university of groningen

faculty of science and engineering

Are capsule networks sufficient for grasping familiar objects?: An approach and experiments with a dual-arm robot

Storm Tomas Martzen van der Velde

university of groningen / faculty of science and engineering

University of Groningen

Are capsule networks sufficient for grasping familiar objects?: An approach and experiments with a dual-arm robot

MASTER'S THESIS

To fulfill the requirements for the degree of
Master of Science in Artificial Intelligence
at University of Groningen under the supervision of

Dr. Hamidreza Kasaei
and
Prof. Dr. Raffaella Carloni

Storm Tomas Martzen van der Velde (s2738260)

August 30, 2022

Abstract

As robots become more accessible outside of industrial settings, the need for reliable object grasping and manipulation grows significantly. In such dynamic environments it is expected that the robot is capable of reliably grasping and manipulating novel objects in different situations. In this work we present GraspCaps: a novel architecture based on Capsule Networks for generating per-point grasp configurations for familiar objects. In our work, the activation vector of each capsule in the deepest capsule layer corresponds to one specific class of object. This way, the network is able to extract a rich feature vector of the objects present in the point cloud input, which is then used for generating per-point grasp vectors. This approach should allow the network to learn specific grasping strategies for each of the different object categories. Along with GraspCaps we present a method for generating a large object grasping dataset using simulated annealing. The obtained dataset is then used to train the GraspCaps network. We performed an extensive set of experiments to assess the performance of the proposed approach regarding familiar object recognition accuracy and grasp success rate on challenging real and simulated scenarios.

Acknowledgments

First, I would like to thank my partner Anna for keeping me focused on the thesis throughout the project and for reminding me to take breaks every now and then. Your help in clarifying which sections of the project needed more explanation and how to more meaningfully explain certain concepts of the thesis was immeasurable and your proofreading has made the thesis a lot more cohesive than it would otherwise be.

Second, I would like to thank my supervisors Hamidreza and Raffaella for the many insightful discussions we had throughout this project and for giving me much needed direction at times when I got stuck.

Third, I would like to thank Marc, Rik, and all my fellow teaching assistants that have worked at the Robotics lab over the past few years. The many discussions we had on all manner of topics regarding robotics, neural networks, and computing have led to insights and ideas that shaped core parts of this thesis.

Finally, I would like to thank the Center for Information Technology of the University of Groningen for their support and for providing access to the Peregrine high performance computing cluster, without which this research wouldn't have been possible.

Contents

List of Figures

List of Tables

1 Introduction

Domestic service robots can be used to provide elderly and disabled people with a higher degree of independence by assisting them with household tasks [3]. Currently, commercial robots such as iRobot's Roomba are becoming more and more commonplace in households, and are able to map and vacuum an entire single-floor house in an efficient fashion. Similar robots are commercially available for purchase to perform tasks such as cutting grass and sweeping floors.

The success of these robots in comparison to more complex domestic robots is likely due to the simplicity of the behaviour that they execute and their size. They are relatively small and light, with close to no possibility of injury or damage in the event that the robot bumps into anyone. Because of this, the behaviour structures of these robots can afford to be simple and imprecise. The earliest version of the Roomba illustrates this point: it vacuumed a house by moving in straight lines and only changed direction when it bumped into an object, effectively employing a random walk approach to cleaning every part of the house. If it were to bump into a human or an animal, the potential of anyone getting injured would be minimal due to the robot's small size. Routinely leaving the robot to clean the house for an extended period of time would ensure that all parts of the house would be relatively clean, even when the robot occasionally doesn't cover the entire floor plan of the house due to the randomness of its behaviour.

Current progress in the field of robotics allowed newer versions of the Roomba to become much more sophisticated. Nowadays, they are able to construct maps of the area they're cleaning by use of SLAM mapping and efficiently keep track of which regions still need to be cleaned. LiDAR sensors ensure they no longer bump into objects, in contrast to their earlier counterparts.

In a related field, the rise in popularity of personal assistants such as the Amazon's Alexa or Apple's Siri that are able to parse complex sentences and fulfill digital tasks such as ordering products, scheduling appointments, and turning on the lights shows that many people are willing to adopt intelligent robotics to aid them in their everyday life.

Where the Roomba can bump into anything without significant chance of damage, larger and more sophisticated robots don't have that luxury. A service robot that is capable of performing more complicated tasks such as cleaning a table or serving a coffee needs to properly plan all its actions before performing a task. Otherwise, it could break objects on the table it is cleaning or spill the coffee it was instructed to serve. This will either create a mess that needs to be cleaned up, or - in a worst case scenario - burn a person by spilling a hot drink over them. A more sophisticated robot also needs to have a detailed map of the house and be able to plan how to move from place to place without assistance from a human. It cannot rely on a random walk like the early versions of the Roomba could.

The final hurdle autonomous assistants need to overcome seems to be squarely in the field of complex physical interactions with their environment.

1.1 Recognition

A robot perceives its surroundings by use of one or more sensors. For the task of object recognition and grasping, a robot usually uses either one or more regular cameras, a depth camera, or a LiDAR camera.

A depth camera produces a 3D image of the surroundings, in which each pixel in a 2D image has a corresponding depth reading. These cameras are also often called RGBD cameras, signifying the four channels of data they produce (red, green, blue, and depth). LiDAR cameras produce a set of points in 3D space, which can be used for detecting and identifying shapes in 3D space. Although they can

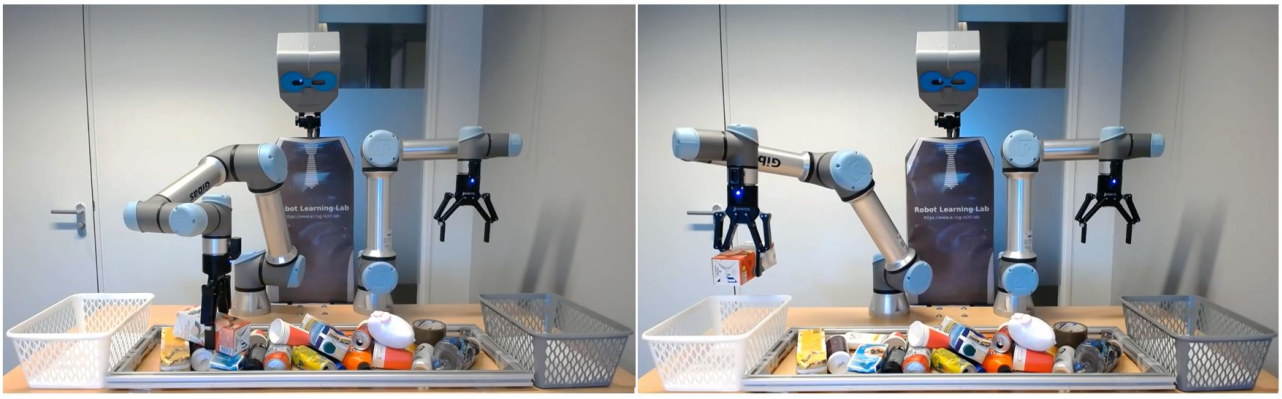

Figure 1.1: In this example, a service robot is instructed to perform a cleaning the table task. In each execution cycle, the robot needs to process the input point cloud, predict reliable grasp configurations, and finally grasp and place objects into the baskets to accomplish this task successfully.

be more precise than RGBD cameras regarding information in 3D space, this precision comes at the cost of not capturing any colour information.

Using 3D representations of the environment over more conventional 2D representations has the distinct advantage in robotics of allowing the robot to reason about an object's placement in 3D space. When using a 2D image, a neural network would be able to classify whether or not an object is present in the image and have a general concept on its location, but it would be non-trivial to correctly determine the exact placement of the object and optimal method of interacting with the object based on the information in the 2D image alone. To bridge this gap, it's possible to use several 2D cameras simultaneously to achieve an internal 3D representation of an object in a method called multi-view imaging.

1.2 Grasping

A robot can interact with its surroundings by use of one or more manipulators.

Industrial robots that work in factories often work using preprogrammed motions that do not change in between execution cycles. Humans are not able to work in close proximity to these robots, as the robots do not take their surroundings into account when performing their task and therefore may hit the human while performing their work.

Using preprogrammed motions would be unusable in a household setting; the robot needs to be aware of its surroundings at any time and be able to change its plans dynamically, as the robot will always work in a space shared with humans. Especially for domestic service robots working in care homes it is important that the robot is able to move items to and from the human and assist the human directly when performing physical tasks such as getting up from a chair. As moving heavy objects requires a considerable amount of strength, it is important that the robot is able to dynamically update its view of the world and adjust its actions accordingly to avoid accidental harm.

In this thesis, we will be focusing on dynamically manipulating the environment by use of multiple robotic arms.

1.3 Our approach and contributions

As noted earlier, the most significant hurdle autonomous assistants currently need to overcome is in complex physical interactions with their environment. A large body of recent efforts for object

grasping has focused on solving 4DoF (x, y, z, yaw) object-agnostic grasping, where the gripper is forced to approach objects from above. The major drawbacks of these approaches are that (1) they do not take into account the semantic function or label of the object, and (2) they inevitably limit the interaction possibilities with the object. For instance, they are not able to distinguish objects from each other (e.g., *plate* vs. *mug*), and are not able to grasp a horizontally placed plate. These limitations motivate the study of "*learning to grasp familiar objects*", where the robot is able to recognize the label of the object and its gripper is free to approach objects from any arbitrary direction it can reach.

Towards this end, we formulate object grasping as a supervised learning problem to grasp familiar objects. The primary assumption is that new objects which are geometrically similar to known objects can be grasped in similar ways using object-aware grasping [4, 5]. Object aware grasping allows the network to specifically generate grasps based on the features and geometry of the specific object it is attempting to grasp, as opposed to object agnostic grasping, which generates grasp vectors based on the input to the network without any deeper knowledge on the features of the object it is attempting to grasp. Since capsule networks are able to extract intrinsic parameters of the input data, we've designed a novel architecture called GraspCaps, that receives a point cloud of the object as an input and generates per-point grasp configurations and a single semantic category label as outputs. Our approach takes the activation of one capsule in the capsule network, and processes that activation to generate per-point grasp vectors and fitness values. The network architecture will be explained in more detail in Section 4.5.2. To the best of our knowledge, this is the first network architecture utilizing a capsule network for object-aware grasp synthesis. The contributions of this thesis can be summarized as follows:

- We propose a novel architecture called GraspCaps for object aware grasping that receives a point cloud of an object as input and produces a semantic category label and point-wise grasp configurations as outputs. We train the model in multiple stages and test it on both simulations and real-world data.
- We propose an algorithm for generating 6D grasp vectors from point clouds and create a synthetic grasp dataset consisting of 12870 samples with corresponding object labels and target grasp vectors.
- We perform an extensive set of experiments in both simulated and real-robot settings to validate the performance of the proposed approach.

1.4 Research question

To formalize the goals of this research project, we will try to answer the following:

- Is it possible to design a neural network architecture based on capsule networks that is able to accurately generate grasps for a multitude of familiar object classes?

with the subquestion:

- Is there significant merit in including capsule networks in a grasping architecture?

2 State of the art

This section will be describing and discussing several state of the art architectures in the field of object recognition and grasping. We will mainly be discussing methods that either use a raw point set for use as input, or use a point set by first transforming it into a different data structure.

2.1 Point based methods

PointNet [6] was one of the first architectures to effectively use point set data for training a neural network in an object recognition task. By design, the PointNet architecture is invariant to point order, which benefits point sets as extracting a natural order from these sets is non-trivial. This does however limit the performance of PointNet, as it is not able to recognize local structures in point sets. Furthermore, prior research has illustrated the importance of order being present in data to aid in the performance of neural networks [7]. Therefore it would be beneficial not to completely disregard order when designing a network architecture for working with point sets.

PointNet++ [8] improves upon PointNet by recognizing local structures in the data, while retaining the ability to handle order-invariant input which is vital when working with point sets. It manages this by compressing the architecture of PointNet into a single layer, and recursively applying PointNet layers to the input data to obtain hierarchical features.

Many following architectures employ some transformation on the input point set to include information on the locality of the points. Klokov and Lempitsky [9] employed kd-trees on point sets, where the input point set is continuously split into subsets of equal size, and the resulting tree structure is used as input for the network. It achieved results comparable with state of the art performance.

Zhao et al. [10] uses a module called an adaptive feature adjustment module to construct features based on the interaction between points. These features contain a representation of the local spatial data that is often lacking in point-based architectures. The constructed features can be used together with the existing point data for more accurate per-point labeling than would otherwise be possible in more simple architectures.

Later research showed successful results when working with point sets by transforming the point set to be processed by a convolutional neural network: PointCNN [11] achieves this by preprocessing the input data by applying a χ-transform on the input point set. DGCNN [1] and Point-GNN [12] first transform the point set into a graph representation and then process the resulting graph edges using a CNN. The performance of DGCNN [1] was further improved by replacing the fully connected classifier with a capsule network module in a study by Cheraghian and Petersson [13].

2.2 Volumetric based methods

Besides using a point set directly as input, a plethora of methods first transform the point set into a volumetric occupancy grid. This changes the unordered point set into a structured 3D voxel grid which greatly simplifies spatial relations between elements. These elements can then be processed using a 3D Convolutional Neural Network (CNN). Due to the high computational complexity of using 3D CNNs the effective maximum size of a voxel grid was estimated in 2017 to be around 30^3 voxels [14]. While with recent computational advances this estimate has most likely increased, volumetric methods will always suffer from a finite effective maximum size that is lower than that of point sets. The complexity of using volumetric methods lies in the fact that when the resolution of the voxel grid becomes larger, the computational complexity and storage requirements grow cubically. Despite these limiting factors, there are a number of studies that show very promising results [15, 16].

VoxNet [15] is one of the first approaches attempting to use a 3D CNN to process a voxel grid and showed good results in a 3D object recognition task.

Riegler et al. [14] managed to mitigate the constrained resolution by replacing the conventional voxel grid with a grid of shallow octrees in a network called octnet. By replacing the voxels in the grid with octrees Riegler at al. managed to allow the grid to contain a high concentration of data in grid cells where a significant amount of information is present, while also keeping the voxels at an efficiently small size in regions with low information density to aid in quick processing of the voxel grid.

2.3 Hybrid methods

Le and Duan [17] created PointGrid, which keeps the point set, but first transforms the set to fall into the unit cube $[-1,1]^3$. The cube is divided into equally sized grid cells, and points are sampled such that each cell contains the same number of points. Empty cells get padded with zeroed out points. The coordinates in the points are used as features of the cell they are in. This architecture allows data from individual points from a point set to be used, while retaining the structure of a volumetric network. Le and Duan noted that despite PointGrid taking slightly longer to execute than PointNet, it does generate better classification accuracies than both PointNet and VoxNet.

When it is impossible to obtain visual data of an object, a tactile sensor can be used to obtain a 3D representation of an object [18]. This can be very useful in low light environments, or when it is otherwise impossible to use a camera. A tactile sensor can also be used in addition to a visual sensor in order to improve the performance of a system [19]. The obtained point clouds can be processed in a similar manner as the point clouds obtained from a visual sensor. A significant downside of tactile sensors when used for object recognition is that, due to the nature of the sensor, the point cloud is often centered on one particular part of the object, which gives an imperfect representation of the object as global features are harder to establish [18]. Despite having these downsides, tactile sensors could nevertheless be an interesting approach in either supporting an existing object recognition system, or implemented as a secondary assessor when the primary system is unable to determine the class of the object.

2.4 Grasp specific methods

Grasp pose detection (GPD) was developed to generate and evaluate the fitness of grasps [20]. It takes a point cloud as its input and generates a number of grasps, which are then filtered by fitness by giving a 2D representation of the grasped slice of the convex hull to a convolutional neural network (CNN) architecture which is based on the architecture of LeNet [21]. The network then classifies the grasp candidate as either successful, or not successful.

PointNetGPD [22] builds upon the idea of GPD and expands on it by employing the PointNet architecture to evaluate the fitness of grasps. A rectangle is created around the intersection between the end effector of the robot arm and the object. The resulting slice is processed using PointNet. Using the PointNet architecture showed significant improvements over the LeNet architecture used in the original GPD paper [20]. Qin et al. [23] created S4G, a grasp generation network based on the architecture of PointNet, which achieved results comparable with GPD and PointNetGPD.

Mousavian et al. [24] created GraspNet, which uses a variable autoencoder to generate a set of grasps for an object using a point cloud. It evaluates the fitness of the generated grasps using an encoder-decoder network. It is able to generate grasps with a success rate comparable to PointNet-GPD.

Breyer et al. [16] created the Volumetric Grasping Network (VGN) that employs convolutional layers, as well as skipped connections on a 3D voxel grid to synthesise grasp configurations for a number of different object shapes. Despite being object agnostic, the approach of training the network on a number of similar shapes is closely related to our method of training GraspCaps.

3 Theoretical background

3.1 Point sets

A point set (or point cloud, both terms will be used interchangeably in this thesis) can be described as an unordered set of vectors [6]. They differ from more conventional data representations in that they are an unordered data structure and are therefore hard to process using commonly used network architectures, as most architectures rely heavily on the spatial relation between elements.

Point clouds cannot easily be processed by convolutional neural networks (CNNs), so many approaches choose to first transform the point cloud into a data structure that is able to be processed by a CNN. As outlined previously in Section 2, a data structure that is relatively straightforward to generate from a point set is a voxel map. A voxel map can be described as a 3D grid of elements, called voxels. Due to their ordered 3D structure, they can easily be processed by a CNN. Converting a point set into a voxel map can be easily achieved by overlaying a 3D voxel grid over of the point set and counting the number of points in each voxel. The resulting voxel map can then be processed by a CNN.

This also brings to light an inherent disadvantage of converting a point set to a voxel grid. Whereas a voxel grid will inescapably contain a number of empty voxels, a point set will only contain informative data. A voxelized representation of a pointset also loses precision, as several points are merged into a single voxel, and therefore cannot be as descriptive as a point cloud. Furthermore, due to the computational complexity in processing 3D data increasing cubically when the input size increases, voxel grids are limited in size to the order of around 30^3 voxels [14]. Point sets are an interesting alternative, as they are quite compact in comparison while containing more representative data. When points are added to a point set, the computational complexity increases linearly, in contrast to the cubic increase present when working with voxel grids.

Working directly with point clouds has a number of challenges that need to be addressed. Firstly, point clouds are by definition unordered. This makes conventional deep learning strategies such as deep convolutional networks hard to apply, as they rely heavily on structure in the input data. Secondly, point clouds captured by depth sensors frequently used on robotic platforms are often highly irregular, having most of their points concentrated on the side of the object that is closest to the sensor [25]. Having a sensor only seeing one side of the object also means that the other side is not captured at all, leaving a sizable gap in the point cloud. Any information-rich features that might exist on the undetected side will therefore not be present in the captured point cloud. Thirdly, point clouds don't contain any information on the surface texture of the object, which can serve as a descriptive visual marker for identification.

Some of the above challenges can be mitigated; for instance, sub-sampling can be applied in order to increase the uniformity of point densities across the object. However, it seems impossible to fully remove the bias with current technological capabilities [25].

3.2 Neural networks

Although research on implementations of artificial neurons can be traced back to the McCullough-Pitts neuron [26], the field of neural networks started in 1958 in the form of the Rosenblatt perceptron [27]. The perceptron is the first model of neurons that was able to learn from the input it was given by changing the weights and threshold values of the artificial neurons that made up its network.

Later, back-propagation [28] was invented as a method to train neural networks with more than one layer. Using back-propagation, classification errors in the final layer of the network can be back-

propagated through the network and used to train neurons in each layer of the network. The ideal parameters of the neurons are then found by using gradient descent. By allowing networks to have more than one layer, the networks are able to solve more complex problems that are not solvable by neural networks with only one layer. A classic example being the XOR function, which cannot be solved by a perceptron due to its non-linearity, but can be solved by networks with more than one layer. This led to the creation of the multi-layer perceptrons (MLP) [29, 30] architecture, which is an architecture with a layer of input neurons, one or more hidden layers, and an output layer. Each of these layers is fully connected with the one before it. While these networks are useful for a myriad of problems, current research interest seems to mainly be focused on deeper, convolution-based networks.

3.3 Deep learning

Deep learning is a form of machine learning in which a neural network composed of a large number of layers is trained to perform a particular task. Because these networks operate as universal function approximators, they are suitable for providing solutions where more traditional methods fail. A popular example is image recognition. It is difficult for a classical model to determine which objects are captured in an image due to the myriad of factors influencing the composition of an image. Colour saturation, object pose, and partial occlusion of the pictured object pictured are some examples of problems commonly encountered when determining what is pictured in detail. Conversely, humans themselves have little issue dealing with these problems in daily life, as we are used to operating in a world where objects are frequently moved, obscured, or badly lit.

Currently, deep learning is the most popular field of study in artificial intelligence. This is likely due to the abundance of large scale data sets that are widely available and the significant interest the field garners from large technology corporations such as Google and Facebook. Advances in computing, especially in the design and availability of graphical processing units (GPUs) allows for networks of ever growing size to be trained in an efficient fashion, accelerating the field even further.

3.4 Activation functions

Activation functions are passive operations on tensors and are applied in neural network architectures in order to constrain the output of a layer of a neural network in a non-linear fashion to better facilitate learning. They contain no trainable parameters and all implement a single, differentiable function. An important aspect of activation functions is that they have to be differentiable in order for the gradient to be accurately calculated. If the gradient cannot be calculated, back-propagation cannot be used to adjust the parameters of the neural network and training will not occur. Constraining the activation of neurons beyond simple linear activation is necessary as unbound activation might lead to the exploding gradient problem, and activation functions supply the necessary non-linearity that is needed for learning complex patterns. The following part of this section will describe all activation functions that have been used during construction of the neural network architecture, along with their specific strengths and use cases.

3.4.1 Sigmoid

Sigmoidal activation [30] constrains the activation of a neuron between zero and one using the sigmoid function, which grows closer to one as the input goes towards infinity, and closer to zero as the input goes towards negative infinity. The equation is defined as follows:

$$\sigma(x_i) = \frac{1}{1 + e^{-(x_i - \theta)/\rho}} \tag{1}$$

where θ can be interpreted as the threshold needed for the neuron to fire. ρ determines the shape of the sigmoid. Large values make the curve flatter, while small values make the curve more steep. At its extreme, $\rho = \infty$ implements a step function set at the threshold θ.

Current implementations often omit the two variables, implicitly setting them to $\theta = 0$, $\rho = 1$, leading to a simplified equation that grows closer to zero as x_i goes to minus infinity, and closer to one as x_i goes to infinity, with the threshold set to 0. The simplified sigmoid function can be defined as follows:

$$\sigma(x_i) = \frac{1}{1 + e^{-x_i}} \tag{2}$$

3.4.2 Rectified Linear Unit (ReLU)

Rectified Linear Unit [31], more commonly called by its abbreviation ReLU, is one of the most used activation functions in current neural networks. The function returns zero if the input is lower than zero, otherwise it returns the input itself without modification. The function can be found in Equation 3. Due to its simplicity ReLU is computationally inexpensive while still being non-linear, which makes it a good candidate for deep networks with a large number of trainable parameters. ReLU is a non-saturated function, opposed to sigmoidal activation. The advantage of a non-saturated activation function is twofold: First, it solves the exploding/vanishing gradient problem. Second, it accelerates the convergence speed [32].

$$\text{ReLU}(x_i) = \begin{cases} x_i & \text{if } x_i \geq 0 \\ 0 & \text{if } x_i < 0 \end{cases} \tag{3}$$

Apart from these advantages the ReLU function has one major disadvantage: the dying ReLU problem [33]. When the weights to a ReLU neuron are excessively negative, the neuron will only output zero. This causes the neuron to stop learning and effectively ”die”, which gives the dying ReLU problem its name. This can happen when the learning rate is too high, which can cause the weights to be updated too extremely, or due to unfortunate initialization parameters. A thorough theoretical analysis of the dying ReLU problem can be found in [33].

Solutions to the dying ReLU problem all introduce some parameter to allow the gradient to flow through the neuron in the event of a negative input. Leaky ReLU [34] adds a parameter a_i that is used to determine the fraction of input that is passed through in case of the input being lower than zero. This keeps the neuron from dying and allows the network to correct itself in the event that the neuron ceases to function. The function is defined as follows:

$$\text{Leaky ReLU}(x_i) = \begin{cases} x_i & \text{if } x_i \geq 0 \\ \frac{x_i}{a_i} & \text{if } x_i < 0 \end{cases} \tag{4}$$

where a_i is a fixed parameter in the range $[1, +\infty]$.

Randomized leaky ReLU (RReLU) [32] introduces noise into the training process by randomly sampling the value of a_i from a uniform distribution in the range $U(l, u)$, where l is the lower bound, and u is the upper bound of the distribution. Randomized leaky ReLU has been shown to improve performance on several datasets when compared to both regular and leaky ReLU [32].

Parametric ReLU (PReLU) [35] takes the leaky ReLU given in Equation 4, but treats the static a_i parameter as a trainable weight instead, allowing the network to tune the value of a_i for each individual neuron.

As a side note, a second problem of ReLU activation is that it's technically not differentiable when the input is zero, which makes it impossible for the gradient to be computed correctly when the input is zero. This is not a problem in practice however, as a value of exactly zero is almost never reached, and nearly all of the available implementations of the activation function add a very small constant to the input in order to mitigate this problem in the event that a value of zero is used as input.

3.4.3 Softmax

While not technically an activation function, softmax is often used in the construction of neural networks, and its function definition lends itself well to inclusion into this section.

Softmax scales all elements of a vector such that their combined value adds up to 1.0. When applied to the final layer of a network this effectively transforms the output vector into a vector of probabilities. This can be useful when training a network to determine the class of an object, as the probabilities obtained from a softmax operation will resemble the target vector more than the raw output vector. This operation should result in a more stable gradient during training, which in turn leads to a more efficient training process. The function is defined as follows:

$$S(\bar{x}_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \tag{5}$$

where $\bar{x}_i$ is a vector input to the function and K is the length of the input vector.

3.5 Adam optimizer

The Adam optimizer [36] computes individual adaptive learning rates for each parameter from estimates of the first and second order moment that are calculated based on the gradient. The rate at which these estimates decay can be given by using hyper parameters. Initial moments are computed using a randomly initialized parameter vector. During training, bias correction terms will adjust the vector in the event that this vector does not align with the direction of the gradient.

The individual learning rates allow for the network to optimize sections of the network that require more training, while not influencing sections that are already functioning properly. As the learning rates are updated dynamically, the optimizer is able to efficiently train the entire network.

3.6 Dropout

Dropout [37] is a method used to reduce overfitting. It functions by randomly dropping out neurons from the network with a preset probability p during training epochs. When a neuron is dropped out it is treated as though it did not exist, so it gives no output and does not have its weights updated. Using dropout reduces overfitting by decreasing the probability of co-adaptation of neurons, and the likelihood that the network starts to depend on a small subset of neurons. It also increases the robustness of the network by forcing the network to be accurate, even in absence of certain information it has grown to rely on [38].

3.7 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) derive their name from the main operation they perform: convolution. Using the convolution operation, CNNs are able to slide small feature detectors in the form of kernels over the input data to find recurring patters in the input data. Each kernel can be trained to respond to specific features, which results in increasingly more information-dense representations of the input as the depth of a CNN increases.

CNNs are especially well-suited to working with image data, as the shape of a large 2D image with an additional dimension for colour lends itself well to processing using a sliding window. By using several convolutional layers in succession, more and more abstract and complicated features can be found in the image. This is useful when performing object detection in an image, as the appearance of an object might differ between several objects that belong to the same class.

CNNs can be composed of a wide variety of different layers, but at their core most networks consist of three basic layer architectures: convolutional layers, pooling layers, and fully-connected layers. A comprehensive survey of CNNs and their architecture can be found in [38].

Convolutional layers are implemented by defining one or more N-dimensional convolutional kernels that can be used to extract features from the input. These kernels consist of trainable weights and are slid over the input data in a step-wise fashion. By sliding the kernels over the input data they're able to use a relatively small number of weights to extract location invariant features from input data that might be several times larger than the kernel, which allows for efficient processing even at large input sizes.

Pooling layers are used to reduce the dimensionality of the data. They aggregate the features into a lower dimensionality to lower the computational load of the network while retaining the learned features of the preceding layers. They tile the entire input matrix using a receptive field that reduces the dimensionality based on set criteria. The two criteria most often used for pooling layers are max pooling and average pooling. In max pooling, the operation reduces the dimensionality of the input by taking the maximum value of the neurons in its receptive field and mapping this value to a single output neuron. Average pooling functions the same as max pooling, but instead of taking the maximum value it takes the average value of its input.

Fully-connected layers are used as the final layers in a CNN, and are often used when the output of a convolutional layer or pooling layer is sufficiently small that the features can be processed with a more interconnected architecture. Fully-connected layers connect, as their name implies, all input neurons to all output neurons, similar to the functioning of a layer in a multi-layer perceptron. As this can be very computationally intensive, fully-connected layers are often only used as the last stage of a CNN. By using a fully-connected layer as the final layer in a CNN, the CNN can be used in tasks such as object recognition by outputting a vector of class probabilities.

3.8 Dynamic Graph CNN

Graph neural networks [39, 40, 1] are a specific type of neural network that are able to function on graph data instead of the more often used Euclidean data, which are used when processing images or voxel grids. Graph data can be more complex than Euclidean data, as nodes can be unordered, the number of connections can differ between nodes, and the number of nodes in a graph can differ between graphs, making operations that are easy and straightforward to apply to the Euclidean domain (e.g. convolution) difficult to apply to the graph domain [39].

When viewing point clouds through the graph domain, we can view the points in the point clouds as nodes in a graph, and we can construct edges between points based on their proximity to each other.

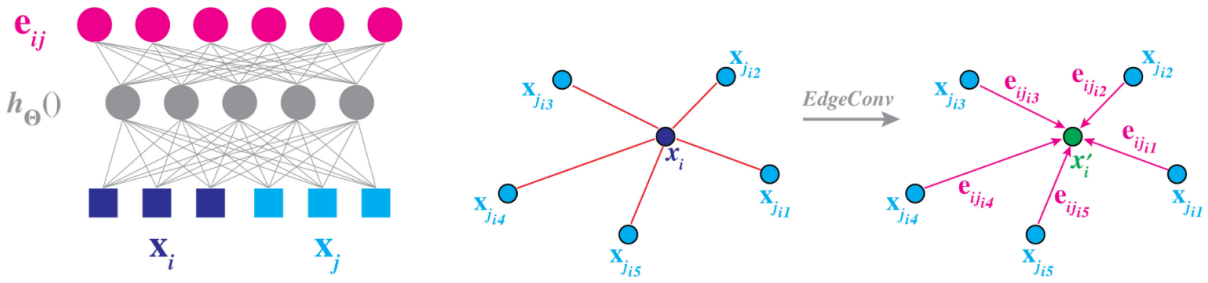

Figure 3.1: Left: Computing an edge feature, $\mathbf{e}_{ij}$ (top), from a point pair, $\mathbf{x}_i$ and $\mathbf{x}_j$ (bottom). In this example, $h_\Theta()$ is instantiated using a fully-connected layer, and the learnable parameters are its associated weights. Right: The EdgeConv operation. The output of EdgeConv is calculated by aggregating the edge features associated with all edges emanating from each connected vertex. Image and caption reproduced from [1].

Dynamic Graph CNN (DGCNN) [1] applies convolution to point clouds by first transforming the unordered point cloud in a graph by constructing a local neighbourhood graph using the k-Nearest Neighbours (kNN) algorithm [41] and applying a convolution-like operation to the edges connecting neighbouring pairs of points called EdgeConv. This operation is applied in multiple layers which allows the network to learn more abstract graph representations of the data in deeper layers. Note that the edges in the graphs in deeper layers most likely consist of abstract semantic similarities, and may have little to do with spatial proximity depending on how the network is trained.

3.8.1 EdgeConv operation

This subsection is mainly reproduced from the explanation given by Wang et al. in [1], with some minor alterations to fit our implementation and use case.

From a point cloud $X = \{\mathbf{x}_1, ..., \mathbf{x}_n\} \subseteq \mathbb{R}^3$, in which $\mathbf{x}_i = (x_i, y_i, z_i)$, are the (x, y, z) coordinates in 3D space, a directed graph $\mathscr{G} = (\mathscr{V}, \mathscr{E}))$ is constructed representing local point cloud structure, where $\mathscr{V} = \{1, ..., n\}$ and $\mathscr{E} = \mathscr{V} \times \mathscr{V}$ are the vertices and edges of the graph, respectively.

In the implementation used for this project, $\mathscr{G}$ is constructed as the kNN graph of X. The graph includes self loops, so each point is connected to itself with an edge of length 0.

Edge features are defined as $\mathbf{e}_{ij} = h_\Theta(\mathbf{x}_i, \mathbf{x}_j)$, where h_Θ is a nonlinear function with learnable parameters Θ.

Finally, the EdgeConv operation is defined by applying a channel-wise *max* operation to the edge features associated with all edges originating from each specific vertex. This can be formalized as:

$$\mathbf{x}'_i = \max_{j:(i,j)\in\mathscr{E}} h_\Theta(\mathbf{x}_i, \mathbf{x}_j) \tag{6}$$

For a visual explanation of the EdgeConv operation, see Figure 3.1.

3.9 Capsule networks

Convolutional neural networks (CNNs) are able to encode location invariant features due to their architecture. The kernels in convolutional layers slide over the entirety of the input, which allows the kernels to learn generic features and identify these features in the input regardless of location. One limitation that convolutional neural networks face is that while they are location invariant, they are

not viewpoint or rotation invariant [42]. This implies that for a single feature, multiple kernels need to be trained to identify the feature when viewed from different viewpoints or rotations. This can potentially lead to larger models with a high amount of redundancy. The increase in size of a model would logically also lead to a higher training time, a higher execution time, and a larger file size of the stored model.

Capsule networks [42, 43, 44] are an approach to generalize the performance of conventional CNNs by replacing the convolutional layers with multiple groups of neurons in structures called capsules. Each capsule is able to train on classifying a particular feature in the input data, and due to their encapsulation they should be able to encode viewpoint-invariant features. By implementing capsules in both shallow and deep layers of the network, the capsules close to the input are able to encode simple features in the data, whereas deeper capsules are able to learn abstract, high-level concepts. In object recognition tasks, the capsules in the deepest layer encapsulate classes. The combined activation of all neurons in these capsules corresponds to the probability that the input is of that particular class. This way, different neurons can encode different aspects of a class within the same capsule, and not all aspects have to be always present in order to get a correct classification. The activity of the neurons in a capsule explicitly represents the instantiation parameters of the object the capsule represents [42, 43].

Our implementation of a capsule network closely resembles the description given by Sabour et al. [42], with the key difference being that, where they encapsulate 2D convolutional layers in capsules we encapsulate 1D convolutional layers, due to our network processing a 1D feature vector instead of 2D images as is the case for [42].

3.9.1 Architecture

The architecture of a capsule network can be split up into three distinct sections: an initial layer that transforms the input neurons for use in the capsule network, a primary capsule layer, and one or more secondary capsule layers. The initial layer is a conventional neural network layer without capsules. Sabour et al. [42] use a 2D convolutional layer as the initial layer for their CapsNet architecture, as CapsNet processes 2D images. We use a fully-connected layer in order to process a 1D feature vector. Our architecture will be explained in detail in Section 4.5.2.

The primary capsule layer consists of a number of capsules that each contain a number of neurons laid out in a conventional layer architecture such as 2D convolutional layers. It is useful to think of the primary capsule layer as a group of parallel layers, all processing the same input simultaneously, but all trying to detect different features in the input.

The secondary capsule layer consists of a single $n \times d$ weight matrix, where n denotes the number of capsules, and d the number of neurons per capsule. The deepest secondary capsule layer contains the instantiation parameters of the object present in the input.

It is stated in [42] that the length of the output vectors of capsules should correspond to the probability that the feature represented by the capsule is present in the current input. Therefore, a non-linear squashing function is used to ensure that short vectors (i.e. capsules with very low activation) get shrunk down to almost zero length, while very long vectors (capsules with very high activation) get shrunk down to a length slightly below one.

$$\mathbf{v}_j = \frac{||\mathbf{s}_j||^2}{1 + ||\mathbf{s}_j||^2} \frac{\mathbf{s}_j}{||\mathbf{s}_j||} \tag{7}$$

where $\mathbf{v}_j$ is the output of capsule j, and $\mathbf{s}_j$ is its input.

The primary and secondary capsule layers are connected to each other using a process called routing. Several types of routing have been developed in the literature, including routing using max-pooling [43], dynamic routing-by-agreement [42], and expectation-maximization (EM) routing [44]. In our GraspCaps architecture we only use routing by agreement as described in [42]. The algorithm for routing by agreement is given in pseudo code in Algorithm 1.

Algorithm 1 Routing By Agreement

procedure ROUTING($\hat{\mathbf{u}}_{j|i}, r, l$)
  for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow 0$ ▷ Initialize logits to zero
  for r iterations **do**
    for all capsule i in layer l: $\mathbf{c}_{ij} \leftarrow \text{SOFTMAX}(b_{ij})$ ▷ softmax implemented in Eq. 5
    for all capsule j in layer $(l+1)$: $\mathbf{s}_j \leftarrow \sum_i c_{ij}\hat{\mathbf{u}}_{j|i}$ ▷ summed input from capsules in layer l
    for all capsule j in layer $(l+1)$: $\mathbf{v}_j \leftarrow \text{SQUASH}(\mathbf{s}_j)$ ▷ squash implemented in Eq. 7
    for all capsule i in layer l and capsule j in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{\mathbf{u}}_{j|i} \cdot \mathbf{v}_j$
  end for
  return $\mathbf{v}_j$ ▷ return squashed capsule output
end procedure

As all secondary capsule layers are implemented as trainable 2D weight matrices, the output of all secondary capsules can be defined as a weighted sum over all prediction vectors $\hat{\mathbf{u}}_{j|i}$ from the capsules in the layer below, and is produced by multiplying the output of one of these capsules $\mathbf{u}_i$ with the weight matrix $\mathbf{W}_{ij}$

$$\hat{\mathbf{u}}_{j|i} = \mathbf{W}_{ij}\,\mathbf{u}_i \tag{8}$$

$$\mathbf{s}_j = \sum_i c_{ij}\,\hat{\mathbf{u}}_{j|i} \tag{9}$$

where c_{ij} are coupling coefficients that are determined by the routing-by-agreement algorithm. In the routing algorithm, the coupling coefficients c_{ij} are iteratively updated by use of the algorithm outlined in Algorithm 1. After r iterations the output of the capsule layer $\mathbf{v}_j$ is returned.

Experiments done by Sabour et al. [42] have shown that convergence of the network increases as r increases, up to a value of $r = 5$, after which the increase in convergence is negligible. When using values higher than $r = 3$, it has been observed that the network starts to overfit on the input data. Due to these factors Sabour et al. recommend a value of $r = 3$ for use in experiments.

3.9.2 Interest for grasping

The fact that the activity in a capsule represents the explicit instantiation parameters of an object (i.e. location, rotation, scale) makes capsule networks an interesting candidate for generating grasps. Capsule networks have been shown to be able to accurately capture the instantiation parameters of multiple types of input data, be it RGB images [42], or point clouds [13], and trained reconstruction networks are able to accurately reconstruct the input data using these instantiation parameters. This indicates that the capsule network architecture is able to extract the instantiation parameters for several types of input data. Intuitively, if the network can accurately replicate the input data from the activation in a capsule, the information in that capsule should also be sufficient for generating grasp vectors.

Intuitively, the activation of a capsule layer should suffice as input for a network creating grasp vectors, as all information regarding the location and orientation of that object is stored in the relevant capsule. The grasp generation network itself does still need to learn a representation of how exactly an object should be grasped, but when looking at the reconstruction network's ability to reconstruct the input of a network based on instantiation parameters found in the capsule layer, it is likely that a network would be able to accurately synthesise grasps using this data.

3.9.3 Application to point sets

Capsule networks have been successfully used to process point cloud data in both object classification [13], as well as object segmentation [45] tasks. To the best of our knowledge this is the first study attempting to apply capsule networks to a grasping task.

3.10 Datasets

As versatile as neural networks are, their performance is often limited by the dataset they are trained on. For example, A neural network trained only on a dataset consisting only of pictures of poodles will most likely fail to classify a pit bull as a dog. For a dataset to be representative of the objects it defines, it needs to have a broad array of samples that are all equally represented. A bias in the number of samples per class can directly result in a bias toward the overrepresented class in the resulting network. One example of this is that many neural networks classifying faces are biased towards white people due to the number of white faces vastly outnumbering other faces in datasets commonly used for training neural networks [46]. This shows not only the importance of creating unbiased datasets, but also the difficulty in creating unbiased datasets due to inherent biases that people creating the datasets might subconsciously hold.

Besides needing to be balanced, a dataset also needs to be of a sufficient size and have a broad array of unique samples showing the classes in a multitude of different settings and configurations. If the dataset is too small, the risk of overtraining on the available data increases significantly, and any results that are achieved on the dataset in terms of accuracy are unlikely to carry over to new and unseen samples. Therefore, these results do not represent the capabilities of the network accurately in this case and might be misleading.

3.11 Grasping

Robotic arms are the primary interface a robot has to interact with the world around it. They allow a robot to perform manual tasks by picking up and placing objects, which greatly increases their functionality and usability.

In the remainder of this section we will elaborate on a number of different setups in which robotic arms are used, including single-armed and multi-armed setups, and multiple types of end-effectors.

3.11.1 Single-armed grasping

Single armed robots are extensively used in industrial settings for pick-and-place tasks, as well as for assembly tasks. In a single-armed robotic system, a robot has to use a single arm to interact with the environment. If an object the robot needed to grasp was covered by another object, it would first need to move that object before being able to interact with the object it needs.

3.11.2 Multi-armed grasping

Multi-armed robotic systems interact with the environment by use of two or more robotic arms. In the most common setting two arms are used in a dual-arm configuration. Using two arms allows robotic platforms to interact with the environment in a manner that is more human-like and efficient, allowing the robot to perform a more extensive and complex set of tasks.

In multi-armed grasping we make the distinction between *synchronous* and *asynchronous* grasping. In a *synchronous* grasping setting, two or more arms are working in a highly coordinated fashion and are all interacting with the same object. This is often useful when having to move large and heavy objects, for example when the robot needs to move a chair, or when an object needs to be handled in a specific manner that requires two arms, like a pan filled with water.

In an *asynchronous* grasping setting, both arms are working in the same environment, but not on the same object. In a pick-and-place task for example, each of the arms could pick up a number of the required objects, which greatly reduces the time it takes to fulfil an order. Asynchronous grasping also solves the problem of stacked objects posed in the previous section, as one arm could pick up the topmost object and hold it while a second arm picks up the target object.

A significant factor in multi-armed systems is that they are often not dynamic, instead relying on pre-programmed paths that do not take the environment into account. What keeps multi-armed robotic systems from becoming more widely used in dynamic settings is mainly the increase in complexity in planning, as the planner now has to ensure all of the arms have a valid trajectory, while not colliding with one another. Planning a complex path for multiple arms is time and resource intensive, so dynamic planning in a multi-arm system is a non-trivial task, especially when it is expected that a robot responds in a timely fashion.

3.11.3 End effectors

A robot that has to interact with small and fragile objects has a different set of needs from a robot that needs to be able to move around big and heavy objects. To this end, multiple end effectors have been developed to work in a myriad of situations.

An end effector is the part of the arm that interacts with the environment, analogous with the human hand. In contrast to human hands, end effectors can be easily switched out for different types, and custom end effectors have been developed for many situations. Grippers are the most commonly used type of end effector, and can exist in a two-fingered symmetric layout, or in three- or more fingered layouts as can be seen in Figure 3.2.

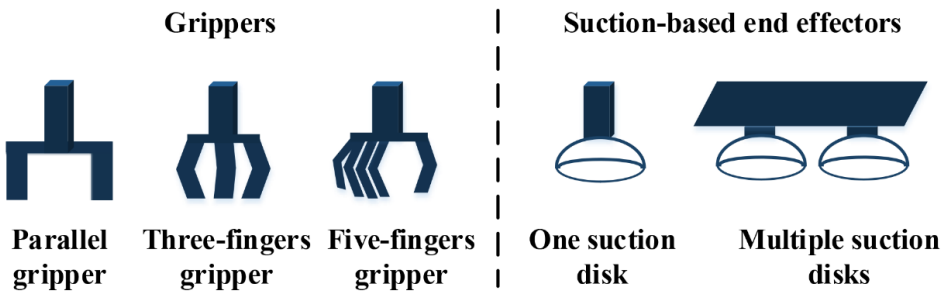

Figure 3.2: A selection of end effectors. Left: two, three, and five fingered grippers. Right: suction-based end effectors. (Image reproduced from [2].)

For grippers, objects can be picked up in a variety of ways depending on the needs of the grasp: *Fingertip grasps* pick up an object using only the fingertips of the gripper, and are utilized for small objects that lay down on a surface such as pencils or tinder boxes.

Parallel grasps pick up an object using two or more parallel fingers, and try to maximize the surface of the grippers with the surface for a stable grasp. This can be very useful with box-shaped objects that have flat sides.

Finally, *enveloping grasps* grasp objects by enveloping the object with the fingers. This requires a specialized gripper that has fingers consisting of multiple individual segments that can move individually, closely resembling the fingers of a human hand. Enveloping grasps are especially useful when grasping objects that have a cylindrical, or polygonal shape, such as water bottles or kitchen appliances.

Apart from grippers, suction cups can be used in order to pick up objects with a smooth surface, which can be beneficial in certain settings such as picking up a plate, which is difficult to do with a gripper as the edges of a plate are very close to the surface it is resting on.

For the research done in this thesis we will exclusively use a symmetric two-fingered end effector without individual finger segments. The exact end effector that will be used for the experiments in this thesis is the Robotiq 2F-140 gripper[1].

3.11.4 Collision avoidance

Collision avoidance in a robotic arm can be roughly divided into two types: self-collision avoidance (SCA) and collision avoidance with the environment.

Self-collision avoidance is needed to ensure that a robotic arm does not collide with itself or another part of the robotic system. The standard way to deal with SCA is to either plan the complete trajectory the arm will take before executing any movement, or by reacting when the arm comes close to colliding with itself [47].

Collision avoidance with the environment prevents the arm from colliding with any objects in its environment. In our experimental setting we can define the environment as the table the robot is positioned on, the object it needs to grasp, and the basket it needs to drop the object into. A full explanation of the environment and task description will be given in Section 5.

In order to avoid collision with the environment, the planner needs to keep track of all the objects in the environment such that it can plan the movement of the arm around them. This is commonly done by creating an internal representation of the location, shape, and size of all objects in a collision map.

An alternative to using planning software is to use pure inverse kinematics to determine movement of the arm. This does not allow for any guarantees regarding collision avoidance, but does ensure both smooth motion from start to end position as well as near-immediate response times.

3.11.5 Inverse kinematics

Inverse kinematics calculates the target joint positions for all joints in a robotic arm given a target position and orientation for the end-effector [48]. It is used to plan a path such that the end effector will be positioned at the target location when the arm has successfully followed the trajectory set out by the path.

A thorough mathematical explanation complete with examples can be found in [49].

[1] https://robotiq.com/products/2f85-140-adaptive-robot-gripper

3.11.6 Planning versus pure inverse kinematics

Using collision avoidance requires the use of a planner, which comes with a number of benefits, as well as several drawbacks. Planning the path beforehand ensures that the arm will not collide with any object while it carries out its trajectory, but has the drawback that it only functions in a static environment, as it can not adapt to changes in the environment during execution of the motion. This significantly limits its usability in a service robot setting, as the state of the environment can not be guaranteed to remain constant, especially when factoring in the time it takes to plan the trajectory of the arm prior to executing the motion. Dynamic planning during execution is therefore desired.

The time-intensive nature of iterative motion planning makes it unsuitable for dynamic use during execution of a grasp. Potentially, the rise of new algorithms and faster processing power might make it a feasible option in the future. However, if dynamic response to the environment is of the essence, pure inverse kinematics might currently be the more desirable option.

Inverse Kinematics has a comparatively low execution time and is suitable for dynamic use during a grasp, albeit at the cost of less robust collision avoidance. Pure inverse kinematics also allows for a smoother and more natural motion from the current position to the target position, as it calculates the minimal motion required to go to the target position. A limiting factor with planners is that they often plan collision free, but inefficient paths, that can cost additional time compared to an inverse kinematics solution.

Some steps could be taken to mitigate the lack of collision avoidance when using inverse kinematics, such as using predefined locations to guide the path along favourable positions to minimize collisions with known locations of objects, but pure inverse kinematics will never have the object avoidance capabilities a planner has in a static environment.

As we have described, both approaches have their pros and cons. For the experiments in this thesis we have chosen to use a pure inverse kinematics planner to allow for more natural movement, and use key locations in order to guide the arm toward favorable positions and limit potential collisions.

4 Methods

This section will explain the architecture of the GraspCaps network in detail, as well as the loss functions that were used to train it. It will also elaborate on the dataset that we've created for this thesis and on the algorithm that was used for generating the target grasp configurations. These grasp configurations were then used as targets for the point clouds in the dataset.

4.1 Architecture of the network

As shown in Fig. 4.1, the GraspCaps architecture consists of three main modules: the feature extraction module, the capsule module, and the grasp synthesis module.

The *Feature Extraction Module* takes a normalized 1024×3 dimensional point cloud as its input, and processes the point cloud using the feature extraction module from the Dynamic graph CNN (DGCNN) [1] network to generate a 1024×1 dimensional feature vector. To achieve this, it processes a point cloud by use of four EdgeConv layers. As described in Section 3.8, an EdgeConv layer first transforms the point cloud into N directed local graphs of k nodes, where N is the number of points in the input point cloud, and k is an integer representing the number of closest neighbouring points that are used for constructing the local graph. The local graph is constructed using k-Nearest Neighbours [41]. The network then applies a shared multi-layer perceptron (MLP) on the edges of the graphs. This process is repeated for each of the five EdgeConv layers in the feature extraction module. The output of all five EdgeConv layers is concatenated and processed using a shared MLP and pooled to a 1024×1 feature vector using an adaptive max-pool operation.

The *Capsule Module* of our network consists of a single fully-connected layer, a primary capsule layer, and a secondary capsule layer. The fully-connected layer consists of 256 neurons using sigmoidal activation that transform the feature vector for use in the primary capsule layer. The primary capsule layer consists of 16 capsules, each containing 4 neurons. The secondary capsule layer consists of 8 capsules containing 30 neurons each. In the secondary capsule layer each capsule corresponds to

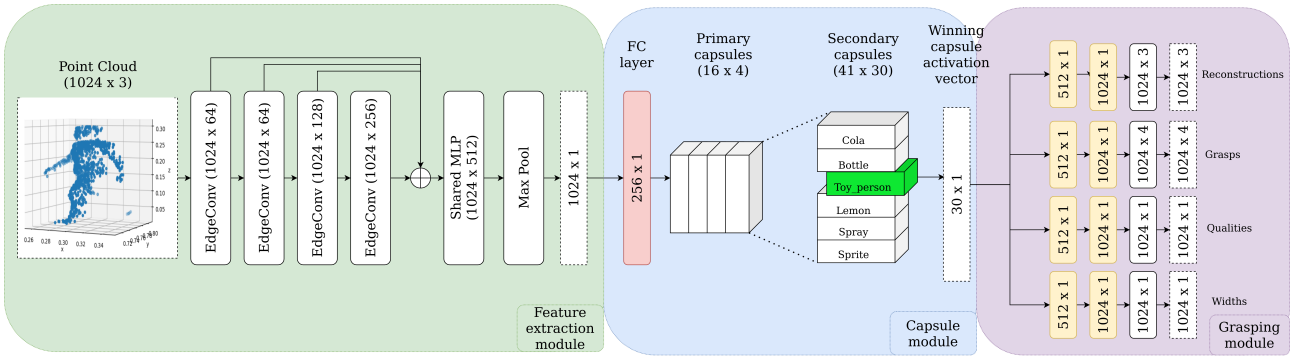

Figure 4.1: Diagram of the GraspCaps architecture. The architecture is divided into several modules: the *Feature extraction module*, the *Capsule module*, and the *Grasping module*. The object classification is determined by looking at the activation of each capsule. The capsule with the highest activation is chosen as the winner, and the corresponding object is determined to be the object in the input data. Layers shown in yellow are fully-connected layers using Leaky ReLU activation. Layers shown in red use sigmoidal activation. The output of the 1024×4 grasping head is normalized to obtain a unit quaternion, which is not shown in the diagram. Output dimensions of the network are shown in rectangular blocks with dotted outline.

one specific class of object. All parameters have been determined using parameter sweeps in earlier stages of the project.

The primary and secondary capsule layers are connected through routing by agreement [42]. It is worth mentioning that capsules in deeper layers encode more abstract and more meaningful features. Our network is trained such that each capsule in the secondary capsule layer corresponds to one object category. In other words, the Capsule module of our network classifies the input data to one of the semantic categories. The Euclidean norms of the activations of the capsules in the secondary capsule layer are ranked. The capsule with the highest norm is chosen as the winner. The activation of the winner capsule is then used for further processing in the next module. The activation of all other capsules is masked off so they don't influence further processing.

The *Grasping Module* consists of four small fully-connected heads. The architecture of each of the heads is based on the architecture of the reconstruction network as described in [42]. Each head consists of three fully-connected layers. The first two layers consist of 512 and 1024 neurons respectively, using leaky ReLU activation. The third layer size is dependent on the output shape of the network and uses linear activation. The reconstruction head has an output shape of 1024×3 to generate point locations in 3D space. The grasp head has an output shape of 1024×4 to generate quaternion rotation vectors for each point. The quality and width heads both have an output shape of 1024×1, as they only need to output a scalar value per point. The output of the grasping head is normalized to obtain a unit quaternion vector.

Initial parameter sweeps indicated that these were the optimal parameters for capsule sizes and number of capsules, outperforming all other configurations over a 1500 epoch test run. The sole exception to this is the number of capsules in the secondary capsule layer. For this parameter the performance of the network seemed to keep increasing as the number of neurons increased. This increase was however not linear, but decayed as the number of capsules increased. The value of 30 neurons per capsule was chosen as an acceptable trade-off between maintaining good performance and keeping the size of the network trainable on a single GPU.

It is also questionable to what degree the performance of the network would increase above 30 neurons, and how much could be attributed to overfitting due to an increase in network size.

4.2 Loss functions

For the training of this network we have used a combination of several custom loss functions. Margin loss [42] is the primary component of the total loss, and is mainly used to train the capsule activation to be representative of the class it encapsulates. It is defined as:

$$\ell_{\text{margin}}(\mathbf{v}_i) = T_i \max(0, m^+ - ||\mathbf{v}_i||)^2 + \lambda(1 - T_i)\max(0, ||\mathbf{v}_i|| - m^-)^2 \tag{10}$$

where T is the existence of the object the capsule corresponds to in the input data, which is set to 1 if the object is present, and to 0 if the object is absent. The λ used is a constant scaling factor that is set to 0.5. It is used to scale down the shrinking of the activity vectors in the event that the output of the capsule is wrong, which stops the initial learning from shrinking the activity vectors [42]. m^+ and m^- are constants set to 0.9 and 0.1 respectively. $\mathbf{v}_i$ is the activation vector of capsule i. By calculating the Euclidean norm, $||\mathbf{v}_i||$, we obtain a scalar that can be treated as the probability that the corresponding class is present in the input.

Reconstruction loss is used as a regularizer that ensures that the capsule learns relevant information such that the object can be re-instantiated from only the activation in the capsule. In our architecture, it is defined as the mean squared error loss between the input point set and the reconstructed point set. It is scaled down by a factor β such that it does not overtake the other losses and functions only as a

regularizer. It is defined as the mean-squared error loss between the input point cloud and the point cloud generated by the reconstruction head:

$$\ell_{\text{mse}}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n}\sum_{i=1}^{n}(\mathbf{y}_i - \hat{\mathbf{y}}_i)^2 \tag{11}$$

The grasping loss is mainly based on the loss function defined in [16], which is composed of separate loss functions for rotations, grasp qualities, and grasp width. The main alteration that was made from the loss function is the inclusion of the T parameter, whose function is identical to the T described in the margin loss above, being set to 1 if the object corresponding to the capsule is present, and to 0 if it is not. Including this factor in the grasping loss ensures that the grasping network does not train on an input that does not correspond to the capsule.

A second alteration from the loss function is the definition of the quality loss $\ell_{\text{quality}}(q_i, \hat{q}_i)$, which we extend to allow any ground truth value $0 \leq q_i \leq 1.0$ instead of the original binary ground truth $q_i \in \{0, 1\}$ as used in [16]. This way q_i is interpreted more like a fitness value instead of a binary classification of either good or bad. This should allow the network to learn from usable but imperfect grasps, while ensuring that the best-fitting grasps will always influence the network most. Bad or colliding grasps are heavily penalized during data generation, so their fitness values are often very close to 0, eliminating the risk of the network training on unsuccessful grasps. As will be discussed in Section 4.3, all grasps with a fitness value below a specified threshold will be removed from the dataset to ensure that the network only trains on usable grasps, which fully eliminates the chance that grasps with a low fitness value influence the network during training. The grasping loss is defined as:

$$\ell_{\text{grasp}}(\mathbf{g_i}, \hat{\mathbf{g}}_i) = T_i[\ell_{\text{quality}}(q_i, \hat{q}_i) + q_i(\ell_{\text{rotation}}(\mathbf{r}_i, \hat{\mathbf{r}}_i) + \alpha\ell_{\text{width}}(w_i, \hat{w}_i))] \tag{12}$$

where both ℓ_{quality} and ℓ_{width} are defined as the mean squared error loss, as found in Equation 11. The rotation loss is defined in Equation 13. α is a constant scaling factor set to 0.001. As the end-effector of the robotic arm we are using is a symmetrical two-fingered gripper, a grasp vector rotated by 180° around the gripper's wrist results in effectively the same grasp [16]. Therefore, we define the rotational loss function to consider both grasps correct:

$$\ell_{\text{rotation}}(\mathbf{r}, \hat{\mathbf{r}}) = \min(\ell_{\text{quat}}(\mathbf{r}, \hat{\mathbf{r}}), \ell_{\text{quat}}(\mathbf{r}\pi, \hat{\mathbf{r}})) \tag{13}$$

where $\ell_{\text{quat}}(\mathbf{r}, \hat{\mathbf{r}}) = 1 - |\mathbf{r} \cdot \hat{\mathbf{r}}|$ is defined to use the inner product to calculate the distance between the ground truth rotation $\mathbf{r}$ and the generated rotation $\hat{\mathbf{r}}$. The loss function that is used for training the network incorporates all these loss functions as follows:

$$\mathscr{L}(\mathbf{t}_i, \hat{\mathbf{t}}_i) = \ell_{\text{margin}}(\mathbf{v}_i) + \beta\ell_{\text{recon}}(\mathbf{p}_i, \hat{\mathbf{p}}_i) + \ell_{\text{grasp}}(\mathbf{g_i}, \hat{\mathbf{g}}_i) \tag{14}$$

where $\mathbf{t}_i$ is the output of the network, consisting of the capsule output $\mathbf{v}_i$, the reconstruction of the input point set $\mathbf{p}_i$, and the generated grasp $\mathbf{g}_i$. $\hat{\mathbf{t}}_i$ is the target vector, composed of the input point set $\hat{\mathbf{p}}_i$, and the target grasp $\hat{\mathbf{g}}_i$. β is a scaling constant set to 0.0005. This constant is set to a small number in order to prevent it from overtaking the margin or grasping loss, making it act exclusively as a regularizer.

4.3 Dataset

The data that was used to generate the dataset is a 90 class dataset composed of domestic objects, such as cups, kitchen utensils, and drink cans. All objects are supplied as SDF files that can be loaded into simulation software [50].

Figure 4.2: All 90 objects from the Gazebo dataset laid out grouped in similarly shaped classes. The classes shown are (from left to right and top to bottom): hardware, boxes, spheres, vertical cylinders, vertical objects, bowls, horizontal cylinders, cutlery.

The dataset used for training the network was generated using Gazebo [51] simulation software. Each of the 90 objects was spawned individually with a random orientation within a random location in the robots visual field. A single sensor measurement was taken each time to obtain a unique point cloud and avoid duplicate data in the dataset. Sensor measurements were obtained using a simulated Microsoft Kinect camera. This process is repeated 143 separate times for each object in order to obtain 100 training samples and 43 testing samples per object using a 70%/30% training data to validation data split. This results in highly balanced training and testing sets of 9,000 samples and 3,870 samples respectively. By randomly initializing the object in the world, we can assume that the dataset generalizes effectively for different placements and rotations of the objects in the world.

Over time the project pivoted away from distinctly identifying all 90 different objects to focus more on familiar shape detection and grasping. This caused the number of classes to lower from 90 classes with a high amount of overlap to 8 distinct shape classes. In Figure 4.2 the grouping of the objects in classes can be observed. All objects that belong to the same class are grouped together. To which class each of the objects belongs is determined solely by the shape of the object. Bottles, drink cans, and Pringles cans all belong to the class of vertical cylinders for example.

In order to still maintain the 70% to 30% training to validation data split, and to ensure that the dataset does not become biased towards the classes with a large number of objects such as the box class, we have to significantly downsample the gathered data. After downsampling, we are left with a dataset of 4,576 unique point clouds (572 per class).

This dataset shall be called the *synthetic recognition dataset*, as its primary function is to train all layers up to and including the capsule module used for recognizing the object in the input. To increase the number of target grasping configurations, a second dataset is created that consists of 120 grasps per point cloud. As the computational costs of generating a large number of point clouds are significant, a balanced subset of 280 unique point clouds (35 clouds per class) were chosen and a total of 34,067 grasps were generated for training the network. This dataset shall be called the *synthetic grasping dataset*, as its function is to fine-tune the grasping module to generate usable grasps.

In order for the grasping network to be trained, grasp targets need to be generated for the data. We opted to design our own grasp generation algorithm based on simulated annealing. The algorithm

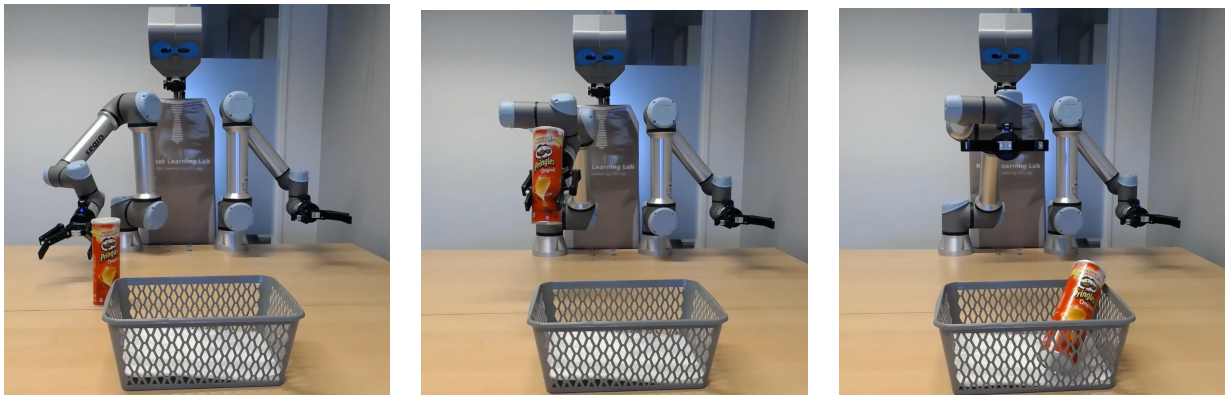

Figure 4.3: Left: Task setting in which the robot is instructed to pick up the Pringles can and drop it in the basket. Middle: The Pringles can has been successfully picked up and the arm moves to the drop-off zone. Right: The Pringles can has been successfully dropped into the basket.

randomly chooses one of the 1024 points of the input point set to center the grasp around, then through simulated annealing it iteratively updates its rotation and gripper width until it reaches an end state. The rotation quaternion, width and fitness of the grasp are recorded and can be used for training. The algorithm is explained in more detail in Section 4.5.

4.3.1 Real world dataset

In order to evaluate the performance of our approach in real world settings, a second dataset was created out of sensor measurements taken with a Microsoft Kinect camera of objects corresponding to the classes used in the simulation. Some of the objects used for creating the dataset can be seen in Figure 4.4.

Two real-world datasets were created. The first consists of 535 point clouds, each with one associated grasp, equally divided over 4 classes (134 per class). The second dataset consists of 220 unique point clouds (55 per class), with 120 grasps generated per cloud for a total of 26,643 grasps for fine-tuning the grasping network. These datasets will follow the same naming convention as the synthetic datasets, and shall be called the *real recognition dataset* and *real grasping dataset* respectively.

During data gathering in the real world it sometimes occurred that, after removing the background points and noise a point cloud would not have the minimum size of 1024 points necessary for training the network. In this event we try to still use the point cloud by generating an airtight mesh that fits the collected point cloud as tightly as possible, and sampling the remaining points from random locations on the mesh using Poisson disk sampling [52]. This procedure was only performed when the point cloud has a minimum of 800 points. Whenever the number of points in the point cloud falls below this threshold, we determine that there is too little information in it to be of use for training the network and the cloud is discarded.

4.4 Dataset augmentation

While training the network, several augmentation techniques are applied to the dataset in order to further increase the variation of data the network is trained on. Firstly, the point set is moved to have its x and y coordinates centered around 0. As the object is stationary on the ground, it can be assumed that the z coordinate is not lower than 0. The point cloud is then normalized.

Figure 4.4: Real robot setup with objects used for creating the real-world dataset.

For the point set, a small amount of noise is added to the location of each point. The noise is sampled from a normal distribution between the bounds $[-0.005, 0.005]$. As this noise is added to the point set after normalization between $[-1, 1]$, the effective noise rate is roughly equal to 5%. This ensures that the network is able to handle a small amount of sensor noise while classifying input data.

As all these augmentation techniques are applied and all the noise is calculated during training of the network, the network is subjected to a large amount of variation in the input data, which improves its robustness to noise, and minimizes the chances of overfitting on consistent noise patters and thereby missing the inherent structure of the data.

4.5 Grasp generation algorithm

In this work, the gripper is simulated as a combination of three separate boxes: two for the fingers, and one for the base (see Fig. 4.5). Initially, we randomly place an object inside the workspace of the robot and capture the object's point cloud. The point cloud of the object is then converted to a watertight mesh. It should be noted that converting the point cloud to mesh significantly speeds up the algorithm, as fewer collision checks are needed per iteration. Our approach then initializes the grasp in a random rotation, either with the gripper facing down towards the object or facing the object from the side. Afterwards, the orientation and opening width of the gripper are iteratively updated to converge on a grasp. The new state is accepted if the grasp has a higher fitness than the previous state, or by random chance using simulated annealing [53].

4.5.1 Simulated annealing

In simulated annealing, the system starts out with a high temperature. This temperature corresponds to a high chance of accepting new states, even if they are worse than the previous state. As the algorithm continues, the temperature of the system decreases and worse states are less and less likely to be accepted. In the final iterations of the program simulated annealing behaves almost exactly as stochastic gradient descent. The high temperature at the start of the process allows the algorithm to

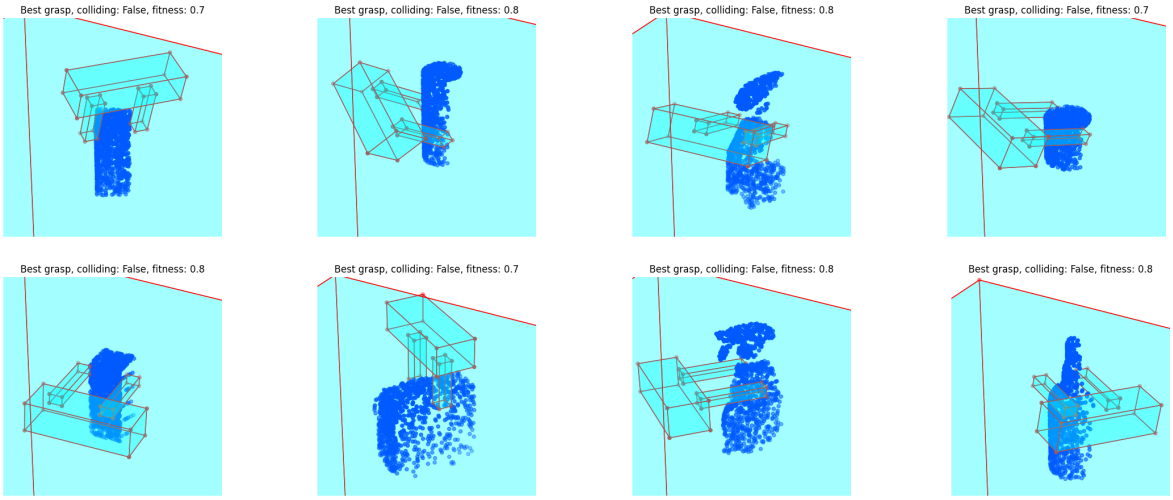

Figure 4.5: Grasps as generated by the algorithm described in Section 4.5. All eight examples have been taken from the synthetic dataset used to train the network. The gripper and floor are shown in light blue, with red lines to denote the edges of the collision objects. The point cloud of the object is shown in dark blue. During the process only the orientation and opening width of the gripper are altered. The location of the grasp center is determined by choosing a random point in the point cloud before converging using simulated annealing.

escape local minima and converge to a more global minimum, while the low temperature at the end of the process ensures that the algorithm does converge to a minimum.

In terms of grasp generation, simulated annealing allows the system to escape usable but imperfect grasps and reach better grasps. The only state transition that is not able to occur even when accepting random state transitions is the transition from a non-colliding state to a colliding state. We have also considered bounds on the rotation to ensure the algorithm does not generate grasps that are difficult for the robotic arm to reach. Examples of generated grasp synthesis for different objects are shown in Fig. 4.5.

4.5.2 Grasp fitness

The fitness of each state is determined by five factors:

- First, we determine a base fitness value based on the distance between the grasping point and the center of the object. The closer the point is to the center of the object, the higher the score is. As we cannot determine the center of mass of the object based on point cloud data alone, the center of the object is a good approximation. This factor is not updated during the simulated annealing process and remains constant.

- Second, we consider what percentage of the points is located between the two fingers of the gripper. This should increase the likelihood of the algorithm converging on grasping a large part of the object, and not on a small part that contains few points.

- Third, we determine the distance of the two fingers to the points located between the fingers. We take the set of points between the fingers and for each of the fingers calculate the distance between the point closest to the finger and the finger itself.

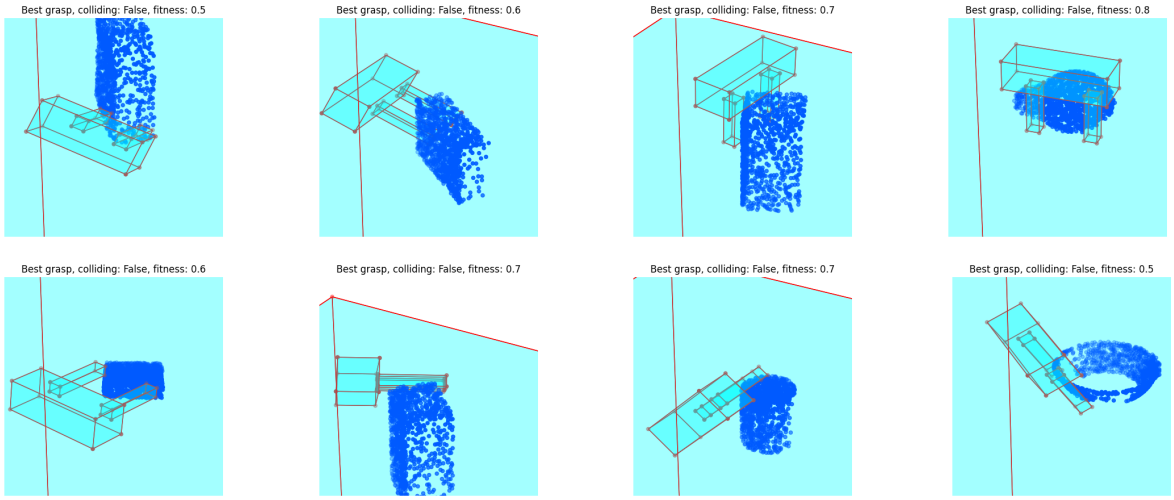

Figure 4.6: Examples of imperfect generated grasps using the algorithm specified in Section 4.5.

- Fourth, we look at how closely the normals of the gripper's fingers overlap with the normals of the object. If the normals are completely opposed to each other, it is likely that the object is grasped at an angle of 90° degrees, and the grasp should be a good fit.
- Fifth, we determine the distance between the two grippers. Since the grasp should be as tight as possible, a small penalty is added to the fitness for the distance the two fingers are apart. This penalty is not weighed too heavily, as there should always be some space between the fingers to account for the object being grasped.

It is possible that the algorithm converges on an imperfect grasp which would not work well when grasping an object. During generation of the synthetic dataset, images are collected of all generated grasp vectors. Several examples of imperfect grasps can be found in Fig. 4.6. It should be noted that these grasps often have a low fitness score of around 0.5 to 0.6, but outliers exist that have higher scores.

4.6 Behaviour architecture

The code architecture of the system we have created consists of multiple ROS action servers that each fulfill one specific purpose, and a single ROS node that completes the recognition and grasping task. A state machine of the full architecture as used for the Gazebo simulation experiments can be found in Figure 4.7. In the state machine failure states have been omitted for clarity and brevity, with the exception of the failure states in the Grasping subbehaviour, where it was deemed important to visualize the recovery behaviours of the system in the event of a failure.

The state machine shown in Figure 4.7 defines the state machine as used in Gazebo. For the real-world experiments the functioning is identical, with the exception of the spawn object server not existing and thus not being called. Classification and grasping success also cannot be automated in real life and needs to be done by a human supervisor.

Please refer to Section 5.4 for a detailed breakdown of the pre- and post-processing steps employed in the classification action server.

The behaviour starts at the top of the figure. For each of the action servers the main node needs to connect to, it initializes an Action Client and waits until all servers are operational. When all servers

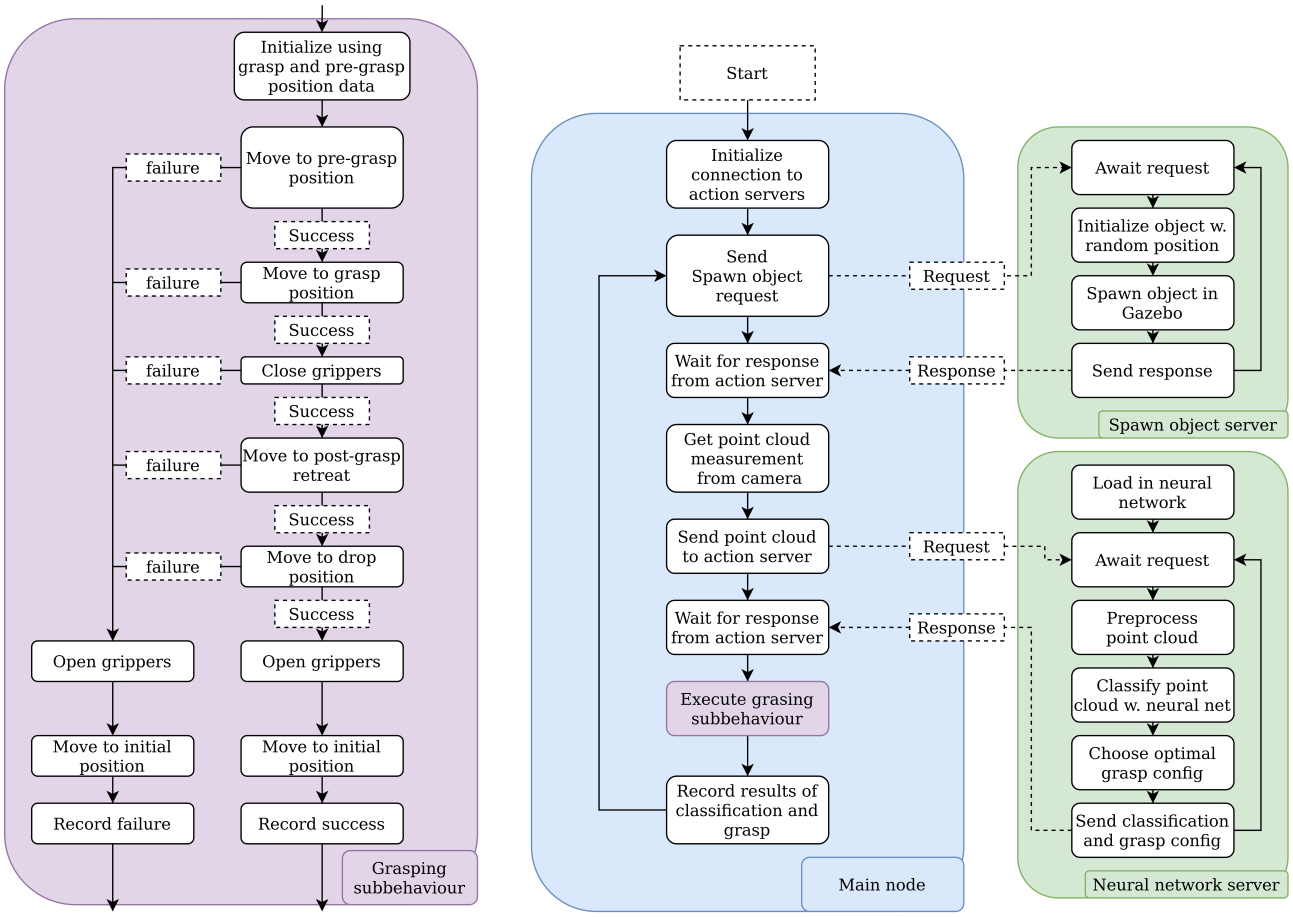

Figure 4.7: Simplified state machine of the behaviour structure used for performing the familiar object grasping task in Gazebo. State transitions are given as solid lines, message passes are given as dotted lines. The main node is given in blue. Action servers are given in green. The state machine shown in violet on the left is a zoomed-in version of the violet coloured state in the Main node.

have been initialized and connected to, the behaviour sends a request to the spawn object action server to spawn an object. This can either be a random object, an object specified by name, or an object from a specific class. Orientation and location parameters can optionally also be supplied, but for all testing in this project we used random position data such that we can accurately measure the performance of the system.

After the object server has spawned the object, the behaviour sends the last point cloud measurement it received from the Kinect camera to the neural network action server to determine the class of the object and the optimal grasping configuration. A more detailed description on the exact functioning of the neural network can be found in Section 5.

When the grasping configuration has been determined, the behaviour executes the grasping subbehaviour. The grasping subbehaviour is given in a separate node in Figure 4.7 as it contains many failure states that might decrease the legibility of the main behaviour if both were shown in the same node. It iteratively moves from the pre-grasp position to the grasp position. As the pre-grasp position is removed from the grasping position by a set distance in a straight line from the grasp position following the orientation of the grasp (see the visualized pre-grasp and grasp positions in Figure 5.7), the gripper can move to the grasping position in a straight line, significantly decreasing the chance of any collision. Planning the motion of the arm is performed by sending position commands to a ROS

service node (not pictured) that generates a path using inverse kinematics. The post-grasp retreat is defined as 20cm above the grasping position. Drop-off positions are defined for both arms above the basket in order to accurately drop the objects in the basket.

5 Experimental Results

In this section we will elaborate on the experiments performed and the achieved results. We will perform extensive experiments to evaluate the performance of the GraspCaps architecture, as well as compare the network's performance to an ablated version of the GraspCaps architecture that does not contain a capsule module in order to accurately evaluate any effect the capsule module might have on the performance of the network.

5.1 Training the network

The entire network is written in Python 3.7.4 and uses the PyTorch [54] version 1.6.0 library for all neural network implementations. All networks are written and optimized for training on Nvidia CUDA GPUs. We have implemented automated mixed precision [55] to increase the speed and efficiency of the training process. Automated mixed precision allows the CUDA cores on the GPU to use half-precision (16 bit) floats instead of the regular (32 bit) floats in areas where the drop in precision does not influence the accuracy of the network. Allowing automated mixed precision can significantly decrease both memory requirements and training time, as many operations such as linear layers and convolutional layers perform much faster with half-precision floats without any significant losses in terms of accuracy. This has allowed us to create larger networks than we would otherwise be able to build, as well as use a larger batch size during training.

A batch size of 16 samples per iteration was used to train both networks. The networks are optimized using the ADAM optimizer [36] using a learning rate of 0.00001 and default initialization parameters as set in PyTorch 1.6.0. As has been done for determining the parameters of the network as well, a multitude of different learning rates and optimizers were explored by use of parameter sweeps. From these experiments we found that the combination of ADAM and a constant learning rate of 0.00001 performed optimally.

The network is trained on the Peregrine high performance cluster on a computing node using a single Nvidia V100 Tensor core GPU[2] for a total of 500 epochs ($\approx$ 5 hours real time) on synthetic data for the simulation experiments.

The network weights are frozen after 300 epochs in order to function as a pretrained network for the real robot experiments, for which the network is trained an additional 500 epochs on real world data.

5.2 Results on dataset

To evaluate the initial performance of the GraspCaps network, we train it on the datasets specified in Section 4.3. Both the GraspCaps network and a similar network architecture without capsules (ablated network) will first be trained for 300 epochs on the synthetic recognition dataset. They will then have their weights in the feature extraction and classification modules frozen before being trained for an additional 200 epochs on the synthetic grasping dataset for fine-tuning the grasp synthesis section of the networks.

Initially it was planned to train the network end-to-end on a single dataset, but during evaluation of this approach we found that this either resulted in a lower classification performance due to overfitting on point clouds that were presented to the network multiple times per epoch; or in poor grasp performance when trained solely on the synthetic recognition dataset. These considerations led us to decide that training the network in two separate phases would likely result in superior performance.

[2] https://www.nvidia.com/en-gb/data-center/v100/

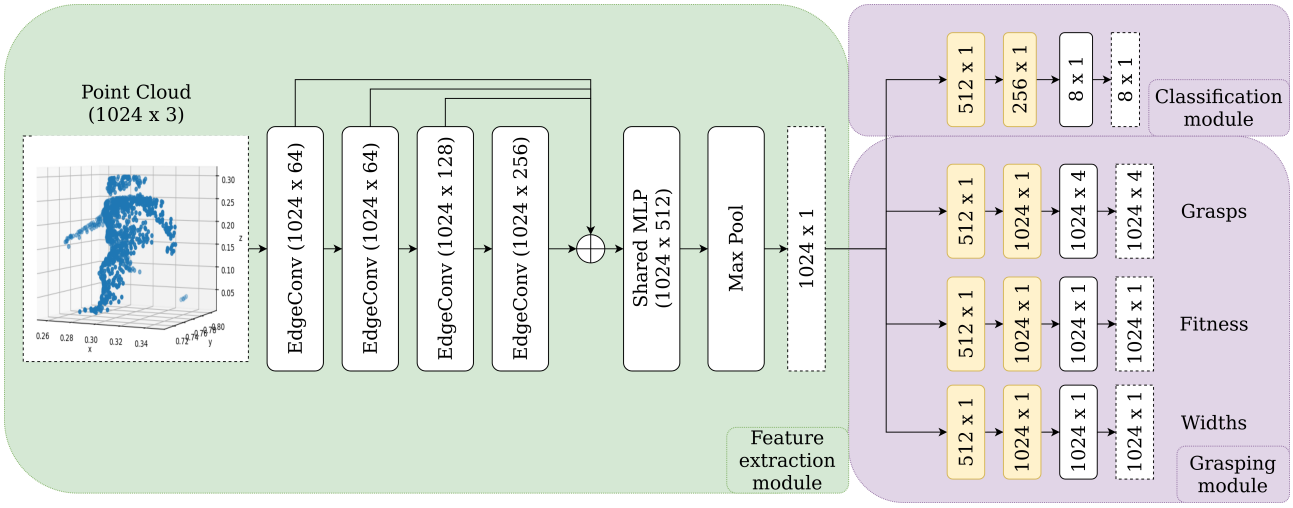

Figure 5.1: Ablated network used for determining the influence of the capsule module in the GraspCaps architecture.

By removing the capsule module from the comparison network we effectively perform an ablation study on the capsule network which should give us a clear indication of whether there is any value added when implementing capsule networks into a grasp synthesis network. The network architecture of the ablated network can be found in Figure 5.1. As the ablated network does not have a capsule layer, it cannot be trained using margin loss. To train the ablated network, we define a new loss function which replaces the margin loss term with a more suitable binary cross-entropy loss with logits term. This operation works well for binary classification problems and should therefore be a good approach to training the recognition head of the ablated network. Apart from replacing the margin loss, the exact same loss function is used as defined in Equation 14.

For training on the real data, an additional third training stage is added. First, we take the network trained for 300 epochs on the synthetic one-pointcloud-one-grasp dataset. Then the network is trained on the one-pointcloud-one-grasp dataset gathered from the real world, and finally the weights of the feature extraction and capsule module are frozen and the grasping module is fine-tuned on the one-pointcloud-120-grasps dataset for 200 epochs.

Classification accuracy is measured by comparing the target classification with the classification output by the network. Grasping accuracy is measured by comparing the generated fitness value with the target fitness using the function:

$$\text{fit}(\mathbf{y}, \hat{\mathbf{y}}) = 1.0 - |\mathbf{y} - \hat{\mathbf{y}}| \tag{15}$$

where $\mathbf{y}$ is the target fitness, and $\hat{\mathbf{y}}$ is the fitness generated by the network. As both the target fitness and generated fitness are values in the range [0, 1.0], the function will equal 1.0 when the generated fitness is equal to the target fitness, and equal 0.0 when the fitness values are completely opposed.

The results of training on the synthetic dataset can be found in Figure 5.2. Results on the real dataset can be found in Figure 5.3. All plots are smoothed using a running average over the last 20 epochs for improved legibility.

The swift increase in grasp accuracy shows that both networks quickly converge to satisfactory performance. The most interesting part of these results is comparing the grasping accuracy on the synthetic dataset with the results obtained on the real-world dataset; in the synthetic dataset, both networks converge to approximately 94% accuracy, with GraspCaps outperforming the ablated network with less than 1% difference. When looking at the real-world dataset results however, the difference

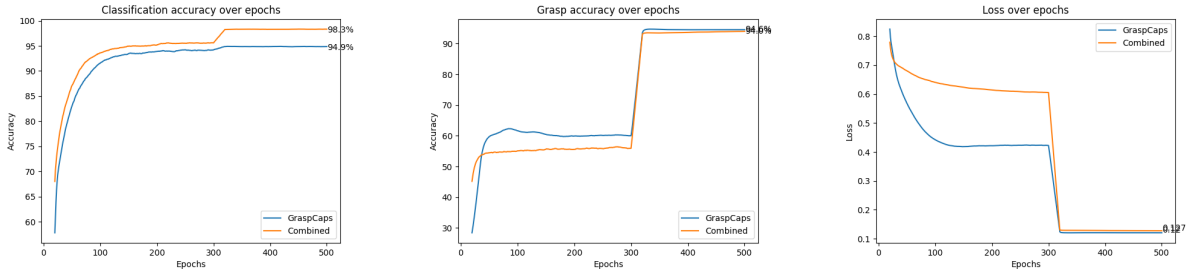

Figure 5.2: Results achieved by training on the synthetic dataset. Left: classification accuracy over epochs. Middle: grasp accuracy over epochs. Right: loss over epochs.

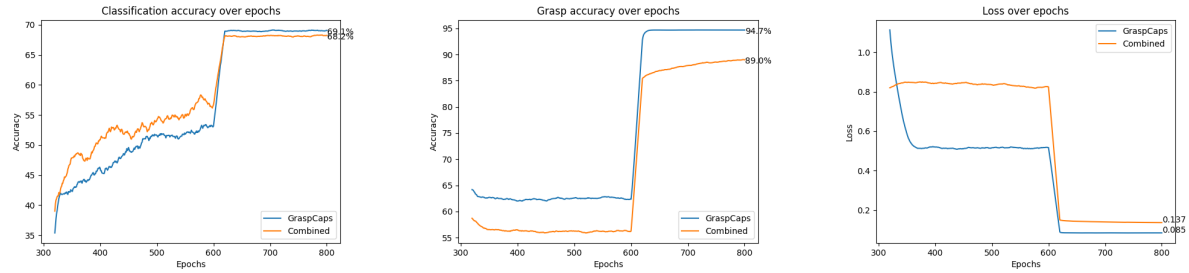

Figure 5.3: Results achieved by training on the real dataset. Left: classification accuracy over epochs. Middle: grasp accuracy over epochs. Right: loss over epochs.

is much more pronounced. Again, GraspCaps achieves an accuracy of 94% while the ablated network converges to a significantly lower accuracy of 89%. This is a strong indication that the GraspCaps architecture has a higher generalization ability than the ablated network. The real dataset is more noisy than the synthetic dataset, so the results indicate that the GraspCaps architecture is more resistant to varying input data and noise than the ablated network, leading to superior performance in the real world.

5.3 Experimental setup

The experimental setup consists of a single Mirosoft Kinect camera and two Universal Robot 5e arms[3] equipped with Robotiq 2F-140 grippers[4] as end effectors. The Kinect camera is situated in the center at a height of 80 centimeters in the z direction, with a pitch of 0.8π (≈ 45.8 degrees). The arms are situated at the left and right sides of the camera at a distance of 25 centimeters from the center in the x direction. The setup is visualized in Figure 5.4.

All experiments in Gazebo were performed on a laptop equipped with an 7th generation Intel core i7 processor and Nvidia GTX 1060 mobile GPU running Ubuntu 18.04. We use ROS Melodic for all of the communication between our code and the robotic platforms simulated in the Gazebo simulator. A more detailed breakdown of the code architecture can be found in Section 4.7.

5.4 Pre-processing and post-processing

To increase the robustness of the performance of the networks during the experiments, we employ several pre- and post-processing steps. For the pre-processing, we take the raw point cloud given by

[3] https://www.universal-robots.com/products/ur5-robot/

[4] https://robotiq.com/products/2f85-140-adaptive-robot-gripper

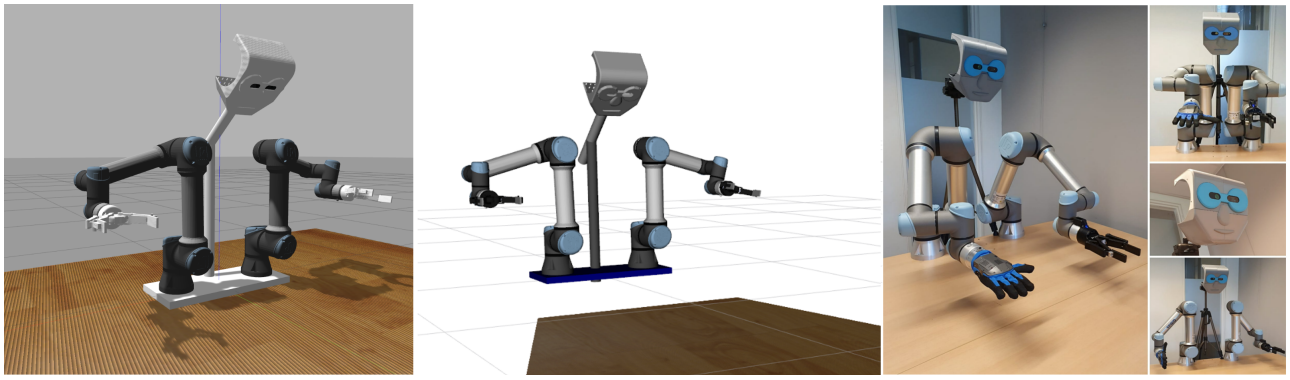

Figure 5.4: Left: The setup using the two UR5e arms and Kinect in the Gazebo simulator. Middle: The simulation setup visualized along with Kinect output in RViz visualization software. Right: The setup with the real life robot.

the Kinect camera and remove the floor using a random sample consensus (RANSAC) operation [56] that removes the most prominent plane in the point cloud, which will always be the floor in our experimental setup. After the RANSAC operation each point with a y value over 0.9 is removed from the point cloud to filter out any points that are part of the basket (see Figure 5.6 for a visualization of the task setup). Finally for each point a check is performed to determine whether it is part of the object or if it might be a faulty point due to sensor noise. We do this by checking if there are at least 300 other points in a 20cm radius from that point. During prior testing of the setup this has shown to be a robust method of removing outliers and faulty points, especially in real-world settings where sensor noise can be a significant issue.

The remaining point cloud, which is often far larger than the 1024 points required for the neural network to function, is sampled to obtain seven random permutations of 1024 points. During classification we present all seven point clouds to the neural network in order to have multiple representations of the object. After the seven point clouds are classified a mode operation is used to determine which classification occurred most often in the output of the network. A mode operation is an often used statistical operation which checks for the most occurring value in its input and then returns that value. In the event of a tie the algorithm is re-run until consensus is reached. By taking several subsets of the point cloud of the object, we significantly reduce the chance of misclassifications due to a single unlucky choice of points. In prior experiments we have found that this procedure significantly decreased the frequency of misclassifications during the object recognition and grasping task.

For the post-processing, the output of the fitness head of the network is smoothed using a custom implemented Gaussian filter. The filter works by determining the 100 closest neighbours of each point using a k-NN operation, and then smoothing the fitness value of each point using the fitness values of its neighbours. By smoothing the fitness values spurious errors in the generated fitness values are reduced in severity, and regions with high or low fitness are more easily identified. An example of the input and output of this smoothing operation can be found in Figures **??** and **??**, in which the generated fitness of a bottle is smoothed.

After smoothing, the point with the highest fitness value should lay in the center of the region with the highest overall fitness. The algorithm then chooses this point to center the grasp around. The grasp corresponding to this point is then chosen from the grasp head and executed.

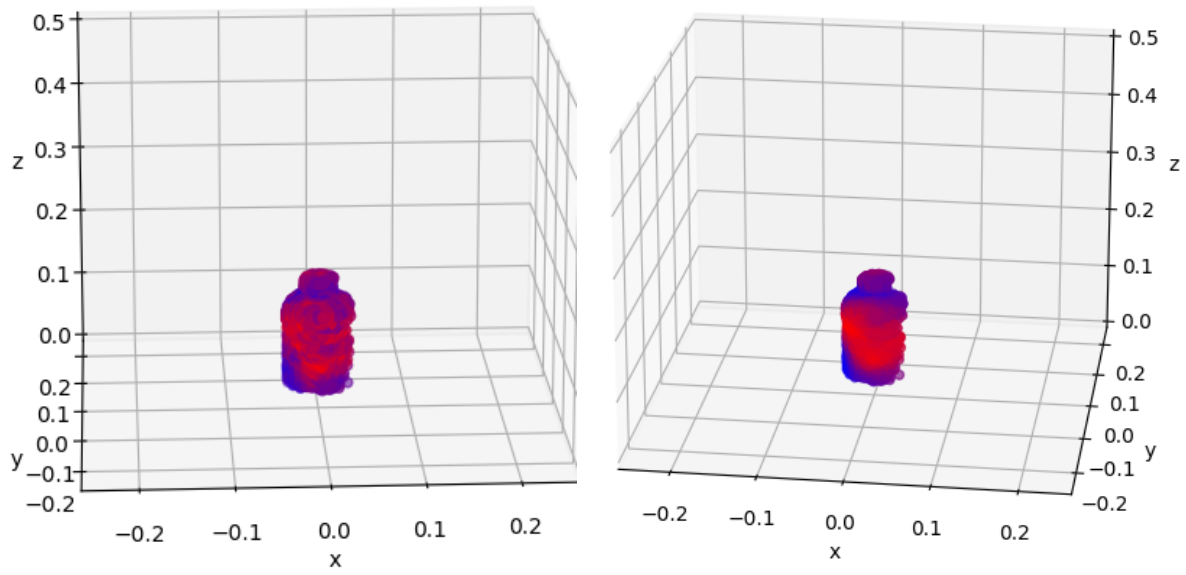

Figure 5.5: Left: Raw output of the fitness head of the GraspCaps network on an object from the vertical cylinder class. Right: The output when smoothed using our Gaussian filter.

5.5 Results in Gazebo

5.5.1 Task description

The task of picking up the object and dropping it in the basket can be divided into two distinct phases: the planning phase and the execution phase.

First, the grasp and pre-grasp positions and rotations need to be determined: the grasp position is chosen by processing the input point cloud as described earlier in this section. This process obtains the position of a single point $\mathbf{p} = (x, y, z)$, and a quaternion rotation $\mathbf{q} = (a, b, c, d)$. The point $\mathbf{p}$ is obtained by determining which point in the point cloud has the highest fitness value. The rotation $\mathbf{q}$ is the rotation vector generated by the neural network that corresponds to that point.

We then apply an offset to this position to account for the length of the fingers in order to obtain a usable grasping position $\mathbf{p}_{\text{grasp}}$ using:

$$\mathbf{p}_{\text{grasp}} = \mathbf{p} - \mathbf{q} \cdot \mathbf{o}_{\text{finger}} \tag{16}$$

where $\mathbf{o}_{\text{finger}} = (0.15, 0, 0)$ is a predefined offset equal to the length from the wrist of the arm to the space between the two fingertips. Adding an offset to the initial position is needed, as all planning algorithms used in this project plan using the location of the wrist as the end-effector instead of the tip of the gripper. Were we to skip this step the gripper would most certainly collide with the object and fail to grasp anything. In a similar fashion we define the pre-grasp pose $\mathbf{p}_{\text{pre-grasp}}$ the arm will move to before moving to the grasp position $\mathbf{p}_{\text{grasp}}$:

$$\mathbf{p}_{\text{pre-grasp}} = \mathbf{p}_{\text{grasp}} - \mathbf{q} \cdot \mathbf{o}_{\text{pre-grasp}} \tag{17}$$

where $\mathbf{o}_{\text{pre-grasp}} = (0.2, 0, 0)$ is a pre-defined offset, and $\mathbf{p}_{\text{pre-grasp}}$ is a position in 3D space translated 20cm from $\mathbf{p}_{\text{grasp}}$ in the orientation defined by $\mathbf{q}$. The orientation the gripper needs to have in the pre-grasp position is equal to the orientation it uses for the grasp position. Visualizations of multiple generated pre-grasp and grasp positions can be found in the Appendix, Figure 5.7.

By defining the pre-grasp position dynamically based on the grasping orientation **q** we ensure that the pre-grasp position is perfectly aligned with the grasping pose, which allows the grasping pose to be reached with a single forward motion in the direction of the grasping orientation. This enables the system to accurately generate a pre-grasp position that is aligned with the grasp position regardless of the grasp orientation, enabling the system to grasp objects in a wide variety of configurations, including top-down and sideways grasps without the need for any static pre-determined approach vectors.

It is important to note that **q** is a directed rotation vector which always points towards the grasping direction (i.e. towards the object), so by applying the calculations above we always obtain a pre-grasp position that is exactly 20cm removed from the object.

Second, after the grasp and pre-grasp positions have been determined, the arm moves to the pre-grasp position using an inverse kinematics planner as explained in Section 3.11.5. If the arm arrives successfully it will move to the grasp position, where it will attempt to close its fingers to grasp the object. After grasping, the arm will move to the post-grasp position, which is located 20 cm above the grasp position. We have chosen to define the post-grasp position as a static addition of 20 cm in the z direction to the grasp position instead of a more dynamic approach as was used for the pre-grasp, as this ensures that the object will not be dragged over the surface it was standing on, but instead will be moved upward in a straight line. As the object is already grasped by the gripper we do not have to take the grasp orientation into account, as there is no chance that any further movement by the gripper can cause the grasp to fail due to collision with the object.

The arm then moves to a predefined drop location above the basket where it will open its fingers, releasing the object into the basket. Finally, the arm retreats back to its initial position as shown in Figure 5.6.

If at any point during the execution of this behaviour the arm is unable to move to one of the locations, it will open its gripper to release the object if it is holding it, and then return to its initial position. The grasping attempt will be marked as a failure in this case.

5.5.2 Experiment description

To test the grasping capabilities of the GraspCaps architecture in the Gazebo simulation environment and to answer our research question of whether it is possible to synthesise usable grasp configurations using capsule networks, we presented 10 objects from each of the 8 classes to both the GraspCaps architecture and the ablated comparison network (see Figure 5.1).

The networks will have to accurately determine the class of the object and generate a grasp configuration that is able to successfully pick up the object. The object should then be dropped in a basket that is placed in front of the robot. A visualization of the task setup can be seen in Figure 5.6. For each of the trials, the networks will get one attempt to classify the object and grasp it. If the grasp fails during that attempt, the object will be removed from the simulation and a new object will be placed with random location and orientation.

Both networks will be evaluated on classification, as well as on grasping success. Classification success will be determined by the system itself by comparing the ground truth class label of the spawned object with the class label generated by the network. Grasping success will be determined by a human supervisor to ensure that faulty grasps cannot be registered as successful.

An implementation of the GraspCaps network, along with trained models and code for generating data can be found on GitHub[5].

[5] https://github.com/Tomasvdv/master_project

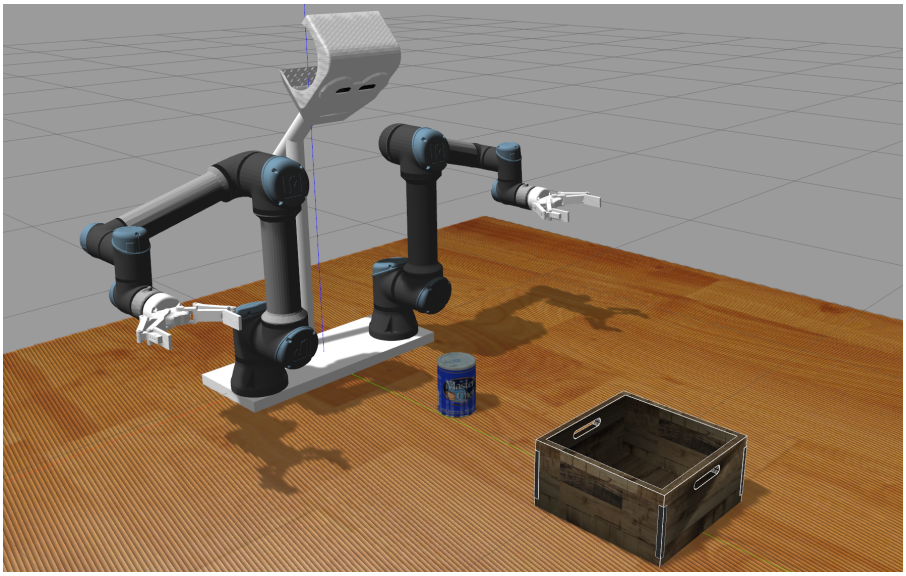

Figure 5.6: Visualization of the task setup in Gazebo containing the robotic platform, a graspable object (master chef can), and a basket used for dropping the object into.

5.5.3 Experiment results

We found that, although both the GraspCaps network and the comparison network achieved an equal classification performance of 80%, the GraspCaps network outperformed the ablated comparison network in terms of grasping success with percentages of 71% for the GraspCaps network versus 53% for the ablated network. See Tables 5.1 and 5.2 for for a per-class breakdown of the achieved classification and grasping accuracy of the two networks. A short video demonstrating the performance of the GraspCaps network can be found on YouTube[6].

When looking at the performance of the GraspCaps architecture in Table 5.1, it can be seen that the cutlery class is a clear negative outlier in terms of both classification accuracy (30% vs. the average accuracy of 86%) and grasping success (50% vs. the average success of 71%). When looking at the confusion matrix of the GraspCaps classifications in Figure 5.8, we observe that all misclassifications of the cutlery class were classified as vertical objects. These are often grasped in a sideways fashion, as opposed to the top-down grasps that should be employed when grasping objects from the cutlery or horizontal cylinder classes, which explains the poor grasping performance.

5.5.4 Results on novel objects

To test the generalizability of the network, we tested the network performance on a set of seven novel objects that are not present in the dataset. The objects are shown in Figure 5.9 and are taken from the 3DGEMS dataset [57].

A number of the novel objects, such as the glass bottle and the tea and tissue boxes, can be easily sorted into the existing eight classes the network was trained on. Other objects however, such as the computer mouse and digital clock, have more abstract shapes and may be hard to sort into the existing classes. For these objects specifically it will be very interesting to see how the network will adapt and if it will be able to successfully grasp the objects.

Both the GraspCaps architecture and the ablated architecture will attempt to grasp each of the

[6]https://youtu.be/89A1gzKJGxI

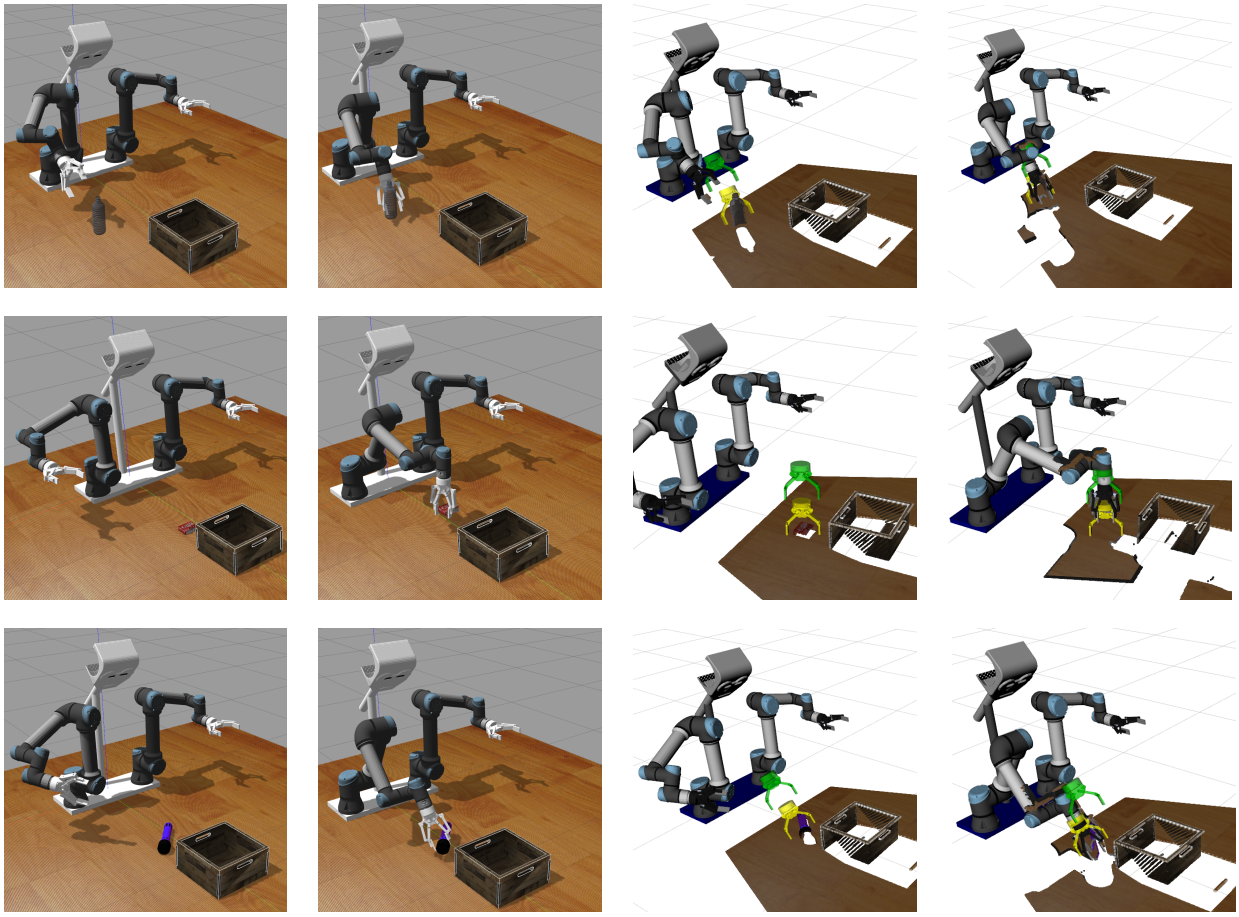

Figure 5.7: Grasps generated by the GraspCaps network on three distinctly different objects as visualized in Gazebo (first two columns) and RViz (final two columns). Images in the first and third column are taken during the grasp approach; images in the second and fourth column are taken during the post-grasp retreat. In the third column, pre-grasp positions are given in green, and grasp positions are given in yellow. In the fourth column, the grasp position is given in yellow, and the post-grasp pose is given in green. From top to bottom, the objects being grasped are: a water bottle, a small box, and a flashlight.

objects five times to get a clear indication of the generalization capabilities of both networks. The experimental setup and task will be equal to the setup in Gazebo described earlier this section.

When comparing the results of the GraspCaps network shown in Figure 5.3 with the results of the ablated network in Figure 5.4 it can be seen that the GraspCaps network outperforms the ablated network with an average of 71% grasping accuracy for the GraspCaps architecture compared to an average of 57% for the ablated network.

During the experiments on grasping the tissue and tea boxes, both networks generated grasp configurations that pierced the box with one of the gripper fingers, which caused the grasp to fail. The grasps that were generated on the boxes were in most cases correctly oriented, with the grippers parallel to the sides of the boxes. The generated grasps piercing the tissue box make sense, as there is a significant opening in the center that could fit one of the fingers of the gripper, which could have been picked up by the network.

An explanation for the tea box is harder to find. One possibility is that, since the tea box is rather large and has equally sized sides, both networks were unable to find grasp configurations that did not collide with the box. Adding large cubic objects to the dataset could potentially solve this problem.

Class	Classification Accuracy	Grasping Success
Bowl	100%	60%
V Cylinder	90%	80%
H Cylinder	100%	60%
Box	80%	80%
Ball	100%	90%
Cutlery	30%	50%
V Obj	100%	70%
Hardware	90%	80%
Average	86%	71%

Table 5.1: Performance of the GraspCaps network in the Gazebo grasping task.

Class	Classification Accuracy	Grasping Success
Bowl	100%	60%
V Cylinder	100%	60%
H Cylinder	100%	40%
Box	100%	20%
Ball	100%	70%
Cutlery	50%	40%
V Obj	40%	60%
Hardware	100%	70%
Average	86%	53%

Table 5.2: Performance of the ablated network in the Gazebo grasping task.

5.5.5 Results on the real robot

The GraspCaps network has been tested in a real world setting and has been found to perform well on objects that exist in its dataset, as well as on a number of novel objects.

The network was able to accurately pick up and place objects in the basket in a pile scenario, where a number of objects were placed on a table with random rotation and significant overlap, causing a number of objects to be partially or fully obscured from view. See Figure 5.10 for an example setup of a pile scenario. The network successfully picked up the objects in the pile with correct orientation of the gripper and was able to drop objects of multiple shapes into the basket without collision. A short video demonstrating the performance of the GraspCaps network in both isolated object scenarios and a pile scenario is available on YouTube[7]. The video contains both successful and unsuccessful grasps, as well as novel objects that were not present in the training data.

[7] https://youtu.be/KXCu4RFdvo

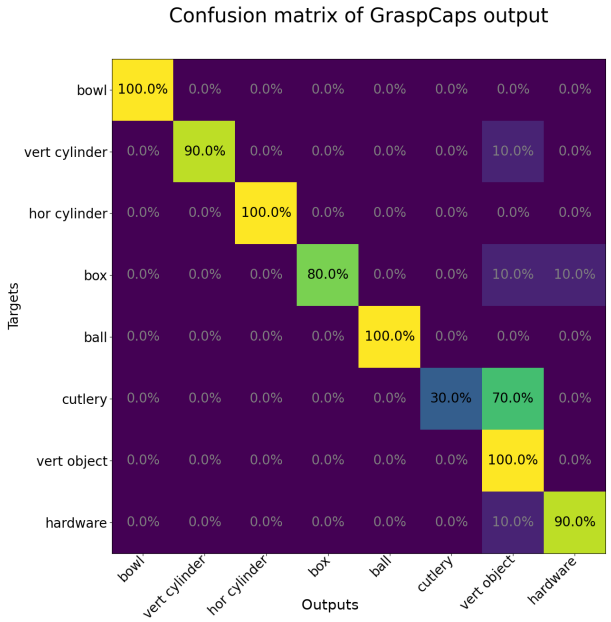

Figure 5.8: Confusion matrix of the classifications given by the GraspCaps network in the Gazebo grasping task.

6 Conclusion

The GraspCaps architecture was able to outperform the ablated architecture in generating grasp configurations both on objects that were in the dataset (71% vs. 53%), and on novel objects neither of the networks had seen during training (71% vs. 57%). It also showed good performance during experiments in a real-world setting. Based on these findings, we conclude that there is significant merit in using capsule networks in a grasping task and that the addition of a capsule module in a grasping architecture could increase both the performance and robustness of such an architecture.

6.1 Answer to the research question

To answer the research question posed in Section 1.4: *"Is is possible to design a neural network architecture based on capsule networks that is able to accurately generate grasps for a multitude of familiar object classes?"* we conclude that, by testing the performance of the GraspCaps architecture in several experiments, we have shown that it is possible to design an effective grasp synthesis architecture using capsule networks.

By comparing the performance of the GraspCaps architecture with the ablated architecture we can also confidently answer the second research question: *"Is there significant merit in including capsule networks in a grasping architecture?"*, as we have found that the GraspCaps architecture outperformed the ablated architecture during all performed experiments. Therefore, we conclude that there is significant merit in employing capsule networks in a grasping architecture.

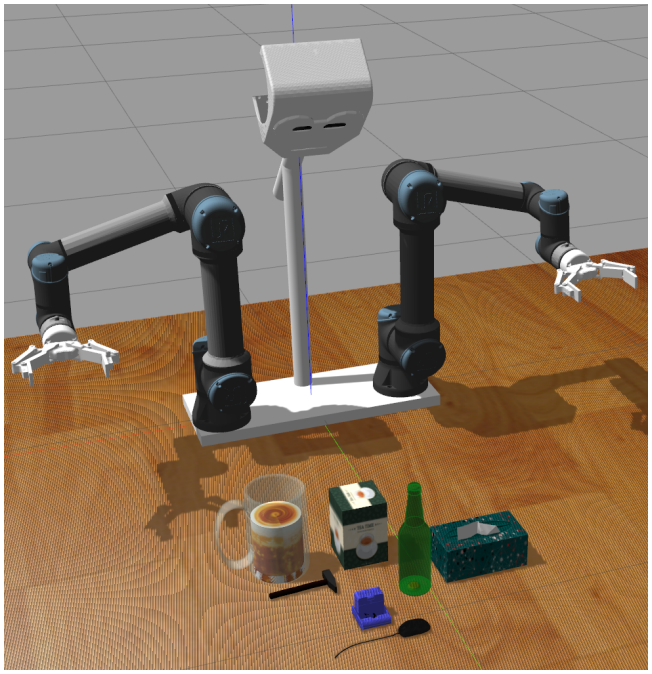

Figure 5.9: Set of novel objects shown in the Gazebo Simulator.

7 Discussion

In this thesis we have introduced GraspCaps, a novel neural network architecture that is able to synthesise per-point grasp configurations based on point cloud input. In our experiments we have shown that our network can successfully generate grasp configurations and have thereby answered our research question whether capsule networks are a viable option for creating grasping architectures.

The project changed directions a number of times. Initially the plan was to focus on synchronous grasping using a two-armed robot, but gradually that changed into creating a grasping network using point clouds as its input. The addition of capsule networks came both from an interest in exploring the effectiveness of unconventional network architecture, and the intuition that capsule networks might be a perfect fit for a grasp generation network, since they have the unique characteristic of extracting the instantiation parameters of the input data.

We also changed from defining the dataset from 90 individual object classes to train the network on simultaneous recognition and grasping, to separating the dataset into 8 distinct shape classes and redefining the task to be on familiar object grasping. The change seems to make logical sense, as many of the objects have very similar shapes, so the feature vectors extracted from the capsule network would also be very similar, resulting in a large amount of redundant data and an unoptimized network.

7.1 Future work

Over the course of this thesis we have shown that capsule networks are an effective approach to object-aware grasp generation. This is an early exploratory study however, and much more work can be done to improve on the premises we set during this project. In this section we will discuss possible avenues for future research using the research performed in this thesis.

Object	Classification		Grasping success	Notes
Bottle	V Cylinder	5/5 (100%)	4/5 (80%)	
Clock	Hardware	2/5 (40%)	5/5 (100%)	
	Box	2/5 (40%)		
	V Object	1/5 (20%)		
Mouse	Box	5/5 (100%)	5/5 (100%)	
Beer Mug	V Object	3/5 (60%)	4/5 (80%)	
	V Cylinder	1/5 (20%)		
	Hardware	1/5 (20%)		
Tea box	Box	5/5 (100%)	1/5 (20%)	3x pierced
Tissue box	Hardware	3/5 (60%)	2/5 (40%)	3x pierced
	Box	2/5 (40%)		
Hammer	H Cylinder	5/5 (100%)	4/5 (80%)	
Average			25/35 (71%)	

Table 5.3: Performance of the GraspCaps network on novel objects.

7.1.1 Affordance segmentation

For future research it would be interesting to extend the network architecture with an additional head in the grasping module that generates an affordance mask, such that the network can be trained to grasp objects only at specific points, like the handles of a pan or the handle of a knife.

Using the current architecture of GraspCaps it would be straightforward to implement and train. The implementation of an affordance head could be similar to the other heads in the Grasping module, consisting of 3 fully connected layers using leaky ReLU activation and outputting a 1024×3 affordance mask for per-point affordance segmentation. To include an affordance head in the training of the network, it would require a loss term to be added to the total loss in Equation 14, potentially using a scaling parameter for tuning the degree to which the loss contributes to the total loss.

The affordance segmentation output could be implemented in the functioning of the network by using the affordance segmentation as a mask. First the network checks which points it is allowed to grasp as determined by the affordance mask. Second, it proceeds as normal, but discards any grasp configuration on points that the affordance mask determined as not graspable.

The idea of including affordance segmentation in the GraspCaps architecture was conceived in the early stages of this project, but ultimately scrapped due to lack of a usable dataset.

A dataset used for affordance segmentation should have a clear delineation of the exact regions where an object should be grasped, and where it should not be grasped. By training on a dataset such as that, a network could learn exactly where it should grasp an object and which regions to avoid if possible.

To elaborate on this thought, it would be very interesting to train a network using an affordance dataset that has ternaries as target values. Ternaries have three values [-1, 0, 1] as opposed to binaries, which only have two values [0, 1] and are more commonly used for segmentation tasks. By using a ternary, we could make a distinction between areas that are favorable to grasp, e.g. the handle of a knife (target value 1); areas that are less desired, but not actively harmful, e.g. the dull side of the blade of a knife (target value 0); and finally regions that we want to actively avoid, e.g. the sharp

Object	Classification		Grasping success	Notes
Bottle	V Cylinder	3/5 (60%)	3/5 (60%)	
	V Object	2/5 (40%)		
Clock	Hardware	3/5 (60%)	5/5 (100%)	
	Box	1/5 (20%)		
	V Object	1/5 (20%)		
Mouse	Box	2/5 (40%)	5/5 (100%)	
	V Object	2/5 (40%)		
	Box	1/5 (20%)		
Beer Mug	V Cylinder	3/5 (60%)	1/5 (20%)	
	V Object	1/5 (20%)		
	Hardware	1/5 (20%)		
Tea box	Box	5/5 (100%)	1/5 (20%)	3x pierced
Tissue box	Box	4/5 (80%)	2/5 (40%)	2x pierced
	Hardware	1/5 (20%)		
Hammer	H Cylinder	5/5 (100%)	3/5 (60%)	
Average			20/35 (57%)	

Table 5.4: Performance of the ablated network on novel objects.

side of the blade (target value -1). As an added benefit, this would also result in a richer and more informative error space while training the network, as the dataset now contains more intrinsic data on the objects it contains.

During the project we were unable to find a suitable dataset that could be used for training the network to perform affordance segmentation.

An alternative to using an existing dataset was to create a new dataset ourselves. This would have the upside of allowing us to use the same objects for both creating the affordance dataset and performing the experiments, which would result in excellent representative data during the experiments, as there is no loss in accuracy due to the objects in the dataset not corresponding to the objects used in the experiments. To this end we did find some resources that might have been of use for generating our own dataset, such as the LabelMe tool [58]. This is commonly used for image segmentation tasks, but also includes functionality for point cloud segmentation.

The amount of time and work it would take to construct a full segmentation dataset is quite significant, and after serious consideration we deemed it was out of the scope of this project and is therefore left for future research.

7.1.2 Bigger datasets

It would be interesting to train the network on bigger datasets and compare it to more established network architectures. Due to the niche this network exists in (familiar object recognition and grasping using point cloud data), no datasets were found that offered a combination of point clouds or depth images as input, and object labels and 6D grasp targets as output. Using per-point grasp generation further increased the requirements of a suitable dataset. Instead we opted to create our own dataset,

which worked very well for the purposes of this research, but does not allow the one-on-one comparisons or benchmark scores needed to accurately and objectively evaluate network architectures.

We hope that the dataset and grasp generation algorithm we have created for this thesis might serve as a starting point for creating such a benchmark.

7.1.3 Skipped layer connections

In a large number of currently popular network architectures skipped layer connections are utilized in order to allow deeper networks to train without the risk of suffering from the degradation problem [59].

The degradation problem can occur when a network of significant depth is trained without taking prior precautions to eliminate the gradient from saturating. With the network depth increasing, the accuracy of the network becomes saturated and starts degrading soon afterwards. This degradation is not caused by overfitting, and adding additional layers results in a higher training error [59], while in an over-fitted network the training error should only ever go down.

Adding skipped layer connections to the feature extraction module in the GraspCaps architecture might have a positive effect on the performance of the network, as it could allow more positional data of the points to flow through to the grasping module, which might improve identifying regions that are most suitable for grasping. It also allows for a deeper feature extraction network to be trained.

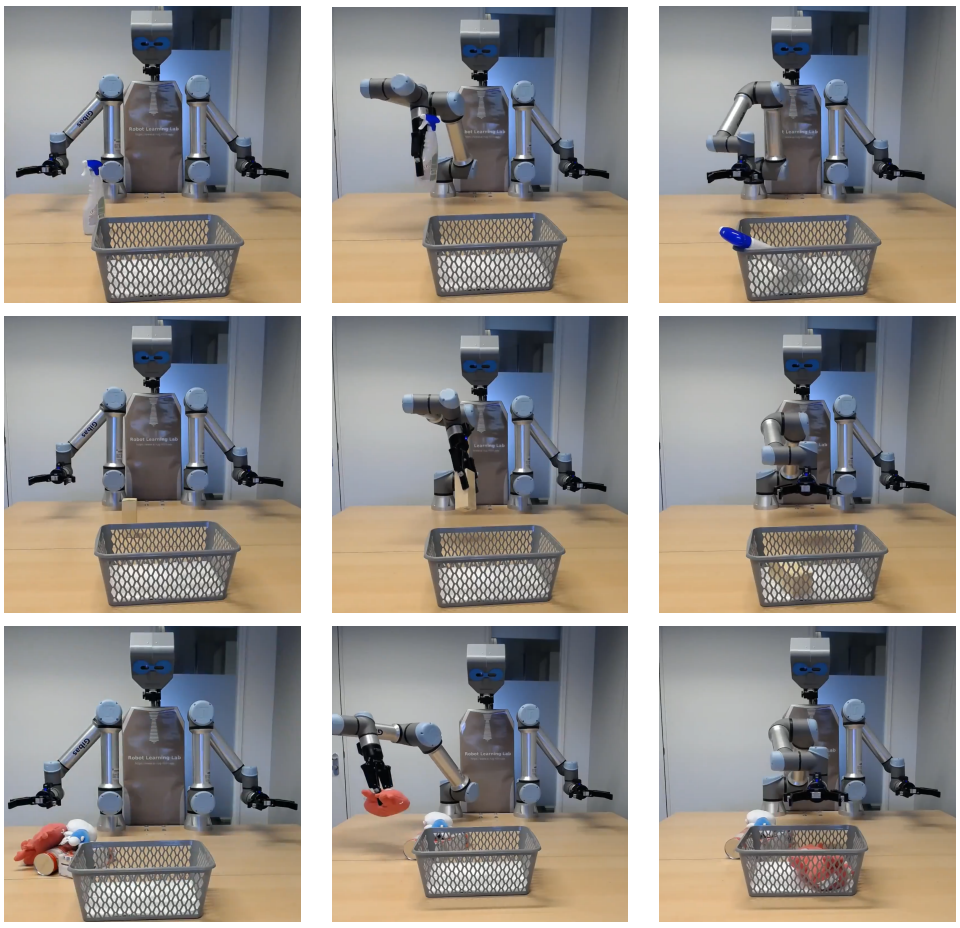

Figure 5.10: Top: Performance on grasping a spray bottle. Middle: Performance on grasping a box. Bottom: Performance on a Pile scenario.

A second possibility is to concatenate the feature vector generated by the feature extraction module and append it to the feature vector generated by the capsule module, effectively creating a skipped layer connection that skips the entire capsule module. This would allow non-object specific features to be propagated through the grasping module, which could potentially have a positive effect on the grasping performance when grasping objects of a class that does not have a large amount of data. This would be an especially interesting avenue to explore in the event that the network is used on a larger number of classes with fewer samples per class to train on.

Bibliography

[1] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph cnn for learning on point clouds," *Acm Transactions On Graphics (tog)*, vol. 38, no. 5, pp. 1–12, 2019.

[2] G. Du, K. Wang, S. Lian, and K. Zhao, "Vision-based robotic grasping from object localization, object pose estimation to grasp estimation for parallel grippers: a review," *Artificial Intelligence Review*, vol. 54, no. 3, pp. 1677–1734, 2021.

[3] S. H. Kasaei, M. Oliveira, G. H. Lim, L. S. Lopes, and A. M. Tomé, "Towards lifelong assistive robotics: A tight coupling between object perception and manipulation," *Neurocomputing*, vol. 291, pp. 151–166, 2018.

[4] S. H. Kasaei, N. Shafii, L. S. Lopes, and A. M. Tomé, "Interactive open-ended object, affordance and grasp learning for robotic manipulation," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 3747–3753, IEEE, 2019.

[5] N. Shafii, S. H. Kasaei, and L. S. 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)*, pp. 2895–2900, IEEE, 2016.

[6] C. R. Qi, H. Su, K. Mo, and L. 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*, pp. 652–660, 2017.

[7] O. Vinyals, S. Bengio, and M. Kudlur, "Order matters: Sequence to sequence for sets," *arXiv preprint arXiv:1511.06391*, 2015.

[8] C. R. Qi, L. Yi, H. Su, and L. J. Guibas, "Pointnet++: Deep hierarchical feature learning on point sets in a metric space," *arXiv preprint arXiv:1706.02413*, 2017.

[9] R. Klokov and V. Lempitsky, "Escape from cells: Deep kd-networks for the recognition of 3d point cloud models," in *Proceedings of the IEEE International Conference on Computer Vision*, pp. 863–872, 2017.

[10] H. Zhao, L. Jiang, C.-W. Fu, and J. Jia, "Pointweb: Enhancing local neighborhood features for point cloud processing," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 5565–5573, 2019.

[11] Y. Li, R. Bu, M. Sun, W. Wu, X. Di, and B. Chen, "Pointcnn: Convolution on x-transformed points," *Advances in neural information processing systems*, vol. 31, pp. 820–830, 2018.

[12] W. Shi and R. Rajkumar, "Point-gnn: Graph neural network for 3d object detection in a point cloud," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 1711–1719, 2020.

[13] A. Cheraghian and L. Petersson, "3dcapsule: Extending the capsule architecture to classify 3d point clouds," in *2019 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pp. 1194–1202, IEEE, 2019.

[14] G. Riegler, A. Osman Ulusoy, and A. Geiger, "Octnet: Learning deep 3d representations at high resolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3577–3586, 2017.

[15] D. Maturana and S. Scherer, "Voxnet: A 3d convolutional neural network for real-time object recognition," in *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 922–928, IEEE, 2015.

[16] M. Breyer, J. J. Chung, L. Ott, R. Siegwart, and J. Nieto, "Volumetric grasping network: Real-time 6 dof grasp detection in clutter," *arXiv preprint arXiv:2101.01132*, 2021.

[17] T. Le and Y. Duan, "Pointgrid: A deep network for 3d shape understanding," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 9204–9214, 2018.

[18] C. Xiao, N. Madapana, and J. Wachs, "Fingers see things differently (fist-d): An object aware visualization and manipulation framework based on tactile observations," *IEEE Robotics and Automation Letters*, vol. 6, no. 3, pp. 4249–4256, 2021.

[19] A. Yamaguchi and C. G. Atkeson, "Recent progress in tactile sensing and sensors for robotic manipulation: can we turn tactile sensing into vision?," *Advanced Robotics*, vol. 33, no. 14, pp. 661–673, 2019.

[20] A. ten Pas, M. Gualtieri, K. Saenko, and R. Platt, "Grasp pose detection in point clouds," *The International Journal of Robotics Research*, vol. 36, no. 13-14, pp. 1455–1473, 2017.

[21] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[22] H. Liang, X. Ma, S. Li, M. Görner, S. Tang, B. Fang, F. Sun, and J. Zhang, "Pointnetgpd: Detecting grasp configurations from point sets," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 3629–3635, IEEE, 2019.

[23] Y. Qin, R. Chen, H. Zhu, M. Song, J. Xu, and H. Su, "S4g: Amodal single-view single-shot se (3) grasp detection in cluttered scenes," in *Conference on robot learning*, pp. 53–65, PMLR, 2020.

[24] A. Mousavian, C. Eppner, and D. Fox, "6-dof graspnet: Variational grasp generation for object manipulation," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pp. 2901–2910, 2019.

[25] A. Dimitrov and M. Golparvar-Fard, "Segmentation of building point cloud models including detailed architectural/structural features and mep systems," *Automation in Construction*, vol. 51, pp. 32–45, 2015.

[26] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[27] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain.," *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[29] M.-C. Popescu, V. E. Balas, L. Perescu-Popescu, and N. Mastorakis, "Multilayer perceptron and neural networks," *WSEAS Transactions on Circuits and Systems*, vol. 8, no. 7, pp. 579–588, 2009.

[30] K. Gurney, *An introduction to neural networks*. CRC press, 1997.

[31] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Icml*, 2010.

[32] B. Xu, N. Wang, T. Chen, and M. Li, "Empirical evaluation of rectified activations in convolutional network," *arXiv preprint arXiv:1505.00853*, 2015.

[33] L. Lu, Y. Shin, Y. Su, and G. E. Karniadakis, "Dying relu and initialization: Theory and numerical examples," *arXiv preprint arXiv:1903.06733*, 2019.

[34] A. L. Maas, A. Y. Hannun, A. Y. Ng, *et al.*, "Rectifier nonlinearities improve neural network acoustic models," in *Proc. icml*, vol. 30, p. 3, Citeseer, 2013.

[35] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[37] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[38] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, *et al.*, "Recent advances in convolutional neural networks," *Pattern Recognition*, vol. 77, pp. 354–377, 2018.

[39] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE transactions on neural networks and learning systems*, vol. 32, no. 1, pp. 4–24, 2020.

[40] Z. Xinyi and L. Chen, "Capsule graph neural network," in *International conference on learning representations*, 2018.

[41] E. Fix and J. L. Hodges, "Discriminatory analysis. nonparametric discrimination: Consistency properties," *Report*, 1951.

[42] S. Sabour, N. Frosst, and G. E. Hinton, "Dynamic routing between capsules," *arXiv preprint arXiv:1710.09829*, 2017.

[43] G. E. Hinton, A. Krizhevsky, and S. D. Wang, "Transforming auto-encoders," in *International conference on artificial neural networks*, pp. 44–51, Springer, 2011.

[44] G. E. Hinton, S. Sabour, and N. Frosst, "Matrix capsules with em routing," in *International conference on learning representations*, 2018.

[45] Y. Zhao, T. Birdal, H. Deng, and F. Tombari, "3d point capsule networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1009–1018, 2019.

[46] J. Tian, H. Xie, S. Hu, and J. Liu, "Multidimensional face representation in a deep convolutional neural network reveals the mechanism underlying ai racism," *Frontiers in computational neuroscience*, vol. 15, p. 17, 2021.

[47] S. Hamidreza Kasaei, J. Melsen, F. van Beers, C. Steenkist, and K. Voncina, "The state of lifelong learning in service robots: Current bottlenecks in object perception and manipulation," *arXiv e-prints*, pp. arXiv–2003, 2020.

[48] A. Goldenberg, B. Benhabib, and R. Fenton, "A complete generalized solution to the inverse kinematics of robots," *IEEE Journal on Robotics and Automation*, vol. 1, no. 1, pp. 14–20, 1985.

[49] S. Kucuk and Z. Bingul, *Robot kinematics: Forward and inverse kinematics*. INTECH Open Access Publisher London, UK, 2006.

[50] H. Kasaei and S. Xiong, "Lifelong ensemble learning based on multiple representations for few-shot object recognition," *arXiv preprint arXiv:2205.01982*, 2022.

[51] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, vol. 3, pp. 2149–2154, IEEE, 2004.

[52] R. L. Cook, "Stochastic sampling in computer graphics," *ACM Transactions on Graphics (TOG)*, vol. 5, no. 1, pp. 51–72, 1986.

[53] S. Kirkpatrick, C. D. Gelatt Jr, and M. P. Vecchi, "Optimization by simulated annealing," *science*, vol. 220, no. 4598, pp. 671–680, 1983.

[54] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32* (H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, eds.), pp. 8024–8035, Curran Associates, Inc., 2019.

[55] E. Rojas, F. Quirós-Corella, T. Jones, and E. Meneses, "Large-scale distributed deep learning: A study of mechanisms and trade-offs with pytorch," in *Latin American High Performance Computing Conference*, pp. 177–192, Springer, 2022.

[56] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM*, vol. 24, no. 6, pp. 381–395, 1981.

[57] A. Rasouli and J. K. Tsotsos, "The effect of color space selection on detectability and discriminability of colored objects," *arXiv preprint arXiv:1702.05421*, 2017.

[58] B. C. Russell, A. Torralba, K. P. Murphy, and W. T. Freeman, "Labelme: a database and web-based tool for image annotation," *International journal of computer vision*, vol. 77, no. 1, pp. 157–173, 2008.

[59] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.