transfer, as the results of the paper indicate that including simulated data can drastically improve vision-based grasping systems, achieving comparable or better performance with 50 times fewer real world samples. The results also suggest that it is not as important to use realistic 3D models for simulated training. Finally, it indicates that including domain adaptation substantially improves performance in most cases.

Unfortunately, we were not able to find the dataset of shapes generated used by Bousmalis et al in their research, so we had to generate our own. In the paper, the researchers mention they generated 1000 shapes by attaching rectangular prisms at random locations and orientations. They then converted the set of prisms into a mesh using Blender and applied a random level of smoothing to each mesh. Each object was given UV texture coordinates and random colors.

We decided to try using V-REP to generate our shapes since we had already gained some familiarity with the simulation platform. Unfortunately, V-REP doesn't offer the option to smooth shapes, so we had to skip this last step. This caused our shapes to be mostly very irregular and generally quite hard to grasp. We then decided to try different approaches to try and make the shapes a bit more regular, such as mixing cylinders, prisms, and spheres at different positions and orientations. We also tried not modifying the orientations of the prisms or combining a smaller number of shapes. All these attempts resulted in shapes that proved to be easier to grasp in simulation and for which Dex-Net managed to generate more reliable grasps. In Figure 4.9 some of the procedurally generated objects are shown.

Finally, we tried to download Blender and rewrite my V-REP script so it could work in Blender. This allowed us to add smoothing to the shapes and obtain shapes very similar to the ones described by Bousmalis et al in their paper. The pseudo-code below is a simplified version of the algorithm we used to generate the smoothed shapes.

```
def create_object(name, n_components):
```

```
# create base object
obj = create_prism(name, radius=1.0)

# add N-1 new objects to base
for _ in range(n_components-1):
  # create a new component
  comp = create_prism('comp', radius=random.random(0.2,1.0))

  # decide where to add it
  add_pt = Vector(random.choice(possible_points))
  comp.location = add_pt

  # pick axis to align to and depending on this alignment
  # add a new set of add points
  axis = random.choice(['x', 'y', 'z'])
  if axis == 'x':
    # note: object created aligned with x axis
    add_pts.add(tuple(add_pt + Vector((2, 0, 0))))
    add_pts.add(tuple(add_pt + Vector((-2, 0, 0))))
  elif axis == 'y':
    comp.rotation_euler.z = pi/2
    add_pts.add(tuple(add_pt + Vector((0, 2, 0))))
    add_pts.add(tuple(add_pt + Vector((0, -2, 0))))
  else:  # z
    comp.rotation_euler.y = pi/2
    add_pts.add(tuple(add_pt + Vector((0, 0, 2))))
    add_pts.add(tuple(add_pt + Vector((0, 0, -2))))

  # add component to obj
  union(obj, comp)

  # smooth by random factor
  smooth(obj, random.random())

return obj
```

**Listing 4:** Pseudocode for an example of the algorithms used to procedurally generate objects

Once a significant amount of objects was generated(we ended up with around 500 objects from different generating algorithms) we created a new Dex-Net database, calculated an SDF for each one of them and saved it in the database. After this, we

run grasp calculations on all of them for 3 different metrics: robust Ferrari-Canny, Force Closure, and Ferrari-Canny. This dataset was later used to create a GQ-CNN dataset that could have been merged with other GQ-CNN datasets such as Dex-Net 2.0.

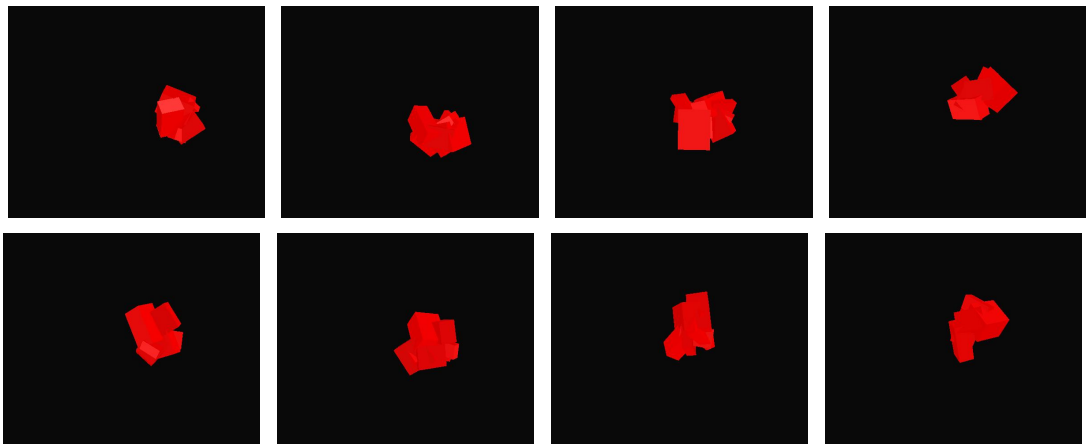

**Figure 4.9:** Some of the procedural objects we generated.

## 4.7 GQ-CNNs Training

After generating our datasets of more than 6 million data points of depth images, grasps details and metrics based on the Dex-Net 2.0 grasp database, it was time to start training our networks. We decided to start trying to replicate Dex-Net's results, therefore we wanted to use the parameters and network architecture described in the Dex-Net 2.0 paper. In the latest GQCNN release, the developers also shared their configuration script to train Dex-Net 2.0. In the original paper, the authors state the network was trained for 5 epochs, using an 80-20% train-validation image-wise split on the data, batch size 128, a momentum term of 0.9, and an exponentially decaying learning rate with step size 0.95. They also added Gaussian process image noise with a standard deviation of $\sigma = 0.005$ to each input image to try and make the network more robust. According to the paper, the GQ-CNN trained on all of Dex-Net 2.0 had an accuracy of 85.7% on a held out validation set of approximately 1.3 million data points. The configuration that was provided in the GQCNN repository was slightly different: the batch size was set to 64 and the network was being trained for 25 epochs. We decided to use the configuration in the GQCNN repository as we expected it would yield better results, even if the cost was longer training time.

For training, we initially had access to lab machines with GPUs(`Graphic`) and later to `Tiger`, a communal machine of the Robot Learning Lab. The `Graphic` machines in the labs support Nvidia CUDA-capable or AMD OpenCL-capable graphics cards. The GPUs capabilities vary quite a lot between different machines, ranging from 6GB Nvidia GeForce GTX Titan to an 8GB Nvidia GeForce GTX 1080. On the Robot Learning Lab `Tiger` machine we had two NVIDIA GeForce RTX 2080 GPUs

with 8GB of memory. The `Tiger` machine turned out to be a great help in training our Neural Networks. The lab machines are shared between all students in the Department, and we would often encounter problems with people shutting off the machines mid-training to kill our processes and use the machines for themselves. We tried solving this problem by writing scripts that would save a version of the network after each epoch of training, and notify us in case training had stopped, so training could be easily resumed from the last version to be saved. This method allowed us to train my networks for many epochs, even 20 or 25, which would have been very hard otherwise as it might have required multiple attempts. On the lab machines, training Dex-Net on all of its data points($32\times32$ pixels depth images, grasp information and binary metrics) for 25 epochs took around 28 hours. On the Tiger machine, the same task would take around 24 hours.

# 5 Testing and Evaluation on Dex-Net

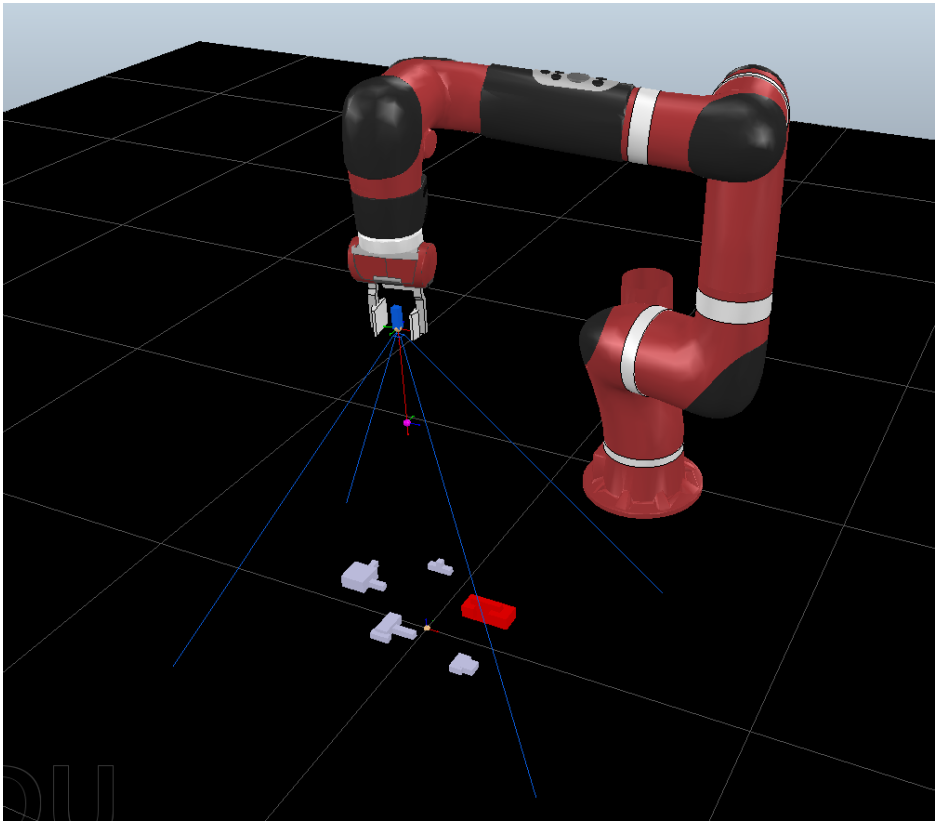

**Figure 5.1:** V-REP environment set-up. Some objects are randomly placed in the workspace area for demonstration purposes.

This chapter will discuss the new environment we built around the Dex-Net and GQ-CNN codebase to allow for quick and easy testing of different networks and policies. This new testing framework tries to automate and simplify evaluation as much as possible so that minimum human intervention is required and hundreds on grasps can be executed and their results accurately recorded with no need for any supervision.

The testing framework can evaluate different combinations of networks, policies, and sets of objects at once. It supports multiple testing configurations, such as grasping the same object multiple times or testing each object in multiple stable positions. Datasets of meshes to test on can be defined either as directories containing `.obj` files or as Dex-Net datasets.

We will start by discussing the metrics we chose to evaluate and compare different Grasp Quality Networks and policies. Following, we will describe our simulation set-up and custom environment.

## 5.1 Evaluation metrics

Our aim is to create different models, with different architectures and trained using different data or different amounts of data. We want to compare the performance of these models using different metrics to identify their weak and strong points and draw conclusions on which one would be most useful in a given situation. We aim to test our grasping models both in simulation and with a real robot and comparing performances. We expect to evaluate the model using the following metrics, some of which are inspired by those that the Dex-Net authors[27]:

- **Success Rate**: The percentage of grasps that were able to lift, transport, and hold the object after shaking.
- **Accuracy**: The percentage of correctly predicted grasp outcomes(above 50% probability is considered as a predicted success, below is considered as a predicted failure).
- **Precision**: The success rate of grasps that have an estimated robustness higher than a certain percentage(Dex-Net uses 50%).
- **Robust Grasp Rate**: The percentage of planned grasps with an estimated robustness higher than a certain percentage(Dex-Net uses 50% for this metric as well).
- **Planning Time**: The time in seconds between receiving an image and returning the planned grasp.

We will evaluate each model using both train and test data, in order to check for overfitting and assess generalization properties on novel objects. We will also compare the performance of our models against a more simple algorithm that doesn't use Deep Learning, to have a baseline to compare against. In particular, we plan on implementing a policy that would rank sampled grasps randomly rather than using a GQ-CNN.

## 5.2 V-REP

The Virtual Robot Experimentation Platform (V-REP) is a robot simulator that provides a unified framework combining many powerful internal and external libraries that are often useful for robotics simulations. This includes dynamic simulation engines, forward/inverse kinematics tools, collision detection libraries, vision sensor simulations, path planning, GUI development tools, and built-in models of many common robots[31].

Its Inverse Kinematics module is able to calculate a list of joint states to follow in order to move the tip of a robot arm to a target position along a desired path. The platform also possesses a remote API in various languages including Python. V-REP also offered a model of the robotic arm we wanted to use (Sawyer) in the default library and it was easy to control it from Lua scripts. V-REP provides a front-end

to a wide variety of physics engines that can be seamlessly interchanged. It also provides a nice interface allowing the user to create new objects, customize them and generally control many aspects of the simulation.

The GUI was very useful to start getting familiar with the simulation environment and its functionalities. The main scripting language V-REP uses is LUA, and not all of V-REP's features are available through the Python API. This meant some of the simulation's code had to be written in LUA, which I didn't have any familiarity with at the beginning of the project.

## 5.3 Environment Set-Up

We tried to keep our environment very simple to reduce complexity and aid processing speed. Our set up, show in Figure 5.1, includes a Sawyer robotic arm fitted with a Baxter gripper and two vision sensors to capture images(one for RGB images and one for depth images) positioned above the work-space area and pointed towards it.

## 5.4 Remote API

V-REP provides a large range of functions that enables communication for various actions in the simulated environment. Communication between our Python application (client) and V-REP (server) are done via socket communication. The server side is implemented via a plugin which we can communicate with via the Python bindings provided. Each object in the simulation has its own unique handle that can be obtained through specific API functions. Different objects have different properties which can be controlled. For example, we can retrieve images from vision sensors, move objects or set their velocities. Even though the number of functions offered by the Python API is quite limited, V-REP offers a more general purpose function, called `simxCallScriptFunction` that allows the user to call any function defined in LUA. By utilizing these remote API functions, the simulated scene can be prepared and manipulated completely in Python.

## 5.5 Physics Simulator Choice

V-REP's dynamic model offers 4 different physics engines: Bullet, ODE, Vortex and Newton[43]. V-REP allows the user to easily switch between the 4 through its GUI. Initially, we tried to use the default engine, ODE, but we had some issues obtaining accurate collisions and realistic object dynamics. This is why we decided to research all the available engines, test them and evaluate the pros and cons of each more in depth:

- **Bullet Physics Library**: Bullet physics is an open source physics engine. It supports real-time 3D collision detection, rigid and soft body dynamics. Bullet probably has the largest user base and most responsive community, it's very well documented and it's easy to find lots of examples, including some of the

robotic simulations. Unfortunately, collision checking and dynamics when using our meshes did not seem very accurate, so we decided not to use it.

- **Open Dynamics Engine(ODE)**: An open source, high performance physics engine. It supports advanced joint types and integrated collision detection with friction. As mentioned before, when we tested it on our simulation, we noticed the movements of the dynamic objects were not very accurate or realistic, with objects that were supposed to be still and resting on the workspace surface moving around or wobbling. Collision also seemed to cause problems and be fairly unreliable, as the gripper would enter the object if it was colliding with it for too long. This could have been due to our meshes not being very accurate.
- **Newton Dynamics**: The dynamics module allows simulating interactions between objects that are near to real-world object interactions.
- **Vortex Dynamics**: A closed source, commercial physics engine producing high fidelity physics simulations. Vortex offers real-world parameters (i.e. corresponding to physical units) for a large number of physical properties, making this engine both realistic and precise. The Vortex plugin for V-REP is based on Vortex Studio Essentials, which requires each user to register with CM Labs, for a free license key. We initially tried using the other engines as obtaining a license key wasn't straightforward and instructions to do so were either unclear or hard to find, especially on Linux. Unfortunately, since none of the other engines seemed to be accurate enough for our simulations or had unpredictable errors in handling collisions between the gripper and the objects, it was necessary to spend some time obtaining the license key and adding it to V-REP.

## 5.6 Vision Sensors

Vision Sensors can detect and render the renderable entities in their field of view. A vision sensor's image content can be accessed via the V-REP API. The sensors can be set to either perspective or orthogonal projection type. For our specific implementation, we set them to perspective mode as this is what Dex-Net uses by default, although it also offers support for orthogonal projection. The near and far clipping planes were set to 0.01 and 1 meters respectively and the perspective angle was set to 60 degrees. These parameters are those that yielded the most similar results to the original Dex-Net ones.

The sensor's image content is retrieved using the V-REP Python API. Then the depth image is modified so that the values at each pixel represent the distance from the camera in meters. This is done with the following code, where `near_clipping_plane` and `far_clipping_plane` represent the corresponding values in the camera configuration.

```
depth_image = near_clipping_plane
            + depth_image * (far_clipping_plane - near_clipping_plane)
```

**Listing 5:** Depth image is rescaled so that the value at each pixel represents the distance from the camera in meters

Both depth and color image are then flipped, as the rows are inverted from what the camera sees in the simulator.

## 5.7 Robotic Gripper

The Robotic Gripper is composed of various components: the main body, two antipodal fingers, and two joints. The gripper's functionality is fairly counter-intuitive, and took me some reading to figure out how to get it working correctly. Initially, I thought the two joints were set up so that each one would move one of the fingers either closer to the center or further away. In reality, only one of the fingers actually moves. We will call the two joints `close_joint` and `center_joint`. While the close `close_joint`'s goal is to move one of the two fingers(either towards the center or away from it), the `center_joint`'s objective is to keep the two fingers equally distant from the center. This means the `center_joint` simultaneously moves both fingers with respect to the body of the gripper while keeping the distance between them equal. This creates the illusion of both fingers opening or closing synchronously while the center of the grasp remains perfectly centered.

## 5.8 Collision Checking

V-REP can detect collisions between two collidable entities in a very flexible way. The collision detection module will only detect collisions but it won't cause any reaction between the colliding objects(the collision response is handled by the dynamics module). The collision detection functionality is illustrated in Figure 5.2. The collision detection module allows registering collision objects which are collidable entity-pairs (collider entity and collidee entity). Objects can be registered in the collision module and set up so that, during simulation, each collision can be easily visualized: the objects will change color when they collide with any of the objects specified in the collision module. For example, a robotic arm can be registered so that it only changes color when it collides with the table surface and doesn't when it collides with any other entity.

Collision can also be checked via the API, providing the object handle of each of the two objects we want to check. It is possible to pass `sim.handle_all` as one of the parameters to check if one object is colliding with any other object in the simulation. The function will return `True` in case of collision or `False` otherwise. Checking for collisions becomes complicated in case of complex objects formed by multiple shapes such as a robotic arm. If we want to check the collision of a robotic arm with another object, checking the collision of every single one of the shapes that

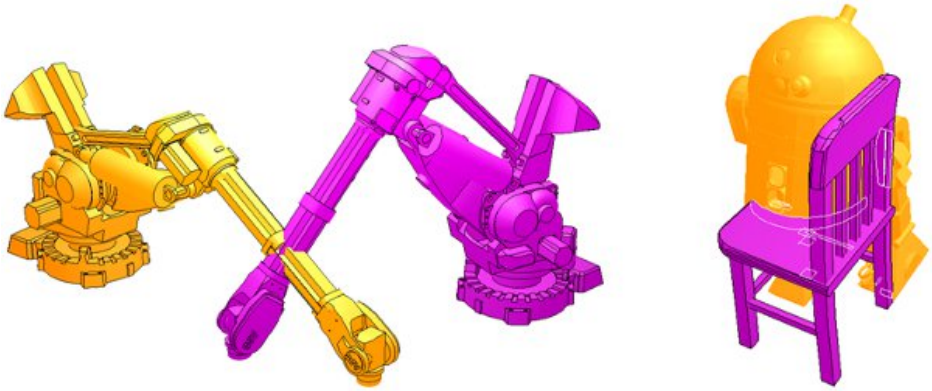

**Figure 5.2:** Simple collision detection between two manipulators(left) and exhaustive collision detection between two shapes(right)[42]

makes up the robotic arm would be tedious and time-consuming. Luckily, V-REP provides the option to create Collections, which are groups of shapes that can all be indicated using the same handle. This handle can then be passed as a parameter to the collision detection function as any other object handle.

But another problem arises: in some situations, such as inverse kinematics, we might need to check if components of the robotic arm are colliding with each other, as shown in Figure 5.3. When performing collision (or minimum distance) calculations between the same collection, V-REP will normally check all collection items against all other items in that collection. Some of these components will inevitably collide, such as two shapes connected by a joint, and we don't want to take those collisions into account as those are to be expected. V-REP provides a workaround to this problem: each shape has a collision self-detection indicator parameter, that can be set both from the V-REP GUI and API. Two items of a same collection will not be checked against each other if their indicator difference is exactly 1, as can be seen in Figure 5.3.

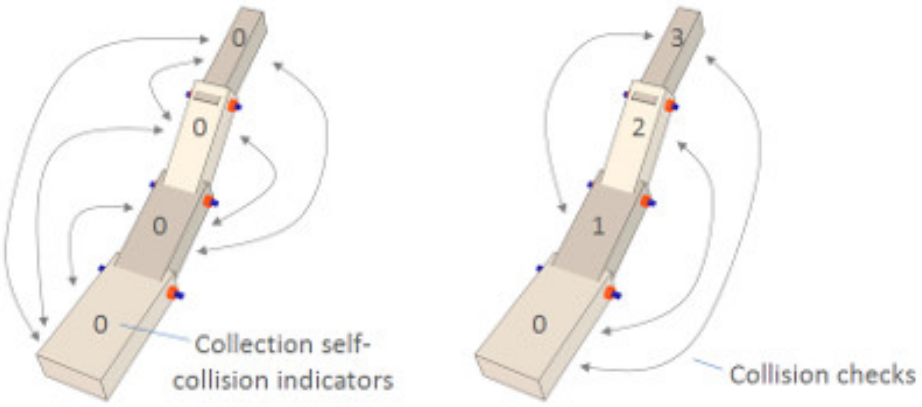


**Figure 5.3:** Collection self-collision indicators[44]

## 5.9 Inverse Kinematics

V-REP uses IK groups and IK elements to solve inverse kinematics tasks. V-REP provides a series of example scenes where either IK is used. This is useful to understand how they work, what they can be used for, and how to set up a similar scene. All their example scenes were in 2D. Although this helped with understanding, as the scenes were very simple and contained the bare minimum of necessary components, it meant we couldn't directly reuse their code for my simulation, which was in 3D. In V-REP, to solve Inverse Kinematics on a simple kinematic chain, we need:

- A **base** It represents the start of the kinematic chain. It can be any type of object, even a joint. In that case however the joint cannot move.
- Several **links**. A rigid part of the object that connects 2 joints
- Several **joints**
- A **tip**. The tip is always a dummy and is the last object in the kinematic chain. The tip dummy should be linked to a target dummy.
- A **target**. The target is always a dummy and represents the position and/or orientation the tip should adopt (or follow) during simulation.

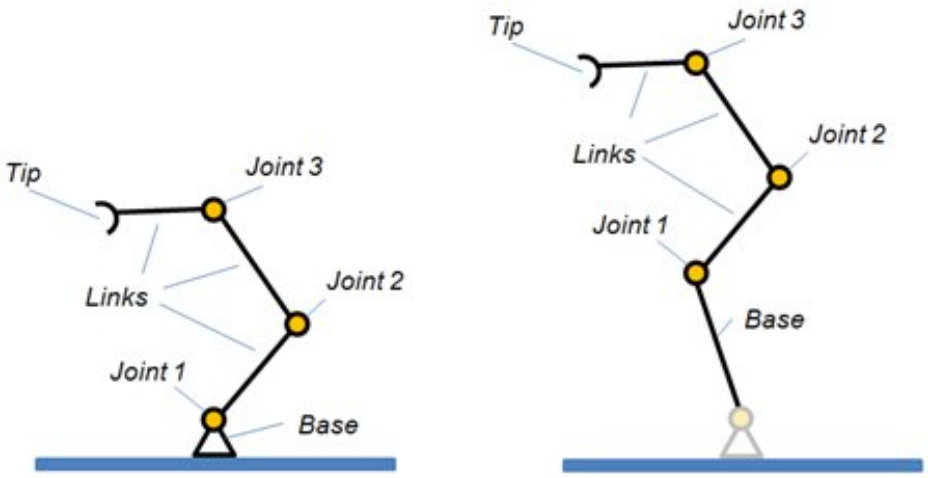


**Figure 5.4:** Two kinematic chains, each describing an IK element [41]

## 5.10 Evaluation Data

Once the simulation was set up and working we had to decide what datasets we wanted to evaluate our Grasp Quality Networks on. First, we decided to evaluate on the training data. This was simple enough as all we needed to do was retrieving meshes from the Dex-Net 2.0 database. Unfortunately, some of these meshes had quite a few orphan and duplicated edges, both of which can cause invalid collision

response during simulation. We also found cases of meshes with two adjacent edges sharing no vertex or only one vertex. As we noticed invalid collision happen multiple times during evaluation, we decided to try and fix these meshes by removing orphan or duplicated edges. The Dex-Net repository seemed to provide a couple of functions to do this, but they didn't seem to work. We eventually decided to just create a "black list" of defective meshes the simulation script would just script during evaluation.

For the test data, we decided to research some mesh datasets online. Unfortunately, most of the datasets we came across online were of very big and complex objects, from furniture to airplanes and engines. We also considered a dataset containing mesh representation of the objects used in the Amazon Picking Challenge dataset but many meshes, like those in the Dex-Net dataset, were neither accurate nor precise. Many objects were also too big for the width of our Sawyer gripper. Other mesh datasets we considered were the Princeton Shape Benchmark, ModelNet, Kit and ShapeNET. In the end, we picked a mix of different objects from the Princeton Shape Benchmark dataset and rescaled them if they didn't fit the width of our gripper. We also decided to test on a mix of the procedurally generated data we generated.

Summarizing, for the evaluation of grasp quality networks, we decided to evaluate performance on:

- **100 known objects** the network was trained on. Each object was evaluated 4 times.
- **100 unknown objects** the networks was not trained on, but were similar in shape and size to the known objects. Each object was evaluated once.
- **100 procedurally generated objects**. Each object was evaluated once.
- **200 randomly selected objects** from the Princeton Shape Benchmark Dataset. Each object was evaluated once.

In order to create a baseline, we evaluated each dataset using a random policy. This policy used the antipodal grasp sampling algorithm to generate candidate grasps and then selected a candidate randomly. The results for each dataset are summarized in Table 5.1.

| **Success Rate** | |
|---|---|
| Mesh Dataset | Random Policy |
| Known meshes | 78.0% |
| Unknown meshes | 92.0% |
| Procedural | 61.0% |
| Princeton ShapeNet | 78.5% |

**Table 5.1:** Baseline for each dataset. Data gathered using a random policy

## 5.11 Evaluation Script

The pseudocode to evaluate the different GQ-CNNs is shown in Listing 6. The testing framework can evaluate different combinations of networks, policies and sets of objects at once. It supports multiple testing configurations, such as grasping the same object multiple times or testing each object in multiple stable positions. Datasets of meshes to test on can be defined either as directories containing `.obj` files or as Dex-Net datasets.

```
for gqcnn in gqcnns:
  for dataset in datasets:
    objects = dataset.get_objects() # dataset can be hdf5 or directory
    for obj in objects:
      simulation.load_object(obj)
      # rescale the object according to which dataset we are using
      simulation.rescale(obj, dataset.scale_value)
      simulation.set_random_color(obj)
      if dataset.random_pose:
        simulation.set_random_pose(obj)
      else:
        pose = dataset.get_object_pose(obj, pose_idx)
        simulation.set_pose(pose)
      object_is_moving = True
      # wait for the object to stop moving
      while object_is_moving:
        object_is_moving = simulation.is_obj_moving(obj)
      depth_im, color_im = simulation.get_images_camera()
      # depending on the type of GQ-CNN, execute a different policy
      if gqcnn.random and gqcnn.color:
        grasp = RandomPolicy(color_im)
        q_value = 0
      elif gqcnn.random:
        grasp = RandomPolicy(depth_im)
        q_value = 0
      elif gqcnn.color:
        grasp, q_value = CEMPolicy(color_im, gqcnn)
      else:
        grasp, q_value = CEMPolicy(depth_im, gqcnn)
      grasp3D = grasp.deproject(simulation.get_camera_pose())
      # set IK target to over 0.2 meters above object
      simulation.set_IK_target(grasp3D.x, grasp3D.y, grasp3D.z + 0.2)
```

```
    # send gripper to IK target avoiding collision with table or itself
    simulation.IK_execute(collisions={table, robot})
    # send gripper straight down to IK target
    simulation.set_IK_target(grasp3D.x, grasp3D.y, grasp3D.z)
    simulation.IK_execute_straight_path()
    simulation.close_gripper()
    # lift the object
    simulation.set_IK_target(grasp3D.x, grasp3D.y, grasp3D.z + 0.6)
    simulation.IK_execute_straight_path()
    # grasp is successful if object is not colliding with the table
    success = not simulation.is_collision(obj, table)
    # save result
    log.save(gqcnn.name, obj.name, success, q_value)
```

**Listing 6:** Pseudocode for GQ-CNN performance evaluation and comparison.

# 6 Optimizing Dex-Net

In this Chapter, we will be discussing different approaches we tried in order to improve Dex-Net's performance and optimize training and computation times. Some examples include:

- Train Dex-Net on subsets of a complete dataset and compare performances
- Hyperparameter search to increase performance while decreasing the total number of parameters in the network
- Generate a new dataset containing grasp images of higher resolution and showing the whole object rather than just part of it, as in some of the images in the original dataset
- Train Dex-Net using Ferrari-Canny and Force-Closure as training labels instead of Robust Ferrari-Canny

## 6.1 Training on subsets of Dex-Net

We decided to try training different GQ-CNNs on different subsets of the Dex-Net's dataset to analyze how the network's performance would be affected by using different amounts of training data. We used scripts to generate datasets of different sizes, all starting from the initial dataset we generated(6.7 million datapoints, as in the original Dex-Net implementation). We created 4 datasets, containing 10%, 20%, 50% and 80% of Dex-Net's total data. We will call these networks GQ-CNN-10, GQ-CNN-20, GQ-CNN-50, GQ-CNN-80, and GQ-CNN-100 respectively. The split we used to create these datasets was object-wise, so each dataset contains the same amount of images for each combination of object, stable pose, and grasp, but contains fewer objects.

Figure 6.4 shows the performance comparison of the 5 different GQ-CNNs across different datasets. Taking aside the known objects dataset, the performance of the network steadily increases as more data is added to it. The exception of the known dataset could be easily explained noting that GQ-CNN-10 and GQ-CNN-20 are probably overfitting, performing better on these specific objects but worst on any other kind. The analysis of the unknown objects dataset is quite straightforward, as both the MSE and false negative rate decrease steadily as the training dataset is augmented. The false positive rate remains flat, but mostly due to the very low number of false positives across all GQ-CNNs.

In the bottom graphs of Figure 6.4, it is interesting to notice the increased amount of false negatives in the GQ-CNN trained on 100% of the data. This is likely due to the particular shapes and meshes contained in the Princeton Shape Benchmark

dataset. A good subset of the meshes contained in this dataset are meshes of plants, and often include complex representations of leaves and thin branches. GQ-CNN-100 rates all grasps on these objects poorly, but these grasps are often successful, as the meshes are static. Therefore, this doesn't indicate a shortcoming on the network's performance, but it highlights one of the challenges of evaluating in simulation.

## 6.2 Different quality metrics

The original Dex-Net GQ-CNN uses the Robust Ferrari-Canny metric to generate binary labels for each grasp. If the Robust Ferrari-Canny metric is above 0.002 the grasp is labelled as a robust grasp. We decided to train two more GQ-CNNs using Ferrari-Canny and Force Closure metrics. As Force Closure is a binary metric, we could use it directly. For Ferrari-Canny, we decided to threshold it at 0.007, as it seemed to give a similar ratio of good/bad grasps compared to the original Robust Ferrari-Canny implementation. Our evaluation confirmed Robust Ferrari-Canny proved to be the best metric to use, as the two new GQ-CNNs performed worse across all datasets, as shown in Table 6.1.

| **Original GQ-CNN** | | | | |
|---|---|---|---|---|
| Mesh Dataset | Success Rate | Accuracy | Robustness Rate | Precision |
| Known | 84.5% | 75.0% | 72.0% | 94.4% |
| Unknown | 99.0% | 92.0% | 91.0% | 100.0% |
| Procedural | 77.0% | 75.0% | 74.0% | 85.1% |
| Princeton | 88.5% | 52.5% | 56.5% | 85.8% |
| **GQ-CNN Ferrari-Canny** | | | | |
| Mesh Dataset | Success Rate | Accuracy | Robustness Rate | Precision |
| Known | 84.0% | 84.0% | 88.0% | 90.9% |
| Unknown | 96.0% | 91.0% | 93.0% | 96.8% |
| Procedural | 72.0% | 72.0% | 84.0% | 76.2% |
| Princeton | 84.0% | 72.0% | 84.0% | 83.3% |
| **GQ-CNN Force Closure** | | | | |
| Mesh Dataset | Success Rate | Accuracy | Robustness Rate | Precision |
| Known | 84.0% | 85.0% | 85.0% | 92.9% |
| Unknown | 94.0% | 95.0% | 95.0% | 96.8% |
| Procedural | 68.0% | 68.0% | 94.0% | 69.1% |
| Princeton | 87.0% | 86.0% | 86.9% | 99.0% |

**Table 6.1:** Comparing results of the original GQ-CNN trained using the Robust Ferrari-Canny metric to label grasps vs GQ-CNN using the Ferrari-Canny and Force Closure metrics. Metrics used for comparison include Success Rate, Accuracy, Robustness Rate and Precision.


**Figure 6.4:** Results for success rate, false positive, false negative rate and mean squared error for multiple GQ-CNNs trained on different subsets of Dex-Net, more specifically 10, 20, 50, 80 and 100% of the total data(this excludes 20% of the data used for validation and testing). The dotted green line in the success graphs represents the success rate of a random policy, implemented as a random selection of one of the possible antipodal grasps. The false positive rate is shown in green, the false negative rate is shown in red. The percentage of data the GQ-CNN were trained on is shown on the $x$-axis of the graphs.

## 6.3 Higher Resolution Images

In our evaluation of the Dex-Net 2.0 GQ-CNN we noticed that, for big objects in particular, the network didn't seem to have a strong preference for grasps closer to the center of mass. We hypothesized this was because the images we were training the network with were relatively small and often cut parts of the objects out of the image. This could have prevented the network from being exposed to important information such as the overall shape of the object, preventing it from extrapolating any information from this data and, instead, leaving it to focus only on a small part of the object. This was only our hypothesis, and the only way to test it was to generate a dataset of images with more context on each object and then train a network with it.

The images in the original Dex-Net dataset were generated by cropping the rendered depth images in smaller 96×96 pixels images around the grasp axis and then reducing the resolution to 32×32 pixels. We decided to crop our images to include twice as much context as the original images in each direction, so we cropped the images at 192 pixels for both height and width. We tried to resize the images at 32×32 pixels as the original dataset, but looking at the results this didn't seem to be enough: the images appeared very blurry and it was impossible to recognize any detail. We reasoned that our images should have the same amount of detail as the originals, just with more context. To achieve this, we resized our images to obtain a final resolution of 64×64 pixels.

While this was certainly the best way to test how increased context was going to affect network performance, we decided to save the images in a 96×96 pixels resolution. This way we would have been able to also consider how increased resolution would affect performance. We choose a 96×96 pixels resolution specifically because this is what the authors of Dex-Net used to train their Dex-Net 4.0 GQ-CNN(this network differs from Dex-Net 2.0 as it is specifically trained to work in a cluttered workspace containing multiple objects).

## 6.4 GQ-CNNs Architecture

### 6.4.1 Original Dex-Net 2.0

The architecture of the GQ-CNN described in the original Dex-Net 2.0 contains 4 convolutional layers separated by ReLU nonlinearities, followed by 3 fully connected layers and a separate input layer for the z, the distance of the gripper from the camera. The filter dimensions of each convolutional layer are 7, 5, 3 and 3 respectively. All convolutional layers have 64 filters and SAME padding(this means the output image of each convolution is the same size as the original image, this is achieved by adding padding around the image). A max pooling operation is performed after the second and fourth convolutional layer. A representation of this network is shown in Figure 6.5. The architecture defined in the configuration file released with the GQCNN repository is very similar to what was described above, the only difference is the addition of an extra max pooling after the fourth convolutional layer.

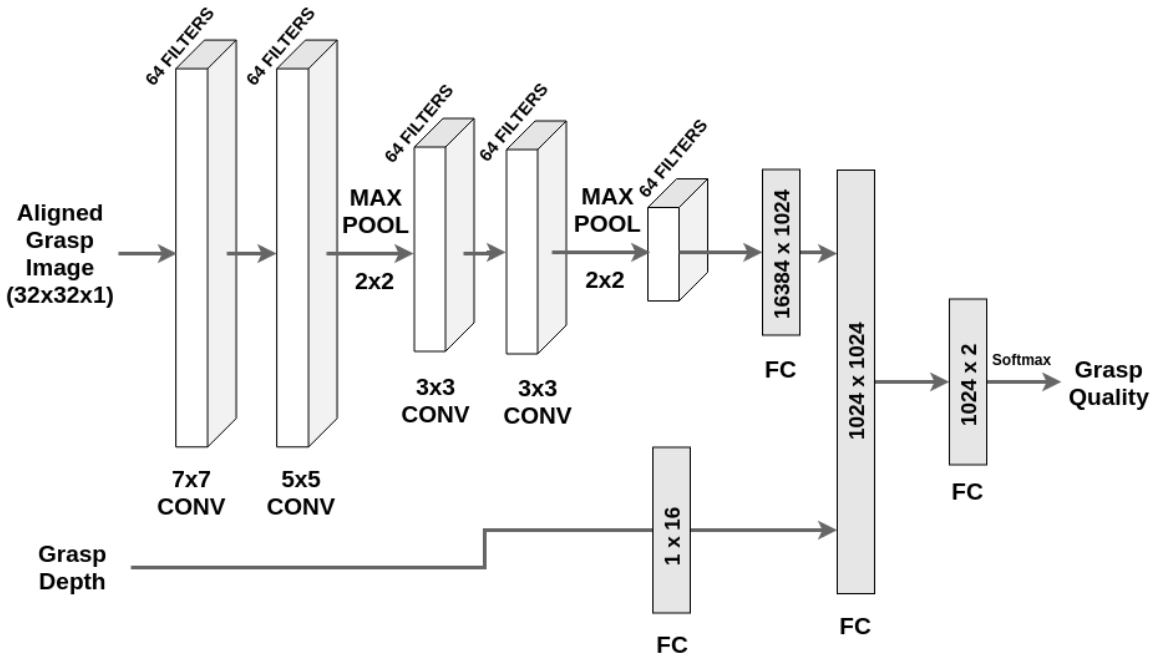


**Figure 6.5:** Original GQ-CNN architecture.

### 6.4.2 96×96 Dex-Net 2.0

For the images with 96×96 pixels resolution, we decided to design the network in the same way Mahler el al[28] did in their Dex-Net 4.0 paper. The GQ-CNN architecture is similar to that used in Dex-Net 2.0 with a few important changes. First, local response normalization was removed. Second, the filter dimension changed: from 7, 5, 3, 3 to 9, 5, 5 and 5 respectively. Also, Dex-Net 4.0 uses 16 filters for each convolutional layer, while Dex-Net 2.0 uses 64. The size of the fully connected layers was changed as well: the output size decreased from 1024 to 128. The architecture is shown in Figure 6.6.

According to the paper, the GQ-CNN was trained using stochastic gradient descent with momentum for 50 epochs using an 80-20 training-to-validation image-wise split of the Dex-Net 4.0 dataset. The learning rate was set to 0.01 with exponential decay of 0.95 every 0.5 epochs, a momentum term of 0.9, and an l2 weight regularization of 0.0005. Other than the number of epochs we trained for(25 instead of 50), we used the same hyper-parameters as the Dex-Net 4.0 to train our GQ-CNN. The results are summarized in Table 6.2 and compared with the performance of the original GQ-CNN of Dex-Net 2.0. The new network seems to perform equally well in most datasets while performing slightly better on the known objects and slightly worst on the unknown ones.

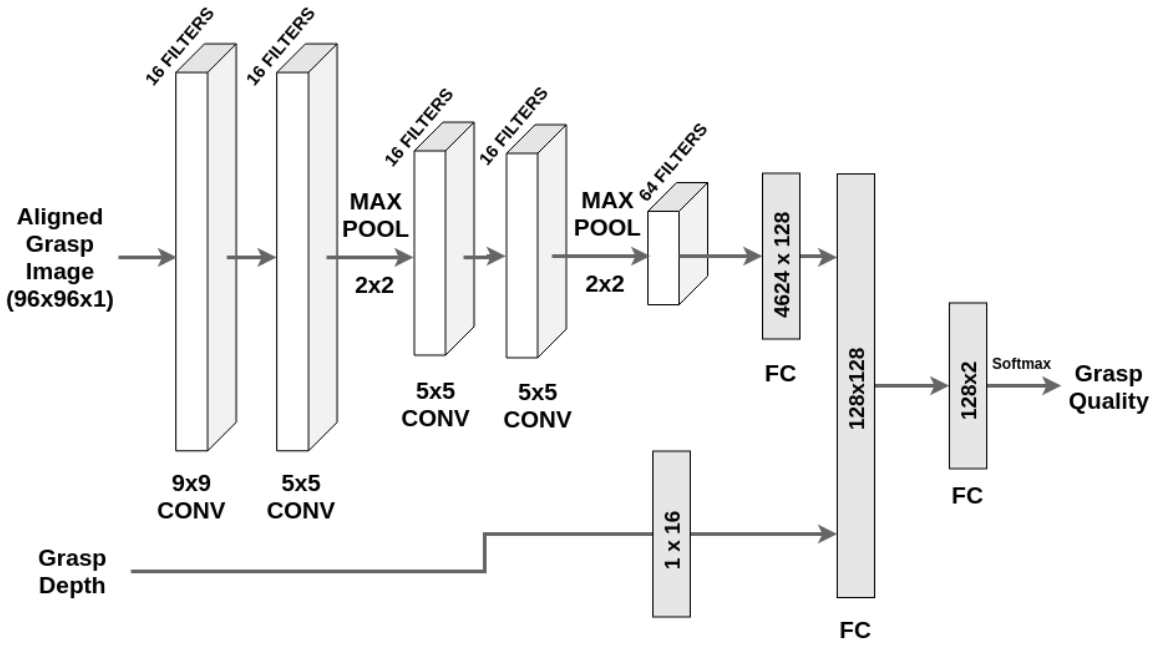


**Figure 6.6:** GQ-CNN architecture for training on 96×96 pixels depth images.

| **Original GQ-CNN** | | | | |
|---|---|---|---|---|
| Mesh Dataset | Success Rate | Accuracy | Robustness Rate | Precision |
| Known | 84.5% | 75.0% | 72.0% | 94.4% |
| Unknown | 99.0% | 92.0% | 91.0% | 100.0% |
| Procedural | 77.0% | 75.0% | 74.0% | 85.1% |
| Princeton | 88.5% | 52.5% | 56.5% | 85.8% |
| **GQ-CNN 96×96** | | | | |
| Mesh Dataset | Success Rate | Accuracy | Robustness Rate | Precision |
| Known | 87.0% | 61.0% | 56.0% | 96.4% |
| Unknown | 94.0% | 78.0% | 76.0% | 97.4% |
| Procedural | 77.0% | 71.0% | 70.0% | 84.3% |
| Princeton | 89.0% | 36.0% | 28.0% | 96.4% |

**Table 6.2:** Comparing results of the GQ-CNN trained on 96×96 pixels images and original Dex-Net 2.0 GQ-CNN. Metrics used for comparison include Success Rate, Accuracy, Robustness Rate, and Precision.

## 6.5 Hyperparameter Search Results

Since the GQCNN codebase provided an easy to use framework for hyperparameter search and network analysis, we decided to take advantage of it to try and improve the performance of Dex-Net. Starting from the Dex-Net hyperparameters, we decided to test some variations in filter number, dimensions, and batch size and analyze the performance of the resulting networks over both test and validation data. An-

other factor we considered when we decided to explore different architectures was the total number of trainable parameters in the network. The Dex-Net 2.0 GQ-CNN currently has over 18 million parameters. Most of these parameters are contained in the 2 fully connected layers that follow the convolutional layers, more specifically, almost 17 million parameters in the first fully connected layer.

Decreasing the total number of parameters could make the network faster to train and less prone to overfitting. It would also make grasp evaluation faster, therefore decreasing the time required to locate an optimal grasp. For the original Dex-Net architecture, the result of the analysis is represented in Table 6.3. To save time

| **Analysis of Original GQ-CNN** | | | |
|---|---|---|---|
| Train Err | Train Loss | Val Err | Val Loss |
| 9.703% | 0.289 | 9.850% | 0.513 |

**Table 6.3:** Analysis of original GQ-CNN architecture

and make our search more efficient, we decided to train each network for 2 epochs during our hyperparameter search, and only train the best performing ones for more epochs. First, we decided to test for batch size, it turned out the best performing network was the one using a batch size of 64, which was the one Dex-Net already used. This experiment was also useful as we could record the performance of Dex-Net after being trained for 2 epochs. The results detailed results are shown in Table 6.4. Next, we trained using different number of filters for each convolution. The

| **Hyperparam Search over Batch Size** | | | | |
|---|---|---|---|---|
| Batch Size | Train Err | Train Loss | Val Err | Val Loss |
| 16 | 15.469% | 0.3480 | 15.443% | 0.8310 |
| 32 | 15.338% | 0.3470 | 15.326% | 0.8240 |
| **64** | 14.402% | 0.3240 | 14.358% | 0.7720 |
| 128 | 15.668% | 0.2770 | 15.687% | 0.8440 |

**Table 6.4:** Analysis on GQ-CNNs with the same architecture as the original Dex-Net GQ-CNN, only difference is batch size. The parameter in bold indicates the original GQ-CNN configuration.

original Dex-Net architecture uses 64 filters for all its convolutional layers. We tried to change the number of filters to 8, 16 and 32. The best performing network turned out to be the one with a filter size of 32 for all convolutional layer. The results of the experiment are summarized in Table 6.5. Finally, we decided to test different convolutional filters dimensions. Smaller filters have a smaller receptive field and are usually better suited for highly local features that don't need much context. The image size decreases slowly, and this makes smaller filters more suitable for deeper networks, where the extracted information could be useful in later layers. Larger filters are more suitable for extracting more generic features spread across the image. They also contain a higher number of parameters, making computation and training less efficient.

| **Hyperparam Search over Number of Filters** | | | | |
|---|---|---|---|---|
| Number of Filters | Train Err | Train Loss | Val Err | Val Loss |
| 8 | 15.902% | 0.3043 | 15.863% | 0.8413 |
| 16 | 14.922% | 0.3263 | 14.884% | 0.7894 |
| 32 | 14.227% | 0.3523 | 14.154% | 0.7507 |
| **64** | 14.436% | 0.3361 | 14.415% | 0.7645 |

**Table 6.5:** Analysis on GQ-CNNs with the same architecture as the original Dex-Net GQ-CNN, only difference is number of filters. The parameter in bold indicates the original GQ-CNN configuration.

To start, we decided to use the same number of convolutional layers as the original Dex-Net architecture. We also used the same number of filters as the original architecture(64). From the results of our experiments, the architecture used by Dex-Net(filter sizes of (7, 5, 3, 3)) was one of the worst performing ones. In general, bigger convolution sizes seemed to result in worst performance. The best performing network, across all the metrics we considered, was the one with filter sizes (5, 3, 3, 3). The results of our experiments are summarized in Table 6.6. After considering

| **Hyperparam Search over Conv Size** | | | | | | | |
|---|---|---|---|---|---|---|---|
| conv1 | conv2 | conv3 | conv4 | Train Err | Train Loss | Val Err | Val Loss |
| 9 | 7 | 5 | 3 | 14.762% | 0.3348 | 14.742% | 0.7665 |
| 9 | 7 | 3 | 3 | 15.372% | 0.3400 | 15.381% | 0.7997 |
| 9 | 5 | 5 | 3 | 14.420% | 0.3418 | 14.401% | 0.7488 |
| 9 | 5 | 3 | 3 | 14.779% | 0.3190 | 14.777% | 0.7683 |
| 9 | 3 | 3 | 3 | 14.095% | 0.3014 | 14.110% | 0.7169 |
| 7 | 7 | 5 | 3 | 14.459% | 0.3132 | 14.491% | 0.7742 |
| 7 | 7 | 3 | 3 | 14.640% | 0.3507 | 14.642% | 0.7825 |
| 7 | 5 | 5 | 3 | 14.318% | 0.3595 | 14.340% | 0.7663 |
| **7** | **5** | **3** | **3** | 15.004% | 0.3268 | 14.978% | 0.8001 |
| 7 | 3 | 3 | 3 | 14.496% | 0.4441 | 14.551% | 0.7393 |
| 5 | 5 | 5 | 3 | 14.195% | 0.3395 | 14.171% | 0.7309 |
| 5 | 5 | 3 | 3 | 14.377% | 0.3365 | 14.299% | 0.7375 |
| 5 | 3 | 3 | 3 | 13.889% | 0.2803 | 13.900% | 0.7062 |
| 3 | 3 | 3 | 3 | 14.657% | 0.2843 | 14.656% | 0.7459 |

**Table 6.6:** Analysis on GQ-CNNs with the same architecture as the original Dex-Net GQ-CNN, only difference is the size of the convolutions. The parameter in bold indicates the original GQ-CNN configuration.

the results of our experiments, we decided to train a network for 20 epochs using 32 filters for each convolutional layer and (5, 3, 3, 3) as filter sizes. Mainly due to the reduction in the number of filters, the total number of parameters decreased to 3 millions. The architecture of the network is shown in Figure 6.7, while the results of the analysis are summarized in Table 6.7. Training this network on the full Dex-Net 2.0 dataset for 20 epochs took only 8 hours, compared to the almost 30 hours

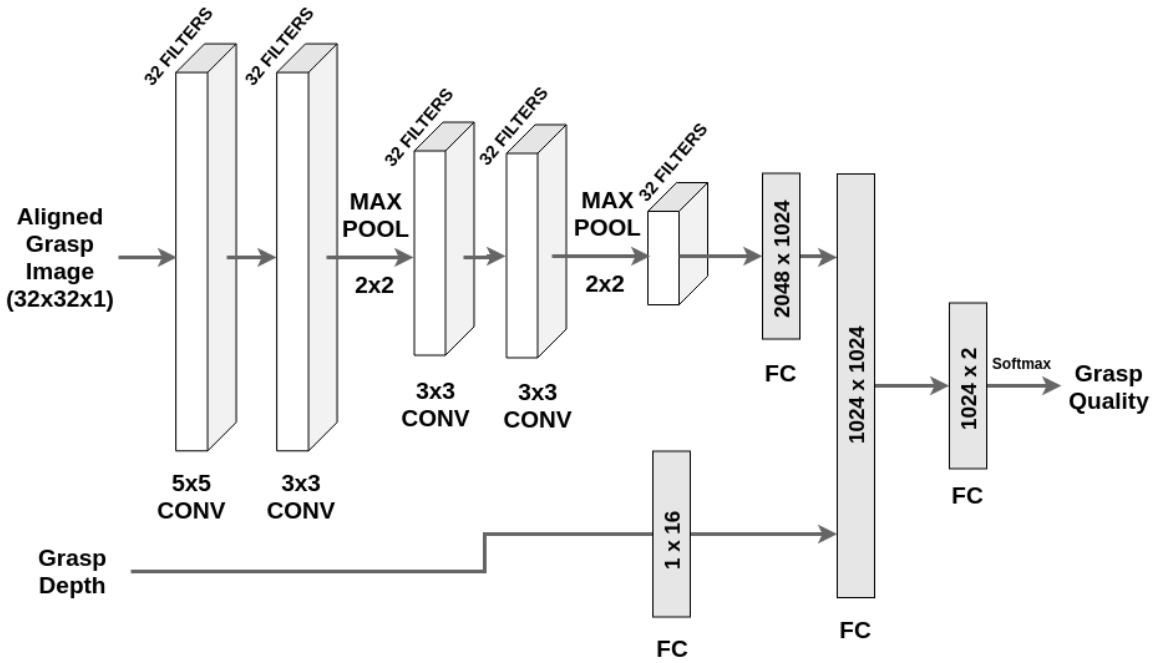


**Figure 6.7:** Architecture of new GQ-CNN. This network contains a total of 3 million parameters, compared to the 18 million parameters of the original GQ-CNN.

| New GQ-CNN 3M Params Analysis | | | |
|---|---|---|---|
| Train Err | Train Loss | Val Err | Val Loss |
| 11.047% | 0.341 | 11.164% | 0.581 |

**Table 6.7:** Analysis of new GQ-CNN architecture

it would usually take for the original Dex-Net architecture(for 25 epochs). We also decided to test the new GQ-CNN in simulation.

As the result show, the new GQ-CNN with reduced number of parameters performed just as well as the original architecture in grasping known objects. It performed slightly worst in capturing unknown but similar objects to the one we used during training but it performed considerably better on procedurally generated objects and objects from the Princeton Shape Benchmark dataset. This stands to show on how the original architecture was probably overfitting to a specific type of objects.

| **Original GQ-CNN** | | | | |
|---|---|---|---|---|
| Mesh Dataset | Success Rate | Accuracy | Robustness Rate | Precision |
| Known | 84.5% | 75.0% | 72.0% | 94.4% |
| Unknown | 99.0% | 92.0% | 91.0% | 100.0% |
| Procedural | 77.0% | 75.0% | 74.0% | 85.1% |
| Princeton | 88.5% | 52.5% | 56.5% | 85.8% |
| **GQ-CNN 3M Params** | | | | |
| Mesh Dataset | Success Rate | Accuracy | Robustness Rate | Precision |
| Known | 84.5% | 68.0% | 72.0% | 96.7% |
| Unknown | 96.0% | 83.0% | 83.0% | 97.6% |
| Procedural | 81.0% | 78.0% | 67.0% | 94.0% |
| Princeton | 91.0% | 43.0% | 42.0% | 90.5% |

**Table 6.8:** Comparing results of the 2 GQ-CNNs in simulation. The percentage shows the overall success rate for a given set of objects. Metrics used for comparison include Success Rate, Accuracy, Robustness Rate and Precision.

# 7 Beyond Dex-Net

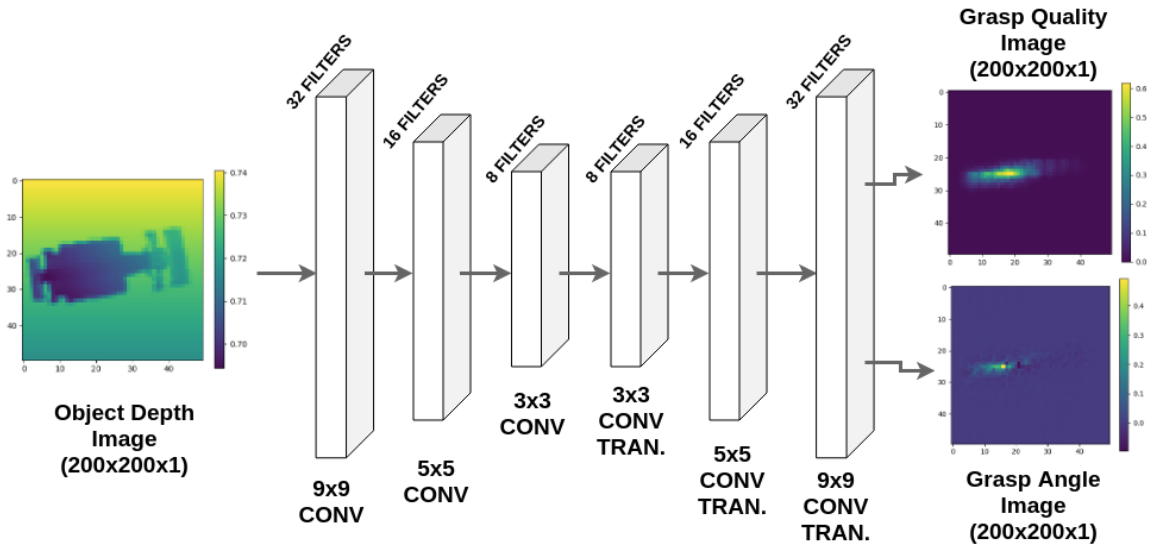


**Figure 7.1:** Our GQ-AE network. The architecture is mostly inspired by the Grasp Generation CNN proposed by Morrison et al.[30] in their research.

In this chapter we will analyze novel ways to make use of the Dex-Net dataset with the ultimate goal of training Grasp Quality Neural Networks for faster and more accurate robust object grasping.

In particular, we will implement a new type of Grasp Quality Neural Network, to then evaluate and compare it with the GQ-CNNs we trained previously. Once the architecture of our new Grasp Quality Neural Network is defined, we will also face the challenges of creating new custom datasets to allow effective training.

## 7.1 Grasp Quality Auto-Encoder(GQ-AE)

Although Dex-Net's GQ-CNN architecture and CEM policy proved to be a very effective approach, identifying successful grasps correctly in more than 90% of cases, it tends to be quite slow. The antipodal grasp sampling algorithm (described in detail in Section 3.9.4) can take between 0.2 and 0.4 seconds to return candidate grasps, depending on the parameters chosen and the image to sample. The GQ-CNN takes an average of 0.2 seconds to evaluate the quality of one 32×32 image(on a 2015 Mac Book Pro) but it's also worth considering each image needs to be cropped, rotated and scaled before being fed into the GQ-CNN. This also takes a considerable amount of time(around 0.01 seconds).

Considering this process needs to be repeated for multiple candidate grasp, and coupled with the antipodal sampling and CEM policy, the whole algorithm takes, on average, between 2 and 4 seconds to return a single grasp. Another drawback of the CEM and grasp antipodal sampling policies is that they must be executed in a serial manner, as each iteration is an improvement on the previous one.

Moreover, as Satish et al[50] noted in their research, evaluating each grasp using a GQ-CNN requires significant computational overhead, particularly when copying data between device and host memory every time the network is queried for a new batch of predictions. Given the time required to identify a successful grasp, it is impossible to implement any kind of closed-loop grasp execution, and precise robot control is fundamental to execute the grasp successfully.

Another downside to the GQ-CNN approach is network evaluation: beyond classic measures such as loss and accuracy(or MSE) over the training and validation set, or manually analyzing the convolutional filters of the network, it is sometimes difficult to tell if the network was trained correctly or if it will react unexpectedly to a new object. The only way of checking is evaluating over vast amounts of data, and even if we come across a problem or an unexpected behaviour, it's hard to identify its root cause and proposing possible solutions.

Another approach, that as not been explored much in research, is to create a network able to return a grasp quality value and gripper pose(depth and angle) for each possible pixel of an image(RGB, depth or RGB-D). This means the network could possibly output 3 images for each input image: a grasp quality image, a grasp angle image, and a grasp depth image.

To be clear, by grasp quality image we mean an image where each pixel has a value between 0 and 1, depending on the predicted quality of the best grasp centered at that pixel(1 being a good grasp and 0 being a bad grasp). Only the best grasp is considered in the grasp quality image because a grasp centered at a certain pixel(or more precisely its corresponding 3D point in the world coordinate frame) can have multiple orientations(grasp angles) or grasp depths, and each pose can have different quality values.

The grasp quality image could be interpreted as representing a probability of grasp success according to different levels of uncertainty in object shape, object position, gripper position, and gripper orientation. The grasp quality image could also represent a value of robustness for a given grasp, such as a Ferrari-Canny metric, or a combination of uncertainty and robustness(Robust Ferrari-Canny). A network able to output a grasp quality, grasp angle and grasp depth images given an RGB, depth or RGB-D image as input, could be easily implemented with an architecture similar to an Auto-Encoder. Because of this, we will refer to it as a GQ-AE(Grasp Quality Auto-Encoder) for here onwards.

The main reason why this approach has not been researched in depth yet is the lack of training data and the difficulty in data generation or data gathering. Training a network such as the one described above would require a vast number of grasp quality images(at least thousands). This is extremely time-consuming to obtain with real-world experiments, as well as in simulation.

## 7.2 Data Generation

### 7.2.1 Challenges

To explain the difficulties in generating data suitable for training a GQ-AE network we will present a simple example: if we consider an image resolution of 200×200 pixels, a camera set up and object size similar to the ones used by Dex-Net, the object would probably take up an area of around 50×50 pixels. Although objects can have different shapes and the overall area can change, this is a reasonable average. For this example, we will also not consider any uncertainty, and our grasp quality image will contain values between 0 and 1 for each pixel, representing a higher probability of grasp success(value closer to 1) or failure(value closer to 0).

To generate the grasp quality image, we first set the quality of each pixel that is not part of the object to 0(making the assumptions grasps not centered on the object will fail, this is not necessarily true, but will simplify calculations). We then need to test the grasp position corresponding to each pixel that is part of the object and test every possible angle. We only need to test for angles in the range [0-179] degrees since opposite angles in parallel gripper grasps are equivalent. If we consider a fixed depth, creating a grasp quality image would require us to evaluate around 447k different grasps. If we wanted to check angles every 5 degrees, we would have to evaluate 87k grasps. This number could be further reduced by checking fewer angles or not checking every single pixel, but checking only one and assuming a small region of neighbouring pixels will have the same quality value, but this could potentially reduce the validity of our data. Regardless, it is easy to understand how creating a single one of these grasp quality images is not at all straightforward, and would require a significant amount of computations.

Previous research [30, 55], when faced with this same problem, decided to generate human-labelled datasets or make use of pre-existing ones, such as the Cornell Dataset[13]. The Cornell Grasping Dataset contains 885 RGB-D images of real objects, with 5110 human-labelled positive and 2909 negative grasps. Although this is certainly a valid solution, the size of these datasets is incredibly small, especially if compared to the 6.2 million datapoints dataset created using Dex-Net. Another solution, proposed by the some of the same researchers that worked on the Dex-Net project[50], avoids the problem entirely by training a GQ-CNN on individual grasps and then converting all fully connected layers into fully convolutional layers, thus creating a Fully Convolutional Network. This eliminates the need for densely-labelled ground-truth images during training.

### 7.2.2 Possible Solutions

Given the number of necessary evaluations for creating a single grasp quality image, we immediately ruled out real life or simulation data gathering. The synthetic, physics-based data from the Dex-Net database could not be used either, as each image only represents the quality of one possible grasp (and not each possible grasp on the object). Even though the Dex-Net algorithm for sampling data from a mesh is very efficient and accurate, it doesn't provide a way of enforcing that each area

of the object's mesh contains at least one grasp. In particular, not all bad grasps are saved, such as the ones where the distance between the two contact points, is greater than the gripper's maximum width. Furthermore, even if every point on the mesh was sampled, this did not mean every point on the image would be evaluated, as only grasps parallel to the table for a certain gripper pose would be added to the image.

Regardless, we decided to use the depth images generated by the `meshrender` library as our starting point. Particularly, the images generated for each object and stable pose, randomized over different camera poses. These were the $400\times400$ pixels depth images from which the smaller images (representing single grasps) used to train the GQ-CNNs were generated. We resized these images to a $200\times200$ pixels resolutions, hoping to cut down on computation time while maintaining reasonable image quality.

#### Using a GQ-CNN

A possible idea we considered to generate our GQ-AE dataset was to use the Dex-Net GQ-CNN to evaluate every possible grasp on a depth image. As mentioned above, we decided to limit the number of possible grasps to evaluate by only checking angles every 10 degrees and only checking pixels that were part of the object. Even considering these optimizations, the creation of each image took a few minutes. Considering this, it was unfeasible to use this method to generate thousands of images. Furthermore, any GQ-AE trained on this data would have been unable to learn anything more than the GQ-CNN, therefore limiting the learning potential of our network.

To further speed up the process, we decided to try first add to the image all the grasps already present in the Dex-Net database. As this did not bring any significant improvement(on average, we had 5-10 grasps for each image), we decided to set an area of 9 pixels(a 3x3 square centered at the grasp center) for each one of the grasps obtained from the Dex-Net database. This further reduced computation time to an average of 40 seconds per image. Depending on the size of the network we intended to create, this computation time was still too high.

After careful consideration, we decided to create a dataset using the second method(combination of grasps from Dex-Net and GQ-CNN), but on only one image per object and stable pose combination, therefore ignoring camera randomization. This allowed us to create a dataset of more than 13k datapoints.

**Optimization** It is worth mentioning the process of image creation is very easily parallelizable. This can be achieved in multiple ways: either having multiple processes evaluating different parts of the same image or by having multiple processes working on different images simultaneously. We did try to implement the latter by using Python's `multiprocessing` library, but it did not seem to speed up the process significantly on my laptop, probably due to overheating. Given the lack of `sudo` access on the lab machines, it was impossible for me to install the required libraries to retrieve grasps from the Dex-Net database. This meant that if we were to parallelize

my data generation across multiple machines, even though we would have been able to crop, rotate, resize and evaluate images using the GQ-CNN, we could not use the grasps present in the Dex-Net database. Because of this, we had to use my laptop to render images using the `meshrender` library, retrieve possible grasps from the Dex-Net database, project them according to the camera position, and save them as 2D grasps in a `.npz` file. This way, the lab machines were able to combine the results of the grasps from Dex-Net and the GQ-CNN grasp evaluation on the depth image in one grasp quality image, without having to access the Dex-Net database or rendering depth images.

#### Partial Grasp Quality Images using Dex-Net

As the creation of grasp quality images dataset through GQ-CNN evaluation took a few days, we decided to explore other options in the meantime. The main problem we had, as explained above, was obtaining accurate grasp quality values for each pixel in the image. On the other hand, it was relatively fast and straightforward to obtain partial grasp quality images. First, every pixel in the image would be set to represent an "unknown" grasp value(we used the -1 value for this) or 0(indicating bad grasp quality), then we would add all the grasp qualities contained in Dex-Net to the image. This could be done by either setting the value of a single pixel, centered at the grasp center, or by setting it together with its neighbouring pixels. Even though this approach generated very high quality, accurate images, it did leave big gaps of unset grasp quality values on the object's surface(the "unknown" values mentioned above). Generating a dataset of about 660k of these images would take between 1 and 3 hours, depending on which grasps we were saving, how and in what order.

## 7.3 GQ-AE Loss Function

The partial grasp quality images dataset was probably the most accurate and complete dataset we would have been able to generate with the given time limits and hardware. However, these images were not suitable to be used as a training set for a traditional auto-encoder, using a loss function such as cross-entropy(CE) or mean squared error(MSE) between predicted image and training image.

As a workaround, we considered creating a network able to learn on partial data from a variety of different objects, hoping it would then be able to "fill in the gaps" by applying the knowledge learned by the partial samples provided. In order to do this, we set any grasp quality value we didn't know to -1. Then, we created a custom loss function that would ignore those values when comparing the predicted image to the provided training image, while applying a traditional loss function(such as BCE or MSE) over all the other known grasp quality values. This meant the network should have been able to learn only from relevant data, ignoring any gaps(marked as -1 in the images).

## 7.4 Pytorch

Even though the previous networks were implemented using Tensorflow, I had much more experience using Pytorch, and I had used it to build auto-encoders before. Pytorch is a Python package that provides two high-level features[32]:

- Tensor computation (like `numpy`) with GPU acceleration
- Deep neural networks built on a tape-based autograd system

Pytorch also offers `torch.nn`, a neural networks library and `torch.utils`, a library offering a variety of utility functions, including Database and DataLoader classes. `torch.nn` also offers multiple pre-implemented, highly optimized loss functions that can be easily integrated in the NN's architecture. These are the main reasons why we decided to implement the GQ-AE network using Pytorch.

## 7.5 GQ-AE architecture

For our GQ-AE, we decided to use a classic auto-encoder architecture. To decide on the initial hyperparameters, filter sizes and the number of layers of the network, we mostly looked at the architecture used by Morrison et al. [30] for their Generative Grasping Convolutional Neural Network and the network architectures of our GQ-CNNs. For the encoder part, we had 3 convolutional layers, with convolution filters of sizes 9, 5 and 3 respectively. For the decoder part, we had a mirrored version of the encoder, using transpose convolution filters of sizes 3, 5 and 9. For our loss function, we tried using cross entropy, binary cross entropy and mean squared error. The final architecture for our GQ-AE network is summarized in Figure 7.1.

## 7.6 Training

As we created new datasets, we tried training the GQ-AE right away to see if any of them proved to be effective. We went through many iterations and kept improving our dataset creation algorithm until the GQ-AE started learning effectively from the data provided. After many iterations and trials, we created over 10 datasets of more than 660k datapoints each. We started out by focusing on the grasp quality images and verifying the network was predicting those correctly. Once we verified this was the case, we moved onto testing different methods for generating training datasets of grasp angle images.

### 7.6.1 Grasp Quality Images Datasets

The main options we had to define when creating our datasets were:

- Only setting pixel values corresponding to examples of robust grasps or setting pixel values corresponding bad grasps as well

- Setting the pixel value to the exact metric of the grasp or use binary labels
- How to set the value of the pixels we did not have a corresponding grasp for. The options we considered were to either set all these pixels to represent an unknown value or only set pixels belonging to the object to represent an unknown value, while all the others would be set to the minimum grasp quality value(0).
- For each retrieved grasp, we could either set only the value of the exact pixel corresponding to it or set the pixel and an area around it. If we decided to set a neighbouring area, we also had to decide how big we wanted this area to be.
- Loss function used. Each loss function requires the data it's used on to have certain characteristics. The binary cross-entropy loss function only works with binary data, while the cross-entropy loss function requires a different channel for each class.

### Unknown-Default, Regression Dataset

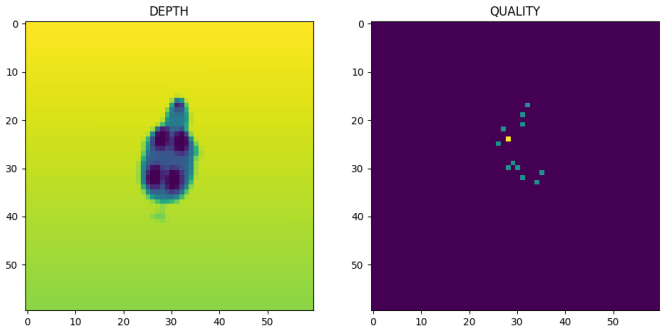


**Figure 7.2:** Sample datapoint in Unknown-Default, Regression Dataset.

For the first dataset, we set all pixels on the image to -1 to begin with. This way, no assumption is made on pixels belonging to the object compared to pixels that do not. We then retrieved all the grasps from the Dex-Net database for that specific object, stable pose and randomized camera position and set the pixels corresponding to the center point of each grasp to the Robust Ferrari-Canny metric divided by 0.002 and capped at 1. This meant the values of each pixel was either -1(unknown) or a value between 0 and 1. Before setting the grasps, we would sort them from worst to best, according to their Robust Ferrari-Canny metric, so that if multiple grasps were laying on the same pixel we would only record the best one. An extract of the algorithm used to create this dataset is shown as pseudocode in Listing 7.

```
quality_image = np.full([200,200,1], -1.0)
grasp_angle_image = np.full([200,200,1], 0.0)

metrics_grasps = sorted(zip(metrics,grasps), key=lambda x: x[0])
```

```
for metric, grasp in metrics_grasps:
  x = int(grasp[0] / 2)
  y = int(grasp[1] / 2)
  if (metric >= 0.002):
    quality_image[x, y, 0] = 1
  else:
    quality_image[x, y, 0] = metric / 0.002
```

**Listing 7:** Pseudocode of algorithm for creation of Unknown-Default, Regression Dataset.

Unfortunately, when we tried training the GQ-AE with this data(using an MSE loss function), the network seemed to be unable to learn and predicted mostly random values for both grasp quality and angle images. We hypothesized this could be caused by the high amount of unknown(-1) pixels, the loss function, or the high amount of negative examples compared to the positive ones.

### Unknown-Default, Classification Datasets

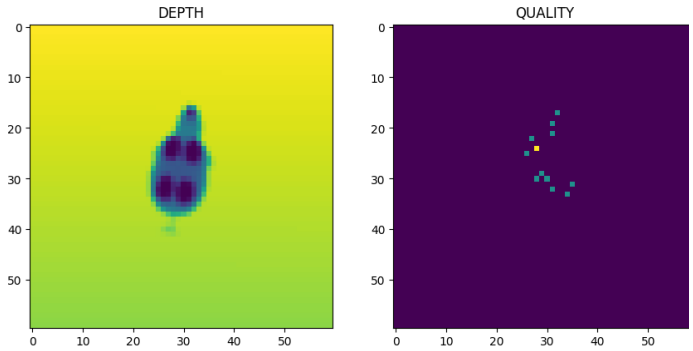


**Figure 7.3:** Sample datapoint in the Unknown-Default, Classification Dataset

Before trying to lower the amount the unknown(-1) pixels, we decided to make the problem simpler by reducing it to a classification problem and use a different loss function, such as BCE and CE. BCE requires all pixels to be either 0 or 1, so we had to create a classification dataset. This new dataset was created using the same logic as the Low Information, Regression Dataset, but using a binary grasp quality value. The value was set to 1 if the grasp Robust Ferrari-Canny metric was above 0.002, or set to 0 otherwise.

When trained using this dataset, the results seemed very similar to the previous Low Information, Regression Dataset, proving the loss function of the network was probably not the cause of the problem. The cross-entropy loss function, on the other hand, requires a quality image for each class, so we had to create an image with multiple channels. Each image had 2 channels for each pixel: if the grasp Robust

Ferrari-Canny value was above 0.002, the second channel of the pixel centered at the image center was set to 1, otherwise, the first value of the first channel was set to 1. Unfortunately, the GQ-AE failed to learn properly even with this dataset and kept returning random values for most pixels.

### Positive Classification Dataset

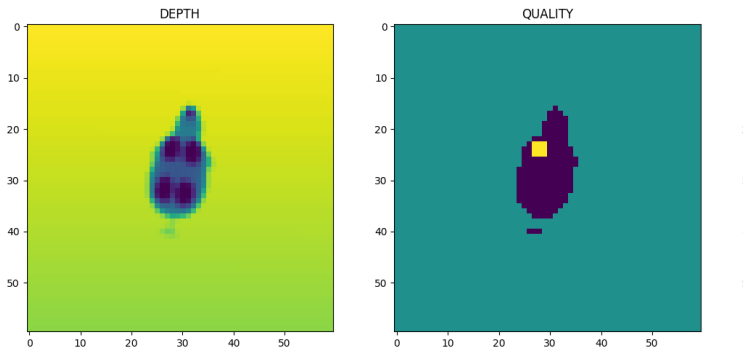


**Figure 7.4:** Sample datapoint in Positive Classification Dataset.

As the network was failing to distinguish possible good grasps on the object from the general background, we decided to help the network by making the assumption that all pixels not part of the object had quality 0. All pixels part of the object are set to -1(unknown). The grasps from the Dex-Net database are retrieved and sorted from worst to best according to their Robust Ferrari-Canny metric value. When we set each grasp value on the image, we also set the neighbouring area. So instead of a single pixel, we set a 3x3 area. Only grasps with a value above 0.002 are recorded in the image, other grasps are ignored. For grasps with Robust Ferrari-Canny metric value above 0.002 we set the value in the image to 1. This means the image will have either -1, 0 or 1 values.

After training this network for 30 epochs, we noticed the network started to predict some plausible grasp quality images around the 15th epoch. Unfortunately, after that, it went back to predicting mostly noise. From this, would could deduce setting the background to a default value of 0 had definitely helped, but we still had to make some more improvements.

```
binary_im = binary_im.resize((200, 200))
quality_image = binary_im.raw_data/255 * -1

grasp_angle_image = np.full([200,200,1], 0.0)

metrics_grasps = sorted(zip(metrics,grasps), key=lambda x: x[0])

for metric, grasp in metrics_grasps:
```

```
x = int(grasp[0] / 2)
y = int(grasp[1] / 2)

if (metric >= 0.002):
  quality_image[x-1:x+2, y-1:y+2, 0] = 1
```

**Listing 8:** Pseudocode of algorithm for creation of Positive Classification Dataset.

### Classification Dataset

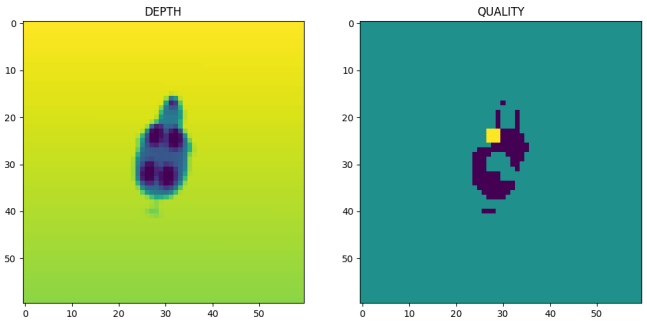


**Figure 7.5:** Sample datapoint in Classification Dataset.

We created this dataset in the same way we create the Positive Classification Dataset, but this time grasps with Robust Ferrari-Canny metric value less than 0.002 were also recorded on the grasp quality image. These value of the pixel area corresponding to these grasps was set to 0 so, similarly to Positive Classification Dataset, the grasp quality images created had values of either -1, 0 or 1.

After training the network for 30 epochs, we could see the network started correctly predicting grasp quality images from the input depth images starting from epoch 10, and kept improving and becoming come accurate as the training continued.

### Summary

The similarity and differences between the 6 datasets are summarized the the Table below.

| Dataset | Grasp Area | Bkgd Value | Bad Grasps | Class. | Regr. |
|---|---|---|---|---|---|
| Unknown-Default, Regr. | 1 px | -1 | ✓ | ✗ | ✓ |
| Unknown-Default, Class. | 1 px | -1 | ✓ | ✓ | ✗ |
| Positive Classification | 3×3 px | 0 | ✗ | ✓ | ✗ |
| Classification | 3×3 px | 0 | ✓ | ✓ | ✗ |

**Table 7.1:** Summary of Grasp Quality Datasets characteristics

### 7.6.2 Grasp Angle Images Datasets

After succeeding in training a network able to reliably predict grasp quality images from depth images, we started considering the options we had for generating datasets of grasp angle images. Some of the variables we considered were:

- Either setting the whole image to represent an unknown value by default, setting it all to 0 by default or only setting the pixels belonging to the object to represent unknown value, while setting everything else to 0.
- Either setting only pixels corresponding to good grasps or setting examples of both good and bad grasps
- Either setting an area for each pixel corresponding to a grasp or only the exact pixel

#### Zero-Default, All Grasps Angle Dataset

We started by setting all pixels in the image to a value of 0. We then retrieved all the grasps from the GQ-CNN dataset, sorted them by metric value(from worst to best) and set the $3\times3$ areas around each pixel to the value of the grasp angle. After training, although this method seemed to still return reasonable grasp quality images, it predicted angles very close to 0 for all grasps.

#### Unknown-Default, All Grasps Angle Dataset

We started by setting all pixels in the image to -1(unknown). Then we proceeded to set the grasp angle values with the same method we used while creating the Zero-Default, All Grasps Angle Dataset. Unfortunately, even after 30 epochs, the network didn't converge and could not even predict grasp quality images accurately.

#### All Grasps Angle Dataset

For this dataset, we set pixels belonging to the object to represent an unknown value and all other pixels to 0. Then we set the grasp angle values with the same method we used while creating the two previous grasp angle datasets. Unfortunately, even in this case and after 30 epochs of training the network did not converge.

## 7.7 Finetuning

At this point, we decided to try a different strategy. First, we trained the model for 30 epochs on grasp quality images and all 0s grasp angle images, then we changed the grasp angle dataset to the All Grasps Angle Dataset, while keeping the same network weights.

This allowed the network to properly learn grasp angles. Even though the results were satisfactory, we noticed the network would often try to predict an angle that

represented an average of good and bad grasps angles, resulting usually in a failed grasp when executed on a real object.

This is why we decided to apply the same strategy but on a different grasp angle dataset. We created a new grasp angle dataset using the same method we applied to the All Grasps Dataset, but only set grasp angles of good grasps. We will call this dataset the Positive Grasps Angle Dataset. The network we finetuned using this grasp angle dataset proved to be our best performing GQ-AE model. Some example predictions of our best performing GQ-AE model are shown in Figure 7.6.

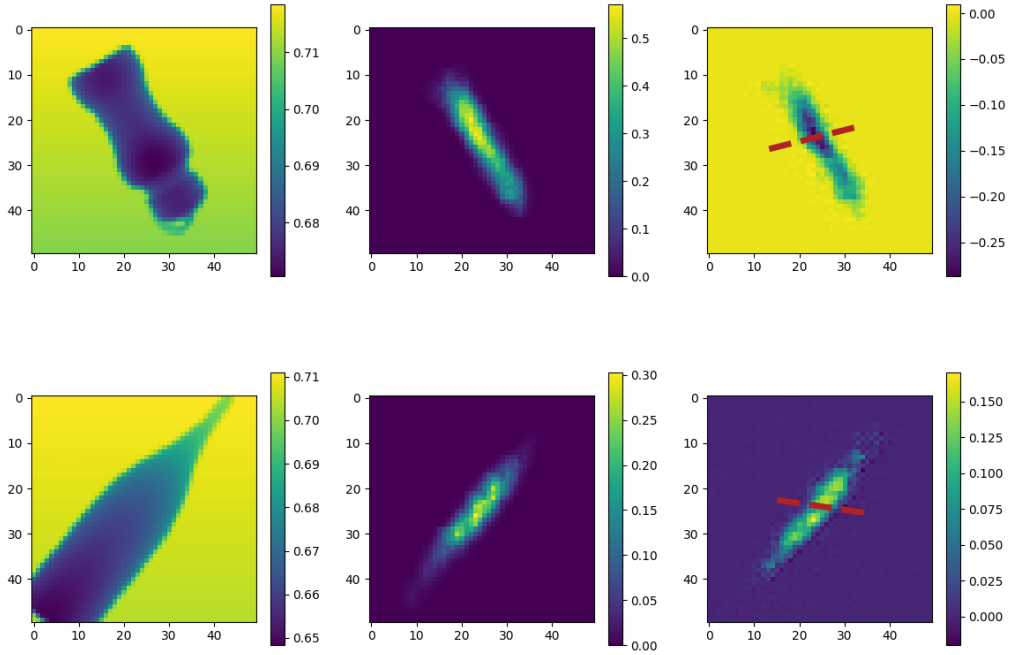

**Figure 7.6:** Results of our best performing GQ-AE, predicting grasp quality image (center) and grasp angle image(right) from a depth image(left). The best predicted grasp pose in shown using the red dotted line.

## 7.8 Evaluation

For the evaluation of our GQ-AE networks, we used the same metrics and testing framework introduced in Chapter 5. It is important to note that for this evaluation, the baselines we calculated for all datasets are no longer applicable. This is because the GQ-AE no longer makes use of the Antipodal Grasping Policy, and the GQ-AE network is the only component of our grasp selection process. With an Antipodal Grasping Policy(described in detail in Subsection 3.9.4), even if grasp are ranked randomly, there is still a good chance the "best" grasp could be robust, as we have seen when creating baselines for our datasets, where the success rate was always above 60%.

The Dex-Net implementation doesn't rely solely on a grasp quality neural network as we do, an uses a grasp sampling algorithm to propose candidate grasps. Furthermore, our implementation is, on average, 100 times faster than the original method proposed by Dex-Net. Our GQ-AE seemed to perform better than the

| **Original GQ-CNN** | | | | |
|---|---|---|---|---|
| Mesh Dataset | Success Rate | Accuracy | Robustness Rate | Precision |
| Known | 84.5% | 75.0% | 72.0% | 94.4% |
| Unknown | 99.0% | 92.0% | 91.0% | 100.0% |
| Procedural | 77.0% | 75.0% | 74.0% | 85.1% |
| Princeton | 88.5% | 52.5% | 56.5% | 85.8% |

| **GQ-AE** | | | | |
|---|---|---|---|---|
| Mesh Dataset | Success Rate | Accuracy | Robustness Rate | Precision |
| Known | 81.0% | 74.0% | 63.0% | 93.7% |
| Unknown | 96.0% | 86.0% | 90.0% | 95.6% |
| Procedural | 65.0% | 61.0% | 52.0% | 75.0% |
| Princeton | 34.0% | 60.0% | 38.0% | 42.1% |

| **Speed** | | |
|---|---|---|
| Random Policy | Original GQ-CNN | GQ-AE |
| 0.1-0.3s | 2-4s | 0.02-0.04s |

**Table 7.2:** Performance comparison between different approaches to the grasping problem: the Dex-Net GQ-CNN approach, that uses a sampling algorithm and then ranks them using a GQ-CNN, and our new GQ-AE approach. Our GQ-AE doesn't rely on an antipodal sampling algorithm and returns a possible grasp for each pixel in a depth image, together with a quality value, the grasp with the highest quality value is selected and executed. Our GQ-AE is about 100 times faster than the original Dex-Net implementation.

GQ-CNN on small irregular objects(Figure 7.7) such as planes, where the Dex-Net GQ-CNN seemed to struggle with. On the other hand, our GQ-AE had difficulties grasping very thin objects, as shown in Figure 7.8. This could be caused by a lack of robust grasp examples for this type of object, and the problem could be easily solved by generating some more. Additionally, we noticed our GQ-AE would occasionally fail to grasp objects due to incorrect gripper depth. This is because we did not include depth information about the grasp when training our network, this could be easily fixed by adding this additional information.

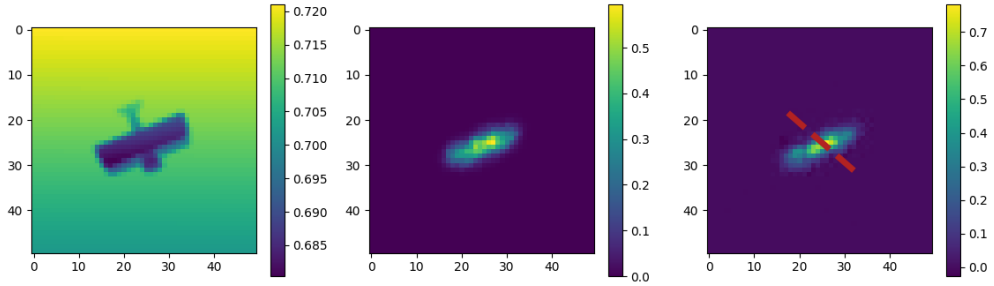

**Figure 7.7:** Example of object our GQ-AE grasped more reliably than Dex-Net GQ-CNN.

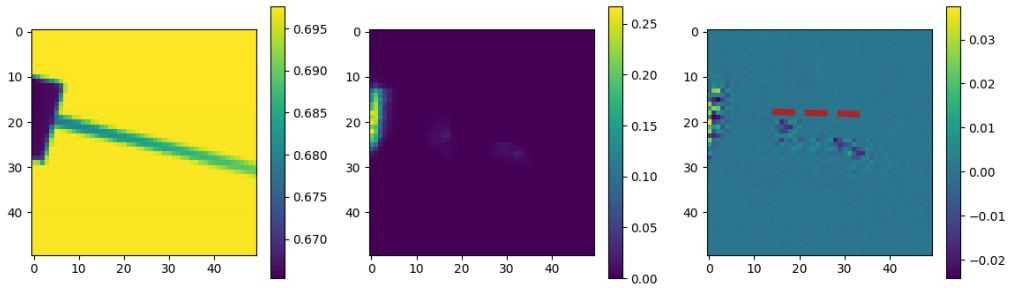

**Figure 7.8:** Example of failure case for our GQ-AE. We noticed the model seemed to struggle predicting accurate grasp quality and angle on thin objects.

# 8 From Simulation to Real World

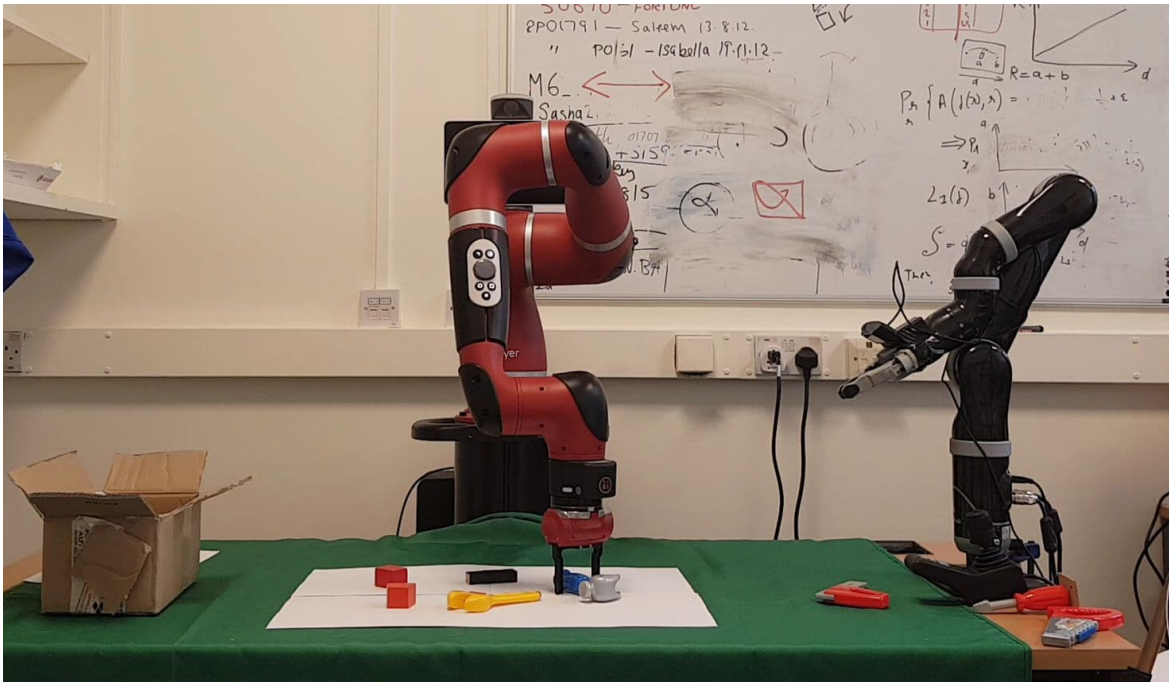

**Figure 8.1:** The picture shows our set up for evaluating our grasping models in the real world.

After thoroughly evaluating and testing our networks in simulation it came time to test on a real robot. The robot used was the same we used in simulation: a Sawyer robotic arm. This robot features a 7 degree of freedom robot arm with a 1260 mm reach. The robotic arm also features 2 cameras: one close to the main screen at the top of the robot and another one placed on the arm(wrist camera), closer to its end effector. To be able to control the robot and execute grasping tasks, we had to implement a script able to interact with ROS(Robot Operating System) and send the appropriate commands. By using the ROS network layer API, any client library or program that interact with ROS and control the robot directly.[46]

## 8.1 Intera SDK

Sawyer offers the Intera SDK, a software interface that helps developers to create applications for their robots. The SDK interfaces with the robot via ROS. The robot can be controlled by connecting to its stand-alone ROS Master through the ROS APIs. One of the main features of the Intera SDK is the interface provided to control and access all of the robot's motors and sensors. In addition, the Intera Interface provides

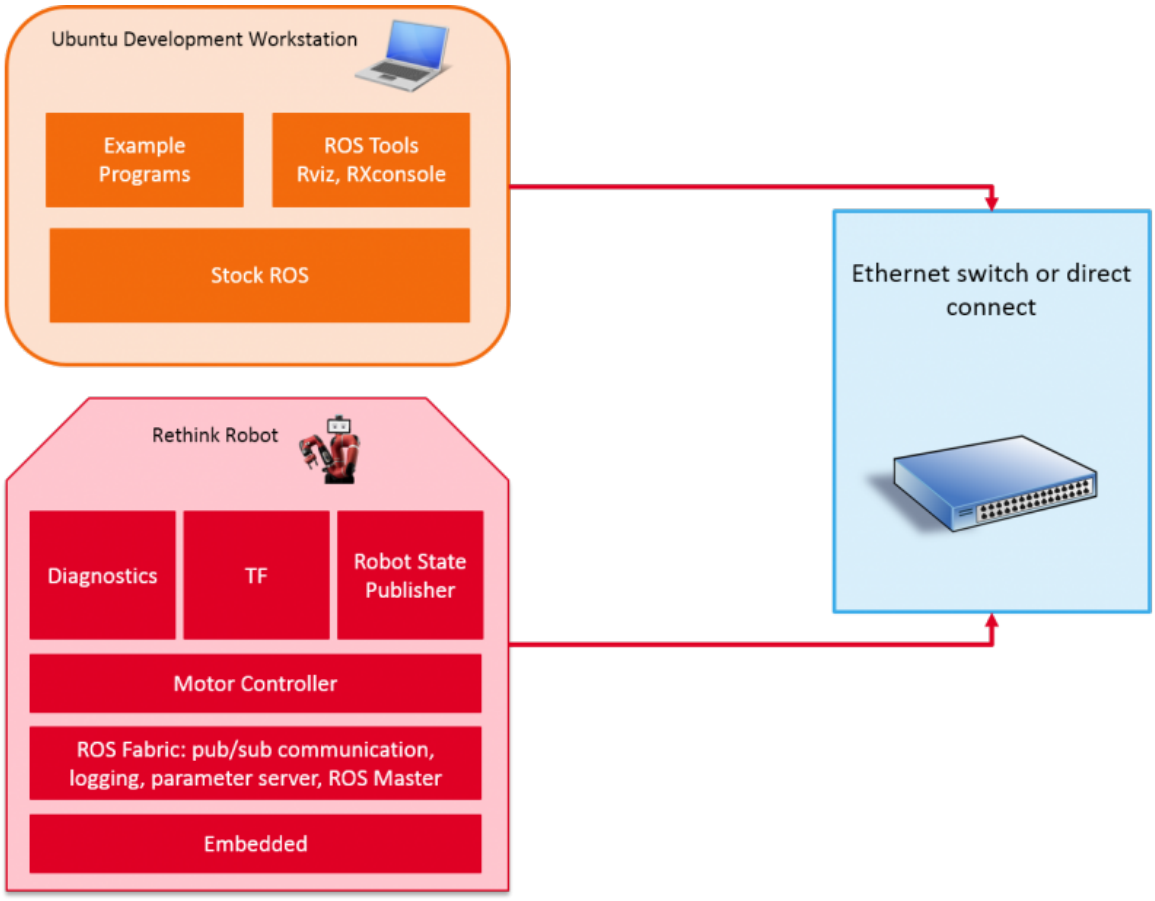


**Figure 8.2:** Intera SDK system overview.[45]

a Python Class-based interface library, which wraps many of the ROS interfaces in Python Classes. First, we installed and set up the ROS workspace on our Ubuntu machine connected to the robot. Then, after testing the robot with a couple of the example scripts provided by the Intera SDK library, we started implementing our own script.

## 8.2 Cameras and Lighting

Our robot had to be able to take a picture of the scene from the camera placed on its arm and send it to the Dex-Net CEM policy script. The policy script would sample possible grasps and evaluate them using our GQ-CNN network, as explained in Section 3.9.4. The script would then return a grasp, defined using its pixel coordinates on the image and its grasp angle. From this information, and knowing the intrinsic parameters of the robot's camera, together with its position relative to the robot's own coordinate frame, the point can be deprojected to 3D space and transformed to obtain its 3D coordinates and gripper orientation(described as a quaternion) relative to the robot's coordinate frame. We would then use Intera's IK service to calculate the angles of each one of the 7 robotic joints in the given position. We decided to

| Wrist Camera | |
|---|---|
| Description | Spec |
| Camera Resolution | 752 x 480 |
| Lens Type | Wide Angle |
| Chromo | Grayscale |

| Head Camera | |
|---|---|
| Description | Spec |
| Camera Resolution | 1280 x 800 |
| Lens Type | Wide Angle |
| Chromo | RGB |

**Table 8.1:** Sawyer embedded camera specifications [46]

implement the grasping execution similarly to how we implemented it in simulation: we would first place the robot hoovering 20cm above the desired grasp position so it could move down in an almost perfect straight line. The gripper would then close and the robot would lift the object and place it in a box.

Even though the implementation of the script itself didn't present many difficulties and the robot was relatively easy to work with, we immediately noticed some problems with the images captured by the wrist camera. Depending on the light conditions in the room, their quality would vary greatly, often going from completely white to completely dark images.

Luckily, the Intera SDK offers a way to regulate the exposure of the image. Adapting the exposure value to the lighting conditions proved to be effective and we managed to obtain very clear images of the workspace. We also found out this camera doesn't support RGB and can only capture greyscale images. This further complicated the adaptation process, as our policy was suddenly not able to distinguish objects from the background, even if they were very different colors. In order to minimize shadows, we used a green matte table cloth, but due to the very limited contrast between the objects and the green background, we had to change our strategy. We used A4, white pieces of paper to create a uniform background, as we hoped they would create images with higher contrast. This trick worked very well, as long as we avoided light colored objects(yellow or white). The downside to this approach was that, unfortunately, the white paper did not minimize shadows as well as the green table cloth did.

Regardless of this, the shadows didn't seem to affect the network to a point where it was unable to differentiate good grasps from bad ones in most cases. The most common issue the shadows caused was offsetting the center of the grasps slightly towards the side where the shadow was cast, causing some grasps to collide with the object or miss it entirely.

## 8.3 Coordinate Frames

When called, the CEM policy returns the $(x, y)$ coordinates, in pixels, of the best grasp it has found on the given depth or RGB image, together with the grasp angle. These $(x, y)$ pixel coordinates had to be deprojected into 3D space. The Dex-Net codebase implements a function to perform this operation and returns a 3D point relative to the camera coordinate frame. In order to transform the grasp position from the camera coordinate frame to the robot coordinate frame we had to understand how both frames were defined, so we could derive their relative transformation

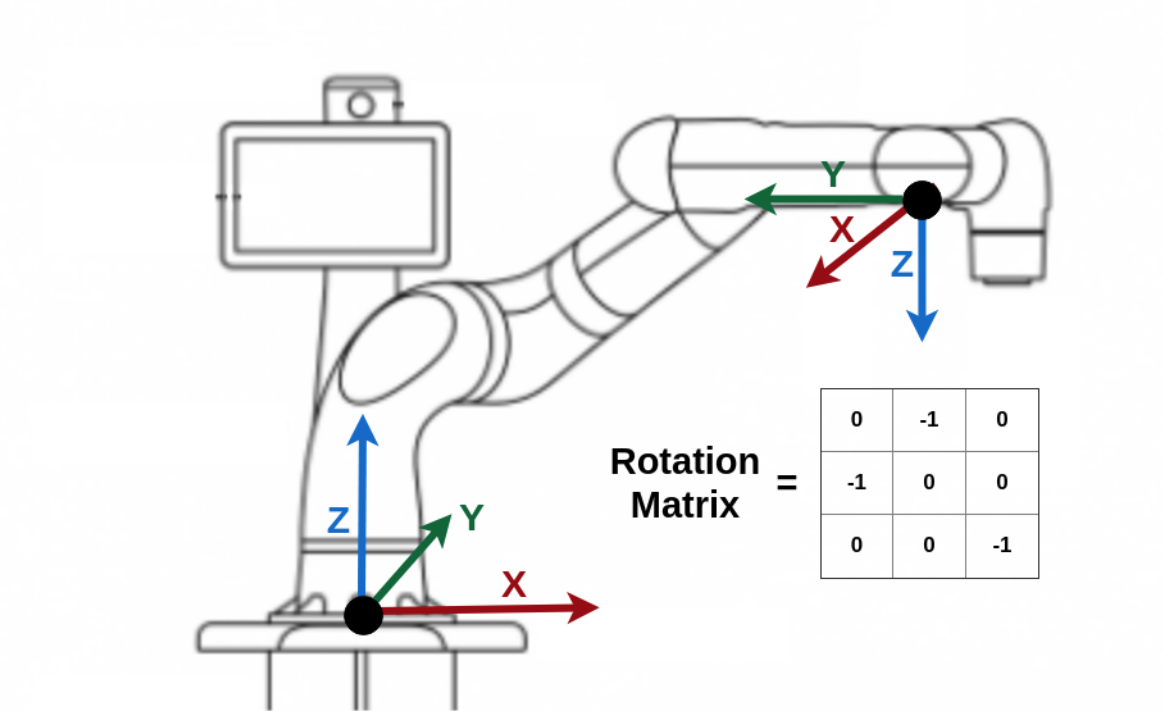


**Figure 8.3:** Coordinate frame of robot and wrist camera.

matrix. A representation is shown in Figure 8.3.

## 8.4 Gripper

Our Sawyer robot was equipped with an electric parallel gripper. The gripper is made up of:

- The base connection (the wrist) that connects the arm of Sawyer.
- 4 sets of two fingers that can be screwed to the base connector.
- 3 sets of fingers tips of different width and shape.

Depending on the set of fingers chosen and how they are screwed onto the base connection of the gripper, the maximum and minimum width of the gripper varies, but the difference between the maximum and minimum gripper width stays the same at 23mm. The maximum possible gripper width for the widest set of fingers is 151mm. In Table 8.2 a summary of all the possible gripper configurations is shown. Considering the size of our object and how our GQ-CNN was trained, we decided to use Narrow fingers and screwed them at position 3, obtaining a gripper width that could vary between 33mm and 56mm. For the fingertips, we decided to use the Basic model, a simple rectangular shape, as they were the most similar to the ones we used in training and simulation. Controlling the gripper was relatively straightforward thanks to the Intera Interface and the examples provided. During the first grasping attempts, we realized the gripper was closing so fast the objects

| Object Width | Finger | Position |
|---|---|---|
| 0-18 mm | Narrow | 1 |
| 14-37 mm | Narrow | 2 |
| **33-56 mm** | **Narrow** | **3** |
| 52-75 mm | Narrow | 4 |
| 71-94 mm | Wide | 1 |
| 90-113 mm | Wide | 2 |
| 109-132 mm | Wide | 3 |
| 128-151 mm | Wide | 4 |

**Table 8.2:** Possible gripper configurations and grasp width ranges. In bold, the configuration we used for our experiments.

were often being moved or rotated during the grasp, causing the execution to fail. After we changed the gripper settings so the gripper would move slower the issue was resolved.

## 8.5 Objects

In order to evaluate our GQ-CNN, we wanted to test on a variety of objects the network had never seen before. Unfortunately, since we didn't have a depth camera, we had to limit our choice of objects to items that were mostly uniform in color. We also had to avoid any objects that contained reflective or transparent components. We also had to consider the limits of our gripper, so we could only use objects with a graspable section between 33 and 56mm. A picture of some of the objects used during the evaluation is shown in Figure 8.4.

## 8.6 Evaluation

We tested grasp execution on each shape 10 times. Objects that appeared different from different angles were considered as different shapes. Overall, we had around 15 objects, but if we considered the different positions that they could be placed in we could test our model on over 30 different shapes. Our results showed a 100% success rate on very simple and linear objects such as cubes and prisms. The network also performed very well on round objects, with a 93% success rate. Examples of round objects we tested on are shown in Figure 8.6.

The network also showed reliable grasping behaviour($\approx$85% success rate) on objects with a fairly large graspable area, such as a toy hammer and toy pliers, or other objects such as pens and screws.

Our algorithm seemed to have difficulties dealing with very small shapes where precision was fundamental. A good example of this is a triangular shape: even though the algorithm would always return a grasp attempting to position one finger on a vertex and the other in the middle of the opposite edge, when executed the gripper would often miss the vertex by a few millimeters, causing the grasp to be

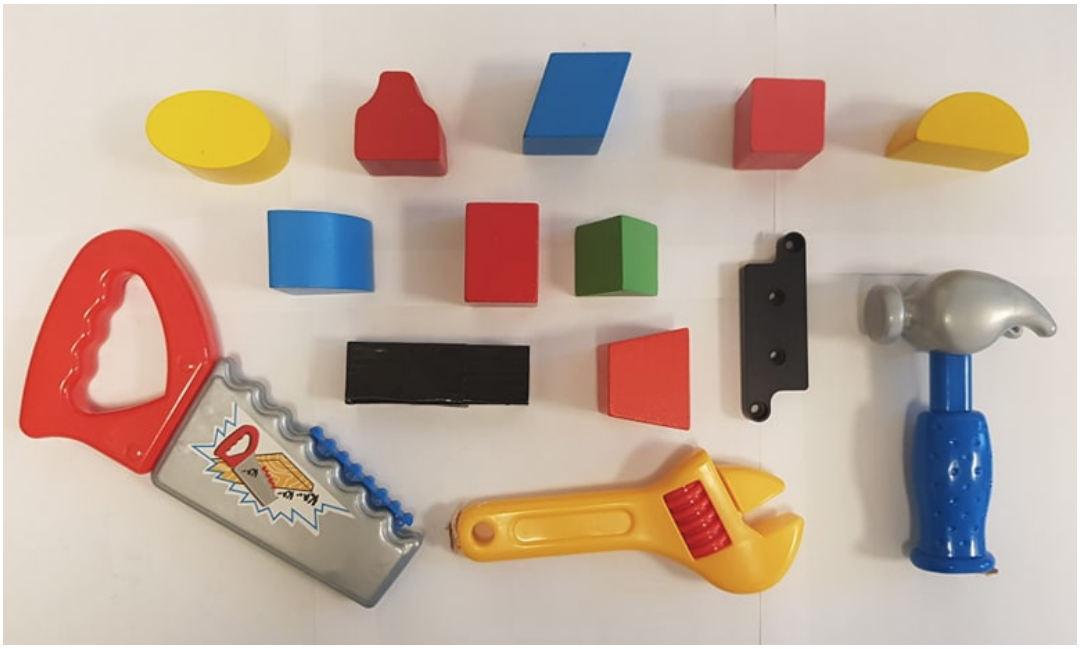

**Figure 8.4:** Some of the objects used during GQ-CNN evaluation on Sawyer.

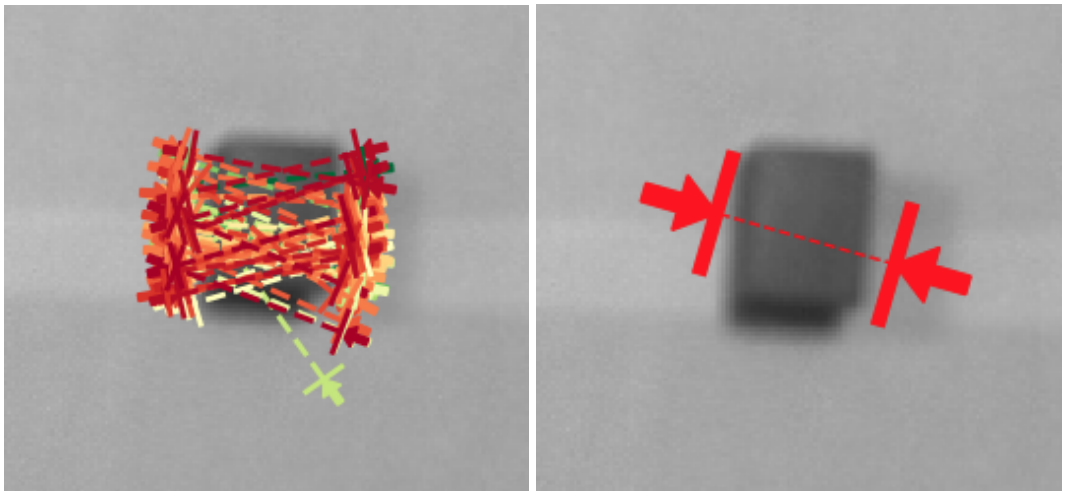

**Figure 8.5:** Elite grasps for simple red prism on white background(Left) and best grasps(Right).

either very unstable or to fail entirely. The success rate on this object was 60%. Another similar example, where the algorithm achieved the same 60% success rate, is shown in Figure 8.7.

When tested on diamond-like shapes, the algorithm performed fairly well, with an overall success rate of ≈80%.

The final test group was composed of very irregular shapes, where multiple robust grasps could be executed and possible collision had to be taken into account. A good example of this is the toy saw.

The algorithm performed very well in this type of shape, executing grasp successfully ≈85% of attempts. Even though the network was able to identify areas of

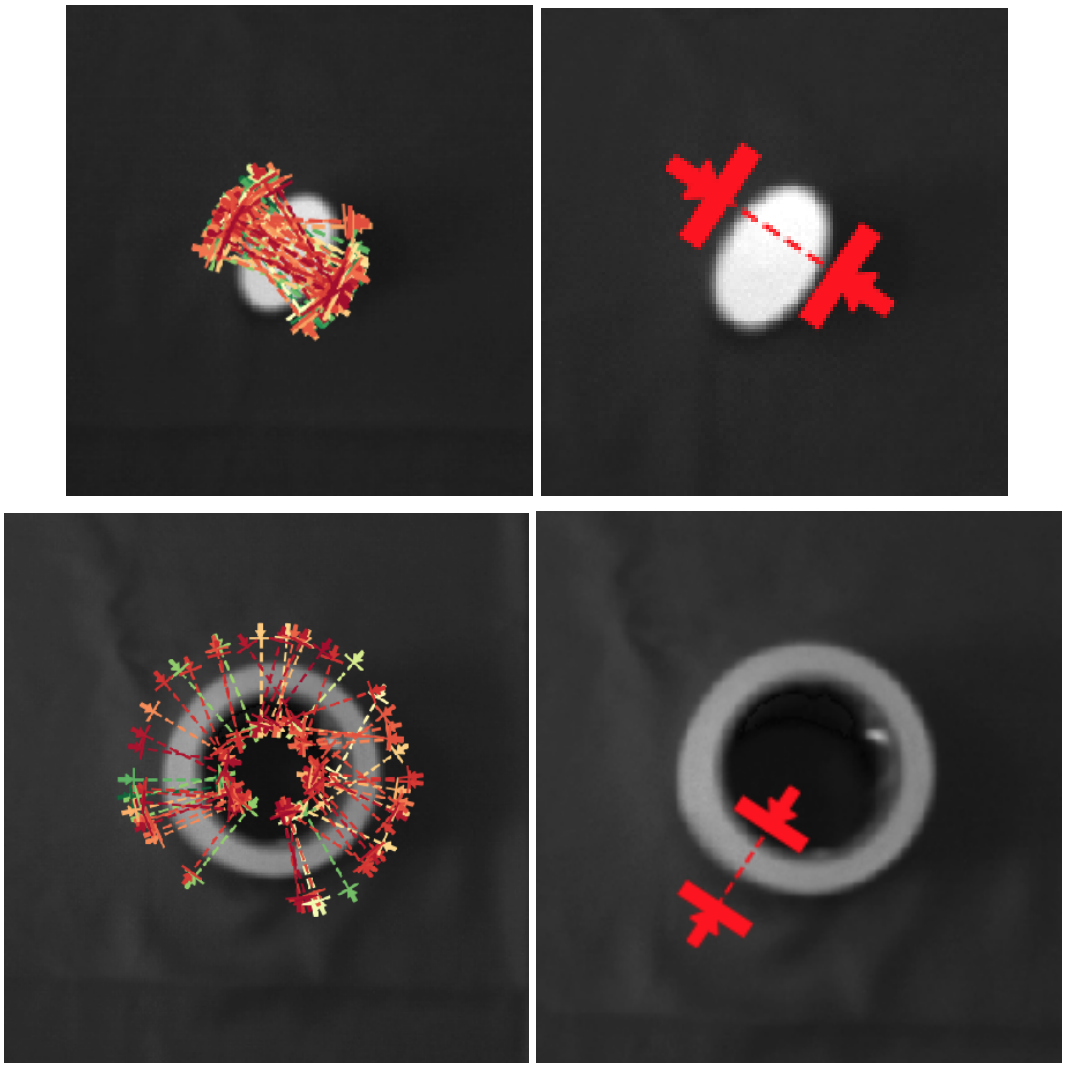

**Figure 8.6:** Elite grasps for round objects on a green background(Left) and best grasps(Right).

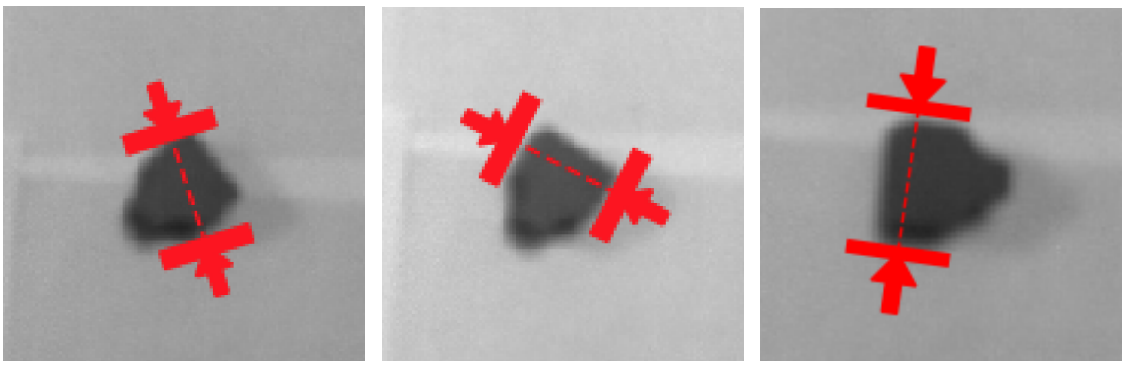

**Figure 8.7:** Best grasp for multiple attempts of grasping a small irregular shape. Even though the best grasp is identified correctly in the majority of attempts(Center, Right) the grasps are often unsuccessful(≈30%) due to imprecision during execution.

optimal grasp robustness correctly, it did sometimes fail in predicting possible collisions. This might be caused either by imprecision in the actuation or by the network being trained for collisions on a slightly different gripper shape than the real one.

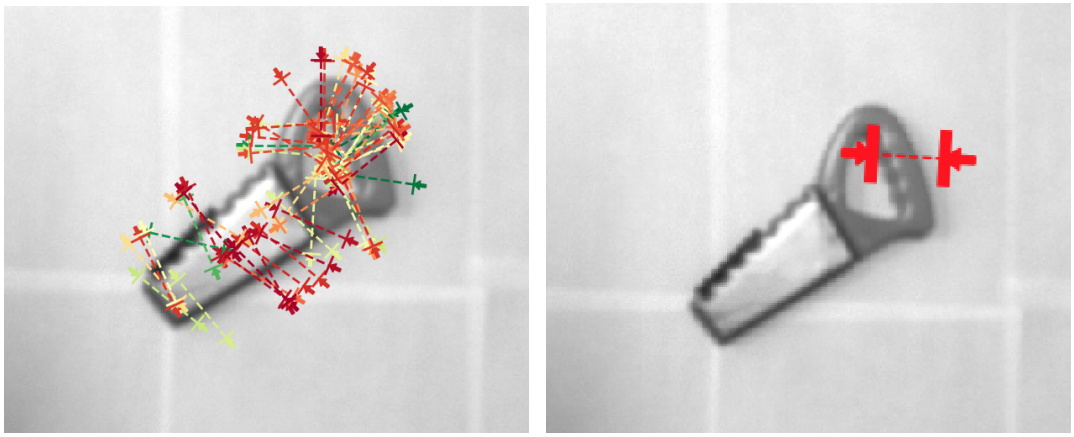

**Figure 8.8:** Elite grasps(Left) and best grasp(Right) for very irregular shape where possible collision has to be considered.

| Object Type | Number of Shapes | Success Rate |
|---|---|---|
| Prisms & Similar | 10 | 100% |
| Round | 3 | 93% |
| Big Irregular Shapes | 5 | 85% |
| Possible Collision | 2 | 85% |
| Diamonds & Trapezoids | 2 | 80% |
| Small Irregular Shapes | 2 | 60% |

**Table 8.3:** Results of grasp execution policies on a real robot grouped by type of shape, ranked from best to worst performance.

# 9 Evaluation

In this chapter we will briefly summarize the results discussed previously, and highlight the strengths and weaknesses of each model.

## 9.1 Original GQ-CNN

**Strengths**

- High accuracy on both seen and unseen objects
- The use of antipodal grasp sampling policy helps greatly by only proposing "reasonable" grasps to the grasp quality network.

**Weaknesses**

- Slow. Takes between 2 and 4 seconds to locate a robust grasp
- Has difficulties grasping small, irregular objects. This happened in both simulation and real-world evaluation, but for different reasons. In simulation, the GQ-CNN wasn't able to accurately rank grasps on objects where the vast majority of possible grasps were likely going to be unstable. Even if grasps on a certain object are bad, the GQ-CNN should still be able to locate the most robust, and in this particular case, it was unable to do it and performed even worse than random choice. In simulation, it was able to locate a robust grasp accurately, but imprecision in grasp execution would often cause the grasp to fail.
- Long training times($>$24h)
- High number of trainable parameters(18 million)

## 9.2 GQ-CNN increased resolution

**Strengths**

- High accuracy on both seen and unseen objects
- Generally, predicts more robust grasps if compared to the original GQ-CNN

**Weaknesses**

- Huge dataset(>80GB)
- Long training times(>24h)
- High number of trainable parameters

## 9.3 GQ-CNN 3 million parameters

**Strengths**

- Faster to train
- Less prone to overfitting

**Weaknesses**

- Slightly less accurate

## 9.4 GQ-AE

**Strengths**

- 100 times faster than original GQ-CNN in predicting a robust grasp
- Doesn't rely on a grasp sampling algorithm
- Able to learn from incomplete data and extrapolate relevant information
- Low number of trainable parameters(≈66k), making it faster to train and less likely to overfit

**Weaknesses**

- Although it has a similar performance to the original GQ-CNN on known and unknown objects similar to the ones it was trained on, it performs significantly worse on completely new shapes. Part of this drop in performance is likely caused by the lack of a grasp sampling algorithm component.
- The fact the network doesn't rely on a grasp sampling algorithm, occasionally causes extremely bad grasps to be executed
- It has difficulty grasping thin objects, this could be likely fixed by adding more examples of these types of objects in the dataset
- Grasps often fail because of wrong grasp depth. This is because grasp depth was not included while training the network

# 10 Conclusions and Future Work

## 10.1 Conclusions

In the first part of our project, we succeeded in reproducing the methodologies described in the Dex-Net 2.0 publication: we generated a dataset of synthetic data starting from 3D meshes and used it to train GQ-CNN that was able to reliably predict robustness of grasp candidates.

We contributed to the Dex-Net repository by suggesting bug fixes, solving library clashes and sharing updated installation scripts. We also verified the validity of the grasps generated by the Dex-Net codebase in simulation. Furthermore, we trained a GQ-CNN able to accept RGB images as input and extended the Dex-Net codebase with a policy able to sample antipodal grasp candidates from an RGB image. Then, we tested our new RGB policy and GQ-CNN on a real robot.

We also implemented an effective testing framework for evaluating and comparing different grasp quality neural networks using V-REP. We generated many variations of the original GQ-CNN dataset to train new GQ-CNNs networks, evaluated and compared their strengths and weaknesses, tested out new architectures and succeeded in creating a new GQ-CNN with 84% fewer parameters than the original GQ-CNN architecture and similar performance. We also trained a GQ-CNN network on higher resolution images with more context on each object. This network achieved slightly better performance than the original GQ-CNN in the majority settings.

In the last part of our project, we worked on an entirely new methodology, building a new type of grasp quality network able to simultaneously evaluate all possible grasp position and locate the most robust one 100 times faster than the Dex-Net algorithm.

This model was trained using a novel training technique we developed to allow the network to learn successfully from incomplete data. The network was able to extrapolate relevant information from thousands of partial data images and correctly merge the knowledge gained from each one to accurately predict complete grasp quality images.

Whilst the performance of our network was slightly lower than Dex-Net's, we believe this approach has great potential to be further researched and improved. Furthermore, our network seemed to perform much better on certain types of objects(e.g. very small, highly irregular shapes such as small planes) that the original Dex-Net GQ-CNN consistently struggled with.

### 10.1.1 Lessons Learned

- **Data Generation** Many times, during our data generation processes, we would wait hours or even days for a dataset to be fully generated only to realize that

for some reason, a small change was necessary before being able to use it to train a network. After the first few times, we learned to test on a small subset of the dataset before starting long computations.

- **Third Party Libraries** As we learned using Dex-Net, it is better to thoroughly test research libraries such as Dex-Net before implementing it in your own application, as they could be old versions or only partial copies of the codebases the researchers actually used.
- **Record all Experiments** At the beginning of our evaluation process, we did not record all of the information we would have needed, such as the pose we were setting the object in when grasping it. This meant we had to repeat some experiments as we were unsure about which object poses we evaluated in our previous experiments.
- **Constantly Back up Code and Data** A few days before the end of our project, one of the machines we were working on, `Tiger`, crashed and became unavailable. This meant the loss of the latest version of our code and models. Luckily, we frequently backed up all data and code on multiple machines, but the latest version we had was still a couple of days old. This caused us to waste some time re-implementing and re-training a few models.

## 10.2 Future Work

### GQ-CNN parameters improvement

In Section 6.5 we succeeded in training a GQ-CNN with 84% fewer parameters than the architecture presented in the original Dex-Net 2.0 publication. We also confirmed this new model performed as well as the original GQ-CNN in most situations. Even though we did not have time to explore other architectures further, we believe there could be room for improvement and it could be possible to reduce the size of the network even further, while still maintaining similar performance.

### Training with Procedurally Generated Objects

Although the procedurally generated objects we generated were useful during training, we did not have time to thoroughly evaluate their potential as training data for our networks.

### RGB Images Generation with Random Illumination

The 6.2 million datapoints dataset of RGB images we created included domain randomization over object color and camera position, but not illumination. Not surprisingly, this caused problems when the network was evaluated in the real world, where shadows were hard to avoid.

### Explore different levels of Domain Adaptation

As we trained our GQ-CNN on different amounts of objects and compared their performance, it would be interesting to train GQ-CNNs on different datasets with different levels of domain adaptation(e.g. lower or higher numbers of randomized camera positions for each grasp) and compare their performances.

### Include Depth in GQ-AE

We decided to train our GQ-AE ignoring gripper depth information, but we noticed this turned out to be a problem during the evaluation phase as many grasps were failing because the depth of the gripper was incorrect. This would be straightforward to implement and hopefully significantly boost the performance of our GQ-AE.

### GQ-AE Training Optimization

Our GQ-AE training script is currently quite inefficient, as Pytorch did not offer any built-in way of ignoring certain values while computing the loss function between two images. This meant we had to change parts of the predicted images to match the training images. Because Pytorch doesn't allow in-place operations on tensors, we had to clone each tensor before applying the loss function and this significantly slowed down the network's training times. In future work, we would like to optimize this part of training by building our own version of Binary Cross Entropy or Mean Squared Error that would consider certain pixels in the images.

### GQ-AE Training Method

In Section 7.6 we explored many different ways of training our network, but we believe we could still improve the network performance significantly by tweaking our data generation and training methodologies further. One idea we had but were not able to test was using some form of "confidence" metric over the value of each grasp, so that the loss function could weight the loss of each pixel differently based on its corresponding confidence value. We thought about this when setting values of neighbouring areas of pixels instead of exact pixels only, as we obviously had more confidence that the exact pixel value represented a correct grasp quality metric compared to the area around it.

### RGB and RGB-D GQ-AE

Our GQ-AE was only trained on depth images, but it would be interesting to train it on RGB or RGB-D images and test how performance varies.

### GQ-AE Applied to Complex and Dynamic Environments

Given how fast our GQ-AE network is in locating a stable grasp on an object($\approx$0.03 seconds), it would be feasible to apply it in a dynamic environment to try and grasp moving objects, as well testing its performance on a real robot.

# Bibliography


[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org. pages 28

[2] A. Aristidou, J. Lasenby, Y. Chrysanthou, and A. Shamir. Inverse kinematics techniques in computer graphics: A survey. *Computer Graphics Forum*, 37(6):35–58, 2018. pages 14

[3] Berkeley Automation. Dex-net. pages 22

[4] T B. Nguyen and Djemel Ziou. Contextual and non-contextual performance evaluation of edge detectors. *Pattern Recognition Letters*, 21:805–816, 08 2000. pages 49

[5] R. Balasubramanian, L. Xu, P. D. Brook, J. R. Smith, and Y. Matsuoka. Physical human interactive guidance: Identifying grasping principles from human-planned grasps. *IEEE Transactions on Robotics*, 28(4):899–910, Aug 2012. pages 16, 24, 38

[6] Muhammet Bastan. Canny edge detection. pages 49

[7] Jeannette Bohg, Antonio Morales, Tamim Asfour, and Danica Kragic. Data-driven grasp synthesis - a survey. 2013. pages 16, 38

[8] Konstantinos Bousmalis, Alex Irpan, Paul Wohlhart, Yunfei Bai, Matthew Kelcey, Mrinal Kalakrishnan, Laura Downs, Julian Ibarz, Peter Pastor, Kurt Konolige, Sergey Levine, and Vincent Vanhoucke. Using simulation and domain adaptation to improve efficiency of deep robotic grasping, 2017. pages 2, 19, 20, 50

[9] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, Nov 1986. pages 49

[10] Angel X. Chang, Thomas Funkhouser, Leonidas Guibas, Pat Hanrahan, Qixing Huang, Zimo Li, Silvio Savarese, Manolis Savva, Shuran Song, Hao Su,

Jianxiong Xiao, Li Yi, and Fisher Yu. Shapenet: An information-rich 3d model repository, 2015. pages 19

[11] Ken Chef. Gradient descent and backpropagation. pages 12

[12] Paul Christiano, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba. Transfer from simulation to real world through learning deep inverse dynamics model, 2016. pages 20

[13] Robot Learning Lab Cornell University. Learning to grasp. pages 18, 76

[14] Li Deng. Three classes of deep learning architectures and their applications: A tutorial survey. 2012. pages 6

[15] Li Deng and Dong Yu. Deep learning: Methods and applications. Technical Report MSR-TR-2014-21, May 2014. pages 6

[16] Guoguang Du, Kai Wang, and Shiguo Lian. Vision-based robotic grasping from object localization, pose estimation, grasp detection to motion planning: A review, 2019. pages 17, 18

[17] C. Ferrari and J. Canny. Conf. robotics and automation - icra, 1992. pages 16, 24

[18] Institut fr Neuroinformatik. Ijcnn2011 competition results. pages 6

[19] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. pages 7, 8, 10, 11

[20] Stephen James, Andrew J. Davison, and Edward Johns. Transferring end-to-end visuomotor control from simulation to real world for a multi-stage task, 2017. pages 2, 21

[21] Z. Ju, C. Yang, Z. Li, L. Cheng, and H. Ma. Teleoperation of humanoid baxter robot using haptic feedback. In *2014 International Conference on Multisensor Fusion and Information Integration for Intelligent Systems (MFI)*, pages 1–6, Sep. 2014. pages 17

[22] B. Kehoe, A. Matsukawa, S. Candido, J. Kuffner, and K. Goldberg. Cloud-based robot grasping with the google object recognition engine. In *2013 IEEE International Conference on Robotics and Automation*, pages 4263–4270, May 2013. pages 16, 24

[23] Sulabh Kumra and Christopher Kanan. Robotic grasp detection using deep convolutional neural networks, 2016. pages 1, 15, 17

[24] Sergey Levine, Peter Pastor, Alex Krizhevsky, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection, 2016. pages 2, 17

[25] J. Mahler, S. Patil, B. Kehoe, J. van den Berg, M. Ciocarlie, P. Abbeel, and K. Goldberg. Gp-gpis-opt: Grasp planning with shape uncertainty using gaussian process implicit surfaces and sequential convex programming. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4919–4926, May 2015. pages 22, 23, 24

[26] Jeffrey Mahler, Jacky Liang, Sherdil Niyaz, Michael Laskey, Richard Doan, Xinyu Liu, Juan Aparicio Ojea, and Ken Goldberg. Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics, 2017. pages 15

[27] Jeffrey Mahler, Jacky Liang, Sherdil Niyaz, Michael Laskey, Richard Doan, Xinyu Liu, Juan Aparicio Ojea, and Ken Goldberg. Dex-net 2.0: Deep learning to plan robust grasps with synthetic point clouds and analytic grasp metrics. 2017. pages 15, 21, 55

[28] Jeffrey Mahler, Matthew Matl, Vishal Satish, Michael Danielczuk, Bill DeRose, Stephen McKinley, and Ken Goldberg. Learning ambidextrous robot grasping policies. *Science Robotics*, 4(26):eaau4984, 2019. pages 68

[29] Jeffrey Mahler, Florian T Pokorny, Brian Hou, Melrose Roderick, Michael Laskey, Mathieu Aubry, Kai Kohlhoff, Torsten Kröger, James Kuffner, and Ken Goldberg. Dex-net 1.0: A cloud-based network of 3d objects for robust grasp planning using a multi-armed bandit model with correlated rewards. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 1957–1964. IEEE, 2016. pages 15, 21, 24

[30] Douglas Morrison, Peter Corke, and Jrgen Leitner. Closing the loop for robotic grasping: A real-time, generative grasp synthesis approach, 2018. pages 74, 76, 79

[31] Neuroscience and Robotics Lab. V-rep introduction. pages 55

[32] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017. pages 79

[33] F. A. Pellegrino, W. Vanzella, and V. Torre. Edge detection revisited. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 34(3):1500–1518, June 2004. pages 49

[34] Felice Andrea Pellegrino, Walter Vanzella, and Vincent Torre. Edge detection revisited. *IEEE transactions on systems, man, and cybernetics. Part B, Cybernetics : a publication of the IEEE Systems, Man, and Cybernetics Society*, 34:1500–18, 07 2004. pages 49

[35] Lerrel Pinto and Abhinav Gupta. Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours, 2015. pages 16, 17

[36] F. T. Pokorny and D. Kragic. Classical grasp quality evaluation: New algorithms and theory. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3493–3500, Nov 2013. pages 16

[37] Florian T. Pokorny and Danica Kragic. Classical grasp quality evaluation: New theory and algorithms. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, Tokyo, Japan, 2013. pages 23

[38] M. Pozzi, A. M. Sundaram, M. Malvezzi, D. Prattichizzo, and M. A. Roa. Grasp quality evaluation in underactuated robotic hands. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1946–1953, Oct 2016. pages 16

[39] Domenico Prattichizzo and J.C. (Jeff Trinkle. *Grasping*, pages 671–700. 01 2008. pages 16

[40] Maximo Roa and Ral Surez. Grasp quality measures: Review and performance. *Autonomous Robots*, 38:65–88, 07 2014. pages 16

[41] Coppelia Robotics. Basics on ik groups and ik elements. pages 60

[42] Coppelia Robotics. Collision detection. pages 59

[43] Coppelia Robotics. Dynamics. pages 56

[44] Coppelia Robotics. Object common properties. pages 59

[45] Rethink Robotics. Networking. pages 89

[46] Rethink Robotics. Sawyer hardware overview. pages 88, 90

[47] Robotis. Robotis emanual - inverse kinematics. pages 14

[48] Alberto Rodriguez, Matthew T Mason, and Steve Ferry. From caging to grasping. *The International Journal of Robotics Research*, 31(7):886–900, 2012. pages 16

[49] W. Rong, Z. Li, W. Zhang, and L. Sun. An improved canny edge detection algorithm. In *2014 IEEE International Conference on Mechatronics and Automation*, pages 577–582, Aug 2014. pages 49

[50] Vishal Satish, Jeffrey Mahler, and Ken Goldberg. On-policy dataset synthesis for learning robot grasping policies using fully convolutional deep networks. *IEEE Robotics and Automation Letters*, 2019. pages 75, 76

[51] OpenCV Dev Team. Camera calibration and 3d reconstruction. pages 13, 14

[52] Theano. Convolution arithmetic tutorial. pages 10

[53] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter Abbeel. Domain randomization for transferring deep neural networks from simulation to the real world, 2017. pages 1, 20

[54] Kevin Zakka. Dex-net 2.0: Deep learning to plan robust grasps. pages 30

[55] Andy Zeng, Shuran Song, Kuan-Ting Yu, Elliott Donlon, Francois R. Hogan, Maria Bauza, Daolin Ma, Orion Taylor, Melody Liu, Eudald Romo, Nima Fazeli, Ferran Alet, Nikhil Chavan Dafle, Rachel Holladay, Isabella Morona, Prem Qu Nair, Druck Green, Ian Taylor, Weber Liu, Thomas Funkhouser, and Alberto Rodriguez. Robotic pick-and-place of novel objects in clutter with multi-affordance grasping and cross-domain image matching, 2017. pages 76

[56] Yujin Zhang. *Volume 2 - Image Engineering*. August 2017. pages 49

[57] Yu Zheng and Wen-Han Qian. Coping with the grasping uncertainties in force-closure analysis. *The International Journal of Robotics Research*, 24(4):311–327, 2005. pages 22

# A Pseudocode

## A.1 Camera Randomization Parameters

```
# Camera pose
# Min-Max radius for viewing sphere
min_radius: 0.65;  max_radius: 0.75  # in meters
# Min-Max elevation (angle from z-axis) for camera position
min_elev: 0.1; max_elev: 5.0  # in degrees
# Min-Max azimuth (angle from x-axis) for camera position
min_az: 0.0;  max_az: 360.0   # in degrees
# Min-Max roll (rotation of camera about
# axis generated by azimuth and elevation) for camera
min_roll: -0.2; max_roll: 0.2 # in degrees


# Object pose
min_x: -0.1; max_x: 0.1   # in meters
min_y: -0.1; max_y: 0.1   # in meters
```

**Listing 9:** Parameters used for camera randomization

## A.2 Obtaining Camera Image and Info with ROS

```
import intera_interface

camera_name = "right_hand_camera"
cameras = intera_interface.Cameras()
cameras.set_exposure(camera_name, 7)

cam_info_topic = "io/internal_camera/right_hand_camera/camera_info"
cam_info_msg = rospy.wait_for_message(cam_info_topic, CameraInfo)
cam_intr = CameraIntrisics(cam_info_msg.header.frame_id,
            cam_info_msg.K[0], cam_info_msg.K[4],
```

```
            cam_info_msg.K[2], cam_info_msg.K[5],
            cam_info_msg.K[1], cam_info_msg.height,
            cam_info_msg.width)

cam_img_topic = "io/internal_camera/right_hand_camera/image_rect"
cam_img_msg = rospy.wait_for_message(cam_img_topic, Image)
bridge = CvBridge()
cv_image = bridge.imgmsg_to_cv2(cam_img_msg, 'rgb8')
```

**Listing 10:** Pseudocode of algorithm used to retrieve Camera Image and Camera Info using ROS, from Sawyer robot

## A.3 Antipodal Grasp Sampling

```
def sample_antipodal_grasps(depth_im, num_samples):
  # compute edge pixels
  edge_im = depth_im.apply_gaussian_filter(sigma=gauss_sigma)
  # add to pixel to edge list if edge value greater than threshold
  edge_pixels = edge_im.threshold_gradients(depth_grad_thresh)

  # compute surface normals
  edge_normals = calculate_surface_normals(depth_im, edge_pixels)
  # uniformly sample pairs of edge pixels
  edge_pairs = edge_pixels.uniform_sample(2)

  # form set of valid candidate point pairs
  valid_pairs = edge_pairs.where(
                    (normal_ip < -np.cos(np.arctan(friction_coef))) and
                    (dists < max_grasp_width_px) and
                    (dists > 0.0))

  # prune out grasps that are not antipodal
  antipodal_pairs = valid_pairs.check_antipodal(edge_normals)

  k = 0
  grasps = []
  while k < sample_size and len(grasps) < num_samples:
    p1, p2 = antipodal_pairs[k] # get the 2 contact points
    k += 1
```

```
    grasp_center = (p1 + p2) / 2  # compute center and axis
    grasp_axis = normalize(p2 - p1)
    grasp_theta = np.arctan2(grasp_axis[0], grasp_axis[1])
    # get depth in the neighborhood of the center pixel
    depth_win = depth_im.get_min_value_in_window(wind_h, wind_w)
    # sample depth between the min and max
    sample_depth = center_depth
                + min_depth_offset
                + ((max_depth_offset - min_depth_offset) * np.random.rand())
    candidate_grasp = Grasp2D(grasp_center_pt, grasp_theta, sample_depth)

    grasps.append(candidate_grasp)
  return grasps
```

**Listing 11:** Pseudocode of antipodal grasp sampling algorithm on a depth image