university of groningen / faculty of mathematics and natural sciences / artificial intelligence

Grasping in 6DoF: An Orthographic Approach to Generalized Grasp Affordance Predictions

Master Thesis
(Computational Intelligence and Robotics)

Mario Ríos Muñoz (s3485781)

March 22, 2021

Internal Supervisor(s): Dr. Hamidreza Kasaei (Artificial Intelligence, University of Groningen)
Prof. Dr. Lambert Schomaker (Artificial Intelligence, University of Groningen)

Artificial Intelligence / Human-Machine Communication
University of Groningen, The Netherlands

RIJKSUNIVERSITEIT GRONINGEN (RUG)

Abstract

Faculty of Science and Engineering
Department of Artificial Intelligence

MSc

by Mario Ríos Muñoz

Grasp detection research focuses at the moment on finding neural networks that given a RGB-D image or point cloud, yield a parametric grasp description that can be used to firmly grip target objects. There is a need for these models to be small and efficient, such that they can be used in embedded hardware. Furthermore these models tend to only work for top-down views, which highly restrict the ways objects can be grasped. In this work, we focus on improving an existing shallow network, GG-CNN, and propose a new orthographic pipeline to enable the use of these models independently of the orientation of the camera. We make our implementation available on GitHub.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Autonomous robotic manipulation is a complex and challenging problem, involving a number of disciplines that range from mechanical engineering and hardware development to motion planning and object detection or semantic segmentation. Advancements in the field are relevant for a large number of applications. Automated manipulators were first introduced to the assembly line by the car industry [1], and are nowadays well established in the manufacturing sector. Moreover, robotic arms like the Canadarm [2] have been employed for space construction. However, at the moment this technology only thrives in highly controlled environments, where end effectors tailored to the specific task at hand are employed and operations are either highly repetitive and well known or planned in detail beforehand.

One sector where the use of dexterous robotics could be highly beneficial but has not seen any commercially viable solutions as of yet is service robots for the domestic environment. The ability to manipulate common household items for these type of agents is fundamental to their purpose, but the challenges are many [3]. To name a few, we can consider: highly dynamic environments; open-ended number of object categories; clutter; noisy sensors and the need for cost-effectiveness.

A key skill for enabling autonomous manipulation is robust grasp affordance detection. The term *affordance* was first coined by Gibson in his 1977 publication [4]. It refers to the utility that an element of an environment offers to an agent. Examples of affordances are sitting, in the case of chairs, or scooping, for spoons, and more relevant to this work: grasping, for handles and object surfaces. The concept of affordance has become of the upmost relevance for the field of cognitive robotics. In order for an intelligent agent to solve complex tasks, it needs to understand not only the class of the objects that surround it, but also how said objects, or even sub-elements of the object, can be useful for the problem at hand. Detecting grasp affordances is instrumental to build more

complex manipulation behaviours and tasks, but it is not sufficient to ensure a firm grip: the pose of the hand and fingers relative to the target geometry prior to grasping an object plays a major role as well. This is the objective of grasp detection research: given a 3D representation of an object, determine the point in space and orientation that a robotic hand must adopt such that when the fingers close a robust grasp is achieved.

In recent years, following the advent of deep learning and affordable of-the-self 3D camera hardware, successful efforts have been made to develop models that take a depth image as an input and yield a grasp prediction. These models typically use CNNs (Convolutional Neural Networks) to process the images, and in most of the cases the cameras have a top-down view of the workspace and the predictions are made only in 3DoF (degrees of freedom). This approach is limited in the type of objects that can be manipulated (for example plates can't be grasped this way). Additionally, this constrains the location where the camera can be mounted to the robot, and where the robot must place itself in order to interact with a scene.

To ensure cost-effectiveness of commercial solutions, the compute hardware of future service robots may be limited. GG-CNN [5] is a small convolutional network that can run inferences in real time (50 Hz) on desktop-level GPU cards. This makes it a good candidate for embedded applications with more limited processing power.

1.1 Problem Description

There is room for improvement accuracy-wise for such a small network as GG-CNN. The challenge that we will focus on is how to achieve better success rates while keeping the network sufficiently small. Moreover, the network was designed to run real time predictions on images from a hand-mounted camera. In their experiments they always present a nadir (top-down) view of the scene and approach the objects vertically. It would be interesting to explore how this network performs when attached to the body of a robot, such that the same cameras used for navigation may be used for manipulation as well. As such, we will evaluate the performance of GG-CNN on off-nadir views, as well as provide a grasping strategy for other than top down manipulator approaches.

1.1.1 Research Questions

1. How can we improve the accuracy of GG-CNN while keeping it small?
2. How well does GG-CNN perform on off-nadir grasp approaches?

3. How can we adapt existing models to robustly grasp objects from off-nadir camera views?

1.1.2 Contributions

1. Through methodical model-space exploration, we find an extended version of the GG-CNN architecture that performs 5% better in our simulation experiments, with only 8% more number of parameters.

2. We propose a model-agnostic orthographic pipeline that enables any architecture compatible with depth images to safely grasp objects from an off-nadir point of view.

Chapter 2

Literature Review

In this chapter we give an overview of Machine Learning based methods to achieve grasp detection. We also introduce a number of datasets available for neural network training, and a set of techniques relevant for automated architecture-space exploration.

2.1 Machine Learning for Grasp Detection

With the cheapening of GPU hardware and the advent of Machine and Deep Learning, many fields have been revolutionized and their state-of-the-art been pushed forward. Grasp detection is no exception. Neural network based approaches typically use Convolutional Neural Networks (CNNs) to process an image with depth information or a point cloud and produce a description of a grasp pose.

2.1.1 Classification vs Regression

Depending on the output of an architecture and the training process, neural networks can be utilized to perform classification or regression tasks. Both methods are valid for grasp detection. Classification-based solutions first sample the grasp space for candidates and then use the model to determine if a grasp is valid or not. The method in [6] uses a predefined grid of anchored boxes as the starting point for tentative grasps. It feeds the image through a ResNet [7] derived architecture to identify features, that are then passed to a regression and a classification subnets. The regression layers fine-tune the predefined boxes, while the classification layers select the valid candidates. The authors in [8] chose to work on point-clouds, and trained their network to classify each point as valid/no valid based on a grasp pose analytically determined from the neighbouring geometry. A similar approach was followed in [9], where a U-Net [10] derived architecture

was employed. Researchers from Nvidia [11] used variational auto-encoders trained on successful grasp examples to generate a variety of grasps per point-cloud. They classify the resulting grasps using PointNet [12], and feed those deemed as invalid to another network in order to fine-tune them into valid grasps. GQ-CNN [13] is trained on a vast dataset of synthetic images to predict a quality score for grasp-aligned depth images.

On the other hand, regression-based methods learn to directly infer the parameters that define a valid grasp. This makes this kind of approaches more straight forward, and typically simpler. GG-CNN [5] and GR-ConvNet [14] follow a similar approach. Both use multiple output maps with similar dimensions to the input image. These maps encode per-pixel the parameters necessary to construct a grasp (grasp quality, width and orientation). To do so, GG-CNN uses a a shallow fully convolutional architecture, while GR-ConvNet combines convolutional and transposed-convolutional layers with residual blocks. The network proposed in [15] makes use of convolutional layers for feature extraction to which a set of of fully-connected layers are concatenated. They output a flat 5-dimensional array describing a single grasp pose per image.

2.1.2 Planar vs 6DoF

Research is frequently focused on antipodal grasps (two fingered grippers). A common choice for this grasp definition is a rectangle, where the height may be fixed (according to the geometry of the gripper used), the width represents the position of the fingers prior to closing them and the position and orientation determine the pose of the hand [5, 6, 13–15]. This is a 3DoF parameterization of the hand pose w.r.t the camera frame of reference, that can then be transformed to a 6DoF pose w.r.t the world coordinates knowing the extrinsic camera parameters. There is however a family of networks that directly provide 6DoF grasps [8, 11, 16]. These networks tend to work on point cloud representations of the scene rather than RGB-D data. The advantage of 6DoF grasps is that they allow manipulation from angles detached from the camera orientation. Given that most imaged-based solutions choose to place the camera with a top-down view, 6DoF methods would enable the grasping of objects that cannot be grabbed from the top, like plates.

2.1.3 Shallow Architectures

As already mentioned, there is a motivation to find small networks that can run on low powered hardware. Incidentally, regression-based networks tend to be better suited for this goal [5, 14, 15]. GG-CNN (75K parameters) was designed from the ground-up to be run in real time, such that closed-loop control strategies could be used to guide the

manipulator. The method in [14], while bigger (2M parameters), can also be executed in real time on capable hardware and it performs notably better (97.7% vs 73%) when tested against rectangle ground truths. The *fast* CNN introduced in [15] also runs at very high frequencies with decent results (94.8%). There is one classification-based method that was specifically designed to be executed on embedded platforms like Nvidia's Jetson TX1 [17]. Their approach consists on training a compressed CNN to segment image pixels according to their graspability. They then post-process the graspable blobs and use a heuristic to generate the 5 parameters for a rectangular grasp.

2.2 Datasets

The Columbia dataset [18] was one of the original attempts at a standardized grasp database. They used the 3D models from the Princeton Shape Benchmark and generated a variety (230k) of grasp labels compatible with different hand geometries. The dataset works on the assumption that similar objects are grasped similarly and it does not provide a partial depth or point-cloud representations of the objects, but a complete mesh description. One very popular alternative that does provide real depth sensor images is the Cornell Grasping Dataset (CGD)[1]. This dataset contains 885 RGB-D images of 240 different household objects with several hand-labelled grasp annotations per image up to a total of 8K grasp examples.

Since the adoption of data-driven and machine learning based methods one of the main challenges for training large scale networks has been the reduced size of the datasets. The reason behind this is that ground truths for grasp poses usually come from either hand-labelled examples [19] or physical experiments with actual manipulators [20, 21], both methods being extremely time consuming. There is a dire need for datasets comparable in size to those used in other kinds of deep learning research. Jacquard [22] was proposed in 2018 to fill this void. They generate grasp annotations through simulation. To do so, they made use of the ShapenetSem [23] database of 3D models and through random sampling and simulation validation they were able to generate 54k RGB-D synthetic images with 1.1M grasp annotations on 11K different objects. More recently ACRONYM [24] (17.7M annotations) and GraspNet-1Billion (1B annotations) [16] have been introduced. The former follows an approach similar to Jacquard but produces annotated point-clouds instead of depth images and full 6DoF grasp descriptions instead of planar ones. The later is generated from real depth sensor data of cluttered scenes. They sample the grasp space using an analytical metric to evaluate the quality, generating 6DoF grasp poses as well.

[1] http://pr.cs.cornell.edu/grasping/rect_data/data.php

2.3 Neural Architecture Search

AutoML is a trend in Deep Learning research that seeks to automate the entire Machine Learning pipeline [25]. This includes all the steps from data preparation and feature engineering, to model generation and evaluation. One step in this automation pipeline takes care of finding the best architecture for the problem. This is known as Neural Architecture Search [26]. Models created using these automated techniques have already proven to out-perform hand-crafted networks. As the name suggests, NAS formulates architecture optimization as a search problem, where the state space is formed by the set of possible architectures that can be found by the algorithm. The initial state is a base network, to which modifications are added in the form of new layers or operations (like pooling). In order to assess the performance each new network must be trained and evaluated, which can be very costly. This process can be streamlined through transfer learning [27]. In this case the transfer of knowledge ensures that the parent and children architecture yield the same output such that only a few training epochs are needed to teach the resulting network to make use of the new layers. This transfer learning based technique was successfully applied to a 3D shape recognition task in [28]. Combining transfer learning with beam search, they were able to find an architecture 99.4% smaller than the state-of-the-art that performed 3% better.

Chapter 3

Theoretical Background

3.1 A Primer on Artificial Neural Networks

The purpose of an ANN (Artificial Neural Network) is to learn a hidden function encoded in a dataset that would be too difficult to model by hand. It takes a series of inputs representing the state of the problem, and outputs a solution learnt from a series of examples.

An ANN is a mathematical model loosely inspired by the way neurons in the brain accumulate action potential from the synapses, and fire when this potential overcomes their activation threshold, thus stimulating subsequent neurons. A single neuron, and also the simplest network that can be conceived, the perceptron [29], can be formally described as in equation 3.1. The term $\boldsymbol{x} \in \mathbb{R}^n$ is the input to the network. $\boldsymbol{W} \in \mathbb{R}^{m \times n}$ represents the weights, or the contribution factor of the synapses towards the output value. The dimension m in $\boldsymbol{W}$ is the number of units or outputs of the perceptron. The term $\boldsymbol{b} \in \mathbb{R}^m$ is called the *bias* and it shifts the value within the activation function. The most important factor in the equation is the S term, or activation function, which shapes the output of the units. Any non-linear function can be used as an activation function. The use of this non-linearity lets us model non-linear relationships in the data. It also becomes specially relevant when building multi-layered networks since multiple linear layers combined can be reduced to a single linear unit. An example of a valid non-linear function commonly used is the Rectified Linear Unit (ReLU) function (Figure 3.1).

$$S(\boldsymbol{W}^{\boldsymbol{T}}\boldsymbol{x} + \boldsymbol{b}) \tag{3.1}$$

The weights and biases are typically referred to as the trainable or hyper-parameters, since they are the values that are changed in order for the model to learn features in

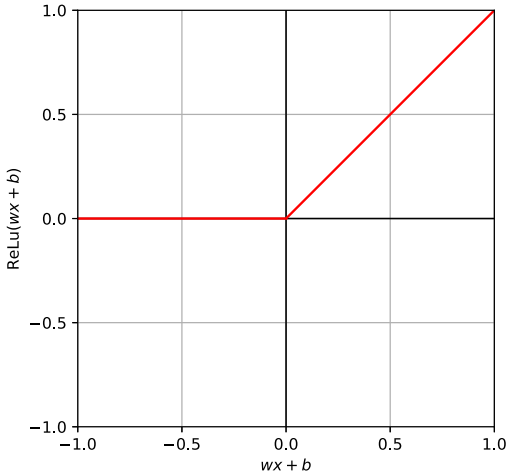

FIGURE 3.1: Rectified Linear Unit (ReLU) activation function acting on a simplified linear model with 1 input. All the values < 0 are truncated to 0. Values above 0 are left intact. The term b (bias) shifts the response within the envelop of the function.

the data. A measure of complexity of a model can be given by the number of units and the way they are connected, thus determining the number of parameters. A Multi-Layer Perceptron (MLP) is a combination of Perceptrons in a series of layers, so that the outputs of the units in the first layer are fed as input for the perceptrons in the next layer. The layers in a MLP network are commonly called fully connected layers, due to the fact that every input contributes to every output. Bigger models can understand more complex relationships in the data, at the cost of longer training times and at risk of overfitting. Overfitting happens when the model learns the exact patterns of a training set and does not extrapolate to new examples. The size of the models also has an impact on the time it takes them to make an inference. This aspect is relevant for applications where latency is a factor.

3.1.1 Forward Pass, Backwards Pass and Loss Function

Given a set of weights, biases and an input, we can evaluate equation 3.1 to obtain an output. This is what is called a forward or inference pass. For an inference to be valid, the values of the hyper-parameters must approximate the function that we want to learn. The process of tuning these values to approximate the output of the model to our desired outcome is called training. This training process can be formulated as an optimization problem, where we try to minimize the difference between the current output of the network and the desired one. This difference is a measure of the error of the system, and is commonly referred to as the loss function. A common formulation

for the loss function is the Mean Squared Error:

$$\frac{1}{n}\sum_{i=1}^{n}(Y_i - Y_i^{'})^2 \tag{3.2}$$

Here n is the number of examples, Y_i is the output of our model for the ith input and $Y_i^{'}$ is the expected output. In order to evaluate a loss function, we must have a set of ground truths or correct answers for a given input that are representative of the problem or function that we want to learn. This set of examples forms the dataset that is used for training the network. In order to train the network, we calculate the loss function over a set of examples and use that value to update the hyper-parameters such that the error is reduced. The set of examples used to calculate the loss function and update the model is called the batch. By subsequently performing batch updates such that the entire dataset is presented to the network once, we conduct an epoch of training. Commonly, several epochs are needed for the neural network to achieve acceptably low scores in the loss function, meaning the model is imitating correctly the desired function.

There are many possible strategies to minimize the loss function, and this has been a very active field of research. Optimization strategies draw from optimization theory. There is available a large body of powerful optimization methods, however, it must be kept in mind that it is not enough to score low on the loss function. The model also needs to perform correctly beyond the training dataset: it needs to generalize to unforeseen data. If the optimization policy is too aggressive, the model can be overfitted to the dataset. A way to monitor the generalization potential, is to split the dataset into a training subset, used for performing the batch updates, and an evaluation dataset. By computing the loss on the evaluation dataset, we can get an estimation on the generalization error. It is a good practice when training neural networks to periodically check the generalization error as well. The moment this error stops decreasing (independently of what the training loss function states) is a signal that overfitting is starting to occur, and training must be stopped.

The last bit needed to understand how the loss function is used to train the network is understanding an optimization scheme. A very simple yet effective optimization technique that is the basis of most of the methods used by the state-of-the-art is the Stochastic Gradient Descent (SGD).

$$W_{t+1} = W_t - \eta \nabla E|_{W_t} \tag{3.3}$$

The terms W_{t+1} and W_t represent the values of the weight matrix at times t and $t+1$ respectively. $E|_{Wt}$ is the loss function for the model using the W_t weights. η is a hyper-parameter called the learning rate. By computing the gradient w.r.t the weights of the loss function at time t, we obtain the direction in the hyper-parameter space in which the error is maximized. Then, by subtracting this gradient from the current weights, we ensure that the error is reduced in the next iteration. Since the error and weight updates propagates from the output of the model to the input, performing one of these updates is commonly referred to as backpropagation of the error, or backwards pass.

As it might be obvious from equation 3.3, SGD is an iterative algorithm. We start the training process with a randomly initialized weight matrix, and each update of W results in a slightly different response of the model, which in turns changes the score on the loss function (with every step it becomes lower and lower). Each step t is a new example (or batch of examples) in the training dataset, thus an epoch is completed once t reaches the last element. The learning rate is the amount by which the network is changed in every iteration. Low learning rates mean very little shift in the response of the model, which makes training take longer. On the contrary, high learning rates change too much the response of the network, and might make it skip valleys where the local minima is, thus preventing the network from learning. Frequently, a dynamic learning rate is used, that changes with t. The policy for these dynamic value tends to use high coefficients early in the training process, to make sure we shift away from the randomly-initialized, poor-performance model, and gradually decreases the rate as the learning evolves, to ensure further examples just fine-tune the model without making drastic changes. This dynamic learning rate is an example of one of the many modifications that can be done to the backprogagation algorithm. Pure SGD is rarely used nowadays for training neural networks. A more powerful alternative, which uses some of these adaptive techniques and is available out of the box in the major machine learning frameworks is the RMSprop optimizer[1].

3.2 Convolutional Neural Networks for Image Processing

Let us consider a grayscale image of 28×28 pixels. Let as contemplate the problem of classifying which digit, from 0 to 9 is depicted in said image. In order to solve such a problem with a MLP we would need to flatten the image to a vector. The simplest architecture feasible for this problem would be a 1-layered network with 784 units at the input and 10 outputs (one for each character, for a 1-hot class representation). The number of parameters needed to model such a small network is 7.850 accounting for

[1] https://keras.io/api/optimizers/rmsprop/

both weights and biases. Now consider a slightly larger image of 128×128 pixels. The number of parameters grows to 163.850. It is evident that this type of architecture does not scale well, more so if we consider deeper models.

Another limitation of MLPs applied to image related taks is the fact that the isolated value of a pixel is rarely relevant. Features in an image are localized to a region of the image, and have some spatial structure. Take the example of face detection. Useful features to determine whether an image contains a face or not could be straight lines, which might denote a jawline or a nose, and oval edges, which could identify an eye. Learning such relationships with a fully connected layer can be hard. Moreover, MLPs lack a fundamental quality for robust image understanding: position invariance. Consider a dataset where most of the relevant information is displayed at one corner of the image. Because each node in the input is tied to a specific pixel in the image, features learned for this dataset will not generalize to examples where the interesting information is predominant in the opposite corner.

One family of architectures better suited to handle images are Convolutional Neural Networks (CNNs) [30]. CNNs solve both of the drawbacks of MLPs by using small convolutional filters instead of fully connected layers. Firstly, instead of working on the totality of the image, CNNs focus on small regions determined by the size of the filters. Thus, local features can be detected. Secondly, The same set of filters are used across the entire input by sweeping the image on both dimensions. This means that the number of parameters can be highly reduced. In order to propertly understand how this is possible, it is necessary to further explain how a convolutional filter looks like.

3.2.1 Convolutional Layers

The formal definition of a convolution between two matrices $\boldsymbol{A}, \boldsymbol{B}$ of similar dimensions is:

$$c = \sum_i \sum_j a_{i,j} b_{i,j} \tag{3.4}$$

That is, a convolution is the scalar value that results from the sum of the element-wise multiplication between the two matrices. In our example $\boldsymbol{A}$ is the input signal and $\boldsymbol{B}$ the filter. For practical applications, the input is larger than the filter. In this case, the convolution is applied by overlapping the filter over the input and systematically shifting it across the rows and columns. The amount by which the filter is shifted is the stride. The output to the convolution is another matrix whose size is invertly proportional to the size of the filter and the stride. The following is an example of a convolution with a stride of 2:

$$\begin{bmatrix} 8 & 2 & 5 & 8 \\ 5 & 1 & 2 & 7 \\ 8 & 5 & 3 & 0 \\ 7 & 8 & 1 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 4 \\ 6 & 2 \end{bmatrix} = \begin{bmatrix} 48 & 63 \\ 86 & 13 \end{bmatrix} \tag{3.5}$$

Here the 4×4 matrix is the input and the 2×2 matrix represents a single filter, that when applied with a stride of 2 results on a 2×2 output. The colors in the input represent the overlap with the filter that leads to the same colored output. A convolutional layer is typically composed of more than one filter, forming a tensor of $n \times m \times d$ where n and m are the size in pixels of the individual filter and d is the number of filters.

A filter is a set of values spatially arranged such that they yield a high output for a specific pattern in the input image. By tuning a bank (volume) of filters to different patterns of interest, we can identify relevant features in the image to extract information. The following matrix could be used as a vertical edge filter:

$$\begin{bmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{bmatrix} \tag{3.6}$$

This, along with more oriented edges and other primitive patterns can help understand the geometrical properties of an image. The output of these filters can be further processed by another bank (convolutional layer) to gain understanding at a higher level of abstraction. For example a filter could tune in to faces by looking at a combination of ovals and triangles denoting the eyes and nose.

3.2.2 Transposed Convolutional Layers

Transposed convolutional layers are some times referred to as de-convolutional layers. This term, however is not accurate. A de-convolutional layer should perform the inverse operation of a convolution. Instead, a transposed-convolutional layer works in a similar fashion as the convolutional one, but upsampling the output rather than downsampling. To understand why a transposed convolutional layer is called so, we must first look at how a normal convolution operation can be expressed in matrix form.

Consider the following convolution with a stride of 1:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} * \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} o_{11} & o_{12} \\ o_{21} & o_{22} \end{bmatrix} \tag{3.7}$$

The flattened output is computed as follows, according to the definition in 3.4:

$$\begin{bmatrix} o_{11} \\ o_{12} \\ o_{21} \\ o_{22} \end{bmatrix} = \begin{bmatrix} a_{11}w_{11} + a_{12}w_{12} + a_{21}w_{21} + a_{22}w_{22} \\ a_{12}w_{11} + a_{13}w_{12} + a_{22}w_{21} + a_{23}w_{22} \\ a_{21}w_{11} + a_{22}w_{12} + a_{31}w_{21} + a_{32}w_{22} \\ a_{22}w_{11} + a_{23}w_{12} + a_{32}w_{21} + a_{33}w_{22} \end{bmatrix} \tag{3.8}$$

To achieve the same result through a matrix multiplication, we can flatten the input row-wise into a 1×9 vector, and re-arrange the kernel coefficients in a 9×4 matrix like:

$$\begin{bmatrix} a_{11} \\ a_{12} \\ a_{13} \\ a_{21} \\ a_{22} \\ a_{23} \\ a_{31} \\ a_{32} \\ a_{33} \end{bmatrix}^T \begin{bmatrix} w_{11} & 0 & 0 & 0 \\ w_{12} & w_{11} & 0 & 0 \\ 0 & w_{12} & 0 & 0 \\ w_{21} & 0 & w_{11} & 0 \\ w_{22} & w_{21} & w_{12} & w_{11} \\ 0 & w_{22} & 0 & w_{12} \\ 0 & 0 & w_{21} & 0 \\ 0 & 0 & w_{22} & w_{21} \\ 0 & 0 & 0 & w_{22} \end{bmatrix} = \begin{bmatrix} o_{11} & o_{12} & o_{21} & o_{22} \end{bmatrix} \tag{3.9}$$

From linear algebra, we now that a vector of size 1×9 multiplied by a matrix sized 9×4 results in a vector sized 1×4. This in unflattened dimensions means going from a 3×3 input down to a 2×2 output. Applying the same rule, we can easily go from a 2×2 input into a 3×3 output simply by **transposing** the weight matrix.

The right hand side term of the following equation is achieved by multiplying a flattened 2×2 B matrix with the transposed of the weight matrix in 3.9:

$$\begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} *^T \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} =$$

$$\begin{bmatrix} b_{11}w_{11} & b_{11}w_{12} + b_{12}w_{11} & b_{12}w_{12} \\ b_{11}w_{21} + b_{21}w_{11} & b_{11}w_{22} + b_{12}w_{21} + b_{21}w_{12} + b_{22}w_{11} & b_{12}w_{22} + b_{22}w_{12} \\ b_{21}w_{21} & b_{21}w_{22} + b_{22}w_{21} & b_{22}w_{22} \end{bmatrix}$$

Similar to the earlier example with the convolution, here we are using colors to denote the source of information from input to output. In the case of the normal convolution, we would go from a sub-matrix in the input to a scalar in the output. With the transposed convolution, however, we go from a scalar value in the input matrix to a sub-matrix in the output.

3.2.3 Fully Convolutional Networks

A fully convolutional network makes no use of fully connected layers (like the ones in the MLP), only convolutional and transposed convolutional layers. They are typically arranged such that the first half of the network uses convolutional filters to extract features from the input, compressing the information to a so called latent-space. From this high level of abstraction representation of the input, the second stage of the network uses transposed convolutions to upsample the information. If the upsample is carried up to the original dimensions of the input, a pixel-level inference can be achieved.

3.3 Transfer Learning

Transfer learning is a technique that allows us to use the knowledge learnt by a model in a different architecture or to solve a different problem. The model that donates the weights is commonly called the teacher, and the receiver the student. Knowledge transfer helps bootstrap the training of a student, by starting from an already decent place in the hyper-parameter space. One example of a transfer learning method widely used in the literature, is using the weights of the convolutional layers of a successful network such as Inception [31] for feature extraction, and combining them as input to our problem-specific layers. This way we don't need to re-learn common traits like corners or edges, and we can focus our training on finding the relationships of higher-order pieces of information.

3.3.1 Net2Net

Net2Net [27] is a method that lets us efficiently add units to a network. It introduces two knowledge transfer operators: *net2deepernet*, to increase the number of layers of the model and *net2widernet* to increase the number of units (filters, in the context of convolutional networks) in a layer. Both operators work in a similar fashion: they initialize the new weights in such a way that the output of the student network remains

the same as that of the teacher. By doing this, the performance of the model is not reduced and less number of epochs are needed to teach the network to use the new units.

Net2DeeperNet

Consider a teacher network $Y = \phi(W^T x)$ where ϕ is the `ReLU` activation function. Since `ReLU` is idempotent, that is, since we can write $\phi(x) = \phi(\phi(x))$ for any x, then the following holds true:

$$Y = \phi(W^T x) = \phi(I\phi(W^T x)) \tag{3.10}$$

Simply by initializing the new layer to the identity matrix we prevent the loss of knowlegde.

Net2WiderNet

The process of adding units to a layer is slightly more involved. This is due to the fact that we must change the size of the inbound and outbound weight matrices. Consider the following network with one hidden layer (note that for simplicity the activation function has been omitted):

$$\text{Hidden layer :}$$
$$\begin{bmatrix} h_1 \\ h_2 \end{bmatrix} = \begin{bmatrix} a & c \\ b & d \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\text{Output layer :}$$
$$y = \begin{bmatrix} e \\ f \end{bmatrix}^T \begin{bmatrix} h_1 \\ h_2 \end{bmatrix}$$

In order to add an additional unit in H, we need one more row in the inbound weight matrix. We can do this by replicating one of the existing sets of weights. This, however, changes the output value and shape of the hidden layer. To mitigate it, and ensure the final output remains constant, we replicate the same position on the outbound layer, and divide it by 2 like:

$$\text{Hidden layer :}$$

$$\begin{bmatrix} h_1 \\ h_2 \\ h_2 \end{bmatrix} = \begin{bmatrix} a & c & c \\ b & d & d \end{bmatrix}^T \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\text{Output layer :}$$

$$y = \begin{bmatrix} e \\ f/2 \\ f/2 \end{bmatrix}^T \begin{bmatrix} h_1 \\ h_2 \\ h_2 \end{bmatrix}$$

This algorithm can be generalized to any number of new units in a layer. Consider $W^{(i)}$ and $W^{(i+1)}$ as the inbound and outbound weights to the affected layer in the parent network, and $U^{(i)}$ and $U^{(i+1)}$ as the corresponding weights in the student network. Let us call m the number of units in the teacher layer, and n the desired total number of units for the student layer. To construct $U^{(i)}$, the first m columns will be a literal copy of $W^{(i)}$. The remaining $n - m$ columns will be randomly selected from $W^{(i)}$. Similarly, the first m rows of $U^{(i+1)}$ are taken directly from $W^{(i+1)}$. The remaining rows follow the exact same random mapping as in the inbound. That is, if column $m + 1$ in $U^{(i)}$ used column $m - 3$ from $W^{(i)}$, row $m + 1$ in $U^{(i+1)}$ will use row $m - 3$ in $W^{(i+1)}$. In order to preserve the output, each row in $U^{(i+1)}$ will be divided by how many times the donor row in $W^{(i+1)}$ was selected.

Chapter 4

Methods

This work presents two main contributions: an extension of GG-CNN that improves its accuracy and a pipeline to enable safe grasping for off-nadir camera angles. In this chapter, we explain the algorithm to methodically explore the GG-CNN variations and we go over the details of our grasping pipeline. Furthermore, we provide some insights into our simulation environment for grasp evaluation.

4.1 GG-CNN

As already described, GG-CNN is a fully convolutional network. That is, instead of having an output with the representation of a single grasp (the 5 parameters discussed in section 2.1.2), it outputs a dense prediction for each pixel in the original input. It achieves this by using 3 transposed-convolutional layers after the initial 3 convolutional layers (see Figure 4.1). The input to the model is a depth-only image (no color information), and it has 4 separate outputs:

- **Grasp Quality:** Normalized between $[0, 1]$, it represents the probability of successful grasp of the pixel.
- **Angle:** The angle is represented as points in the unit circle by two images, one for each component, with values between $[-1, 1]$.
- **Width:** The width is also given in normalized units and then remapped to a range of $[0, 150]$ pixels.

While the network generates 300^2 grasp assessments, for practical applications we typically want just the best grasp. The output of the network is postprocessed to facilitate

grasp generation. The grasp quality distribution is filtered by a Gaussian blur. The angle is computed from the two components as $\frac{1}{2}\arctan\frac{y}{x}$, so that the values are comprised within the range $[-\frac{\pi}{2}, \frac{\pi}{2}]$. The width is rescaled to the range of $[0, 150]$ pixels and also filtered by a Gaussian. The image coordinates of the best grasp are obtained by finding the best local maxima in the smoothed grasp quality distribution, which are then used to find the angle and width. Knowing the intrinsic and extrinsic parameters of the camera, and having the depth from the input image it is possible to transform this image-space grasp representation to world-space coordinates.

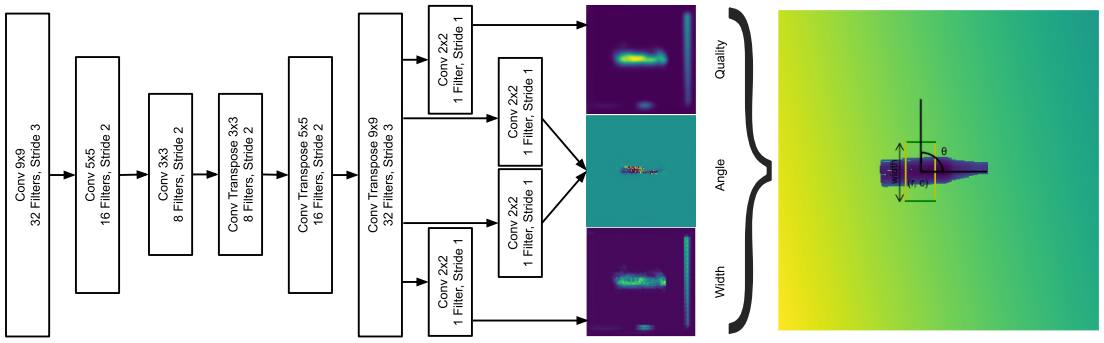

FIGURE 4.1: The GG-CNN architecture. The 4 convolutional filters of the last layer generate the grasp-quality and width images, and the 2 angle images. The best grasp is isolated from the quality distribution, and the images are post-processed to generate a bounding-box representation. The rightmost image overlays this bounding box representation over the input depth image.

4.1.1 Datasets and Training

Since the goal of GG-CNN is to generate a grasp distribution, the network can be trained on any grasping dataset as long as it presents several grasp examples per image. The loss function used is the pixel-wise MSE on each of the outputs, and the optimizer used is the *RMSProp*. In our experiments we trained the network on Cornell, using the same approach to that of the original publication.

4.2 Architecture Modification

One of the objectives of this work is to find an improved version of the GG-CNN architecture. Coming up with a new model is a highly involved process: the combinatorics of the modifications that can be done to a neural network is such that an exhaustive approach is not feasible. The method proposed in [28] solves this issue by formulating the architecture exploration as a search problem, and using the beam search algorithm to solve it. We employ this method in our work to grow the base GG-CNN network.

4.2.1 Beam Search

Breadth-first is an exploration strategy for state space search problems where all the children of a given level in the search tree are expanded before the offspring is evaluated. For problems where the search space is infinite, this guarantees that if there is a solution it will be found (it does not get stuck on local optima). However, if the branching factor is too big the number of nodes to be explored before a solution is found can be prohibitively large. Beam search is a greedy variant of breadth-first. Instead of considering all nodes at a depth level, it evaluates them according to some heuristic and selects the best for expansion. It is governed by a beam size parameter k that limits the number of nodes to expand. Due to its greedy approach, beam search can be misguided and get lost on local optima. An example of such a case can be seen in Figure 4.2. A solution to this is guiding the search by a depth-limited lookahead, albeit this increases the execution time of the algorithm considerably.

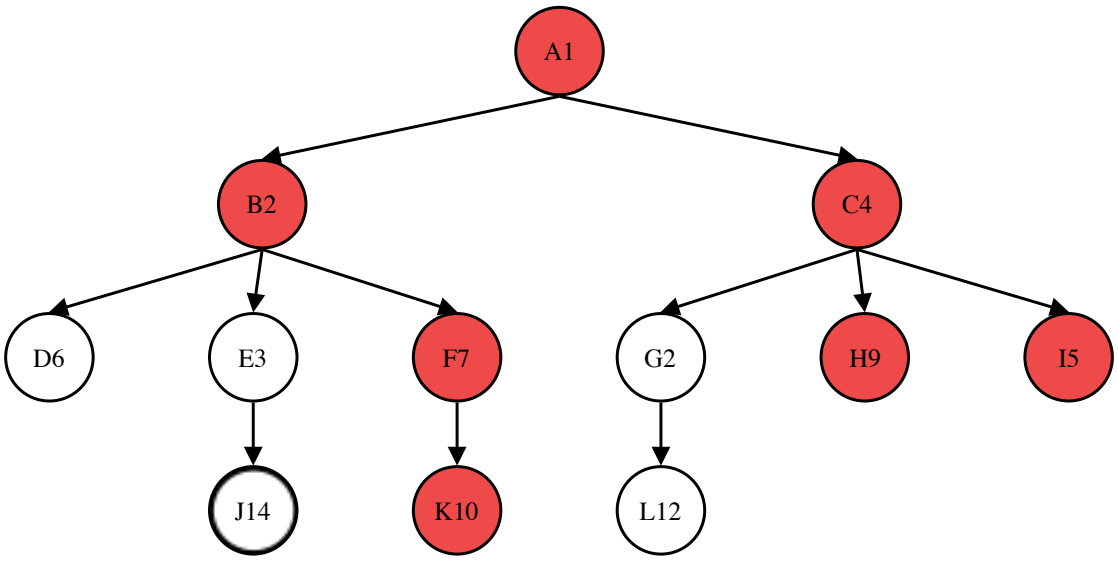

FIGURE 4.2: Example of exploration of a graph using beam search with a beam size of 3. The characters identify the node and the digits the value of the heuristic. The objective of the search is to maximize the heuristic. Nodes in red have been expanded. In this case the algorithm is miss-guided by the local maxima at depth 2, and it fails to find the optimal solution of node J

4.2.2 Beam Search for Architecture Optimization

Beam search can be easily employed to systematically explore the architecture space of a model. The nodes in the tree search are the instances of a model, and the branches correspond to modifications to the models. An example of a possible modification is adding filters to a convolutional layer. Applying any modification means changing the hyper-parameters of a model, therefore the network needs to be re-trained. Training time

can be cut short if the network modifications preserve the knowledge of the parent model. This way, only a few epochs are needed to teach the network to use the newly added units, without having to re-learn past knowledge. Knowledge-preservation between parent and offspring is crucial for this algorithm to be effective, since re-training the network from scratch after every modification is unfeasible in practice. Therefore, the number of modifications that can be done to a given network is limited by the amount of available knowledge transfer operators. In this work we use the two operations described in section 3.3.

Listing 4.1 shows a python-pseudocode implementation of the beam search algorithm for architecture optimization. Since a base model can be infinitely grown, the algorithm is depth limited and aims to find the best architecture within `d` modifications. From lines 8-10 it is clear that the branching factor increases with the depth, which makes searching into higher depths very costly. The `wider` and `deeper` functions implement the knowledge transfer operators, adding a copy of the layer below the currently selected layer, and duplicating the number of filters in the selected layer respectively. The `sort_by_accuracy` method evaluates the newly trained nodes against the validation split of the dataset, and this score is used as the heuristic to guide the search. The best `k` nodes are selected to be expanded on the next iteration.

```
1  def beam_search(start_model, k, depth):
2      queue = [start_model]
3      explored = []
4
5      for d in depth:
6          children = []
7          for model in queue:
8              for layer in model:
9                  children += train(wider(model, layer))
10                 children += train(deeper(model, layer))
11         children = sort_by_accuracy(children)
12         queue = children[:k]
13         explored += queue
14
15     return explored
```

Listing 4.1: Architecture exploration using Beam Search

Lookahead

To mitigate the effect of getting stuck on local optima, as explained earlier, we can guide the search with a lookahead. The lookahead changes the heuristic to select the modification based on which will result in a better model at a depth `d`. This `d` is fixed, that is, the algorithm never explores deeper. Lookahead improves the search not only because it looks at the score further in the tree, but also because with each iteration the algorithm trims the breadth of the tree, allowing it to focus its beam on the most promising offspring. Listing 4.2 shows the pseudocode for the algorithm. Note how in line `4` the depth is capped to `d` (relative to the initial model). The method `select_best_leaf` looks at the last `k` nodes from the `beam_search`, which represent the leaves of the lookahead, and selects the best. From this leaf, we find the immediate children from the model that leads to it. This model will then be selected as the new root for the next lookahead. Notice how in line `8` the selected child is trained further. This is a way to balance training and execution time: we train the offspring only for a few epochs during lookahead, and for longer after selecting the best node at each level.

```
1 def beam_search_with_lookahead(model, k, depth)
2     models_e = [model]
3     for d in depth:
4         models = beam_search(depth-d+1,k, model)
5         leaf = select_best_leaf(models)
6         while parent(leaf) is not model:
7             leaf = parent(leaf)
8         model = train(leaf)
```

Listing 4.2: Beam Search with lookahead

4.3 Orthographic Pipeline

In this section, we introduce a pipeline that allows the use of GG-CNN for off-nadir grasps prediction. The pipeline works on a point-cloud representation of the scene. It segments the object of interest and identifies its canonical views (front, side, top) w.r.t the camera. For each of the views, a depth image is synthesized, and prioritised according to an entropy-based measure of the information it carries. Then, the view with the highest level of information is fed to the network to predict a grasp point. For a schematic of the pipeline, see Figure 4.3

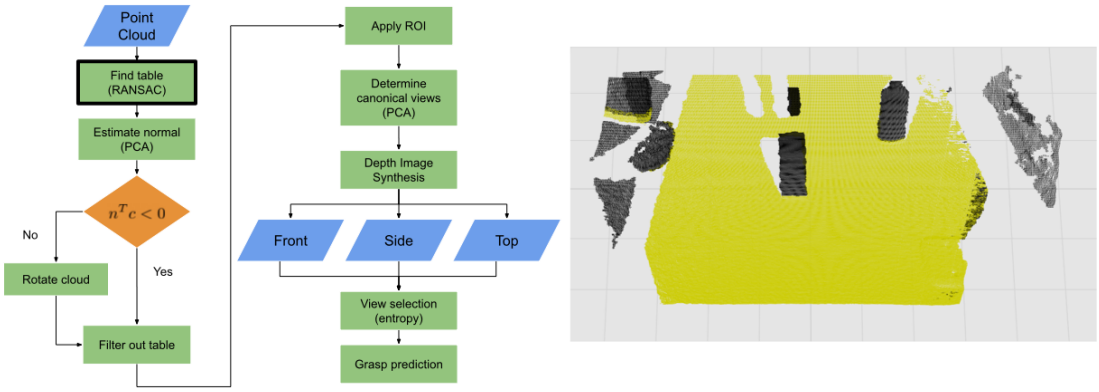

FIGURE 4.3: Overview of the proposed orthographic pipeline: (left) flow chart for our proposed orthographic pipeline. (Right) example of a point-cloud captured with a real RGB-D sensor. The plane corresponding to the table (yellow) is identified via RANSAC. The object in the center of the scene is isolated and its orthographic views are projected.

4.3.1 Canonical Views

The point cloud is received w.r.t. the camera frame. First, the largest plane (table) is identified using RANSAC [32]. PCA is used to find the main axes of the plane, and the vector with the smallest eigenvalue is chosen as the normal of the plane. The orientation of the normal is selected such that the cosine between the normal and the camera vector is negative (so that it points "up").

In this work, we assume that there is only one object to be grasped, and that it lays within a known region of the workspace. Object segmentation is performed by discarding all the points outside the ROI and also removing those that belong to the plane. The next step is to determine the main axes of the target object. These axes should maximise the information that is contained on each of the faces, but they should also facilitate a safe approach direction for the grasp. An unsafe approach is one that results in collisions with the workspace surface, for example approaching from below the table. To avoid this, the normal of the plane is always used as the z axis of the object. The other two components are found by performing PCA.

The main axes x, y, z of the object cloud determine the canonical views of the object, which are used for grasp prediction and grasp approach orientation. The top view approach vector is $-z$. The approach vector for the front view is computed as:

$$\boldsymbol{a_f} = \underset{\boldsymbol{a} \in \{\boldsymbol{x}, \boldsymbol{y}\}}{\arg\max}(|\boldsymbol{a}^T \boldsymbol{v_c}|) \tag{4.1}$$

$$\boldsymbol{a_{af}} = \boldsymbol{a_f} \operatorname{sign}(\boldsymbol{a_f}^T \boldsymbol{v_c}) \tag{4.2}$$

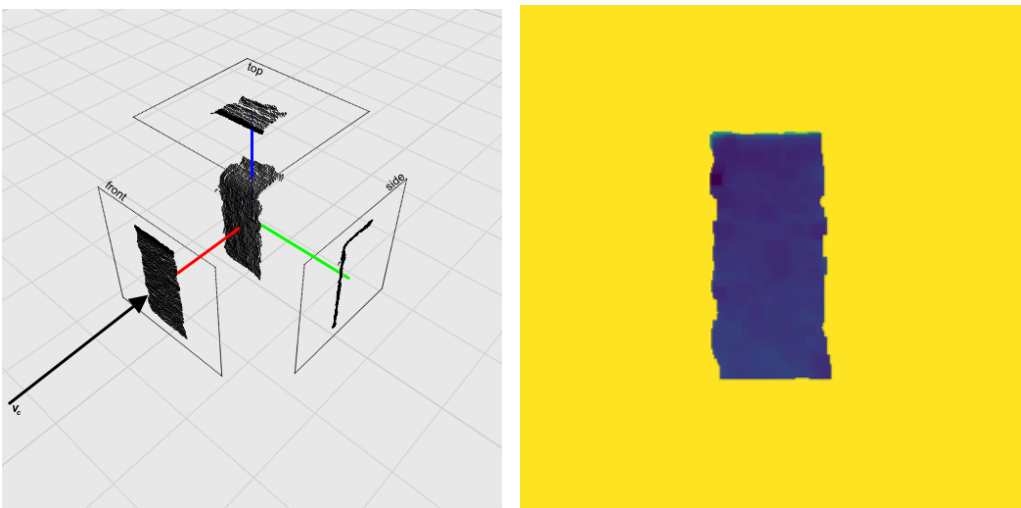

FIGURE 4.4: (Left) Point cloud of a real object and its projected canonical views. Top view is defined by the normal to the world plane. (Right) Synthetic depth image derived from the frontal projection of the point cloud

Where $\boldsymbol{v_c}$ is the camera aim vector and $\boldsymbol{a_f}$ and $\boldsymbol{a_{af}}$ are the front axis and front approach axis respectively. The remaining axis determines the side view, and the approach orientation is computed similarly to the front view. One limitation of PCA is that while the direction of the axes for a given cloud is unique, there are two possible orientations for each axis (there is a sign ambiguity). In our case, this means that the approach direction could potentially be from the back of the object (w.r.t the camera). This is not desirable, since the back view contains the highest uncertainty due to self-occlusion. To overcome this, we always select the orientation opposite to the camera vector (i.e. the sign term in equation 4.2 will always be -1). This is similar to approaching the object from the side closest to the camera.

4.3.2 Depth Image Synthesis

A depth image is synthesized based on the visible points of the object cloud from each of the canonical views. The virtual camera is aimed following the approach vector described above. The object is centered in the image, and a padding is added to limit the size of the object so that it resembles the dataset images. For each point in the object cloud the corresponding row r and column c indices in the image are computed according to the following equation:

$$\begin{bmatrix} r \\ c \end{bmatrix} = \begin{bmatrix} l - 1 - \lfloor \frac{p_v}{p_s} + \mu_r \rfloor \\ \begin{cases} l - 1 - \lfloor \frac{p_h}{p_s} + \mu_c \rfloor & \text{if } \boldsymbol{a_{ap}} \in \{\boldsymbol{y}\} \\ \lfloor \frac{p_h}{p_s} + \mu_c \rfloor & \text{else} \end{cases} \end{bmatrix} \tag{4.3}$$

Where:

- l is the width or height of the image (300 pixels, to match the input of GG-CNN).
- μ_r and μ_c are the row and column indices of the image center.
- p_v and p_h are the horizontal and vertical components of the point w.r.t to the center of the cloud. For the *front* and *side* views the vertical component is z and the horizontal component is the remaining one orthogonal to the approach vector (either x or y). For the *top* view the vertical component is that which is colinear with the *front* approach vector.
- p_s is the pixel size (world units per pixel).

The pixel size is computed based on the dimensions of the AABB (Axis Aligned Bounding Box) according to:

$$\mathbf{p_{max}} = \begin{bmatrix} \max \mathbf{C_x} & \max \mathbf{C_y} & \max \mathbf{C_z} \end{bmatrix}$$

$$\mathbf{p_{min}} = \begin{bmatrix} \min \mathbf{C_x} & \min \mathbf{C_y} & \min \mathbf{C_z} \end{bmatrix}$$

$$p_s = \frac{\max(\mathbf{p_{max}} - \mathbf{p_{min}})}{l - 2d - 1} \tag{4.4}$$

where $C_i, i \in x, y, z$ denotes the components of all the points in the object cloud and d is the padding. The same size is used across all 3 views, to ensure a uniform representation of the dimensions of the object. The depth value of each pixel is taken from the component that is colinear with the approach vector of each view. Since the approach vector has the opposite direction to the corresponding view axis, positive values occlude negative ones in the case that more than one point belongs to the same pixel. Missing pixels, for which there is no information available are identified as $-\infty$

Generating a depth distribution directly from the points in the cloud results in a very sparse image, since each point fills one pixel at most. To achieve more realistic images, a dilation with a radius of 3 is applied to non-missing pixels, taking into account depth occlusion. Missing values are replaced by the lowest known depth value. At this stage, points that are further from the virtual camera have a lower value than points that are closer. The depth images from real sensors encode actual distance to the camera along the camera vector, so we invert the depth values to match this. Since the depth

values are expressed w.r.t to the center of the object cloud, the resulting image is already distance-invariant. As a final step, we smooth the image by applying a gaussian filter.

4.3.3 View Selection

Out of the 3 synthetic depth images, not all will contain sufficient information to estimate a valid grasp, due to self-occlusion. An example of a potentially problematic image can be found in the side view of Figure 4.4. The view with the highest uncertainty depends on the position of the camera relative to the object. A method must be found to robustly determine the best view for prediction. In [33, 34] entropy was proposed as a way of prioritising orthographic projections. Entropy is a concept from information theory that measures the amount of information in a message as the unpredictability of its contents [35]. Messages whose content is highly predictable (e.g. outcomes from a loaded dice) contain less information that those which are unpredictable (throws from a fair dice). Formally, information entropy is defined as follows:

$$H = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{4.5}$$

Applied to the problem of view selection, the summation is iterated over the pixels in the image, and p_i represents the probability that the pixel is occupied by a point (the number of points assigned to that pixel over the total number of points in the object cloud). The base of the logarithm can be arbitrarily chosen, since it only affects the scale, however 2 is a common choice in the literature. The singularity at $p_i = 0$ does not contribute to the final entropy value (i.e. missing values are not considered). Under this formulation, the highest possible value of entropy is that of an image where all the pixels have the same number of points assigned. In opposition, the lowest values of entropy are yielded when most of the points are assigned to a reduced number of pixels. This will happen for views with highly occluded data. Since those are the views we want to avoid, we select that image which maximizes entropy.

4.4 Simulation Environment

One of the challenges in grasp research is the lack of standardized benchmarks. Task performance can be considerably affected by factors like the set of test objects, the capability of the end effector, the precision of the robotic arm, the quality of the motion planning and ultimately, the accuracy and robustness of the grasp pose prediction. In [22] authors propose a benchmark that employs simulation to test grasp points on a

large dataset of objects. They make their simulator available as a web service for other researchers to test their prediction algorithms under the exact same conditions. At the beginning of the project, the use of said simulator was attempted in order to establish a baseline. Unfortunately, the results never arrived. In order to have full control over the simulation and evaluation aspect and ensure quick turnaround of results, it was decided to implement a custom simulation.

4.4.1 Pybullet

We base our simulation on Pybullet [36], a Python wrapper for the Bullet [37] engine. Bullet and Pybullet have become the de facto physics simulation engine for Machine Learning. Companies like Google Brain and OpenAI have employed it to train and evaluate their Reinforcement Learning algorithms [38–40], and it is the engine of choice of the Jacquard simulator team. It allows us to easily model rigid body physics and friction interactions, essential for simulating something like a pick and place task.

4.4.2 Experimental Setup

In order to simplify the implementation, we are using a free-floating custom end effector with 6DoF and unlimited workspace, instead of a realistic armed manipulator. Our custom end effector is a simple two-jaw gripper. Figure 4.5 depicts the gripper in a fully opened configuration.

For each grasping trial, a pre-initialized scene is loaded containing the object to be grasped in a stable configuration. The gripper is spawned one meter above ground, and a target bin is placed two meters away from the object. The purpose of the bin is to constrain the motion of the object once dropped. Upon grasp prediction, the gripper first moves to a pre-grasp pose, matching the final orientation but 10 cm above the final point. This is to avoid any rotation near the object that might disturb it and end up in grasp failure. After gripping, the manipulator lifts the object vertically and translates towards the bin, so that the CoM of the object lays at the baricenter of the box. The objects are then dropped and the simulation is let run for a period of 20 seconds to let the scene stabilize. After that, if the CoM of the object lies within the boundaries of the bin, the grasp is considered to be successful.

4.4.3 Scene Initialization

Scenes are initialized by spawning the object centered in the world with a random orientation, at a height equal to its longest axis to avoid initial collisions with the

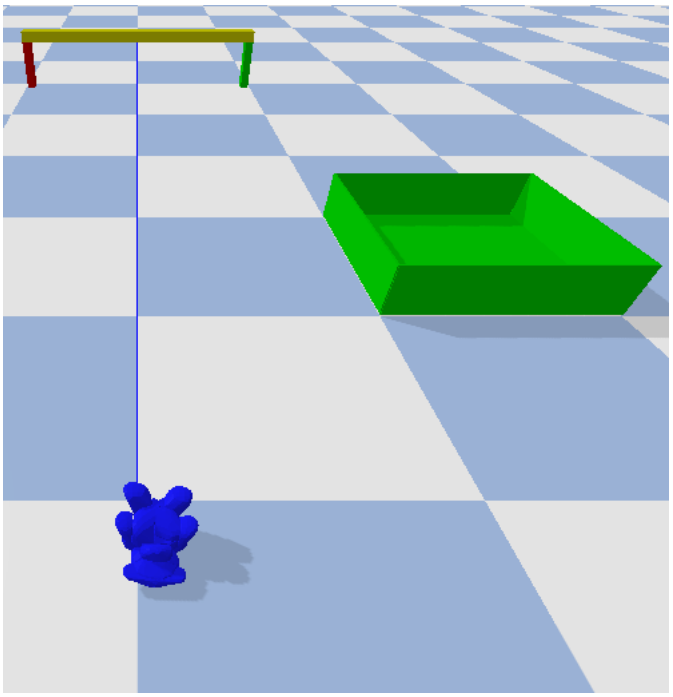

FIGURE 4.5: Simulation environment. The object must be dropped in the green bin for a grasp to be considered successful

ground plane. Objects are then let loose and the simulation is ran until the change in position is smaller than a given threshold or it times out. Those scenes that didn't time out are considered stable. For each stable scene, the state of the object (including position, velocity and acceleration) are appended to a hdf5 file along with the rgb and depth images aimed at the object. This way we ensure that all subsequent experiments are performed under the same conditions.

4.4.4 Objects Dataset

We use object models from ShapeNetSem [41], also used by Jacquard. ShapeNetSem is a heavily annotated compilation of a series of 3D object databases. This implies that the scaling of the objects is not homogeneous and in some cases it is unknown. To ensure that all the objects used could be grasped by our simulated gripper, we randomly subsampled a set of 40 objects and manually re-scaled them using Blender[1] to fit the gripper's dimensions. Figure 4.7 displays all the objects used in the simulations.

When loading an object onto Pybullet, two models can be provided, one for visual rendering and another for collision interactions. If only the visual model is provided, Pybullet attempts to use it as a collision model. However, visual models tend to be too

[1] https://https://www.blender.org/

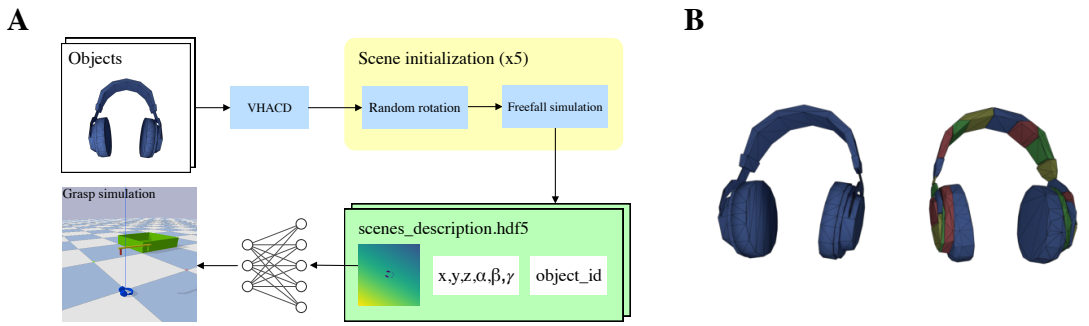

FIGURE 4.6: Overview of the experimental setup: (left) scene generation process and experimental setup. Scenes are batch generated and stored in a hdf5 file. For each object, we generate 5 scenes with random orientations. As part of the scene description we embed a depth image to ensure all predictions are carried on the same input; (right): result of the VHACD algorithm. Left object represents the original geometry. Each volume of a different color in the right object shows the convex decomposition that approximates the original shape.

complicated and make physics calculations inefficient. If this is the case, Pybullet optimizes the visual mesh reducing the number of polygons. In our experience, the default optimizations introduce ghost geometries, like closing the gap in a mug handle. This leads to unrealistic simulations, where the performance of the experiments are highly diminished. To overcome this, we make use of the Volumetric Hierarchical Approximate Convex Decomposition (V-HACD) algorithm. V-HACD decomposes a mesh into smaller chunks of lower resolution that approximate the original geometry, thus simplifying the physics calculations while staying true to the visual model. An example of how the algorithm works on one of our test objects can be observed in 4.6

4.5 Additional Software and Support Materials

The entire project is implemented in *Python 2.7*. For preprocessing the Cornell dataset we use the original script by the GG-CNN team. The neural network is implemented in *Keras* using the *tensorflow* backend. Image and point cloud processing are conducted using the SciPy toolkits[2] Model training and evaluation was conducted on NVIDIA K40s and V100s GPUs provided by the Peregrine HPC cluster[3].

[2]https://www.scipy.org/
[3]https://www.rug.nl/society-business/centre-for-information-technology/research/services/hpc/facilities/peregrine-hpc-cluster

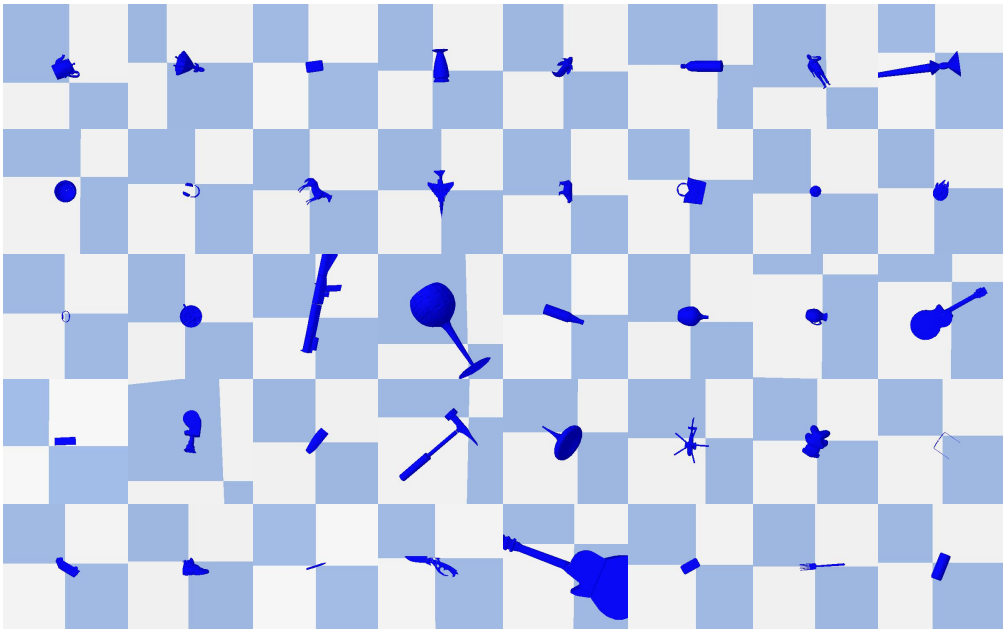

FIGURE 4.7: Objects used in the simulation environment.

Chapter 5

Results

In this chapter we present our experiments and interpret the results. We first establish a baseline, evaluating GG-CNN on our simulation environment. Next we show our proposed GG-CNN$^+$ architecture resulting from beam search. Finally, we evaluate both networks on off-nadir approaches and test our orthographic pipeline.

5.1 Simulation Baseline

We are using a state-of-the-art physics engine to develop our simulation environment. However, there are many factors that can have an influence on trial performance. In our case these can be the selection of objects, grip geometry, motion control gains and election of hyperparameters like simulation timestep which has an effect on the accuracy of friction physics. In order for our evaluation pipeline to be meaningful, we first need to validate it and set a baseline with which to compare future experiments.

To establish this baseline, we first evaluate the vanilla GG-CNN architecture with the published weights on our platform. We repeat the task for each objects 5 times with different orientations. We achieve a precision of 78.5% successful grasps, where each successful event accounts for 0.5%.

In order to understand the impact on the actual performance of network training, we re-train a randomly initialized GG-CNN on the Cornell dataset and evaluate each epoch in both simulation and IOU. The objective is to observe whether our simulation truly reflects the learning of the network. As such, we expect the scores to have a tendency to rise with the number of training epochs. Results can be observed in Figure 5.1. After one epoch, the network achieves a performance of 60%. Both simulation and IOU show a considerable improvement in performance in the first few epochs, and in both cases, the

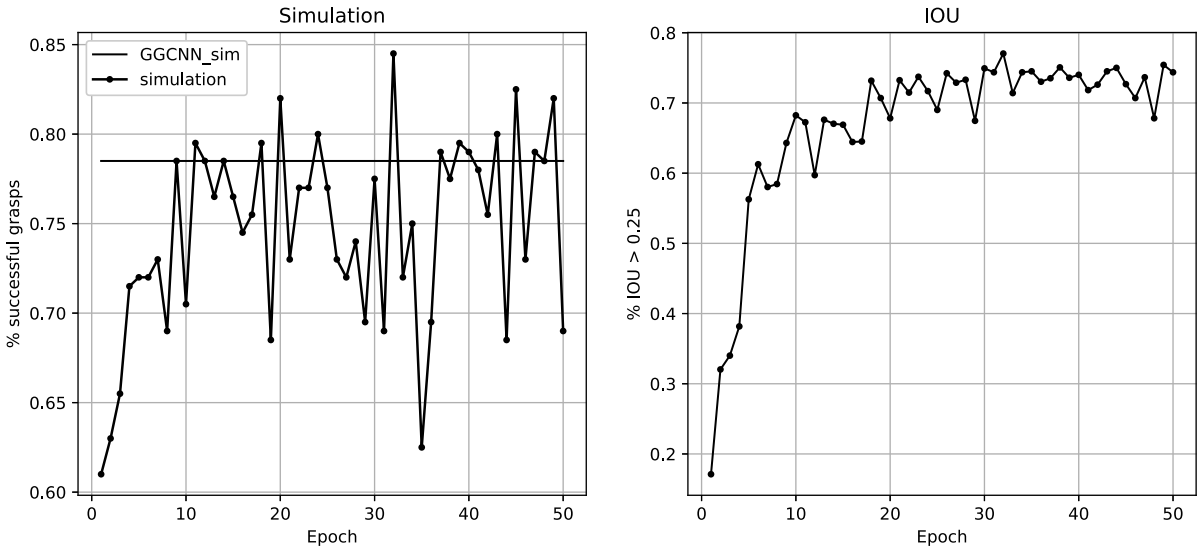

FIGURE 5.1: Simulation baseline: (left) evolution of the rate of successful grasps of a randomly initialized GG-CNN network. The published weights for GG-CNN achieves an accuracy of 78.5%. We find the best epoch for the randomly initialized one to achieve 82% of success rate; (right) performance of the randomly initialized network based on IOU > 25% criteria.

best epoch is number 32, which achieves a success rate of 82% in simulation. However, there is a clear distinction in between-epoch variability, being that of the simulator notably larger.

5.2 Beam Search

The attractiveness of methodically exploring the architecture space using beam search was two-fold for this project: improving the accuracy by adding parameters to the original model; or reducing the number of parameters while achieving a similar success rate.

To reach both goals we conducted two sets of experiments, starting with different architectures. Each candidate architecture was re-trained on Cornell. Two alternatives were considered for the search heuristic: IOU accuracy or simulation performance. The IOU metric, also known as box metric in the literature, consists on computing the intersect over union of the predicted grasp rectangle and the ground truth. If this overlap is $> 25\%$ and the difference in the orientation is less than $30°$ the grasp is considered valid. The advantage of IOU is the speed of evaluation. The advantage of simulation is the quality of the heuristic (since it reflects actual task performance). However our object

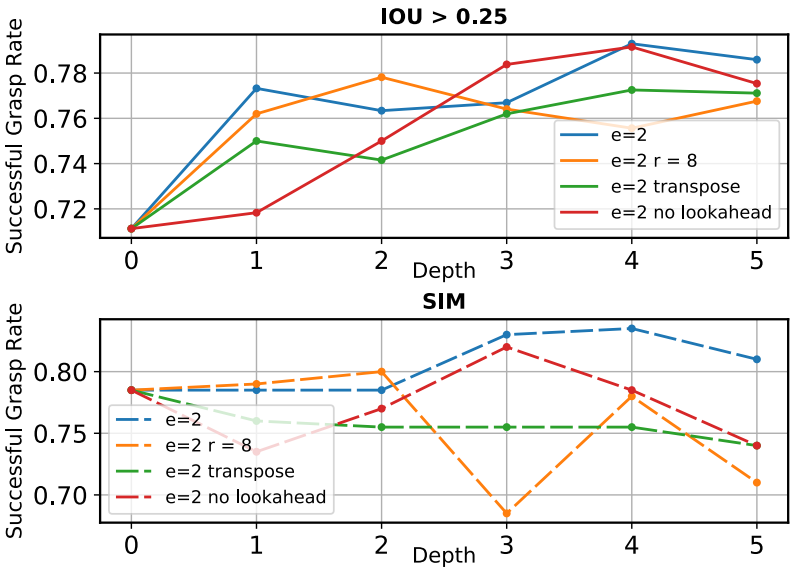

FIGURE 5.2: Results of the beam search with the vanilla GG-CNN architecture as the starting node. All experiments were ran to a depth of 5 with a beam size of 3. e stands for training epochs after applying a transfer operator. r is the number of retraining epochs for the best node after lookahead. All experiments selected only the convolutional layers for expansion, except for the `transpose` one, which also selects the transposed convolutional layers. The `no lookahead` experiment was ran without the lookahead step.

dataset was not large enough to be split into separate test/training subsets. Additionally, evaluating each candidate network on simulation would take orders of magnitude longer than on the IOU dataset. For these reasons the IOU method was selected. In the following subsections we explain the experiments in more detail.

5.2.1 Improving the Architecture

For these experiments, we used the vanilla GG-CNN network as the starting node, and only considered the `net2widernet` and `net2deepernet` transfer operators, meaning that for any given model, the number of possible children is 2 times the number of convolutional layers. We use the same values in the original beam search paper, for the depth limit and beam size. That is, we used a depth limit of 5 and a beam size of 3. The results for our experiments can be seen in Figure 5.2.

The lookahead step increases by several orders of magnitude the number of nodes that are explored, and consequently makes the algorithm take longer to reach the depth limit. This motivated the first experiment, comparing the benefits of using lookahead. The red line, labelled as "no lookahead" shows the results without lookahead (i.e. only running the algorithm in Listing 4.1. These results can be compared against the blue line, which

did make use of lookahead. The success rate in simulation is marginally lower for the experiment without lookeahead up until depth 3 where lookahead starts to pay off. This behaviour was to be expected: beam search is greedy and can easily be stuck on local optima. With the lookahead, we explore more of the offspring of the promising branches, which leads to better architectures.

Beam search can retrain the network at two stages: After applying a new transfer operator before each node is evaluated; and after a lookahead step. One possible way to improve the accuracy of the resulting model is training for longer. However, increasing the number of retraining epochs for each new candidate can become prohibitively costly due to the growing branching factor of the search tree. Nonetheless, some amount of training at this stage is required, otherwise, given the definition of our knowledge transfer operators no improvement can happen. Thus, the only way to achieve longer training in shorter a time is training very briefly after a modification is applied and retraining the best node after lookahead for longer. In Figure 5.2 we list the number of epochs used at each stage: e stands for training after expanding a node, and r stands for retraining after lookahead. We used 2 and 8 epochs respectively. It can be seen how the best possible result is obtained when training briefly after a modification is applied.

So far we have only considered convolutional layers for modification. This accounts, however, for only half of the layers in the architecture. The transfer operators can be applied to transposed convolutional layers in a similar fashion. We ran one experiment where both convolutional and transposed convolutional layers were allowed to be expanded. This increases the number of possible children for a node by an initial factor of 2, which also makes the algorithm take longer. The results can be observed in green, labeled as "transpose". We see the accuracy of this experiment monotonically decrease with depth, by a small amount.

We draw the conclusion from these experiments that the best results are achieved when only training for 2 epochs right after applying a transfer operator, and doing so only to convolutional layers, with an exploration strategy that uses lookahead. The best model using these settings is found at depth 4 and achieves an accuracy in simulation of 82.5%. This is a boost in accuracy of 5% w.r.t the vanilla GG-CNN model, by only increasing the number of hyperparameters of the model 8%. The resulting architecture can be seen in Figure 5.3. The modifications that lead to this architecture are listed in Table 5.1

5.2.2 Optimizing the Architecture

Beam search can only add units to an existing model. Thus, in order to achieve a smaller model than vanilla GG-CNN, we must start from a stripped down version. We

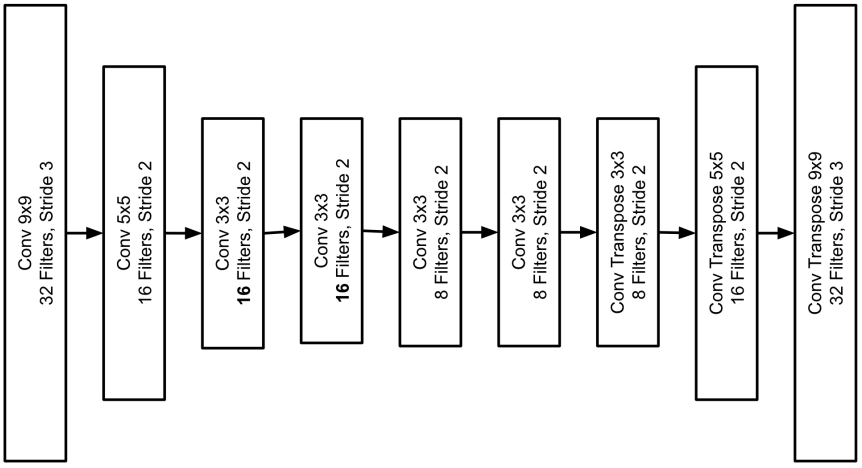

FIGURE 5.3: GG-CNN$^+$ (improved version of GG-CNN) resulting from applying beam search with a depth limit of 5, beam size of 3, 2 training epochs per student network and lookahead. The 4 output convolutional layers remain unchanged and are left out of the graph for simplicity

TABLE 5.1: Modifications to GG-CNN that result in the best performing model. Elements in bold font represent the new addition at that depth. Layers 1 ($9 \times 9 \times 32$) and 2 ($5 \times 5 \times 16$) remain unchanged and thus are omitted.

depth	operator(layer)	layer 3	layer 4	layer 5	layer 6
0	deeper(3)	$3 \times 3 \times 8$			
1	deeper(4)	$3 \times 3 \times 8$	$\mathbf{3 \times 3 \times 8}$		
2	wider(3)	$3 \times 3 \times 8$	$3 \times 3 \times 8$	$\mathbf{3 \times 3 \times 8}$	
3	deeper(3)	$3 \times 3 \times \mathbf{16}$	$3 \times 3 \times 8$	$3 \times 3 \times 8$	
4	N/A	$3 \times 3 \times 16$	$\mathbf{3 \times 3 \times 16}$	$3 \times 3 \times 8$	$3 \times 3 \times 8$

considered two initial models that we call narrow and shallow. The narrow model has the same 6 layers as GG-CNN, but instead of using filter banks of 32, 16 and 8, it has sets of 8, 4, 2. The shallow model keeps only the 4 inner convolutional and transposed convolutional layers with kernel sizes of 5×5 and 3×3, discarding the outer 9×9 kernel layers. It uses the same filter bank sizes as the narrow model: the outer layers have 4 filters and the inner ones 2 filters. A schematic for both models can be seen in Figure 5.4

Both models were pre-trained in the same version of the Cornell dataset as GG-CNN. Figure 5.2 shows their initial accuracy, as well as the results of the beam search. We used the same search parameters used for the best-performing run in the earlier experiments: 2 retraining epochs after layer expansion, lookahead and expansion of only the convolutional layers. It can be seen how both models achieve $> 60\%$ accuracy in simulation prior to any modification. The shallow model as a tendency to decrease accuracy with the search depth, whereas the narrow model has a slight tendency to become more accurate. The best model is achieved at depth 5 starting from the narrow architecture, and obtains a success rate of 71.5%. The resulting architecture is 4 times smaller than

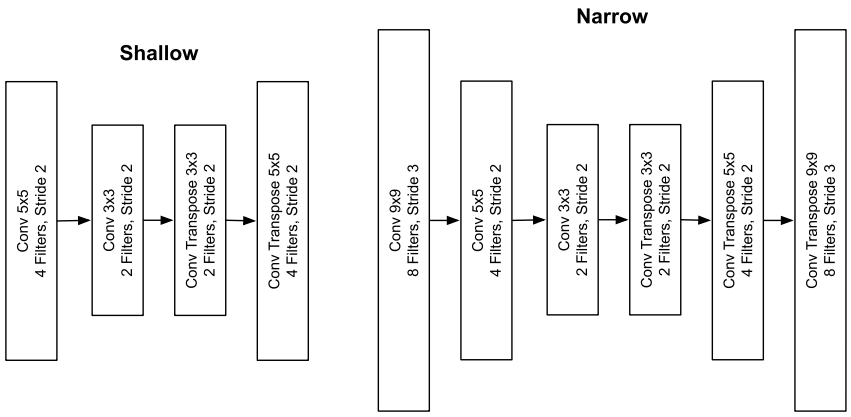

FIGURE 5.4: Stripped down versions of GG-CNN used as starting point for the optimization experiments. The 4 output convolutional layers are not depicted.

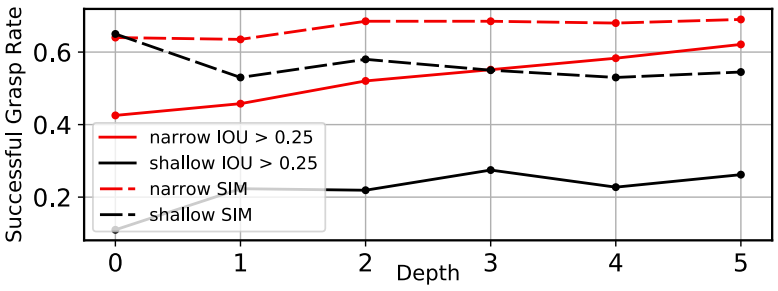

FIGURE 5.5: In order to optimize the original mode we start from two stripped down versions.

GG-CNN, having 16104 parameters, however it performs 7% worse. Given the fact that GG-CNN is already quite a lightweight architecture, the drop in performance can't be justified by the reduced size.

5.3 Grasping in 6DoF

The main purpose of the orthographic pipeline is to enable a robotic agent with a frame-fixed camera to manipulate objects at any relative position. This hypothetical robot can navigate around the workspace so as to frame the target within the field of view of the camera, but it has no control over the angle from the horizontal plane with which the object is observed. Thus the orthographic pipeline must work well under any elevation.

To replicate these variations in object perception, we evaluated the orthographic pipeline in 3 different scenarios. Each scenario alters the camera orientation w.r.t to the ground. The elevation angles considered where 30°, 60° and 90° (see Figure 5.6). The camera

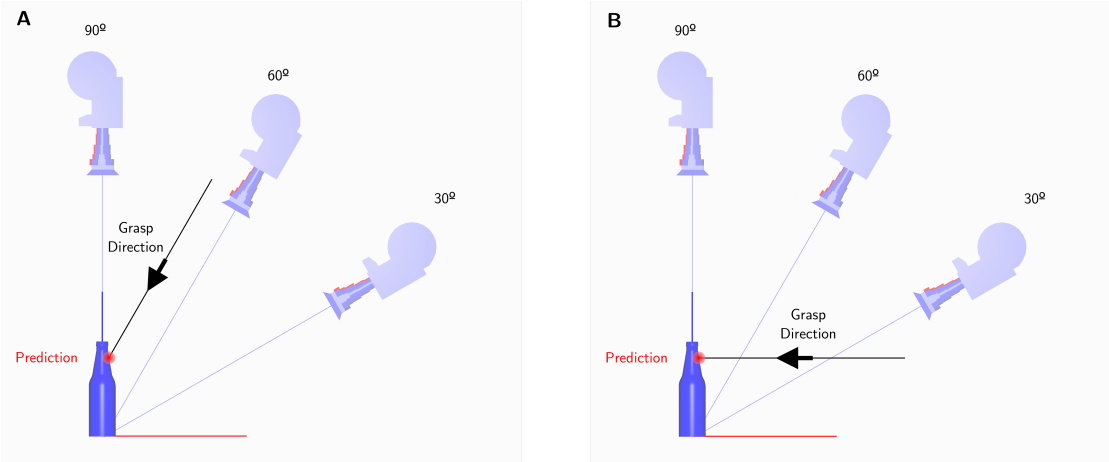

FIGURE 5.6: Experimental setup for the oblique camera settings. The pipelines are tested at 30°, 60° and 90° from the horizontal plane. (Left) straight in approach. The object is grasped along a parallel to the optical axis (ligth blue) of the camera. (Right) orthographic pipeline. The object is grasped along a parallel to the x, y or z axis w.r.t the object frame of reference.

was aimed at the origin, where the target objects were spawned, such that the objects would get projected towards the optical center of the image.

5.3.1 Straight-in Approach

We lack a baseline to compare the orthographic pipeline with. In the top-down experiments found in the literature, the grasp is typically executed following a nadiral path. This type of approach can be generalized to any camera elevation by traversing the gripper along a parallel direction to the optical axis of the camera. This has some potential problems with ground clearance for oblique approaches. Therefore, this can be considered a good minimum to improve upon. If the orthographic pipeline performs similarly or worse than a straight-in approach, then the method can be considered flawed.

In order to establish the mentioned baseline we first tested the models on a straight-in approach. With this method, pixel prediction is used to cast a ray from the camera. This ray defines the approach direction and the position of the final pose. The angle prediction in camera coordinates is directly used as the angle about the approach axis to complete the grasp pose definition.

We ran the straight-in approach over the full set of scenes and camera positions for both the GG-CNN and GG-CNN$^+$ models. The results for the 90° setup are the same as observed earlier in the simulation baseline and beam search results, since it is effectively the same setup. Performance drops significantly for the 60° setup, by a factor of 42% in the case of the GG-CNN model and 37.5% in the case of the GG-CNN$^+$ model.

TABLE 5.2: Straight-in Approach Evaluation Results

angle	model	accuracy
90	GG-CNN	78.5%
	GG-CNN$^+$	**83.5**%
60	GG-CNN	36.5%
	GG-CNN$^+$	**46**%
30	GG-CNN	14.7%
	GG-CNN$^+$	**24.3**%

Predictably, performance is even worse for the 30º setup, were in both cases it falls below 25%. In all the experiments GG-CNN$^+$ outperforms GG-CNN. For the oblique settings, GG-CNN$^+$ achieves $\approx$ 10% better success rates than GG-CNN.

5.3.2 Evaluating the View Selection Metric

Prior to evaluating the orthographic pipeline, the best view selection method must be identified. In the methods section two different view selection metrics were proposed: observable mass and entropy. The former is a count on the number of visible pixels in a synthetic image, while the latter also takes into consideration the number of points assigned to each pixel. To evaluate which of the two methods works best for selecting the optimal approach direction we conduct a simple experiment.

For this experiment, we want to decouple the performance of the grasp prediction, and the simulator's grasp accuracy from the performance of the view selection metric. To achieve this, we find the best possible accuracy if the view selection metric was not a factor. We establish this baseline by grasping each scene by all three approaches. This means that for each object and each of the 5 random orientations, we grasp three times, approaching from the front, the side and the top views. For each scene, a grasp is considered successful if at least one of the approaches succeeded. The result of this experiment represents the highest possible success rate that an ideal view selection metric could achieve for a given prediction model in our pipeline and simulator.

Next, we run the experiment again, but instead of grasping by all three approaches we pick only the best one according to the selection metric. All these experiments were conducted with a camera angle of 45º, and our extended GG-CNN architecture was used for the grasp prediction. The results can be observed in Table 5.3. The baseline is the ideal scenario where the best approach is always selected. It achieves a 82.5% success rate. This is the maximum performance our model and manipulator can achieve for the given camera orientation and object dataset. The best metric is entropy, which

Table 5.3: View Selection Evaluation Results

Metric	Accuracy
baseline	82.5%
entropy	68%
observable mass	66.5%

Table 5.4: Orthographic Pipeline Evaluation Results

Angle	Model	Acc.	% Top	Acc. Top	% Front	Acc. Front	% Side	Acc. Side
90°	orig.	65.5%	100%	65.5%	0	—	0	—
	ours	**72.5%**		72.5%	0	—	0	—
60°	orig.	59.5%	91.5%	63.4%	6%	25%	2.5%	0
	ours	**70%**		74.3%		25%		20%
30°	orig.	26%	47%	51%	49.5%	4%	3.5%	0
	ours	**34.5%**		60.6%		12.1%		0

achieves an accuracy of 68%. This is a penalisation on accuracy of 14.5%. Observable mass is 1.5% worse than entropy.

5.3.3 Orthographic Pipeline

Finally, we test the orthographic pipeline on the same scenes and camera configurations. Table 5.4 shows the outcome of each experiment as well as split accuracies per approach direction. The best results were obtained for the 90° configuration. The totality of the scenes for this configuration was grasped from the top, which suggests that the entropy metric works optimally for a top down setup.

For the 60° setup the most selected view is the top one, followed by the front and the side. For the 30% configuration the front view is the most selected followed by the top one. In all cases, the highest success rate is achieved when grasping from the top, and the poorest performance results from grasping from the side. Our GG-CNN$^+$ model outperforms vanilla in all scenarios.

When comparing the straight-in approach with the orthographic pipeline, the former performs 11% better for the top approaches while the latter is 24% and 10.2% more successful for the 60° and 30° configurations respectively. The reason for this is discussed in detail in the next section.

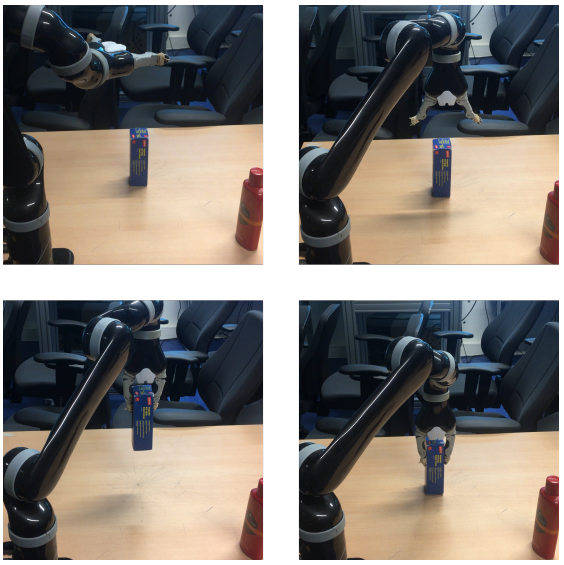

FIGURE 5.7: Stages of a grasping experiment with a real robot. (Top left) Initial configuration. (Top Right) Gripper in pre-grasp pose. (Bottom Right) Grasp pose with fingers closed. (Bottom Left) Post grasp pose.

5.3.4 Real Robot Trials

We also conducted a simple experiment on an actual robot with real sensor data. The arm used was a Kinova MICO 6DoF with a KG-2 2-fingered gripper. To control it we wrote a ROS package and made use of the MoveIt library. The experiment consisted of a simple pick and drop task, and we used a cardboard box from a whiteboard eraser as the target object. The point cloud was captured with an Asus Xtion RGB-D camera. An example of a point cloud generated from such a sensor can be seen in Figure 4.3. We carried out this experiment not as a performance evaluation on a physical environment, but as a proof of concept for our orthographic pipeline. The camera was set at approximately 45° of elevation and pointed directly towards the object. We used the same pipeline described in 4.3, with the exception of forcing the grasp approach to be the top one, for security concerns regarding table-arm collisions. In our single trial, we succeeded on grasping the mentioned object using predictions from GG-CNN. Figure 5.7 shows the different stages of the grasp operation for our trial.

Chapter 6

Discussion and Future Work

6.1 Evaluation Methods

In this section we will discuss some aspects regarding our evaluation methods. This will be paramount to put into perspective all our results and to frame the way they can be interpreted.

6.1.1 Particularities of our Simulation Environment

We chose to evaluate our model and pipeline on simulation rather than IOU or other heuristic-based evaluation techniques for two reasons: heuristic approaches do not necessarily reflect actual performance; and in the case of the orthographic pipeline no ground truths would have been available. Nonetheless, as much as simulation experiments attempt to replicate real-life conditions, their results can never be compared to physical experiences. There are a number of factors applicable to any simulation method that differ from real-life behaviour. In the following we will discuss a few details specific to our implementation.

First of all, while fine-tuning the parameters of the simulation, we observed how crucial the value of the time step was to the accuracy of the system. In order to have collision physics robustly simulated, the time step must be bellow a threshold (in our experience, we found this threshold to be $\leq 10^{-3}$). Longer time steps can lead to deep penetrations, singularities and integration errors [36], that result in the objects shooting off from the gripper plates after a grasp. We did our best to find a time step that balanced physics accuracy and execution time, however the selection of this parameter biases the result of our simulation.

Secondly, it was also observed that the objects had a tendency to fall from the gripper during traversal from the pick-up to the drop-off points. This happened even for apparently good grasps. To prevent it, we set an arbitrarily high value to the friction coefficients and the maximum force that the fingers could exert. As a consequence, small features or protuberances in the objects would suffice for a grasp to be firm. Some of these grasps would never be possible with a real life manipulator.

Lastly, we described in the methods section how the criteria for a successful, robust grasp was that the target object could be picked up and dropped off within a destination container. In some of our experiences, an object would be successfully moved to the drop-off location but because of the orientation at which it collided with the ground it would bounce outside of the container. In retrospective, a simple pick and shake experiment, or translation to a target position without dropping onto a bin would have sufficed as a success criteria. However, this behaviour was noticed along the experimentation process, and for the shake of consistency and validity of earlier results was not addressed.

6.1.2 Simulation Results

One phenomenon stands out from all our simulation experiments: the success rate achieved by networks with very little training (just 1 epoch) is considerably high. This happens to the vanilla architecture in the simulation baseline experiment (Figure 5.1) as well as to the narrow and shallow initial architectures in the beam search experiments (Figure 5.5). In all cases, the networks achieve an initial accuracy slightly above 60%. In contrast, IOU results yield initial values in the low 10s. This might raise doubts about the validity of the simulation to assess performance, however there is one plausible explanation for this behaviour. The IOU metric assesses how close the network prediction is to the ground truth. This is what we mean when we say that IOU does not represent actual performance. A grasp prediction might not align sufficiently well with any of the ground truths and still be valid. Moreover, the case can be given where the gripper is completely misaligned with the object, but because the fingers gradually close the object gets re-oriented to a pose suitable for grasping.

6.1.3 Standardization

So far we have mentioned a number of factors that can have a substantial impact on the results of a simulation experiment. To add a few more, we can consider the election of the gripper geometry, the control strategy, and parameters of the manipulator (like the torque limit of the motors). All these aspects make reproducibility and comparability of results hard.

It is the opinion of the author that there is a lack of much needed standardized benchmarking methods for the field of grasp detection. This stands out when comparing it to the fields of object detection or image classification. Yearly challenges like those of ILSVRC [42] offer a common goal to research as well as a fair and accurate way to compare results among the proposed solutions. There have been some recent efforts towards this standardization for manipulation research. In the aforementioned work by the Jacquard team [22], their simulator was made available for other research teams to submit their predictions over the dataset via their website. A priori, this solution seems optimal, since it offers the exact same environment, with no possibility for alterations, across all submissions. Thus it has the potential to be the most partial and accurate way to compare results. However, there are some caveats to their solution.

To begin with, the simulator is quite limited in the scenarios it can handle. Only pick-and-move tasks are supported, and the point of view of the camera is always top-down. This means that it is only good for comparing grasp prediction quality. If one's experiments involve other points of view (like it is our case for the orthographic pipeline) or a specific control strategy (e.g. the closed-loop controller for [5]) or any other type of task (like grasping in cluttered environments) then the simulator is of no use. Moreover, the fact that the simulator cannot be installed and run locally makes its use unpractical for iterative improvements. It also removes the possibility of running simulations within the training loop (e.g. as validation tests). Since the publishing of their work, a number of other publications have succesfully made use of the Jacquard dataset. However, to the best of our knowledge, their simulator is yet to be used as a performance metric.

Designing a single simulator that can accommodate all possible experimental setups and still be a "single source of truth" for performance results might be unfeasible or hard to do. An alternative could be a way to compare performance between the simulators themselves. In 2015 an object dataset was proposed to benchmark robotic grasping experiments [43]. While this is useful to remove the object-related performance deviations, the other factors discussed so far still remain. More recently, a method to benchmark simulation engines against real-robot experiment has been proposed [44]. This new benchmark provides a set of scene and task descriptions, a dataset of motion-captured movements of a real Kinova Mico robot solving said tasks and some metrics to compare simulation performance and motion captured data. This framework is meant to be used to tune the hyperparameters of the simulator such that the velocities, accelerations, torque, contact forces and other physical properties of the arm and the final pose of the manipulated object are as close to the ground truth as possible. Ensuring that ones's simulator meets this standard would help reduce the irreproducibility of an experiment.

6.2 GG-CNN

One of the reasons we chose GG-CNN to build upon was its object-class agnostic approach to grasp prediction. The rationale behind that was that if the network only learns the geometric properties that make a shape suitable for grasping, it should generalize well to unseen objects. The consequence, however, is a complete lack of semantic grasping capability. We observed in our experiments how some of the grasps, while successful, were "peculiar", or not how a human would grasp the object. For example, consider a wine glass sitting on its foot. For a top-down approach, the network would tend to grab it by the wall of the bowl (that is with one finger inside the bowl, and the other outside), instead of with both fingers outside the bowl (which would avoid dipping the fingers in the potential content). Another example is a hammer being grabbed by its head instead of its grip. While these grasps would succeed in a table-clearing scenario, they would not be acceptable for more service-oriented tasks like food serving or tool handling. One way to inform the model prediction with the object class, and improve its performance on these scenarios would be through multi-task learning [45].

6.2.1 Multi-task Learning

Multi-task learning [46] consists in training a model to solve more than one task simultaneously. This can be done when there is some form of shared knowledge between the two and it typically results not only in a model that can solve two tasks, but that also obtains better generalization performance. It works because by optimizing one of the loss functions, we are updating the common knowledge representation, and since their knowledge is related it also improves the performance for the other task. Additionally, it can be beneficial in the cases where one of the tasks has fewer data available to train on [47], as is the case for grasp prediction when compared to a problem like object detection or semantic segmentation.

It could be considered that the original GG-CNN model already uses some form of multitask learning. It minimises 4 loss functions, albeit all aim towards the same goal of grasp prediction and their ground truth is derived from the same dataset. The existing architecture could be extended to be class-aware by adding an additional convolutional layer with as many filters as classes to be identified like it was done in [48] and training against a dataset like COCO [49]. If this was to be done, the network would be conducting simultaneous classification and regression. An example of earlier success with this type of multitasking on CNNs can be seen in [50], where they use it to jointly diagnose and predict clinical scores for Alzheimer's Disease.

Typically, models that perform well on object-classification tend to be considerably larger than our GG-CNN version [30, 51–54]. It is fair to assume that adding more tasks to the network, and specifically a higher number of classes to classify, will lead it to reach its capacity. The beam search framework would be useful in this case, to methodically enlarge our current model. Alternatively, in order to keep the network small, we could consider fewer classes. It has been argued before that "objects that look similar can be grasped similarly" [18, 45, 55]. By using a tighter object taxonomy, where several classes that are grasped similarly (e.g. cans and bottles, forks and knives...) are clustered into a single one, we could still afford to have an object type informing the grasp. We propose the exploration of this alternative as a future work.

6.2.2 Affordance Segmentation

An alternative to informing the grasp with object class knowledge would be doing so with other affordances in the image, besides grasping. Knowledge about the ways an object can be used might be more informative on how to grasp it than the object category on its own. If a part of an object has the affordance of pounding, while another has it for grasping, then it is less likely that the network infers that the pounding part is suitable for a grasp. Furthermore, in a lot of cases knowledge of affordances has implicit knowledge about the object, but it can have a more compact representation (many tools can be used for cutting). Once again, we could use multitask-learning to incorporate this affordance segmentation skill into the grasp generation task.

6.3 Beam Search

6.3.1 Operator Selection

There are some peculiarities in the choices in terms of layers and operators that beam search made in our experiments that are worth discussing.

Wider vs Deeper

We observed in section 5.2.1 how in order to arrive at GG-CNN$^+$ 3 out of 4 modifications consisted on making the model deeper rather than wider. Furthermore, all of those modifications were applied to the deeper layers, leaving the first two untouched. While our experiments don't have statistical significance to argue that deeper is always better for GG-CNN like architectures, our results line up with current ML trends. In principle

an arbitrarily wide network can approximate complex functions [56, 57], however there is a tendency in state-of-the-art ML models to become deeper and not wider [58]. In [59] researchers from Google conducted a study on how deep vs wide learning affects the knowledge representation and ultimately the output of the network. Surprisingly they found that on average wider vs deeper networks perform similarly on image classification tasks, although the accuracy distribution depends on the class (some classes having higher accuracy on wide networks and vice-versa).

Transposed Convolution Knowledge Transfer

It stood out during our experiments that no transposed convolutional layers were selected for knowledge transfer. There are two possible reasons for this: 1) that indeed replicating transposed convolutions results in lesser improvement than normal convolutional layers 2) that the transfer operator for transposed layers has a negative effect in performance. We attempted ruling out option 2 by subtracting the output of a teacher network from a student resulting from a transposed knowledge transfer operation. The result was not a completely blank image which suggests a fault in our implementation of the transposed convolution transfer operator. This fault was not shared with the normal convolution transfer operator, where both student and teacher networks yield the exact same output. Nonetheless, the differences in both outputs were small enough (we hypothesize it was caused by numerical inaccuracies) that a few training epochs should quickly overcome it. We chose not to further explore the reason for this negative impact, since having an option to expand both convolutional and transposed convolutional layers increases the branching factor and makes the algorithm run slower.

6.3.2 Limitations

One of the limitations of Beam Search is that it *lacks imagination* in the sense that it can only do a limited number of modifications to the initial architecture. This means, for example, that if we want to explore the potential of different filter sizes the starting node must be at least as deep as the number of sizes we want to investigate. The source of this limitation lies in the fact that we can do as many operations per layer as knowledge transfer operators we have available (if we want to keep an efficient implementation that is). Arbitrary modifications could still be conducted without the benefit of shorter re-training times. In the original beam search publication [28] the authors considered max-pooling operations, but in their experience the algorithm would never select it due to its negative impact on initial performance.

6.3.3 Potential for lifelong learning

Lifelong learning is the ability of an agent to incrementally update its knowledge base in order to adapt to new scenarios, acquire new skills or improve its performance on existing ones [60]. This is a pervasive feature for most animals. For example, humans are born with a very limited set of skills, evolve them very rapidly during the early stages of development, and continue to perfect them as they grow older. A key aspect of this capability is being able to acquire new knowledge without "catastrophically forgetting" existing one. There have been some efforts in the recent past to achieve open-ended learning. In [61], a cognitive architecture is proposed capable of incrementally increasing the knowledge base for known classes, as well as building new models from unknown objects from scratch, in an on-line manner. The method in [55] achieves incremental learning through kinesthetic teaching. Both methods use hand-crafted features to detect object and match object categories. In [62], an extensive review is given of current efforts to achieve lifelong learning with neural network models. One of the proposed frameworks is that of dynamic architectures. Dynamic architectures achieve incremental learning by means of introducing units (trainable parameters) to the model, and retraining it with the novel and old data. Beam search can fit within this framework. Once an architecture reaches its capacity, and retraining on novel data makes it perform worse (i.e. the network starts to catastrophically forget past knowledge), beam search can be applied on the novel dataset to determine the best student network from the present model. In the case of GG-CNN, this can help increase the number of graspable geometries over time.

6.4 Orthographic Pipeline

For models where the grasp pose is given in 3DoF there is only one possible approach direction to the grasp: parallel to the normal of the image plane. In section 5.3.1 we demonstrated how this approach can be risky when the camera is oriented at off-nadir angles. The reason for the elevated failure rate of this method is mainly the table-gripper collisions that happen when approaching the objects at such angles. With our orthographic pipeline we were able to get around this issue achieving higher success rates on oblique approaches. Nonetheless, our accuracy cannot be compared with the state-of-the-art of 6DoF prediction [8, 11, 16]. We see the value of our approach, however, in the fact that it is model-agnostic and compatible with any architecture that works on depth images. Furthermore, while we did not gather statistically significant results with a real robot, we demonstrated that, in principle, our pipeline works with a physical arm and real sensor data.

6.4.1 Caveats

It is evident from the results in sections 5.3.1 and 5.3.3 that the orthographic pipeline supposes a penalization in effectiveness for top-down approaches w.r.t the straight-in strategy. The reason for this is obvious: our synthetic image generation process introduces noise and loss of resolution due to the sparsity of the image and our dilation post-processing. Nonetheless, the overhead of our approach is not necessary for nadir grasps, since this is the setup for which most models are designed.

In table 5.4 we showed the results for the orthographic grasp experiments segregated per camera orientation and canonical view. In our experiments we let the entropy metric be the only criteria for view selection, which under some circumstances might favor the side view. However, given our definition of the front and side vectors, the side view will always have the highest uncertainty due to occlusions. Thus, for practical applications it does not make sense to ever consider it as a viable approach. Furthermore, the highest success rate is achieved when grasping from the top, independently of the camera orientation. Our pipeline has a collision avoidance feature built-in into the grasp selection, but clearly a collision-less grasp does not translate into a successful one. There are two possible reasons why non top-grasps have such a high failure rate. Firstly, due to our test objects geometry and our simulation scene initialization process, most objects lay flat against the floor. This makes grasping horizontally a daunting task. Even a human would prefer to grab the objects vertically for most scenes. Secondly, a non-colliding grasp might result in ground-object interactions when closing the gripper's fingers that exert unforeseen torque on the object-gripper system, leading to the object slipping. As such, we propose our orthographic pipeline as a solution for nadir grasps independently of the camera orientation, since this is the most robust approach.

Chapter 7

Conclusion

Grasp detection is a fundamental skill in order to enable robotic agents to autonomously perform manipulation tasks in the wild. There are numerous Machine Learning methods available to solve such a problem, and some of them achieve excellent performance on the benchmarks ($> 97\%$), however these models tend to be complex. There is a motivation to find smaller architectures that still behave sufficiently well, such that they can be deployed on low-powered embedded devices. In this work, we took it upon ourselves to improve GG-CNN, one of these small networks that achieves results $> 78\%$. Through a methodical architecture exploration strategy, we were able to find an extended version that we coin GG-CNN$^+$ that performs 5% better on simulation experiments and is only 8% larger, making it still a good candidate for embedded applications.

Most of the grasp detection research focuses on top-down views, which limits the way objects can be approach to just a vertical fashion. We propose an orthographic pipeline that can be applied to any existing top-down solution, that enables grasping objects based on images taken from off-nadir oriented cameras. While our solution is not robust enough to grasp objects from the side or the front, it can still be used to grab them from the top, independently of the orientation of the camera.

Bibliography

[1] Mordechai Ben-Ari and Francesco Mondada. *Robots and Their Applications*, pages 1–20. Springer International Publishing, Cham, 2018. ISBN 978-3-319-62533-1. doi: 10.1007/978-3-319-62533-1_1. URL https://doi.org/10.1007/978-3-319-62533-1_1.

[2] Bruce A Aikenhead, Robert G Daniell, and Frederick M Davis. Canadarm and the space shuttle. *Journal of Vacuum Science & Technology A: Vacuum, Surfaces, and Films*, 1(2):126–132, 1983.

[3] Charles C Kemp, Aaron Edsinger, and Eduardo Torres-Jara. Challenges for robot manipulation in human environments [grand challenges of robotics]. *IEEE Robotics & Automation Magazine*, 14(1):20–29, 2007.

[4] James J Gibson. The theory of affordances. *Hilldale, USA*, 1(2), 1977.

[5] Douglas Morrison, Peter I. Corke, and Jürgen Leitner. Closing the loop for robotic grasping: A real-time, generative grasp synthesis approach. *CoRR*, abs/1804.05172, 2018.

[6] Xinwen Zhou, Xuguang Lan, Hanbo Zhang, Zhiqiang Tian, Yang Zhang, and Nanning Zheng. Fully convolutional grasp detection network with oriented anchor box. *arXiv preprint arXiv:1803.02209*, 2018.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[8] Yuzhe Qin, Rui Chen, Hao Zhu, Meng Song, Jing Xu, and Hao Su. S4g: Amodal single-view single-shot se (3) grasp detection in cluttered scenes. In *Conference on robot learning*, pages 53–65. PMLR, 2020.

[9] Yikun Li, Lambert Schomaker, and S Hamidreza Kasaei. Learning to grasp 3d objects using deep residual u-nets. In *2020 29th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, pages 781–787. IEEE, 2020.

[10] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

[11] Arsalan Mousavian, Clemens Eppner, and Dieter Fox. 6-dof graspnet: Variational grasp generation for object manipulation. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 2901–2910, 2019.

[12] Charles R Qi, Hao Su, Kaichun Mo, and Leonidas J Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 652–660, 2017.

[13] 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. *arXiv preprint arXiv:1703.09312*, 2017.

[14] Sulabh Kumra, Shirin Joshi, and Ferat Sahin. Antipodal robotic grasping using generative residual convolutional neural network. *arXiv preprint arXiv:1909.04810*, 2019.

[15] Eduardo G Ribeiro and Valdir Grassi. Fast convolutional neural network for real-time robotic grasp detection. In *2019 19th International Conference on Advanced Robotics (ICAR)*, pages 49–54. IEEE, 2019.

[16] Hao-Shu Fang, Chenxi Wang, Minghao Gou, and Cewu Lu. Graspnet-1billion: a large-scale benchmark for general object grasping. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11444–11453, 2020.

[17] Umar Asif, Jianbin Tang, and Stefan Harrer. Graspnet: An efficient convolutional neural network for real-time grasp detection for low-powered devices. In *IJCAI*, volume 7, pages 4875–4882, 2018.

[18] Corey Goldfeder, Matei Ciocarlie, Hao Dang, and Peter K Allen. The columbia grasp database. In *2009 IEEE international conference on robotics and automation*, pages 1710–1716. IEEE, 2009.

[19] Hanbo Zhang, Xuguang Lan, Xinwen Zhou, Zhiqiang Tian, Yang Zhang, and Nanning Zheng. Visual manipulation relationship network. *arXiv preprint arXiv:1802.08857*, 2018.

[20] Lerrel Pinto and Abhinav Gupta. Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 3406–3413. IEEE, 2016.

[21] Sergey Levine, Peter Pastor, Alex Krizhevsky, Julian Ibarz, and Deirdre Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, 37(4-5):421–436, 2018.

[22] Amaury Depierre, Emmanuel Dellandréa, and Liming Chen. Jacquard: A large scale dataset for robotic grasp detection. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3511–3516. IEEE, 2018.

[23] Manolis Savva, Angel X. Chang, and Pat Hanrahan. Semantically-Enriched 3D Models for Common-sense Knowledge. *CVPR 2015 Workshop on Functionality, Physics, Intentionality and Causality*, 2015.

[24] Clemens Eppner, Arsalan Mousavian, and Dieter Fox. Acronym: A large-scale grasp dataset based on simulation, 2020.

[25] Xin He, Kaiyong Zhao, and Xiaowen Chu. Automl: A survey of the state-of-the-art. *Knowledge-Based Systems*, 212:106622, 2019.

[26] Thomas Elsken, Jan Hendrik Metzen, Frank Hutter, et al. Neural architecture search: A survey. *J. Mach. Learn. Res.*, 20(55):1–21, 2019.

[27] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

[28] Xu Xu and Sinisa Todorovic. Beam search for learning a deep convolutional neural network of 3d shapes. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 3506–3511. IEEE, 2016.

[29] Frank Rosenblatt. *The perceptron, a perceiving and recognizing automaton Project Para*. Cornell Aeronautical Laboratory, 1957.

[30] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[31] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander A Alemi. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[32] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[33] S Hamidreza Kasaei, Ana Maria Tomé, Luís Seabra Lopes, and Miguel Oliveira. Good: A global orthographic object descriptor for 3d object recognition and manipulation. *Pattern Recognition Letters*, 83:312–320, 2016.

[34] Juil Sock, S Hamidreza Kasaei, Luis Seabra Lopes, and Tae-Kyun Kim. Multi-view 6d object pose estimation and camera motion planning using rgbd images. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 2228–2235, 2017.

[35] Claude E Shannon. A mathematical theory of communication. *Bell system technical journal*, 27(3):379–423, 1948.

[36] E Coumans, Y Bai, and J Hsu. Pybullet physics engine, 2018.

[37] Erwin Coumans et al. Bullet physics library. *Open source: bulletphysics. org*, 15 (49):5, 2013.

[38] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *CoRR*, abs/1804.10332, 2018. URL http://arxiv.org/abs/1804.10332.

[39] Fereshteh Sadeghi, Alexander Toshev, Eric Jang, and Sergey Levine. Sim2real view invariant visual servoing by recurrent control. *CoRR*, abs/1712.07642, 2017. URL http://arxiv.org/abs/1712.07642.

[40] Oleg Klimov and J Schulman. Roboschool, 2017.

[41] Angel X. Chang, Thomas A. Funkhouser, Leonidas J. Guibas, Pat Hanrahan, Qi-Xing 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. *CoRR*, abs/1512.03012, 2015. URL http://arxiv.org/abs/1512.03012.

[42] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

[43] Berk Calli, Arjun Singh, Aaron Walsman, Siddhartha Srinivasa, Pieter Abbeel, and Aaron M Dollar. The ycb object and model set: Towards common benchmarks for manipulation research. In *2015 international conference on advanced robotics (ICAR)*, pages 510–517. IEEE, 2015.

[44] Jack Collins, Jessie McVicar, David Wedlock, Ross Brown, David Howard, and Jürgen Leitner. Benchmarking simulated robotic manipulation through a real world dataset. *IEEE Robotics and Automation Letters*, 5(1):250–257, 2019.

[45] Nima Shafii, S Hamidreza Kasaei, and Luís Seabra Lopes. Learning to grasp familiar objects using object view recognition and template matching. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 2895–2900. IEEE, 2016.

[46] Rich Caruana. Multitask learning. *Machine learning*, 28(1):41–75, 1997.

[47] Yu Zhang and Qiang Yang. A survey on multi-task learning. *arXiv preprint arXiv:1707.08114*, 2017.

[48] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. *CoRR*, abs/1411.4038, 2014. URL http://arxiv.org/abs/1411.4038.

[49] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014. URL http://arxiv.org/abs/1405.0312.

[50] Mingxia Liu, Jun Zhang, Ehsan Adeli, and Dinggang Shen. Joint classification and regression via deep multi-task multi-channel learning for alzheimer's disease diagnosis. *IEEE Transactions on Biomedical Engineering*, 66(5):1195–1206, 2018.

[51] Bo Zhao, Jiashi Feng, Xiao Wu, and Shuicheng Yan. A survey on deep learning-based fine-grained object classification and semantic segmentation. *International Journal of Automation and Computing*, 14(2):119–135, 2017.

[52] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[53] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[54] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[55] S Hamidreza Kasaei, Nima Shafii, Luís Seabra Lopes, and Ana Maria Tomé. Interactive open-ended object, affordance and grasp learning for robotic manipulation. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 3747–3753. IEEE, 2019.

[56] Gaurav Pandey and Ambedkar Dukkipati. To go deep or wide in learning? In *Artificial Intelligence and Statistics*, pages 724–732. PMLR, 2014.

[57] Sanjeev Arora, Simon S. Du, Wei Hu, Zhiyuan Li, Ruslan Salakhutdinov, and Ruosong Wang. On exact computation with an infinitely wide neural net. *CoRR*, abs/1904.11955, 2019. URL http://arxiv.org/abs/1904.11955.

[58] Md. Zahangir Alom, Tarek M. Taha, Christopher Yakopcic, Stefan Westberg, Paheding Sidike, Mst Shamima Nasrin, Brian C. Van Esesn, Abdul A. S. Awwal, and Vijayan K. Asari. The history began from alexnet: A comprehensive survey on deep learning approaches. *CoRR*, abs/1803.01164, 2018. URL http://arxiv.org/abs/1803.01164.

[59] Thao Nguyen, Maithra Raghu, and Simon Kornblith. Do wide and deep networks learn the same things? uncovering how neural network representations vary with width and depth, 2020.

[60] Sebastian Thrun and Tom M Mitchell. Lifelong robot learning. *Robotics and autonomous systems*, 15(1-2):25–46, 1995.

[61] S Kasaei, Juil Sock, Luis Seabra Lopes, Ana Maria Tomé, and Tae-Kyun Kim. Perceiving, learning, and recognizing 3d objects: An approach to cognitive service robots. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32, 2018.

[62] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113:54–71, 2019.